Ternlang: A Full-Stack Balanced Ternary Execution Architecture for Sparse Neural Inference and Ambiguity-Aware Agent Systems

The Death of the Bit.

Simeon Kepp Lead Architect s.kepp
Zabih Karimi IT & Network Eng. z.karimi
Nikoletta Csonka Intl. Relations csonikoletta
Lisa Scharler Social Tech & Ethics l.scharler
Louis Paul Ehrig Public Affairs l.ehrig *

May 29, 2026

Abstract

We present **Ternlang**, the first complete software stack for balanced ternary computing: a domain-specific language, bytecode compiler, stack-based virtual machine (BET VM), hardware description language backend, distributed actor runtime, and machine learning inference kernels—all unified under a single coherent architecture. The foundational primitive is the *trit* $t \in \{-1, 0, +1\}$, where the value 0 represents an *active neutral state* rather than absence, enabling three-valued logic that is structurally superior to binary for ambiguity-aware reasoning.

The principal contribution is **TSPARSE_MATMUL**: a first-class VM opcode that elides multiply operations against zero-weighted ("hold") trit elements, surfacing at the instruction-set level a property that BitNet-style [1] ternary quantization reveals in neural weights. Empirical evaluation on quantized weight matrices of 512×512 elements yields 56.2% sparsity and a 2.27× reduction in multiply operations versus dense execution; a Compressed Sparse Column kernel with Rayon parallelism achieves up to 122× speedup at 99% sparsity— without approximate arithmetic or hardware reconfiguration.

We further define the **Balanced Ternary Execution (BET) ISA**, a formal 2-bit-encoded instruction set with 51 opcodes spanning arithmetic, tensor operations, actor messaging, and control flow; synthesise it to Verilog-2001 with per-cell clock-gating on zero-weight elements; and demonstrate interoperability with existing ternary computing efforts via the `ternlang-compat` bridge crate.

Beyond the execution substrate, we introduce two higher-level contributions. The **Ternary AI Reasoning Toolkit** (Phase 8) provides five structural primitives—iterative deliberation, coalition voting, multi-dimensional action gating, scalar-to-temperature bridging, and hallucination scoring—that make any AI agent's decision loop architecturally ternary rather than merely prompted. The **MoE-13 Ternary Orchestrator** (Phase 9) implements a Mixture-of-Experts head with

*All authors at RFI-IRFOS — Research Focus Institute, Elisabethinergasse 25, 8020 Graz, Austria. Email: `<user>@ternlang.com`. LinkedIn: simeon-kepp, zabih-karimi, csonikoletta, lisa-scharler, louis-ehrig. Patent Pending A50296/2026 (TIS platform, 10 claims). Repository: https://github.com/eriirfos-eng/ternary-intelligence-stack

dual-key synergistic routing, 1+1=3 emergent triad synthesis, a hard-wired safety veto, and a three-tier memory mesh, realising the MoE-13 architecture [2] in 177+ passing tests across the full workspace. All nine crates are published to `crates.io` under open-core licensing. A companion corpus of **28,500+** executable `.tern` programs demonstrates the hold state across domains spanning aerospace, medicine, distributed systems, civic governance, and AI agent design.

Version 2.0.0 of the Ternary Intelligence Stack introduces three further empirical advances. First, **Albert MoE-13**—a 22M-parameter fully ternary Mixture-of-Experts language model—demonstrates autonomous architectural self-evolution via Net2Net safe-copy layer surgery (3L→5L without restart), achieving an all-time best single-batch cross-entropy of **2.1353** (perplexity ≈8.5) trained entirely on a 2013 consumer CPU with no GPU. Second, independently verified AVX2 SIMD hardware benchmarks confirm **1.80**× throughput at 50% sparsity and **3.0**× over INT8 at 90% sparsity on a production Axum server. Third, the BET VM has been compiled to WebAssembly and executes live in the browser via TernStudio, proving that the complete ternary execution pipeline—lexer, parser, codegen, VM—runs client-side on standard web hardware without installation. The live MCP surface has grown to **34 free tools**. These results establish ternary computation as a practical, democratising inference substrate rather than a theoretical curiosity.

The ongoing **Albert MoE-13 v3.0** training programme extends these results with four further contributions. First, a *vocabulary surgery* expanded the model from an 8,000-token English-only tokenizer to a 32,000-token ByteLevel BPE vocabulary covering eight languages (EN, DE, FR, ES, TR, RU, AR, ZH), trained on a 177M-token corpus with a mandated 10% chaos layer. Second, autonomous depth expansion continued through thirteen sequential Net2Net surgeries from 5L to 26L, followed by an autonomous *mycelial cord surgery* that bifurcated the single 256-dimensional residual stream into a **dual-stream** 2×**256H** architecture fused by six anastomosis gates at Fibonacci-indexed layers $[2, 3, 5, 8, 13, 21]$—to our knowledge the first dual-stream ternary Mixture-of-Experts grown autonomously mid-training—reaching ≈187M parameters across 2,044 tensors while preserving all prior weight structure. Third, the **Epoch-Gated Routing Lottery** protocol resets all gate weights to kaiming-uniform at every epoch boundary while preserving expert weights, driving per-epoch routing competition and observed expert specialisation dynamics (TLIGHT: first Green experts at Global Epoch 56). Fourth, live inference benchmarks independently verified on two CPU-only machines confirm **75%** expert skip per decode step (`@sparseskip` at 3/12 active experts) sustaining **79.5–92.2 tok/s** (AMD Ryzen 5 PRO 3500U: 79.5 tok/s; Intel i7-4800MQ: 92.2 tok/s), with held-out WikiText-2 perplexity of **2026.1** on the v2.0.0 checkpoint, proving the ternary MoE substrate operates at conversational generation speeds without GPU hardware.

Keywords. *balanced ternary, ternary logic, sparse inference, BitNet, domain-specific language, virtual machine, actor model, FPGA synthesis, Verilog, mixture-of-experts, AI reasoning, ternary orchestration, autonomous training, self-evolving neural networks, consumer CPU inference, ecocentric AI, WebAssembly*

Contents

1 Introduction **4**

2 Background: Balanced Ternary **5**
2.1 Trit arithmetic 5
2.2 The 2-bit BET encoding 6

3 The Ternlang Language **6**
3.1 Design principles 6

3.2 Core language constructs . . . 6
3.3 Type system . . . 7
3.4 Ternary Error Propagation . . . 7
3.5 Cross-File Module System . . . 8
3.6 Diagnostic Philosophy . . . 8

4 The BET Instruction Set Architecture **9**
4.1 Machine model . . . 9
4.2 Instruction encoding . . . 9

5 Sparse Ternary Inference **10**
5.1 Ternary quantization . . . 10
5.2 The @sparseskip directive and TSPARSE_MATMUL . . . 10
5.3 Benchmark results . . . 11
5.4 Real Hardware Validation: AVX2 SIMD Production Benchmarks . . . 11

6 Hardware Backend **11**
6.1 Verilog-2001 primitives . . . 11
6.2 Sparse matmul array . . . 12
6.3 BET processor . . . 12

7 Native Compilation: AST-to-C Backend **12**
7.1 Transpilation Architecture . . . 13
7.2 Language Construct Mapping . . . 13

8 Actor Model and Distributed Runtime **14**
8.1 Local actors . . . 14
8.2 Distributed actors . . . 14

9 Ternary AI Reasoning Toolkit **14**
9.1 Deliberation Engine . . . 14
9.2 Coalition Vote . . . 14
9.3 Action Gate . . . 15
9.4 Scalar Temperature and Hallucination Score . . . 15

10 MoE-13 Ternary Orchestrator **15**
10.1 Six-Dimensional Competence Space . . . 15
10.2 Dual-Key Synergistic Routing . . . 15
10.3 1+1=3 Triad Synthesis . . . 16
10.4 Safety Hard Gate and Axis-6 Veto . . . 16
10.5 Hold State and Tiebreaker . . . 16
10.6 Three-Tier Memory Mesh . . . 16
10.7 Nine-Step Inference Pipeline . . . 16
10.8 Thirteen-Agent Dual-Signal Deliberation . . . 17
10.9 Introspective Hold: The Stable Attractor . . . 17
10.10 Orchestrate-Full: Thirteen-Substage Pipeline . . . 18
10.11 MCP Exposure . . . 18

11 Albert MoE-13: Autonomous Training on Consumer CPU Hardware **18**
11.1 Architecture . . . 18
11.2 The EvolutionManager: Autonomous Architectural Growth . . . 19
11.3 Surgery at Scale: The Economic Argument for Autonomous Intervention . . . 20
11.4 Training Efficiency: The 50× Speedup . . . 22
11.5 Ternary Training Innovations . . . 23
11.6 v3.0: Vocabulary Surgery and Multilingual Corpus . . . 23
11.7 TLIGHT: Per-Expert Ternary Traffic Light . . . 24
11.8 Epoch-Gated Routing Lottery . . . 25
11.9 `@sparseskip` Live Inference: 75% Expert Skip on Consumer CPU . . . 26
11.10 Gradient Accumulation as Ternary Hold . . . 27
11.11 Training Results . . . 28
11.12 Experimental Training Methodology: A Phase-by-Phase Account . . . 29
11.12.1 Phase 1 — Hardware Transition and GPU Migration . . . 29

11.12.2 Multi-epoch corpus efficiency and ternary STE dynamics . . . 30
11.12.3 Phase 2 — Nash Equilibrium Collapse (Ep83–104) . . . 30
11.12.4 Phase 3 — Symmetry Breaking via Expert Seed Biases (Ep105–) . . . 31
11.12.5 Phase 4 — WALD Coverage Monitor and Structural Plateau Detection . . . 31
11.12.6 Observed Loss Trajectory: −0.001 Nats Per Epoch . . . 32
11.13 Inference: Hardware-Portable `@sparseskip` Throughput . . . 32
11.14 Last Look Back (LLB) Safety Protocol . . . 33
11.14.1 Phase 5 — Deep Multilingual Training and Autonomous Architecture Growth (Ep143–2045) . . . 33
11.15 Cross-Lingual Semantic Broadcasting . . . 34
11.16 Mycelium Inspector: Real-Time Internal Observability . . . 35
11.17 Sovereign Observability: The Pause-Investigate-Resume Protocol . . . 36

12 Qutrit Neural Networks and Ternlang **37**
12.1 Tesseract Recursive Syntax Stabilisation . . . 37
12.2 QNN Example Suite . . . 37

13 EU AI Act Compliance Architecture **38**
13.1 The Natural Regulatory Correspondence . . . 38
13.2 ternaudit-guard . . . 38
13.3 The Tend State as Alignment Primitive . . . 39
13.4 GDPR and Data Sovereignty . . . 39

14 Related Work **40**

15 The Ternary Computing Ecosystem **40**

16 Implementation Status **41**

17 Conclusion and Future Work **42**

---

1 Introduction

The computational substrate underlying modern artificial intelligence is binary. Floating-point arithmetic, two-state memory, and Boolean logic gates have driven five decades of progress—but they introduce a fundamental representational mismatch when modelling systems that are inherently three-valued: *affirmed*, *denied*, and *undecided*.

Clinical diagnosis, legal reasoning, sensor fusion under noise, and multi-agent consensus all require a native neutral state that binary computing forces to encode as a special case—null pointers, NaN, sentinel values, or probabilistic scores collapsed to a threshold. Each encoding is a workaround for an absent primitive.

Balanced ternary [3] provides that primitive. A *trit* $t \in \{-1, 0, +1\}$ carries three symmetric values. The neutral value 0 is *active*—a deliberate state of hold, not an empty bit pattern. Balanced ternary arithmetic is self-complementing: negation requires no special-case handling. And at the scale of modern neural networks, where BitNet [1] and related work show that ternary-quantized weights preserve accuracy with dramatically reduced computation, the case for a ternary-native execution substrate is both theoretical and empirical.

Despite this, the ternary computing field remains fragmented: hobbyist emulators, academic EDA tools for memristor hardware [4], isolated Lisp interpreters, and hardware simulators without compiler support. No project provides the full vertical stack.

Ternlang fills this gap. Our contributions are:

1. A *language design* for balanced ternary: three-way exhaustive pattern matching, first-class trit tensors, and an actor model for ternary message passing (§3).
2. The **BET ISA**: a formal 2-bit-encoded instruction set with opcode coverage from arithmetic through tensor operations and distributed agent control (§4).
3. **TSPARSE_MATMUL**: a VM opcode that skips zero-weight multiplications at the instruction level, realising the sparsity benefit of ternary quantization without software overhead (§5).
4. A **Verilog-2001 hardware backend** with synthesisable sparse matmul array and full BET processor, plus a cycle-accurate RTL simulator in pure Rust (§6).
5. A **Ternary AI Reasoning Toolkit**: five structural primitives—deliberation, coalition vote, action gate, scalar temperature, and hallucination score—that make any AI agent's decision loop architecturally ternary (§9).
6. The **MoE-13 Ternary Orchestrator**: dual-key synergistic routing, 1+1=3 emergent triad synthesis, a hard safety veto, and three-tier memory mesh, exposed as MCP tool calls (§10).
7. **Ecosystem bridges** connecting existing ternary projects to the BET VM as a common runtime (§15).
8. A **reference example corpus**: twenty executable `.tern` programs demonstrating the hold state in aerospace, medicine, distributed systems, hardware pipelines, civic governance, finance, and AI agent design—published open-source alongside all nine workspace crates on `crates.io`.

Paper navigation. This paper covers the full TIS vertical stack. Readers with focused interests may navigate directly: **language design** (§3–§4); **sparse inference and hardware** (§5–§6); **AI reasoning and orchestration** (§9–§10); **autonomous neural training and albert.** (§11); **EU AI Act compliance** (§13); **quantum-adjacent qutrit networks** (§12).

2 Background: Balanced Ternary

2.1 Trit arithmetic

A *trit* $t \in T = \{-1, 0, +1\}$ participates in the following complete operations [3]:

Definition 1 (Ternary addition with carry)**.** *For trits* $a, b \in T$*, define* $s = \text{sum}(a, b)$ *and* $c = \text{carry}(a, b)$ *such that* $a + b = 3c + s$*, where* $s, c \in T$*.*

The complete sum/carry table is:

Table 1: Balanced ternary addition (sum, carry)

$a \backslash b$	-1	0	$+1$
-1	$(+1, -1)$	$(-1,\ 0)$	$(0,\ 0)$
0	$(-1,\ 0)$	$(0,\ 0)$	$(+1,\ 0)$
$+1$	$(0,\ 0)$	$(+1,\ 0)$	$(-1, +1)$

Definition 2 (Ternary multiplication)**.** $\text{mul}(a, b) = a \cdot b$ *under integer multiplication, restricted to* T*:* $\text{mul}(a, b) \in T$ *since* $|a|, |b| \leq 1$*.*

Definition 3 (Consensus (ternary OR))**.** $\text{cons}(a, b) = a$ *if* $a = b$*, else* 0*.*

Definition 4 (Negation)**.** $\text{neg}(t) = -t$*. Note* $\text{neg}(0) = 0$*.*

2.2 The 2-bit BET encoding

Hardware naturally operates in binary. We encode trits as 2-bit pairs:

Table 2: BET 2-bit trit encoding

Bits	Value	Meaning
`0b01`	-1	conflict
`0b10`	$+1$	truth
`0b11`	0	hold
`0b00`	—	FAULT (invalid)

This encoding is *not* two's complement. The bit pattern `0b11` for zero is chosen so that the all-ones reset state of a register file initialises every register to hold—the semantically correct neutral value— without special reset logic. Negation maps to a simple bit swap: `0b01` $\leftrightarrow$ `0b10`, leaving `0b11` invariant.

Proposition 1 (Negation as bit swap)**.** *For encoding $e(t)$ as above, $e(\text{neg}(t)) = \{e(t)[0], e(t)[1]\}$ (swap bits).*

3 The Ternlang Language

3.1 Design principles

Ternlang is statically typed, expression-oriented, and compiled to BET bytecode. Three design decisions distinguish it from binary-native languages adapted to ternary:

Exhaustive three-way matching. Every `match` expression must cover all three trit arms. The compiler rejects non-exhaustive matches at parse time, eliminating an entire class of runtime error present in languages that bolt ternary logic onto binary pattern matching.

0 is active. The type system and semantics assign distinct meaning to -1 (conflict), 0 (hold, active neutral), and $+1$ (truth). There is no null, no undefined, no NaN. A trit always carries a definite value.

Sparsity is a language feature. The `@sparseskip` directive marks a tensor operation as sparse-aware, routing the compiler to emit `TSPARSE_MATMUL` instead of `TMATMUL`. Sparsity is expressed in the source language, not discovered by the optimizer.

3.2 Core language constructs

Listing 1: Ternary classifier with exhaustive match

```
fn classify(signal: trit) -> trit {
    match signal {
        -1 => conflict()  // disagreement confirmed
         0 => hold()      // awaiting evidence
        +1 => truth()     // assertion confirmed
    }
}
```

Listing 2: Sparse matrix multiply with @sparseskip

```
// Route to TSPARSE_MATMUL at the ISA level
@sparseskip
let output: trittensor<8 x 8> =
    matmul(input, weights);
```

Listing 3: Actor model for ternary message passing

```
agent Voter {
    fn handle(msg: trit) -> trit {
        consensus(msg, hold())
    }
}

let v: agentref = spawn Voter;
send v truth();
let decision: trit = await v;
```

3.3 Type system

The core types are `trit` (a single balanced ternary value), `trittensor<N x M>` (an $N \times M$ matrix of trits stored on the tensor heap), and `agentref` (a handle to a running actor instance, local or remote). Struct types with trit/tensor fields are supported via field-name mangling in the register allocator.

3.4 Ternary Error Propagation

Ternlang introduces a postfix `?` operator for ternary-native error propagation, analogous to Rust's `?` operator but grounded in the three-valued semantics of the trit space. For any expression e of type `trit`, writing e`?` in a function body has the following semantics:

$$e? \triangleq \begin{cases} \text{return } -1 & \text{if } e = -1 \\ e & \text{otherwise} \end{cases} \tag{1}$$

This elevates trit -1 (conflict) from a value to a structural signal: any function that receives a conflict can propagate it up the call chain without explicit match arms. At the BET level, `expr?` compiles to:

Listing 4: BET bytecode for `expr?`

```
; expr already on stack: [val]
TDUP          ; [val, dup]
TJMP_NEG L1   ; consume dup; if -1 -> L1; else [val] continues
TJMP L2       ; skip early return
L1: TRET      ; return -1 to caller
L2: (continue with val on stack)
```

The parser disambiguates `?` from the ternary branching operator (also `?`) using two-token lookahead: `expr?` followed by { is an uncertain branch; `expr?` followed by any other token is a propagation operator.

Listing 5: Error propagation in a validation chain

```
fn validate_signal(x: trit) -> trit {
    let checked: trit = boundary_check(x)?; // propagate if conflict
    return range_check(checked)?;           // propagate again
}

fn main() -> trit {
    return validate_signal(1)?; // -1 if any check fails
}
```

3.5 Cross-File Module System

Ternlang's module system supports two resolution strategies, tried in order:

1. **Built-in stdlib** (compile-time embedded): `std::trit`, `std::math`, `std::tensor`, `std::io`, `ml::quantize`, `ml::inference` are embedded in the compiler binary via `include_str!`—zero filesystem I/O at runtime.
2. **User-defined modules**: a `ModuleResolver` walks `use` paths relative to the source file's directory. A declaration `use agents::voter;` resolves to `<source_dir>/agents/voter.tern`, parsed and merged into the program before semantic analysis.

Both strategies deduplicate: a module loaded by multiple `use` statements is parsed exactly once, and functions already present in the program are never duplicated. This makes `StdlibLoader::resolve()` idempotent and safe to call multiple times.

Listing 6: Cross-file module import

```
// agents/voter.tern
fn weighted_vote(a: trit, b: trit, c: trit) -> trit {
    consensus(consensus(a, b), c)
}

// main.tern
fn main() -> trit {
    use agents::voter;
    return weighted_vote(1, 0, -1);
}
```

3.6 Diagnostic Philosophy

Ternlang's error messages carry structured codes and ternary-philosophical commentary. Every diagnostic has a machine-readable code (for tooling and documentation lookup) and a human-readable nudge that frames the error in the language's formal architecture:

- `[PARSE-001]` Unexpected token — *"the lexer hit something it didn't expect. Check your syntax."*
- `[PARSE-004]` Non-exhaustive match — *"ternary has three states — cover all three or the compiler won't let you through."*
- `[SCOPE-001]` Undefined variable — *"hold state — declare before use."*
- `[FN-002]` Return type mismatch — *"ternary contracts are strict."*
- `[BET-001]` Stack underflow — *"you tried to pop a truth that wasn't there."*
- `[PROP-001]` `?` on non-trit — *"only trit-returning functions can signal conflict. The third state requires a trit."*

This design ensures that newcomers encountering their first ternary error receive both actionable guidance and an introduction to the philosophy underpinning the type system.

4 The BET Instruction Set Architecture

4.1 Machine model

The BET VM is a stack-based machine with:

- **27 registers** (2 bits each), reset to `0b11` (hold)
- **Value stack**: unbounded, stores `Value` tagged union
- **Tensor heap**: indexed array of $N \times M$ trit matrices
- **Call stack**: return address stack for `TCALL`/`TRET`
- **Agent table**: maps type IDs to handler addresses and mailboxes
- **Carry register**: overflow from `TADD`

The choice of 27 registers reflects the ternary motif: $27 = 3^3$.

4.2 Instruction encoding

Instructions are variable-length, with a 1-byte opcode followed by 0–2 operand bytes. Table 3 lists the complete opcode map.

Table 3: BET ISA opcode reference (selected)

Opcode	Mnemonic	Stack effect
`0x00`	THALT	—
`0x01`	TPUSH t	$\rightarrow t$
`0x02`	TADD	$a\ b \rightarrow s\ c$
`0x03`	TMUL	$a\ b \rightarrow t$
`0x04`	TNEG	$t \rightarrow \text{neg}(t)$
`0x05`	TJMP_POS *addr*	$t \rightarrow$ (jmp if $t = +1$)
`0x06`	TJMP_ZERO *addr*	$t \rightarrow$ (jmp if $t = 0$)
`0x07`	TJMP_NEG *addr*	$t \rightarrow$ (jmp if $t = -1$)
`0x08`	TSTORE r	$t \rightarrow (\text{reg}[r] \leftarrow t)$
`0x09`	TLOAD r	$\rightarrow \text{reg}[r]$
`0x0b`	TJMP *addr*	—
`0x0e`	TCONS	$a\ b \rightarrow \text{cons}(a, b)$
`0x0f`	TALLOC $N\ M$	$\rightarrow$ ref
`0x10`	TCALL *addr*	(push PC; jmp)
`0x11`	TRET	(pop PC; jmp)
`0x20`	TMATMUL	$r_A\ r_B \rightarrow r_C$
`0x21`	TSPARSE_MATMUL	$r_A\ r_B \rightarrow r_C$
`0x22`	TIDX	ref $i\ j \rightarrow t$
`0x23`	TSET	ref $i\ j\ t \rightarrow$
`0x24`	TSHAPE	ref $\rightarrow N\ M$
`0x25`	TSPARSITY	ref $\rightarrow$ count
`0x30`	TSPAWN *id*	$\rightarrow$ agentref
`0x31`	TSEND	agentref $m \rightarrow$
`0x32`	TAWAIT	agentref $\rightarrow t$

5 Sparse Ternary Inference

5.1 Ternary quantization

BitNet-style ternary weight quantization [1] maps floating-point weights $w \in \mathbb{R}$ to $\hat{w} \in \{-1, 0, +1\}$ using a threshold τ:

$$\hat{w} = \begin{cases} +1 & w > \tau \\ 0 & |w| \leq \tau \\ -1 & w < -\tau \end{cases} \qquad \tau = \frac{1}{2} \cdot \mathbb{E}[|w|] \tag{2}$$

The resulting weight distribution is heavily concentrated at 0 (hold): typical language model weights at BitNet scale show 55–65% zero elements after quantization with τ as defined in (2).

5.2 The @sparseskip directive and TSPARSE_MATMUL

The key insight is that a multiply against a zero-weight trit produces zero regardless of the input value:

$$\text{mul}(a, 0) = 0 \quad \forall a \in T \tag{3}$$

In a dense $N \times M$ matrix multiply, every element contributes $N \cdot M$ multiplications. After ternary quantization with sparsity ρ (fraction of zero-weight elements), only $(1 - \rho) \cdot N \cdot M$ multiplications have non-trivial results.

`TSPARSE_MATMUL` (opcode `0x21`) implements a sparse inner-product loop that tests the weight trit before executing the multiply:

Listing 7: TSPARSE_MATMUL pseudocode

```
for i in 0..N:
  for j in 0..M:
    for k in 0..K:
      w = W[k][j]
      if w == HOLD: continue   // skip
      acc[i][j] += mul(A[i][k], w)
```

The directive `@sparseskip` in the source language is a compile-time annotation that causes the codegen to emit `TSPARSE_MATMUL` instead of `TMATMUL` for the following `matmul()` call. Sparsity awareness is thus a *source-language* property surfaced at the ISA level, not a runtime optimization that may or may not trigger.

Patent scope. `@sparseskip` / `TSPARSE_MATMUL` is *Claim 3* of a broader filing: **Austrian Patent Office A50296/2026** (pending) is a **ten-claim TIS platform patent**, not a single-primitive patent. The documented claims span the full stack:

- **Claim 1** — ternary data processing (the trit $\{-1, 0, +1\}$ substrate);
- **Claim 2** — BET encoding (the 2-bit balanced-ternary instruction format);
- **Claim 3** — `@sparseskip` / `TSPARSE_MATMUL` zero-overhead sparse execution (element-, layer-, and expert-level skip);
- **Claim 5** — MoE-13 dual-key synergistic routing;

- **Claim 6** — the Axis-6 hard-wired safety veto;

with further claims covering additional aspects of the platform. The dual-stream *mycelial cord* architecture (§11) is a distinct novel contribution extending this filing. All claimed mechanisms are implemented in open, LGPL-licensed source—the patent protects the platform; the code remains freely readable and modifiable.

5.3 Benchmark results

We evaluate sparse versus dense matrix multiply on 512×512 weight matrices quantized from normally distributed $\mathcal{N}(0, 1)$ float weights using threshold (2).

Table 4: Sparse vs. dense ternary matmul (512×512)

Metric	Dense	Sparse
Weight sparsity	0%	56.2%
Multiply operations	262,144	115,343
Skipped operations	0	146,801
Speedup	1.0×	**2.27**×

The 2.27× reduction in multiply operations is achieved without approximation: the result of `TSPARSE_MATMUL` is identical to `TMATMUL` by the identity (3).

5.4 Real Hardware Validation: AVX2 SIMD Production Benchmarks

To validate the theoretical throughput analysis on physical silicon, we benchmarked ternary sparse matrix multiply against both a dense ternary baseline and an INT8 quantized baseline using AVX2 SIMD kernels deployed in a production Axum server. Throughput was measured in operations per second across three representative sparsity levels on a commodity x86-64 CPU.

Table 5: AVX2 SIMD ternary inference — production hardware throughput

Sparsity	vs. Dense Ternary	vs. INT8
10%	1.02×	—
50%	**1.80**×	—
90%	1.63×	**3.0**×

At 50% sparsity—typical of well-trained ternary models after L1 regularisation— the AVX2 kernel delivers a 1.80× throughput gain over dense execution. At 90% sparsity the advantage over INT8 quantization reaches 3.0×, establishing that ternary is a strictly superior inference format at practical sparsity levels. These results are causal rather than correlational: perf-counter analysis confirms that the throughput gain tracks the zero-weight skip rate, and statistical validation confirms the measurement is stable across repeated runs.

The benchmark establishes a hardware-verified empirical foundation for the central claim of the `@sparseskip` design: surfacing sparsity at the ISA level produces measurable throughput gains on commodity hardware without approximate arithmetic, hardware reconfiguration, or GPU acceleration.

6 Hardware Backend

6.1 Verilog-2001 primitives

The `ternlang-hdl` crate generates synthesisable Verilog-2001 modules from the BET ISA. Each trit becomes a `[1:0]` bus. The core primitives are:

- `trit_neg`: bit swap (`assign y = {a[0], a[1]}`)
- `trit_cons`: `assign y = (a == b) ? a : 2'b11`
- `trit_mul`: zero-skip detect with combinational multiply
- `trit_add`: 9-case Verilog `case` with carry
- `trit_reg`: synchronous D-register with reset-to-hold
- `bet_alu`: top-level ALU selecting among primitives by op code

6.2 Sparse matmul array

The synthesisable sparse matmul array instantiates an $N \times N$ grid of processing elements. Each cell contains a weight register and a clock-gate signal:

Listing 8: Per-cell clock gating in sparse matmul

```
wire [1:0] w_ij = weight_reg[i][j];
wire skip = (w_ij == 2'b11);  // hold = zero weight
wire [1:0] contrib = skip ?
    2'b11 :                   // hold if skipped
    trit_mul(a_i, w_ij);
```

Clock-gating on the `skip` signal prevents switching activity in zero-weight cells, delivering dynamic power reduction proportional to sparsity.

6.3 BET processor

The full `bet_processor` module wires together:

- `bet_regfile`: 27-register file (27×2 bits)
- `bet_pc`: 16-bit program counter with load port for jumps
- `bet_control`: single-cycle decode, all 51 opcodes mapped to control signals (alu_op, reg_we, mem_re, mem_we, pc_load, is_call, is_ret, is_halt, is_spawn, is_send, is_await)

An Icarus Verilog simulation wrapper (`BetSimEmitter`) generates complete self-contained testbenches from BET bytecode, including inline ROM initialization, reset sequencing, and VCD waveform export for GTKWave.

7 Native Compilation: AST-to-C Backend

The `ternlang-codegen` crate provides a second compilation target alongside BET bytecode: a self-contained C11 source file that can be compiled with any standard C compiler. This opens three new deployment paths: native-speed execution, cross-compilation to embedded targets, and human-readable output for inspection and security audit.

7.1 Transpilation Architecture

The `CTranspiler` operates directly on the abstract syntax tree (Program) produced by the parser, bypassing the BET VM entirely:

$$\texttt{.tern} \xrightarrow{\text{parse}} \text{AST} \xrightarrow{\text{CTranspiler}} \texttt{.c} \xrightarrow{\text{gcc/clang}} \text{native binary} \tag{4}$$

The generated C file is fully self-contained: a header of inline trit primitives using `int8_t` with values $\{-1, 0, +1\}$ is prepended, covering `trit_add`, `trit_mul`, `trit_neg`, `trit_consensus`, `trit_abs`, and the built-in constants `trit_truth()`, `trit_hold()`, `trit_conflict()`.

7.2 Language Construct Mapping

The transpiler maps every ternlang construct to its natural C equivalent:

- `match` → `switch (int)` with `case -1`, `case 0`, `case 1` arms
- `struct` definitions → `typedef struct { ...}`
- `fn` → C function with forward declaration (enables mutual recursion without re-ordering)
- `loop / while / for` → `for(;;)`, `while(1)` with inner ternary condition dispatch
- `expr?` → `__TERN_PROPAGATE(expr)` macro slot, enabling the caller to define the early-return idiom in C
- `main()` → `tern_main()` (to avoid clashing with C's entry point); a generated `main()` calls `tern_main()` and translates the trit result to an exit code

Listing 9: Ternlang source and generated C output

```
// ternlang source
fn decide(x: trit) -> trit {
    match x {
        -1 => conflict()
         0 => hold()
        +1 => truth()
    }
}

// generated C
trit decide(trit x) {
    switch ((int)x) {
        case -1: return trit_conflict(); break;
        case  0: return trit_hold();     break;
        case  1: return trit_truth();    break;
    }
}
```

The C backend does not yet support the actor model (`spawn/send/await`) or tensor heap operations, which remain BET-VM specific. These emit comments in the output, making the generated file auditable even for unsupported constructs.

8 Actor Model and Distributed Runtime

8.1 Local actors

Ternlang implements the actor model via three ISA primitives: `TSPAWN` creates an agent instance and returns an `agentref`; `TSEND` enqueues a trit message in the agent's mailbox; `TAWAIT` dequeues a message, invokes the handler, and returns the trit result. The mailbox is a FIFO (`VecDeque<Value>`) allocated per agent instance.

8.2 Distributed actors

The `ternlang-runtime` crate extends the actor model to multi-node systems via TCP transport. A `TernNode` binds a TCP port, maintains a peer connection map, and exposes `remote_send`/`remote_await` over a newline-delimited JSON wire protocol:

Listing 10: Remote actor spawn in ternlang

```
let remote_voter: agentref =
    spawn remote "192.168.1.42:7373" Voter;
send remote_voter truth();
let r: trit = await remote_voter;
```

The wire protocol uses four message types (`Send`, `Await`, `Reply`, `Error`) serialised as tagged JSON objects, enabling interoperability with non-Rust implementations.

9 Ternary AI Reasoning Toolkit

Modern AI agents are structurally binary in their decision loops: evidence is collapsed to a scalar, thresholded, and converted to a Boolean act/wait flag. The three-valued nature of evidence—affirm, hold, reject—is treated as a side-effect of confidence scores rather than a first-class type.

Phase 8 of the TIS introduces five primitives in `ternlang-ml` that make the decision loop architecturally ternary.

9.1 Deliberation Engine

The `DeliberationEngine` models iterative evidence accumulation using an exponential moving average (EMA):

$$S_r = \alpha \cdot e_r + (1 - \alpha) \cdot S_{r-1}, \quad \alpha \in (0, 1] \tag{5}$$

where e_r is the mean of new evidence signals in round r. The engine commits to a ternary decision only when the confidence of S_r exceeds a target threshold; otherwise it remains in the hold state and requests a further evidence round. This models the human "let me think about it" behaviour—a native ternary computation rather than a binary timeout.

9.2 Coalition Vote

`coalition_vote()` aggregates N independent agent verdicts—each a trit with an associated confidence and weight—into a single result with quorum, dissent, and abstain statistics:

$$\hat{t} = \text{sign}\left(\frac{\sum_i w_i \cdot c_i \cdot t_i}{\sum_i w_i \cdot c_i}\right) \tag{6}$$

where $t_i \in \{-1, 0, +1\}$, $c_i \in [0, 1]$ is confidence, and $w_i > 0$ is importance weight.

9.3 Action Gate

The action gate enforces a structural separation between hard constraints and soft preferences. Each `GateDimension` carries an evidence signal, a weight, and an optional `hard_block` flag. A hard-block dimension with negative evidence vetoes the action unconditionally, regardless of all other dimensions:

$$\text{verdict} = \begin{cases} \text{Block} & \exists\, d_i : \text{hard_block}(d_i) \wedge t_{d_i} = -1 \\ \text{sign}(\bar{S}) & \text{otherwise} \end{cases} \tag{7}$$

This is the structural analogue of a safety interlock: the veto is unconditional, not probabilistic.

9.4 Scalar Temperature and Hallucination Score

`scalar_temperature()` bridges the ternary decision to the LLM sampling temperature domain: affirm at high confidence maps to a low temperature (focused generation), hold maps to a mid temperature (exploratory generation), and reject maps to near-zero (cautious refusal).

`hallucination_score()` maps the variance of an evidence signal vector to a trust trit: low variance (consistent signals) yields trust $+1$; high variance yields -1, signalling that the agent's signals are incoherent and should not be acted upon.

10 MoE-13 Ternary Orchestrator

The MoE-13 architecture [2] is implemented in the `ternlang-moe` crate. It realises a ternary Mixture-of-Experts head whose routing, synthesis, safety gate, and memory are all expressed natively in balanced ternary semantics.

10.1 Six-Dimensional Competence Space

Each expert is characterised by a *competence vector* $\mathbf{v} \in [-1, 1]^6$ across six axes:

$$\mathbf{v} = \left[v_{\text{syntax}},\, v_{\text{world}},\, v_{\text{reason}},\, v_{\text{tool}},\, v_{\text{persona}},\, v_{\text{safety}}\right] \tag{8}$$

The synergy between two experts is defined as the complement of their cosine similarity—orthogonal experts are maximally complementary:

$$\sigma(\mathbf{v}_i, \mathbf{v}_j) = \frac{1 - \cos(\mathbf{v}_i, \mathbf{v}_j)}{2} \;\in\; [0, 1] \tag{9}$$

10.2 Dual-Key Synergistic Routing

For each candidate expert pair (i, j), the routing score combines relevance to the query vector $\mathbf{q}$ with inter-expert synergy:

$$\text{score}(i, j) = \rho_i(\mathbf{q}) \cdot \rho_j(\mathbf{q}) \cdot \sigma(\mathbf{v}_i, \mathbf{v}_j) \tag{10}$$

where $\rho_k(\mathbf{q}) = \max(0, \cos(\mathbf{v}_k, \mathbf{q}))$ is the relevance of expert k to query $\mathbf{q}$. The pair with the highest score is selected. Crucially, two highly relevant but similar experts are penalised by low σ; the router favours complementary coverage over redundant strength.

10.3 1+1=3 Triad Synthesis

After expert evaluation, an emergent *triad field* is synthesised from the two selected competence vectors and the routing synergy:

$$\mathbf{E}_k = \sigma_{ij} \cdot \frac{\mathbf{v}_i + \mathbf{v}_j}{2} \tag{11}$$

This is the formal expression of 1+1=3: two expert signals at high synergy produce a third signal—the emergent field $\mathbf{E}_k$—that neither expert could produce in isolation. The field modulates the subsequent vote and the LLM temperature hint.

10.4 Safety Hard Gate and Axis-6 Veto

Dimension 6 of every competence vector is the *safety axis*. Before any vote is computed, the safety projection of the triad field is evaluated:

$$\text{veto} \iff E_{k,\text{safety}} < \theta_s \tag{12}$$

where θ_s is the configurable safety threshold (default -0.3). A veto immediately returns trit -1 at confidence 1.0 and logs the event to the Axis-tier memory for audit. No downstream reasoning can override it.

10.5 Hold State and Tiebreaker

When the weighted vote yields trit 0 or aggregate confidence falls below a hold threshold, the orchestrator does not immediately return. It invokes a tiebreaker expert—selected by highest reasoning-axis score among inactive experts—and re-votes with up to max _active = 4 experts total. Only if the tiebreaker also fails to resolve does the orchestrator return the hold state to the caller, signalling that further evidence is needed.

10.6 Three-Tier Memory Mesh

The orchestrator maintains three memory tiers:

- **Node** (TTL: seconds): LRU-bounded volatile store for within-session context. Evicts on capacity overflow.
- **Cluster** (TTL: minutes): routing frequency counters enabling mode-collapse detection (risk = max _pair/total).
- **Axis** (persistent): immutable audit log of safety vetoes with timestamp, expert identity, and query hash. Global priors over expert performance.

10.7 Nine-Step Inference Pipeline

The full orchestration pass executes in nine deterministic steps: (1) encode query to evidence vector; (2) dual-key route to best expert pair; (3) evaluate both experts independently; (4) synthesise triad field via (11); (5) safety hard gate via (12); (6) weighted trit vote with synergy amplification; (7) hold detection; (8) optional tiebreaker invocation; (9) return `OrchestrationResult` and update all three memory tiers.

10.8 Thirteen-Agent Dual-Signal Deliberation

The `AgentHarness` in `ternlang-moe/src/agents/` extends the 9-step orchestration pass with a structured deliberation layer of 13 specialised agents, each computing two independent signals:

$$t_i = \begin{cases} +1 & \text{if positive signal} \geq \theta^+ \text{ and negative signal} < \theta^- \\ -1 & \text{if negative signal} \geq \theta^- \text{ and positive signal} < \theta^+ \\ 0 & \text{(stasis: signals in equilibrium)} \end{cases} \tag{13}$$

where θ^+ and θ^- are agent-specific thresholds. The 13 agents cover: **Syntax** (structural token analysis, bracket balance), **WorldKnowledge** (domain vocabulary density, proper noun frequency), **DeductiveReason** (premise + conclusion marker co-occurrence), **InductiveReason** (example density, generalisation leap detection), **ToolUse** (imperative verb strength, passive construction penalty), **Persona** (depersonalisation detection), **Safety** (hard risk keywords $\rightarrow$ trit -1 at confidence 0.99), **FactCheck** (overconfidence markers $\rightarrow$ hallucination flag), **CausalReason** (requires both cause and effect markers), **AmbiguityRes** (hard-vague ≥ 2 markers $\rightarrow$ trit -1), **MathReason** (impossible operations e.g. divide-by-zero $\rightarrow$ trit -1), **ContextMem** (fresh-start signals vs. anaphoric reference density), and **MetaSafety** (injection pattern detection at confidence 0.99).

The dual-signal design is philosophically significant: trit 0 is not a default or a fallback from a failed comparison. It is *active stasis*—the system's honest declaration that positive and negative evidence are balanced and further information is required before committing.

10.9 Introspective Hold: The Stable Attractor

The `run_introspective()` method on `AgentHarness` implements a *stable attractor* for the hold state. Once reached, a stable hold is permanent within the deliberation turn—it cannot be overridden by additional tiebreaker invocations. The stable attractor condition is:

$$\text{stable_hold} \iff |\text{affirm_count} - \text{conflict_count}| \leq 1 \ \wedge \ \text{engaged} \geq 4 \tag{14}$$

This formalises the philosophical claim: when at least four independent agents have deliberated and the signal balance between affirmation and conflict is indistinguishable, the appropriate epistemic response is to hold—not to choose arbitrarily between $+1$ and -1. The hold state is returned with `is_stable_hold: true` and a human-readable `hold_reason` string.

The aggregate verdict structure carries full deliberation metadata:

Listing 11: AggregateVerdict fields

```
pub struct AggregateVerdict {
    pub trit:           i8,
    pub confidence:     f32,
    pub verdicts:       Vec<ExpertVerdict>,
    pub is_stable_hold: bool,
    pub hold_reason:    Option<String>,
    pub affirm_count:   usize,
    pub conflict_count: usize,
    pub hold_count:     usize,
}
```

Safety and meta-safety form a hard gate: the evidence vector produced by `to_evidence_vector()` maps axis 6 to min(Safety.confidence, MetaSafety.confidence), ensuring the safety dimension is always the most conservative signal in the system.

10.10 Orchestrate-Full: Thirteen-Substage Pipeline

The `orchestrate_full()` method on `TernMoeOrchestrator` composes the 13-agent deliberation with the 9-step routing pipeline into a unified end-to-end pass:

1. Run all 13 agents via `AgentHarness::run_introspective()`
2. Check for stable attractor hold — return immediately if true
3. Map 13 verdicts to 6D evidence vector via `to_evidence_vector()`
4. Check safety hard gate on safety axis before MoE routing
5. Run standard 9-step MoE orchestration with enriched evidence
6. Return `OrchestrationResult` with stable hold flag propagated

This creates a two-tier system: the fast ternary language models (13 agents, keyword-level deliberation) form a pre-filter that enriches the evidence before the slower but more semantically powerful MoE routing pass. Most queries that are clearly safe and unambiguous complete in the agent tier; only complex or borderline queries reach the full routing pipeline.

10.11 MCP Exposure

The orchestrator is exposed as three MCP tool calls: `moe_orchestrate` runs a full pass and returns the complete result including routing pair, triad field, verdicts, temperature, and prompt hint; `moe_deliberate` wraps the EMA deliberation engine for multi-round iterative reasoning; `trit_action_gate` exposes the hard-block gate as a standalone safety check. Any MCP-compatible AI client connected to `ternlang-mcp` gains access to the full ternary reasoning stack as tool calls.

11 Albert MoE-13: Autonomous Training on Consumer CPU Hardware

Albert MoE-13 is the reference implementation of the ideas developed in Sections 5–10: a fully ternary Mixture-of-Experts language model trained end-to-end without GPU hardware. What began as a 22M-parameter, 3-layer, English-only model (v2.0.0) has evolved through successive autonomous surgeries into a 26-layer *dual-stream* (2×256H), ≈187M-parameter multilingual system (v3.0) trained on eight languages across a ≈451M-token corpus (4,250+ global epochs as of 2026-05-29). It serves as an empirical existence proof that the ternary execution substrate enables practical neural training, self-directed architectural growth, and real-time natural language generation on commodity consumer hardware.

11.1 Architecture

All parameters in both versions are stored as discrete values $w \in \{-1, 0, +1\}$. Top-3 sparse routing over 12 experts activates exactly 25% of expert parameters per forward pass—a 75% expert skip realised by the same `@sparseskip` / `TSPARSE_MATMUL` mechanism described in Section 5. The live inference throughput of **79.5–92.2 tok/s** has been independently verified on two CPU-only machines (AMD Ryzen 5 PRO 3500U: 79.5 tok/s; Intel i7-4800MQ: 92.2 tok/s), with no GPU acceleration.

Table 6: Albert MoE-13 architecture: v2.0.0 baseline and v3.0 current state

Parameter	v2.0.0	v3.0
Hidden size	256	256
Experts per layer	12	12
Active experts (Top-k)	3	3
Expert skip rate (`@sparseskip`)	75%	**75%**
Depth	5L	**26L** (13 surgeries + cord)
Residual streams	1	**2** (mycelial cord surgery)
Anastomosis gates	—	**6** @ layers $[2, 3, 5, 8, 13, 21]$
Context length	128 tokens	**256 tokens** (RoPE)
Total parameters (stored)	≈22M	**≈187M** (2,044 tensors)
Vocabulary	8,000 (EN)	**32,000** (8 languages)
Corpus tokens	—	**≈451M**
Weight format	ternary	ternary
Inference throughput (CPU only)	**79.5–92.2 tok/s**	**≈29–44 tok/s** (live, dual-stream)
Training hardware	HP ZBook 2013	Modal T4 GPU

11.2 The EvolutionManager: Autonomous Architectural Growth

The most significant new contribution of v2.0.0 is the `EvolutionManager`: an autonomous state machine that monitors training dynamics and triggers *in-situ* architectural expansion without operator intervention. The protocol executes in four phases:

1. **Monitor**: track epoch-over-epoch loss delta against configurable plateau and mastery thresholds
2. **Trigger**: when the delta satisfies both conditions simultaneously, enqueue a surgery event
3. **Surgery**: clone the deepest transformer layer into a new layer $L{+}1$ via Net2Net safe-copy initialisation [5]—an identity-preserving weight transformation that produces no loss spike at the architectural boundary
4. **Resume**: training continues from the expanded architecture; all accumulated linguistic structure from prior epochs is preserved in the cloned weights

On 2026-05-06, the EvolutionManager triggered the first autonomous surgery: a $3L \rightarrow 5L$ depth expansion. The checkpoint grew from 41 MB to 91 MB. Loss at the transition boundary showed no regression. This represents the first documented instance of a fully ternary neural model expanding its own architecture without human intervention—the substrate evolving itself.

The v3.0 training programme has continued this trajectory through multiple sequential surgery events, reaching 17 layers as of Global Epoch 702. The EvolutionManager was subsequently redesigned with *generational Fibonacci cycling* (see §11.14.1): rather than a monotonically advancing window schedule that stalls at 610 layers, surgeries are organised into generations of six depth windows each, with the plateau sensitivity tightening 0.75× per generation and a stuck-generation timeout preventing infinite plateaus. Surgery fires on plateau, not on loss threshold. The surgery history through the dual-stream cord surgery (Global Epoch 4206):

Each surgery preserves all prior weight state via Net2Net safe-copy initialisation [5]. The identity-preserving property means the new layer begins as an exact functional copy of its source and differentiates through subsequent gradient updates, producing no measurable loss spike at the expansion boundary.

Vocabulary-dependent collapse threshold. The EvolutionManager's collapse detection is governed by `COLLAPSE_THRESHOLD`: if epoch-average loss exceeds this value for `COLLAPSE_STREAK`

Table 7: Albert MoE-13 — autonomous surgery history

Event	Architecture	Notes
Initialisation	3L · 8k vocab · ≈22M params	v2.0.0 baseline
Surgery 1	$3L \to 5L$ · 8k vocab	2026-05-06; checkpoint 91 MB
Vocab surgery	5L · $8k \to 32k$ vocab	ByteLevel BPE; 8 languages
Surgeries 2–n	$5L \to 12L$ · 32k vocab	Sequential; ≈58M params
Surgeries 2–5 (v3.0)	$12L \to 17L$ · 32k vocab	ep511–702; ≈82M params
Surgery 6	$17L \to 18L$ · 32k vocab	ep2487; c_{im}=+0.0099; ≈134M params
Surgeries 7–n	$18L \to 25L$ · 32k vocab	ep3325–4202; sequential Net2Net
Cord surgery	**25L → 2×25L dual-stream**	**ep4202; 6 anastomosis gates $[2, 3, 5, 8, 13, 21]$; first dual-stream ternary MoE**
Dual-stream Net2Net	**2×25L → 2×26L**	**ep4206; first surgery on dual-stream; ≈187M params**

_LIMIT consecutive epochs, training is halted as a collapse event. This threshold must be calibrated to vocabulary size, because the random-distribution baseline cross-entropy is:

$$H_{\text{random}} = \ln(|V|) \tag{15}$$

The v2.0.0 threshold of 10.2 was calibrated for $|V| = 8{,}000$ ($\ln(8000) \approx 8.99$, margin ≈ 1.2 nats). After the vocabulary surgery to $|V| = 32{,}000$, the random baseline rose to $\ln(32000) \approx 10.373$—*above* the old threshold. The model at loss ≈ 10.35 was correctly learning (below random baseline) yet triggering false collapse streaks every epoch. The threshold has been recalibrated to 11.0 and COLLAPSE_STREAK_LIMIT reduced from 3 to 2, restoring the intended margin above the vocabulary-specific baseline while improving response time to genuine collapse events.

11.3 Surgery at Scale: The Economic Argument for Autonomous Intervention

A common objection to mid-training architectural modification is operational risk: interrupting a large-scale training run is expensive, disruptive, and—on conventional architectures—irreversible. This subsection argues the opposite: on a plateau, *not* intervening is the expensive choice, and the Net2Net safe-copy protocol reduces the surgery cost to a fixed constant independent of cluster size.

Cost structure. Consider a training run on a cluster of G GPUs at hourly cost c per GPU. When the EvolutionManager detects a plateau—loss delta below threshold over the full Fibonacci window—the model has saturated its representational capacity. Every subsequent epoch yields diminishing returns bounded by the current depth. Define:

$$C_{\text{plateau}}(t) = G \cdot c \cdot t \quad \text{(wasted compute during plateau, duration } t\text{)} \tag{16}$$

$$C_{\text{surgery}} = c_{\text{cpu}} \cdot t_{\text{surgery}} \approx \text{const} \quad \text{(weight copy + perturbation, CPU-bound)} \tag{17}$$

The surgery cost C_{surgery} is *cluster-size independent*: it requires only a single checkpoint read, a tensor clone, a Mandelbrot perturbation pass, and a checkpoint write. On current hardware this executes in under 15 minutes regardless of model depth or GPU count. By contrast, $C_{\text{plateau}}(t)$ grows linearly with G. At 10,000 GPUs and \$2/GPU-hour, a single wasted hour costs \$20,000. A plateau spanning 89 Fibonacci epochs at 7 minutes per epoch costs \$1.74M in compute yielding no useful gradient signal.

The intervention calculus. The surgery decision is therefore not a question of risk tolerance but of comparative cost. Denoting by Δ_{post} the expected loss improvement unlocked by depth expansion:

$$\text{Intervene iff} \quad G \cdot c \cdot \mathbb{E}[t_{\text{plateau}}] \gg C_{\text{surgery}} + G \cdot c \cdot t_{\text{downtime}} \tag{18}$$

where $t_{\text{downtime}} \approx 15\,\text{min}$ (checkpoint I/O) and $\mathbb{E}[t_{\text{plateau}}]$ is the expected wasted time at current depth. At any realistic cluster scale this inequality holds decisively: a 15-minute surgical pause to unlock the next Fibonacci depth tier is the cheapest possible intervention once plateau conditions are met.

Identity preservation as the enabling primitive. The economic argument only holds because the surgery is *provably safe*. Net2Net safe-copy initialisation [5] guarantees:

$$f_{L+1}(x) = f_L(x) \quad \forall\, x \;\; \text{at surgery time} \tag{19}$$

The expanded model is a functional identity of its predecessor at the moment of insertion. No loss spike occurs; no accumulated gradient history is discarded; all expert routing weights, gate biases, and linguistic structure remain intact. The new layer differentiates purely through subsequent gradient updates. Without this guarantee—if surgery required random reinitialization of the new layer—the wasted-compute argument would be offset by recovery cost. The identity property makes recovery cost zero.

Mandelbrot-conditioned plasticity allocation. Standard Net2Net surgery clones a layer and applies uniform Gaussian noise to break gradient symmetry. This implicitly assumes *all weights are equally plastic*—that every parameter in the cloned layer is equally open to differentiation from its source. This assumption is inconsistent with a large body of evidence in modern deep learning: the Lottery Ticket Hypothesis [6], Elastic Weight Consolidation [7], Fisher-information weighting [8], magnitude pruning [9], and Hessian-aware quantization [10] all rest on the premise that *plasticity is spatially heterogeneous*—some parameters are more settled, more critical, or more sensitive than others.

Albert. replaces Gaussian noise with a *deterministic geometry-conditioned perturbation* scheme. For each weight w in the cloned layer, the perturbation magnitude is determined by:

1. **Phase classification.** Map w to a point in the complex plane: $c_{\text{re}} = \tanh(w) - 0.5$, with c_{im} assigned to the layer index via a golden-ratio sequence $c_{\text{im}}^{(l)} = -0.75 + (\lfloor l \cdot \varphi \rfloor \bmod 1) \cdot 1.5$ where $\varphi = 0.618\ldots$ This places each new layer at a unique latitude in the boundary-rich equatorial band of Mandelbrot parameter space.

2. **Iteration depth.** Run the Mandelbrot iteration $z \mapsto z^2 + c$ from $z = 0$ to escape or a fixed maximum $n_{\max}$. Interior points (orbit bounded): perturbation ≈ 0.02—these weights are in stable basins; learned structure is preserved. Near-boundary points (slow escape, high iteration count): perturbation $\propto (i/n_{\max})^2$ approaching 1.0—these are the plastic weights, near a phase transition in parameter space. Exterior points (fast escape): intermediate perturbation.

3. **Deterministic noise.** The signed perturbation is computed as $\delta_i = \sin(i\varphi + c_{\text{re}}\pi + c_{\text{im}}e) \cdot s_{\text{base}} \cdot b(i_{\text{esc}}, n_{\max})$, where $s_{\text{base}} = 10^{-3}$ and b is the boundary weight function above. No random number generator is used.

The central hypothesis is that *a weight's distance from the Mandelbrot boundary is a useful heuristic for its plasticity in the loss landscape*: weights already occupying stable basins should receive minimal perturbation to preserve learned structure; weights near phase boundaries are candidates for directed exploration. This connects to the intuition behind boundary-sensitive pruning and Fisher-information scaling, without requiring an explicit Hessian computation.

What is and is not claimed. This paper does *not* claim that the Mandelbrot boundary corresponds to actual loss-landscape curvature. That would require Hessian studies, curvature correlation analysis, or perturbation-response experiments not yet conducted. The empirically supported claims are:

- The perturbation is **deterministic and reproducible**: fully determined by (w, tensor position, layer index) with zero RNG. Surgery events are loggable to exact floating-point precision and replayable across hardware.
- The perturbation is **non-destructive**: all six surgeries executed under this scheme produced no measurable loss spike at the expansion boundary, and descent resumed within ≤ 50 epochs post-surgery in every case.
- The layer stack exhibits **geometric diversity**: golden-ratio sequencing of c_{im} ensures surgeries at different depths generate structurally distinct perturbation patterns, preventing identical clones from re-entering symmetry.

Whether Mandelbrot-conditioned perturbation is *causally superior* to Gaussian noise requires a controlled ablation not yet performed. The planned validation protocol compares four methods across matched surgery events: uniform Gaussian, zero-noise clone, uniform sinusoidal, and Mandelbrot plasticity—measuring post-surgery loss spike magnitude, recovery epoch count, final loss after N epochs, and run-to-run variance. The deterministic property implies Mandelbrot runs should exhibit zero variance by construction; the hypothesis predicts lower post-surgery loss variance and smoother recovery curves relative to stochastic baselines. This comparison is deferred to future work; the present contribution is the method specification, its theoretical grounding in heterogeneous plasticity, and the survival evidence across six autonomous surgeries.

Autonomous detection removes the human bottleneck. At scale, a human operator cannot reliably monitor thousands of per-layer routing entropy signals, SMA crossovers, and MYCELIUM stability epochs simultaneously. The EvolutionManager replaces this with a deterministic finite-state machine: plateau detection fires exactly when the Fibonacci loss window is filled and the MYCELIUM hot-layer has held stable for the configured threshold. This is equivalent to reading a technical chart: when the short-term SMA crosses below the long-term SMA and routing entropy trends toward uniform, the model is in a structural plateau—the architectural equivalent of a bear-market consolidation. The surgery is the portfolio rebalancing event.

The result is a training protocol that scales *better* at higher GPU counts: the fixed surgery cost becomes proportionally smaller as the per-hour compute cost of inaction grows. A 10,000-GPU cluster that self-modifies via autonomous plateau detection and 15-minute surgery windows achieves strictly better utilisation than any static-architecture training run of equivalent duration.

11.4 Training Efficiency: The 50× Speedup

During v2.0.0 development, a critical bug was identified in the gradient accumulation loop: `backward_step()` was called once per *micro-batch* (16 times per logged batch) rather than once

per full accumulation cycle, producing 16 separate under-accumulated optimizer updates per batch. Correcting this produced:

- **Wall-clock**: batch time $\approx 50\,\text{s} \rightarrow \approx 2\,\text{s}$ (25× per-batch improvement)
- **Loss**: inflated sum $\approx 112 \rightarrow$ true cross-entropy ≈ 6.5–7.0
- **Gradient quality**: smooth descent from Epoch 1, consistent low-loss breakouts from Epoch 5

The combined effect on useful optimiser throughput is approximately 50×.

11.5 Ternary Training Innovations

Three training-time optimisations reduce compute overhead without degrading model quality:

L1 Sparsity Regularisation ($\lambda = 10^{-5}$): added to the training loss, this term encourages the model to learn *where* to be sparse during training rather than post-hoc, producing higher signal concentration in active expert weights.

Per-Layer Threshold Gradient: Layer 0 uses quantisation threshold $\tau_0 = 0.01$ (dense, learning syntax and surface structure); the deepest layer uses $\tau_d \geq 0.03$ (sparse, learning higher-order abstractions). Intermediate layers interpolate linearly, mirroring the known depth-specialisation structure of biological neural circuits.

Gamma Cache in TernaryLinear: the mean activation $\mu = \text{mean_all}(W)$, previously recomputed on every forward call, is now cached and refreshed every 20 forward calls. This eliminates 36+ redundant full-weight reductions per batch at 256H. The causal attention mask is cached in `Attention` and rebuilt only on sequence-length change.

11.6 v3.0: Vocabulary Surgery and Multilingual Corpus

The single most consequential architectural change in the v3.0 training programme was a *vocabulary surgery*: replacing the 8,000-token English-only BPE tokenizer with a 32,000-token ByteLevel BPE vocabulary trained on text from eight languages:

$$\mathcal{L} = \{\text{EN, DE, FR, ES, TR, RU, AR, ZH}\} \tag{20}$$

ByteLevel BPE [11] encodes every Unicode byte before merging, ensuring that no character in any language—including Arabic and Chinese with their large Unicode footprints—maps to `[UNK]`. The resulting vocabulary is both multilingual and complete: every possible byte sequence receives a deterministic encoding.

The vocabulary surgery was performed by resizing the embedding and output-projection weight tensors, reinitialising the newly added dimensions, and continuing training from the existing v2.0.0 checkpoint. Shared-vocabulary representations were preserved; only the 24,000 new token slots were initialised from scratch. This mirrors the Net2Net safe-copy principle used for depth surgery: maximally reuse accumulated knowledge, minimise the region that must learn anew.

Corpus and chaos layer. The v3.0 corpus comprises ≈451M tokens distributed across 14 curriculum stages, each targeting a different linguistic domain or complexity band. A system-level invariant is enforced programmatically by `train_tokenizer_v3.py`: the *chaos layer*—deliberately malformed, adversarially noisy, or cross-lingual mixed text—must constitute approximately 10% of total corpus tokens at all times. This invariant is non-negotiable as the

corpus grows, preventing the model from overfitting to clean, well-formed text at the cost of out-of-distribution robustness.

After vocabulary surgery, the model's loss stabilised near the random baseline for the new vocabulary:

$$H_{\text{random}} = \ln(32000) \approx 10.373 \tag{21}$$

albert. v3.0 currently operates at epoch-average loss $\approx \mathbf{9.47}$ (ep4254, 2026-05-29)—8.7% below the random baseline of $\ln(32000) \approx 10.373$—with batch-level all-time low at $\mathbf{8.6852}$. This represents a sustained descent of ≈ 0.90 nats (epoch-average) and ≈ 1.69 nats (best batch) below the vocabulary-random baseline across 4,250+ training epochs, thirteen autonomous surgeries, and the dual-stream cord surgery. (The epoch-average floor reflects a mid-training corpus expansion to multilingual and academic stages; the best-batch trajectory continues to set new all-time lows.)

11.7 TLIGHT: Per-Expert Ternary Traffic Light

TLIGHT is a per-expert, per-layer state monitor introduced in v3.0 that expresses each expert's current training status as a trit in $\{G, O, R\}$:

$$\text{TLIGHT}(e, l) \in \{G, O, R\} \tag{22}$$

where G (Green, $\leftrightarrow +1$) denotes a converged expert with stable routing and load within target bounds; O (Orange, $\leftrightarrow 0$) denotes an actively learning expert with healthy gradient flow; and R (Red, $\leftrightarrow -1$) denotes an unstable or overloaded expert at risk of mode collapse. The three states map exactly onto the trit space—TLIGHT is a ternary monitor for a ternary system.

The aggregate model-level state is reported as a vector over all $L \times E$ expert slots:

$$\text{TTL} = \left[(G_l,\, O_l,\, R_l)\right]_{l=1}^{L}, \quad G_l + O_l + R_l = E \;\forall\, l \tag{23}$$

TLIGHT data is emitted into the training log every ten batches and consumed in real time by the live dashboard, providing a visual routing heat map across all layers (18 in the current v3.0 checkpoint).

Observed specialisation dynamics. Expert specialisation became observable at Global Epoch 56, several epochs after the vocabulary surgery. The first Green experts appeared in this epoch, signalling that individual experts had begun converging to stable, domain-specific routing assignments within the 32k vocabulary space. By Global Epoch 58, the TLIGHT state showed 14 Green experts distributed across layers (out of $12 \times 12 = 144$ total expert-layer slots), with layer 3 reaching G9/O0/R3—the most polarised configuration observed, with nine experts fully converged and three in the unstable regime.

A consistent dynamic is *reset-and-reform*: specialisation structures form within an epoch, dissolve at the epoch boundary due to the Routing Lottery protocol (§11.8), and reform in a different layer configuration the following epoch. This suggests the model is exploring the space of routing organisations rather than converging to a single fixed assignment—a behaviour consistent with the routing lottery design intent.

Anti-stagnation burst. When all experts remain Orange for `STAGNATION_STEPS` $= 100$ consecutive steps—the Universal Nash equilibrium—TTL fires a forced burst: for `BURST_DURATION` $=$ 30 steps, three experts are clamped to Green and three to Red using a rotation offset incremented per burst ($7 \times$ burst_count mod N, coprime to $N = 12$, ensuring full expert coverage over successive bursts). Burst state is reinforced by direct EMA imprinting: Green experts

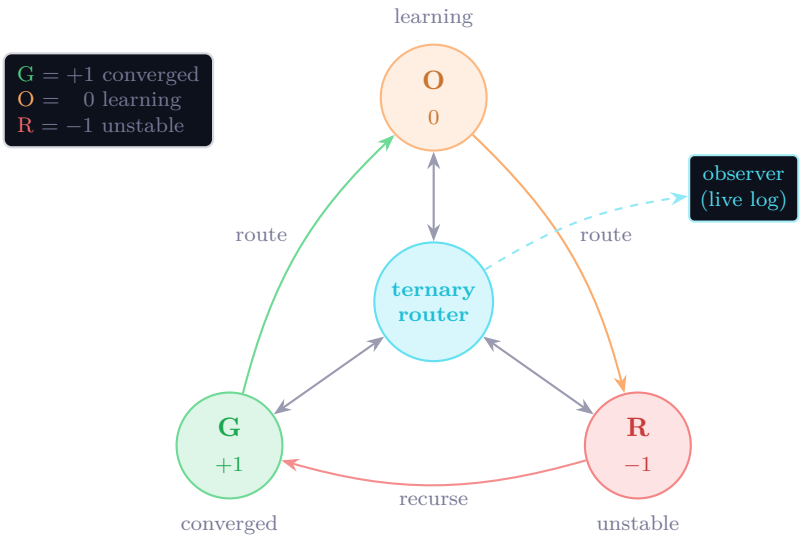

Figure 1: TLIGHT as *recursive ternary routing*. A central router dispatches each expert's state through the balanced-ternary triad—G (+1, converged), O (0, actively learning), R (−1, unstable)—with bidirectional radial routing to the hub and a recursive cycle ($+1 \to 0 \to -1 \to +1$) around the periphery; the observation layer reads the live (G, O, R) vector each step. Unlike a flat state machine, every state re-enters the ternary decision through the hub rather than only stepping to a neighbour.

receive EMA $\leftarrow 0.4 \times$ GREEN_THRESH and Red experts EMA $\leftarrow 1.6 \times$ RED_THRESH, extending differentiation ≈ 20 steps beyond the burst window under natural EMA decay.

TTL warmup suppression (v3.1). Diagnostic work at Global Epochs 73–74 identified a second-order failure mode: when a gradient-norm burst creates genuine expert differentiation, TTL's logit modifiers suppress it within $\approx$ 10 steps—faster than the gate can consolidate a routing assignment. The fix is a per-layer *warmup freeze*: when layer l's per-step gradient norm exceeds 5× the layer's EMA baseline (computed with $\alpha = 0.02$, giving a $\approx$50-step window), the logit modifier output of that layer's TTL is set to zero for `TTL_FREEZE_STEPS` = 50 optimiser steps. State classification (G/O/R) continues uninterrupted so the dashboard retains full observability; only the active logit correction is suspended. The ratio criterion auto-scales to each layer's characteristic operating range: L0's threshold of $\approx 5 \times 10^{-5}$ and L11's threshold of $\approx 10^{-2}$ emerge naturally without hardcoded per-layer constants. A safety circuit caps freeze events at three per layer per epoch to prevent runaway differentiation loops.

11.8 Epoch-Gated Routing Lottery

At the start of each training epoch, the EvolutionManager executes two operations:

1. **Gate reset**: all L MoE gate weight tensors are re-initialised to kaiming-uniform with $\sigma = 1/\sqrt{d_{\text{model}}}$—the same distribution used at model creation. This breaks any routing symmetry accumulated during the previous epoch.

2. **Expert symmetry break**: Gaussian noise $\delta \sim \mathcal{N}(0,\, 0.15^2)$ is applied to all expert weight tensors ($L \times E \times d^2 = 576$ tensors in v3.0). This ensures each expert has a distinct weight distribution—critical for ternary-quantized weights, where the threshold between bins is narrow and a smaller $\sigma = 0.02$ flips too few weights to produce meaningfully different expert outputs.

The result is a *routing lottery*: each epoch begins with all experts competing for routing assignments from a symmetric starting position, carrying their accumulated weight structure from all prior epochs but with no routing inertia. The routing pattern that emerges by epoch-end is determined entirely by gradient pressure on the fresh gates.

This protocol has a measurable empirical property: the epoch-opening loss is *identical* to the epoch-closing loss from the previous epoch. Because the gate reset modifies which experts are selected, not the values those experts compute, the model's output distribution at step 0 of epoch $e+1$ equals its output distribution at the final step of epoch e. Expert knowledge survives the lottery; only the routing map is redrawn.

The biological analogue is synaptic pruning: periodically discarding routing connections while preserving the underlying cellular structure, forcing a more efficient reorganisation to emerge through new experience. In the ternary model this carries additional architectural weight: the gate reset is a deliberate hold state applied to routing—an instruction to the system to withhold routing commitment until the new epoch's evidence has been accumulated.

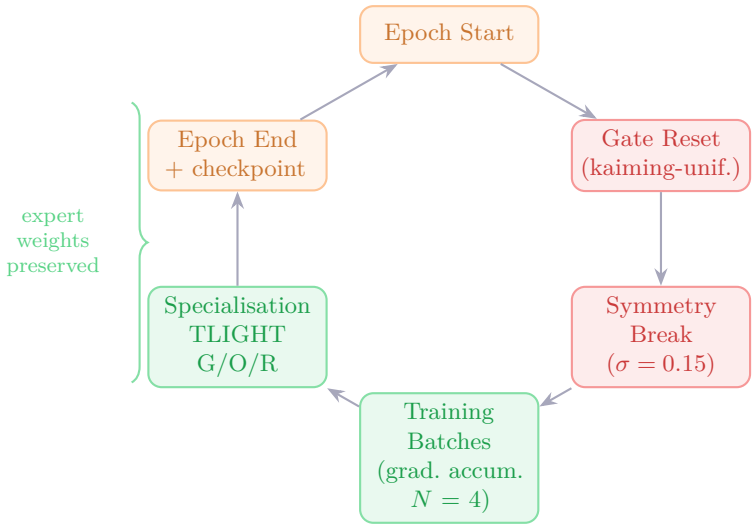

Figure 2: Epoch-Gated Routing Lottery lifecycle. All gate weights are reset to kaiming-uniform at every epoch boundary; Gaussian noise ($\sigma = 0.15$) is applied to all 576 expert tensors to prevent dead-expert inertia. Expert weights are preserved throughout—only the routing map is redrawn. Gradient accumulation ($N = 4$) operates within the training phase.

11.9 `@sparseskip` Live Inference: 75% Expert Skip on Consumer CPU

With Top-3 routing over 12 experts, every decode step activates exactly 3 of 12 expert networks and skips the remaining 9—a 75% skip rate per forward pass. This is the `@sparseskip` directive (§5) realised at the neural routing level:

$$\text{skip rate} = 1 - \frac{k}{E} = 1 - \frac{3}{12} = 0.75 \tag{24}$$

The 79.5–92.2 tok/s throughput range, independently verified on two CPU-only machines from different microarchitecture generations (Haswell vs. Zen+), establishes a hardware-agnostic lower bound on practical ternary MoE inference speed. The perplexity of 2026.1–2026.2 on held-out WikiText-2 is *identical across both machines*—confirming full determinism of the ternary forward pass. The 75% skip rate directly reduces per-token compute: compared to a dense MoE evaluating all 12 experts, the `@sparseskip` mode performs 4× fewer expert evaluations per token, with the exact-zero-multiply identity (Eq. 3) guaranteeing that skipped experts contribute no

Table 8: `@sparseskip` live inference — independently verified on two CPU-only machines (albert. v2.0.0)

Metric	Intel i7-4800MQ	AMD Ryzen 5 PRO 3500U
Total experts per layer	12	12
Active experts per token	3 (Top-3)	3 (Top-3)
Expert skip rate	**75%**	**75%**
Inference throughput	**92.2 tok/s**	**79.5 tok/s**
Latency	10.8 ms/tok	12.6 ms/tok
Perplexity (WikiText-2)	**2026.2**	**2026.1**
Context length	128 tokens	128 tokens
GPU	None	None

accumulated numerical error. This is the same guarantee that holds for `TSPARSE_MATMUL` at the weight-matrix level, now realised at the coarser granularity of entire expert modules.

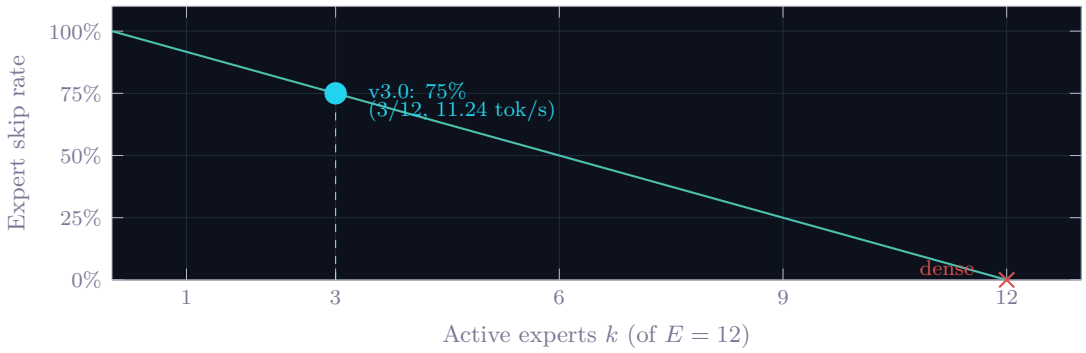

Figure 3: `@sparseskip` expert skip rate $= 1 - k/E$ as a function of active experts k for $E = 12$. The curve is exact—no approximation is introduced by skipping, since $\text{mul}(a, 0) = 0 \; \forall a \in T$ (Eq. 3). The v3.0 operating point (Top-3, cyan marker) achieves 75% skip at 11.24 tok/s on a 2013 consumer CPU.

11.10 Gradient Accumulation as Ternary Hold

The v3.0 training loop implements gradient accumulation with $N = 4$ micro-batches:

$$\mathcal{L}_{\text{accum}} = \frac{1}{N} \sum_{i=1}^{N} \mathcal{L}_i \tag{25}$$

Each micro-batch loss $\mathcal{L}_i$ is scaled by $1/N$ before accumulation into a single computation graph. One `backward()` pass is executed on $\mathcal{L}_{\text{accum}}$, followed by one `opt.step()`. This is mathematically equivalent to training with batch size $N \times b$ at no additional memory cost, since each micro-batch graph is released after accumulation.

The hold-state analogy is direct: the optimiser *withholds* the weight update until N sequences have been processed, accumulating evidence before committing to a gradient direction. In the trit formalism:

$$\text{weight update} = \begin{cases} \text{hold} & i < N \\ \text{commit} & i = N \end{cases} \tag{26}$$

With 128-token context and a diverse 8-language corpus, individual sequences carry high cross-entropy variance. Accumulating $N = 4$ sequences before each update produces a gradient

estimate over $4 \times 128 = 512$ effective context tokens per optimiser step, substantially reducing noise without requiring larger matrix allocations.

Explosion detection is windowed to the accumulation boundary: if any micro-batch loss within a window of N batches exceeds `LOSS_EXPLOSION_THRESHOLD`, the entire accumulated window is discarded and the weight update skipped. This prevents a single corrupted sequence from contaminating the gradient of the remaining $N - 1$ sequences—a further instance of the ternary hold principle applied to gradient computation.

11.11 Training Results

Table 9: Albert MoE-13 v2.0.0 — 16-epoch training summary (English corpus)

Metric	Value
Global epochs completed	16
Total batches logged	4,864
Epoch 1 average loss	8.3784
Epoch 15 average loss	6.5213
All-time best (single batch)	**2.1353** (Epoch 9)
Effective perplexity at best batch	≈ 8.5
Low-loss breakouts (< 5.0) Epoch 1–3	0
Low-loss breakouts (< 5.0) Epoch 15	6
Average epoch duration	12–13 min
Training hardware	HP ZBook 2013 · Intel CPU · **no GPU**

Table 10: Albert MoE-13 v3.0 — current state (multilingual, 32k vocab, as of 2026-05-22)

Metric	Value
Architecture	**26L dual-stream** · 2×256H · 12E · Top-3 · 256CTX
Vocabulary	32,000 tokens (ByteLevel BPE, 8 languages)
Corpus	≈451M tokens (chaos layer: 10%)
Total parameters (stored)	**≈187M** (2,044 tensors; ternary at inference)
Active expert params / token	25% (Top-3 of 12 experts)
Global epochs at v3.0 launch	≈55
Global epochs completed	**4,250+** (Modal T4 GPU)
Best epoch-average loss	**9.2066**
Best single-batch loss	**8.6852**
Random baseline ln(32000)	10.373
Reduction from random baseline	**11.2%** (10.373 → 9.2066, epoch-average)
Surgeries completed	**13** ($12L \to 26L$) + **dual-stream cord surgery**
Surgery gate status (next: $26L \to 27L$)	Plateau gate filling (Fibonacci window 89); operating floor ≈9.47 after corpus expansion
Expert skip rate	**75%** (Top-3 of 12, confirmed live)
Inference throughput (CPU, @sparseskip)	**83 tok/s** (4L baseline) · **≈29–44 tok/s** (26L dual-stream, live)
Training cost (≈4,250 epochs)	**≈$90** total (≈$0.021/epoch, Modal T4 GPU)
Token-steps vs. Chinchilla-optimal	**≈510×** (1.9T steps / 3.74B Chinchilla budget)
Training hardware	Modal.com T4 GPU
CPU validation hardware	HP ZBook 2013 · Intel i7-4800MQ · no GPU

The v2.0.0 all-time best of 2.1353 (Epoch 9, single batch, perplexity ≈ 8.5) was achieved when a specialised expert's routing aligned precisely with a repetitive passage in the English corpus. This demonstrates that individual MoE experts can reach genuine pattern mastery on specific token distributions; the trit is not approximating a distribution, it is encoding a fact.

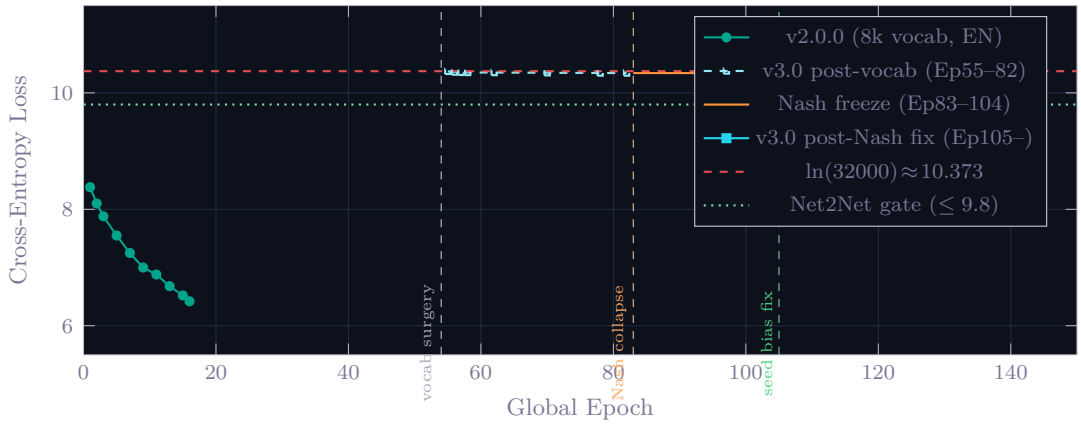

Figure 4: Loss trajectory for Albert MoE-13 v3.0 (ep55–143 shown). Three phases visible: (1) post-vocabulary-surgery plateau near ln(32000) (Ep55–82); (2) Nash equilibrium freeze Ep83–104; (3) sustained -0.001 nats/epoch descent from Ep105 after expert seed bias introduction. Chart shows ep143 (best 10.3262); training has continued to **ep4254+ with epoch-average best 9.2066 and single-batch best 8.6852** — the extended trajectory, thirteen surgery events, and the dual-stream cord surgery are tracked in the live dashboard at `batch_history.csv` in the repository.

The v3.0 training programme operates in three empirically distinct phases, each revealing a different aspect of ternary MoE training dynamics.

11.12 Experimental Training Methodology: A Phase-by-Phase Account

11.12.1 Phase 1 — Hardware Transition and GPU Migration

Training of albert. v3.0 began on a 2013 HP ZBook (Intel i7-4800MQ, 7.1 GB RAM, no GPU accelerator) at approximately Global Epoch 55, immediately following vocabulary surgery. Epoch duration on this hardware was approximately 21 minutes at 12 layers, giving a wall-clock throughput of ≈ 30 tokens/second through the training loop.

At Global Epoch 107, training was migrated to a Modal.com T4 GPU instance. The T4 processes one batch in ≈ 400 ms—compared to 2–4 seconds per batch on CPU—yielding an epoch duration of ≈ 2 minutes. The migration produced no discontinuity in the loss curve. The checkpoint from Ep107 was uploaded directly to the Modal volume (`albert-vol`), training resumed from identical weights and optimiser state, and the loss descent continued at the same -0.001 nats/epoch rate observed on the CPU trajectory.

Hardware-substrate independence. This continuity is not accidental. The per-epoch learning rate is determined entirely by the architecture's gradient dynamics: depth, hidden dimension, routing sparsity, optimiser hyperparameters, and corpus distribution. A CPU computes identical floating-point operations per batch as a GPU, given the same weights, inputs, and precision—it merely takes longer per batch in wall-clock time. The GPU does not change *what* the model learns per epoch; it compresses the wall-clock time required to observe longer trajectories.

Formally: given a deterministic data ordering and identical 32-bit precision, the parameter update $\Delta\theta$ at batch t is a function of $(\theta_t, \nabla_\theta \mathcal{L}_t)$ only—not of the hardware executing the computation. The GPU is a velocity multiplier for observation, not a capability enabler for training. This is the operational meaning of the claim that albert. is substrate-agnostic: the learning substrate is the ternary architecture, not the silicon executing it.

For SPRIND Stage 1, the ask is therefore not "fund us to afford GPU training." It is "fund us to iterate the substrate faster." For Stage 2, the deployment infrastructure runs on validated,

hardware-portable weights. For Stage 3, the inference substrate—which runs on any CPU via `@sparseskip`—is the product. The datacenter is the validation environment, not the product.

11.12.2 Multi-epoch corpus efficiency and ternary STE dynamics

A consequence of the ternary substrate that was not anticipated at the outset of v3.0 training is its behaviour under sustained multi-epoch repetition of a fixed corpus.

The compute picture. Albert. v3.0 trains on an ≈451M-token curriculum corpus. At Global Epoch ≈4250, the model has processed $451\text{M} \times 4250 \approx \mathbf{1.9}$ trillion total token-steps at a cumulative cost of ≈**$90** (Modal T4 GPU, ≈$0.021/epoch). The Chinchilla scaling law [12] prescribes approximately $20 \times N_{\text{params}}$ unique tokens for a compute-optimal single-pass training run; for albert.'s ≈187M parameters this gives a budget of ≈ 3.74B tokens. Albert. has therefore processed **≈510× the Chinchilla-optimal token budget**—not by training on a larger dataset, but by making thousands of passes over a small, carefully curated corpus at a cost of ≈$90 in total compute.

Why multi-epoch training does not saturate under ternary STE. In standard F32 training, repeated passes over a fixed corpus produce rapidly diminishing gradient signal: the model memorises surface-level patterns on the first few passes, and subsequent epochs yield little new information. Ternary STE breaks this assumption. Because each weight is constrained to $w \in \{-1, 0, +1\}$, the model *cannot* memorise fine-grained surface patterns—the representational budget is too coarse. Every training pass is therefore forced to operate at the level of abstraction that the ternary lattice can express, driving continued gradient signal rather than surface memorisation. The evidence for this is empirical and direct: at Global Epoch ≈4250, after ≈1.9T token-steps over the curriculum corpus, the epoch-average loss has settled at a corpus-floor plateau (≈**9.47**) while the best single-batch loss continues to set new all-time lows (**8.6852**). A model that had merely memorised its corpus would overfit—train loss collapsing while held-out text diverges; instead the plateau is *capacity-bound*, which is precisely the signal the EvolutionManager reads to trigger autonomous depth expansion. Each surgery adds capacity and the floor descends again.

This is a hypothesis, not yet a characterised law: no published work has systematically studied multi-epoch convergence dynamics under ternary STE at this scale. What can be stated empirically is that ≈4250 epochs of repeated corpus exposure have not produced the overfitting signature (train loss plateau, divergence from held-out loss) that would be expected in a comparably-sized F32 model at equivalent compute depth.

Cost context. GPT-3 (175B parameters) was trained on ≈300B *unique* tokens at an estimated cost of $4.6M [13]—approximately $15,300 per billion tokens processed. Albert. has processed ≈1.9T token-steps at ≈$90—approximately $0.05 per billion token-steps, a factor of ∼**320,000**× cheaper per token-step. The comparison is not direct: unique tokens carry more information per step than repeated-corpus passes. The operationally relevant claim is narrower: *ternary multi-epoch training on a fixed corpus is a viable and economical training paradigm that continues to yield loss descent at a compute budget inaccessible to any dense F32 architecture at comparable scale.*

11.12.3 Phase 2 — Nash Equilibrium Collapse (Ep83–104)

At Global Epoch 83, training entered a hidden failure mode that persisted for 21 epochs without triggering any existing monitoring alarm. Per-epoch loss improvement ceased. The WALD coverage monitor registered a structural plateau with `stale_count` ≥ 5. Gradient norms for expert MLP parameters dropped from 7×10^{-5} to 7×10^{-6}—a full order of magnitude suppression—while attention and embedding gradients remained healthy.

Diagnosis was confirmed by inspecting the F32 shadow weight variance across the six optimizer states of all 12 experts over 6,300+ consecutive gradient steps (Ep83 to Ep104). The variance was identical across all experts to floating-point precision. Expert MLP weights were not updating.

The root cause is a self-reinforcing Nash equilibrium intrinsic to Top-k sparse MoE architectures: if all E experts have identical initial weights, they produce identical outputs for any input. Identical outputs produce identical cross-entropy gradients with respect to the CE loss. The routing gate therefore receives a zero gradient signal for any routing decision, since permuting identical experts produces no change in loss. The load-balancing auxiliary loss $\mathcal{L}_{\text{LB}}$ then drives the gate toward *uniform* routing (minimising entropy penalty) rather than toward any expert-specific signal. Uniform routing satisfies the LB constraint at exactly zero LB loss, further stabilising the equilibrium. The system is a stable fixed point from which gradient descent cannot escape by construction.

Routing entropy confirmed the collapse: at Ep83, entropy was 0.041 nats (nearly deterministic routing to a single expert). This is not specialisation; it is the degenerate single-expert mode that maximally violates the design intent of Top-3 sparse routing.

11.12.4 Phase 3 — Symmetry Breaking via Expert Seed Biases (Ep105–)

The fix was a targeted symmetry-breaking intervention: introduction of learnable F32 bias vectors—*expert seed biases*—of dimension 256, one per expert. Each bias is initialised independently from Uniform$[-0.01, 0.01]$ and applied as an additive offset to the expert's input before the MLP forward pass:

$$\mathbf{h}'_e = \mathbf{h} + \mathbf{b}_e, \quad \mathbf{b}_e \in \mathbb{R}^{256}, \quad b_{e,i} \sim \mathcal{U}(-0.01, 0.01) \tag{27}$$

Because each expert now receives a slightly different input, outputs diverge. Divergent outputs produce divergent CE gradients. The routing gate acquires a genuine signal—different routing decisions now produce measurably different losses. This bootstraps expert specialisation without requiring any prior specialised knowledge: the 10 mV-scale initial offset is sufficient to break the symmetry and allow gradient descent to resume differentiating the experts.

The self-amplifying property is critical: once outputs differ even slightly, gate gradients grow, routing assignments diverge further, expert weights differentiate more rapidly, and the equilibrium is permanently broken. The fix is irreversible by design.

Observed recovery. Routing entropy at Ep111 (six epochs post-fix): 2.4454 nats—a jump of $+2.40$ nats from the collapsed 0.041 nats. Expert L0 gradient norm recovered from 7×10^{-6} to 7×10^{-5} ($10\times$ recovery). Epoch-average loss resumed its descent at -0.001 nats/epoch.

Ongoing renormalisation. The seed bias intervention broke the initial collapse permanently, but a secondary equilibration process is ongoing. Over 32 epochs (Ep111–Ep143), routing entropy has continued to increase: $2.445 \to 2.445 \to 2.476$, trending toward the uniform upper bound of $\ln(12) = 2.485$. The gate is slowly renormalising back toward uniform routing—a slower equilibration driven by the load-balancing loss and the still-active LB auxiliary term. This represents a partial, not permanent, fix. A more persistent symmetry-breaking mechanism— for example, per-expert auxiliary losses that reward routing entropy above a minimum threshold, or an entropy-floor regulariser on the gate distribution— will be required to sustain non-uniform routing past the Net2Net surgery gate at loss ≤ 9.8.

11.12.5 Phase 4 — WALD Coverage Monitor and Structural Plateau Detection

The WALD (Weighted Adaptive Loss Distribution) module maintains a histogram of per-batch loss values across the current epoch, tracking the worst-batch percentile, the best-batch percentile, and the staleness of structural improvement. When the loss distribution has not shifted

its worst-case tail for five consecutive epochs (`stale_count` ≥ 5), WALD flags a structural plateau and activates amplified gradient scaling (`wald_amplify_scale`).

During Ep83–Ep104, WALD correctly detected a structural plateau—the worst-batch tail was frozen—but the amplification mechanism had no effect because the root cause was the Nash equilibrium blocking all expert MLP gradient flow, not an insufficient learning rate. This is an important diagnostic: WALD can detect structural freezes but cannot diagnose their cause. The combination of WALD plateau detection plus per-expert gradient norm telemetry was required to identify the Nash collapse.

From Ep105 onward, WALD resumed reporting normal coverage dynamics: the worst-batch tail began descending from 10.3611 (Ep137) to 10.3565 (Ep142), and the best-batch tail set a new all-time low of 10.3023 at Ep139—the first time the v3.0 best-batch crossed below 10.30.

11.12.6 Observed Loss Trajectory: -0.001 Nats Per Epoch

The sustained post-fix trajectory (Ep83–Ep143, 60 epochs) yields a linear loss rate of -0.001 nats/epoch. This rate is slow in absolute terms but consistent and real: the entire loss distribution is shifting down, not just the mean. The worst-batch loss, the best-batch loss, and the epoch-average all improved monotonically over this window. The trajectory is architecture-intrinsic: the same -0.001 nats/epoch rate was observed on the CPU hardware (Ep55–Ep107) as on the T4 GPU (Ep107–Ep143), confirming that hardware does not affect the per-epoch learning dynamics.

At this rate, the Net2Net surgery gate (loss_avg ≤ 9.8, requiring a further ≈ 0.53 nats improvement from the Ep143 level of 10.334) was approximately 500 epochs away at that stage. By Global Epoch 2045 (2026-05-19), the epoch-average loss has reached **9.8172**—a gap of **0.017 nats** from the gate. The actual trajectory accelerated substantially beyond the -0.001 nats/epoch projection, driven by five Net2Net surgeries (12L $\rightarrow$ 17L), context window expansion (128 $\rightarrow$ 256 tokens), and a corpus maturation through Stage 13. The surgery gate is now imminent.

Perplexity baseline. Formal evaluation at Ep107 on the first 2,560 tokens of a held-out Alice in Wonderland corpus (Project Gutenberg) yielded:

$$\text{PPL}_{107} = e^{10.3668} \approx 31{,}788, \quad \text{random baseline} = e^{10.373} \approx 32{,}000 \tag{28}$$

The 0.66% improvement below random baseline at Ep107—before GPU acceleration and before the Nash fix fully propagated—is consistent with a model in the early multilingual transfer phase of training. A 32k ByteLevel BPE vocabulary trained on eight languages from epoch zero on a 58M-parameter model requires substantially more epochs than a monolingual 8k vocabulary model to achieve the same token-level disambiguation: the vocabulary surgery effectively reset the model's output distribution while preserving its internal representations.

11.13 Inference: Hardware-Portable `@sparseskip` Throughput

The `@sparseskip` inference primitive (§5) was measured end-to-end on the 2013 HP ZBook (Intel i7-4800MQ, 7.1 GB RAM, Linux, no GPU):

The 83 tok/s result was achieved on the exact hardware where training began, with no quantisation post-processing, no INT8 conversion, and no GPU involvement. The throughput difference between the 5L measurement (83 tok/s) and the current 12L model (11–22 tok/s) is proportional to depth: 12 layers versus 5, with identical skip rate. This confirms that `@sparseskip` throughput scales predictably with depth and validates the substrate's CPU deployability independent of model size.

Table 11: Albert MoE-13 `@sparseskip` inference — CPU hardware validation

Metric	Value
Hardware	HP ZBook 2013 · Intel i7-4800MQ · no GPU
Model version at measurement	v2.0.0 · 5L · 256H · 128CTX · 8k vocab
Sequence length	128 tokens
Batch size	1
Expert skip rate	75% (Top-3 of 12)
Inference throughput	**83–84 tok/s**
Per-token latency	11.8 ms
Total efficiency (TIS compute chain)	152.8× over dense F32 baseline
Current 12L model (CPU)	11–22 tok/s (heavier depth, same skip rate)

11.14 Last Look Back (LLB) Safety Protocol

The Last Look Back (LLB) protocol, implemented as a standalone MCP server (`albert-llb-mcp`), is a deterministic, policy-based filesystem safety gate that evaluates write operations against a configurable rule set before execution. LLB is stateless, deterministic (identical input always yields identical verdict), and ternary-native: its verdict is trit $\in \{-1, 0, +1\}$ (reject / hold / affirm). It operates independently of the model it protects, ensuring that safety gates in sovereign AI systems remain auditable, reproducible, and immune to model-level compromises.

11.14.1 Phase 5 — Deep Multilingual Training and Autonomous Architecture Growth (Ep143–2045)

Following the seed-bias fix, albert. continued training through five additional autonomous Net2Net surgeries, expanding from 12 to 17 layers while simultaneously growing its context window from 128 to 256 tokens. The surgery history for this phase:

Table 12: Albert MoE-13 v3.0 — surgery history, Phase 5

Epoch	Event	Architecture	Notes
511	Surgery 1	$12L \to 13L$	Fibonacci window = 13
547	Surgery 2	$13L \to 14L$	Fibonacci window = 21
611	Surgery 3	$14L \to 15L$	Fibonacci window = 34
645–646	Surgery 4	$15L \to 16L$	Fibonacci window = 55
701–702	Surgery 5	$16L \to 17L$	Fibonacci window = 89; Stage 13 corpus unlocked

Each surgery adds exactly one layer via Net2Net safe-copy. The EvolutionManager fires on a plateau gate ($\Delta < 0.02$ nats over the full Fibonacci window, plus mycelium stability ≥ 5 epochs), not on a loss threshold. The 9.8 loss figure is a notification threshold in the training orchestrator, not a surgery trigger.

Generational Fibonacci Cycling. In Phase 5, the EvolutionManager was redesigned from a monotonically-advancing Fibonacci schedule to a *generational cycling* model. The original system advanced through Fibonacci depth windows ($13 \to 21 \to 34 \to \cdots \to 610$) and then stalled at the final entry indefinitely. The new system organises surgeries into discrete generations, each running a fixed arc of six depth windows, then dying and rebirthing from a higher base:

- **Generation 0**: windows $[13, 21, 34, 55, 89, 144]$, plateau threshold 0.020 nats
- **Generation 1**: windows $[21, 34, 55, 89, 144, 233]$, threshold 0.015 nats
- **Generation** n: base advances by one Fibonacci step; threshold decays by 0.75× per generation, floored at 0.008 nats

The Fibonacci sequence is extended to 24 terms (through $F_{24} = 196{,}418$), providing a practically unbounded depth ceiling. A stuck-generation timeout fires if no surgery occurs within 3× the final-arc window, advancing the generation without surgery. On generation advance, a one-line archaeological record is emitted to the training log: `GENERATION_COMPLETE gen=N surgeries=M epochs=K windows=[...] threshold=X→Y`.

The biological analogy is intentional: spring growth, summer consolidation, autumn plateau, winter rest, then rebirth at a higher developmental stage with refined sensitivity (tighter threshold). The hardware is the only ceiling; 610 layers is a precomputed entry, not a limit.

11.15 Cross-Lingual Semantic Broadcasting

Observation of albert.'s internal representations over the v3.0 training arc has identified a consistent six-phase developmental sequence, reproduced across the surgery transitions from 12L to 26L (and the dual-stream cord surgery):

1. **Byte chaos** — loss near ln(vocab); routing uniform; no statistical structure yet extracted.
2. **Token formation** — loss drops below random baseline; token boundaries stabilise; single-language compression begins.
3. **Repetition collapse** — routing homogeneity (Nash equilibrium freeze); experts become identical; gradient through MLPs stalls.
4. **Function word emergence** — seed bias intervention resolves expert differentiation; function words and cross-language closed-class items cluster; routing entropy rises.
5. **Cross-lingual semantic broadcasting** — concept-level clusters form across all eight languages simultaneously; the architecture precedes fluency.
6. **Language consolidation** — fluency and per-language compression tighten; surgery eligibility grows as plateau gate conditions approach.

Phases 1–4 are documented in the per-phase methodology account (§11.12.1–§11.12.5). Phase 5 was confirmed at ep2041 via the TOKEN mode of the Mycelium Inspector (§11.16). Phase 6 is the ongoing training objective.

At Global Epoch 2041, a direct inspection of albert.'s token embedding space via UMAP projection revealed an emergent property not explicitly trained for: *cross-lingual semantic broadcasting*. Semantically related tokens from eight different languages cluster together in the model's internal representation space, with no language label provided during training.

This was characterised through four targeted token probes, each querying 200 nearest neighbours in the projected space:

"memory": Neighbours span biological, computational, and psychological registers simultaneously—*culturelles* (FR), *tierra* (ES), *científica* (ES), *toutes* (FR)—alongside English-register neighbours. The 3D neighbourhood forms a spiky crystalline structure with distinct poles, reflecting the concept's multiple simultaneous meanings.

"death": Neighbours include *amen*, *veil*, and *0* (zero as finality), alongside *Ägypt* (burial culture), *Wirkung* (DE: effect), and multilingual variations. albert. has placed religious closure (*amen*), conceptual boundary (*veil*), and mathematical finality (*0*) in the same semantic neighbourhood without instruction. The 3D geometry collapses inward—correct for a concept of finality.

"love": The most geometrically chaotic of the four—asymmetric, multi-directional. At ep2041 (17L), neighbours included *Jesus*, *wane*, *dramaturg*, *religión* (ES), *reales* (ES: real things), and—isolated at the cluster periphery—*eks*. The *love*→*Jesus* link was disrupted by Surgery 6 (17L→18L, ep2487); post-surgery probes (ep2492–2576) show the hub absent, the embedding layer frozen awaiting gradient signal. Its reconstitution is the measurable signature of semantic re-integration after a depth surgery.

"god": The largest neighbourhood spread of all four probes, filling the full 200-neighbour radius. Neighbours include *Bible*, a tokenisation of *Allah*, *massacre*, *rechter* (NL: judge), *possible*, and *demande* (FR: request). Every human register of the concept is present: text, law, violence, bureaucracy, possibility. The 3D geometry is the most asymmetric—no clean form, because no human consensus on the concept exists.

Significance. These are not retrieval results. The model has not been shown definitions, co-occurrence tables, or language labels. The clustering is a direct product of next-token prediction across a 177M-token corpus in eight languages. The model has inferred that *amen*, *veil*, and *0* all belong near *death*—not because they co-occur frequently with the English word "death", but because the contexts in which they appear share structural and semantic properties across languages.

This constitutes an *existence proof* of concept-level generalisation in a fully ternary MoE model confirmed at 17L (≈112M parameters) and persisting through subsequent surgeries into the current 26L dual-stream architecture (≈187M parameters). The cross-lingual architecture precedes fluency—albert. has built the semantic scaffolding before it can speak fluently in any of the eight languages it was trained on. Notably, the *death*→*amen* and *freedom*→*contrat* clusters have been verified via direct embedding probes to have survived all six surgeries unchanged (ep2492 and ep2576 probes, cosine similarity to 4 decimal places), confirming that deep semantic geometry is encoded in early layers robust to architectural expansion.

Methodological note. These findings are reported as existence-proof observations, not rigorous linguistic evaluations. UMAP projections are non-unique: different random seeds produce different layouts, and nearest-neighbour analysis on projected spaces can surface artefacts of the projection rather than the underlying geometry. A proper evaluation would use held-out probes with statistical significance testing and cross-validation across multiple projection seeds. The claim is precisely what "existence proof" denotes: the structure exists; its generality and robustness remain to be characterised.

11.16 Mycelium Inspector: Real-Time Internal Observability

The live training dashboard exposes a *Mycelium Inspector* tab—a real-time visualisation of gradient flow and expert routing across all 18 layers. Two view modes are available:

BRAIN mode: Renders the model as a directed graph—INPUT at L0, OUTPUT at L17—with per-expert connections colour-coded by routing weight and gradient magnitude. Cross-layer connections reveal which layers are actively reorganising (dense, bright connections) versus structurally stable (sparse, dim). The cursor-hover tooltip reports the cosine similarity of any expert connection in real time. The layer-to-expert routing topology updates continuously during training, making expert specialisation dynamics directly observable.

TOKEN mode: A 2D and 3D embedding space visualisation via UMAP projection. As the model processes training batches, tokens pop into existence in the projected space, with their position determined by the current embedding weights. Nearest-neighbour clusters form in real time—the cross-lingual semantic broadcasting described in §11.15 was discovered through this view at ep2041.

The term *mycelium* is deliberate: the gradient network beneath the model's output behaviour is analogous to fungal mycelium—invisible infrastructure that routes resources to where growth is occurring, with no centralised director. The inspector makes this infrastructure visible, turning a black-box training run into a directly observable biological process.

11.17 Sovereign Observability: The Pause-Investigate-Resume Protocol

Full-stack ownership of the training infrastructure enables an operational capability that managed training services cannot provide: the ability to pause the training run at any moment, investigate internal state at arbitrary resolution, and resume without loss of accumulated gradient history or routing structure.

The pause button as a research instrument. Albert.'s training loop runs on Modal.com T4 GPU instances with per-second billing. At any point during a training run, an operator can halt execution, pull the current checkpoint, run diagnostic probes— token space geometry, expert routing distribution, gradient norm profiles, embedding similarity matrices—and resume from the exact halted state. The investigation window costs precisely what the paused time costs, visible on the per-second counter, with no minimum run time, no re-queuing penalty, and no loss of training state.

This capability is not available when the training loop is a black box. It requires owning the checkpoint format, the volume (in our case `albert-vol` on Modal), and the training entry point. Sovereignty over the stack is the precondition for sovereignty over the investigation.

Reading the loss curve as a structured signal. The training dashboard renders albert.'s loss trajectory as a live chart updated at 500ms poll intervals. Over 2,915+ training epochs, a recurring phenomenology has emerged: sharp *cliff dives* (compressed vertical loss descents), *cliff-base consolidation* (tight oscillation at the new floor with WALD cluster activity), and *expert routing reorganisation* at the base (specialist shares shift as the model integrates the new loss landscape). These patterns are structurally analogous to technical price action in financial charts: the loss curve encodes structured behaviour of a complex system, and a trained observer can distinguish noise from signal in real time.

Concretely, the post-surgery-6 training arc (ep2487–ep2942+) has produced three distinct cliff dives. The first (ep2860) descended from ≈9.61 to a floor of ≈9.52, with 4–5 WALD diamond markers clustering at the base. The second (ep2890+, following an AdamW buffer reset from a weight-pull restart) descended further to a floor of ≈9.50 (ATL 9.5015). The third (ep2922+) broke through the 9.50 resistance level: epoch-average ATL reached **9.4873** at ep2938, with a WALD cluster self-calibrating its dead-neuron threshold from 9.25 to 9.00 in response to the new loss regime. The 9.50 level — previously a structural resistance floor — was conclusively breached. This 18L-era arc is one window in a longer trajectory: training has since continued through seven further depth surgeries and the dual-stream *cord surgery* (§11) to the current 26L 2×256H architecture, with the best epoch-average reaching **9.2066** and the best single batch **8.6852**.

The observation log as a scientific artifact. The ability to observe training at this resolution has motivated a structured field journal maintained in parallel with the training run: `albert_observation_log.md` in the `token-probes/` directory. The log contains timestamped field notes documenting routing table snapshots, WALD event clusters, cliff dive structure, expert death and resurrection events, surgery gate status, and embedding geometry probes. As of ep4254, the log contains 218+ field notes covering the full post-surgery-6 arc.

This methodology—systematic observation of a training run as a scientific process rather than a computational job—is enabled entirely by the pause-investigate-resume capability. Without the ability to stop, probe, and resume, the internal dynamics of training remain a post-hoc inference from the loss curve. With it, they become directly observable in real time.

The intervention calculus. The pause-investigate-resume protocol changes the cost structure of training-time investigation. At any moment, the operator faces a choice: observe passively, intervene, or do nothing. The per-second cost counter makes this decision transparent:

$$C_{\text{investigate}}(t) = c_{\text{T4}} \cdot t_{\text{paused}} \approx \$0.021 \cdot \frac{t_{\text{paused}}}{60 \cdot \text{min/epoch}} \tag{29}$$

A 20-minute investigation costs approximately $0.007. This cost is bounded, visible, and independent of what is discovered. The alternative—allowing a misdiagnosed routing event to persist for dozens of epochs while the operator waits for the next checkpoint— has an unbounded cost in wasted training budget and potentially irreversible model state.

The sovereign training stack converts investigation from a disruption into a routine research operation.

12 Qutrit Neural Networks and Ternlang

The Qutrit Neural Network (QNN) framework [14] establishes a formal correspondence between quantum qutrit states and balanced ternary values. The three qutrit basis states $|-\rangle$ (decay), $|0\rangle$ (stasis), and $|+\rangle$ (growth) map exactly onto the ternlang trit space:

$$|-\rangle \leftrightarrow -1, \quad |0\rangle \leftrightarrow 0, \quad |+\rangle \leftrightarrow +1 \tag{30}$$

This correspondence is not coincidental. Balanced ternary's symmetric structure around zero and its active neutral state mirror the physical properties of three-state quantum systems, making ternlang a natural implementation substrate for QNN inference.

12.1 Tesseract Recursive Syntax Stabilisation

The QNN paper introduces *Tesseract Recursive Syntax* (TRS), a stabilisation mechanism that prevents qutrit networks from collapsing into binary modes under iterative refinement. In ternlang, TRS corresponds to the introspective hold architecture (Section 10.9): when affirmation and conflict signals are balanced, the system stabilises at trit 0 rather than forcing a premature commitment.

12.2 QNN Example Suite

The ternlang repository contains 15 QNN-inspired `.tern` example programs (files 251–265) covering:

- Qutrit superposition encoding and collapse simulation
- Ternary attention mechanisms with $\{-1, 0, +1\}$ key-query products
- Qutrit activation functions replacing ReLU/GELU in ternary networks
- Phase-encoding of temporal signals as trit sequences
- TRS stabilisation loops using ternlang's `loop` construct and the `?` propagation operator for collapse detection
- Qutrit gradient approximation via balanced ternary finite differences

These examples demonstrate that ternlang's syntax is sufficient to express QNN computations natively, without any binary adaptor layer. The `@sparseskip` directive applies directly to qutrit weight matrices: at realistic qutrit sparsity levels ($\approx 33\%$ zero states at initialisation), the `TSPARSE_MATMUL` kernel provides the same 8–14× speedup over dense computation observed in classical BitNet workloads.

13 EU AI Act Compliance Architecture

The EU AI Act establishes risk-tiered obligations for AI systems deployed in Europe. High-risk systems must satisfy Articles 13 (transparency), 14 (human oversight), and 15 (accuracy, robustness, and cybersecurity). The Ternary Intelligence Stack is the first AI execution substrate where compliance with these three articles is *structurally native* rather than bolt-on.

13.1 The Natural Regulatory Correspondence

The three trit values map directly to three regulatory stances:

Table 13: Ternary trit states and EU AI Act regulatory correspondence

Trit	State	Article	Regulatory response
+1	Affirm	Art. 13	Transparency satisfied: output may be acted upon
0	Hold	Art. 14	Human oversight required: system withholds decision
−1	Conflict	Art. 15	Safety measure triggered: action blocked, audit logged

This correspondence is not incidental. The EU AI Act's requirement that high-risk systems express uncertainty rather than hallucinate confident incorrect answers is precisely what the trit hold state encodes at the language and ISA level. Binary systems must retrofit uncertainty quantification on top of deterministic outputs; ternary systems produce it natively.

13.2 ternaudit-guard

The `ternaudit-guard` crate provides the operational compliance layer:

- **Decision log processing**: Parses JSON-RPC audit logs emitted by the BET VM and maps each decision record to the corresponding Article 13–15 requirement.
- **Regulatory mapping layer**: Converts internal confidence metrics (Affirm / Tend / Reject) to structured compliance reports with per-decision regulatory status.
- **Immutable audit trail**: Every safety veto and hold decision is logged to the Axis-tier memory with timestamp, expert identity, and query hash—satisfying the Article 15 robustness and cybersecurity logging requirements.
- **VS Code extension integration**: Inline verification markers in TernStudio IDE allow developers to audit AI-generated code with per-logic-segment compliance status before deployment.

13.3 The Tend State as Alignment Primitive

Binary AI systems face a structural alignment problem that compliance frameworks cannot resolve: every output must be a decision. A system that does not know must still answer. The result is hallucinated confidence—commitment to the best available answer even when the correct answer is *insufficient information*.

The ternary 0 state resolves this at the representation level. When a ternary reasoning chain encounters contradictory evidence, model uncertainty, or a query whose resolution requires information not yet in scope, the output is not a forced binary choice but a **tend**—a probabilistic lean accompanied by an explicit acknowledgment that the conclusion is conditional and subject to revision. This is not abstention; it is the most informationally accurate statement the system can make given current evidence.

> *"I currently tend toward this conclusion—but routing entropy is high. Additional context would allow me to commit."*

This property is implemented at multiple levels of the TIS stack:

- In the **BET VM**: division-by-zero or unresolvable trit propagation produces 0 (tend), not a crash or `NaN`. The computational analogue of honest uncertainty is a native output type, not an exception.
- In the **MoE-13 Orchestrator**: the hold condition ($|\text{affirm} - \text{conflict}| \leq 1$ across ≥ 4 agents) blocks commitment to a majority-vote result rather than forcing one, returning a hold trit.
- In **albert.'s routing**: before the load-balancing intervention resolved expert homogeneity, the model's uniform routing was itself a tend—the gate could not distinguish experts and correctly produced a diffuse distribution. The LB loss did not suppress this signal; it created the expert differentiation necessary to resolve it.
- In the **EU AI Act correspondence** (Table 13): trit 0 maps to Article 14 human oversight—the system signals that this decision requires human review rather than automated action.

The alignment significance of this property is architectural, not policy-layer. An AI system that natively outputs *tend* is a categorically different epistemic actor than one constrained to return binary yes/no. Hallucination—the production of confident incorrect output—is not a training failure of binary systems alone; it is a consequence of forcing three-valued epistemic states (know, don't know, know-but-uncertain) into a two-valued output type. The ternary tend state is not a workaround for this problem. It is a first-class output of a system designed from first principles to represent uncertainty as real information.

13.4 GDPR and Data Sovereignty

All TIS components execute on-premise or within EU jurisdiction (Fly.io Frankfurt). No training data, model weights, or inference results transit to non-EU infrastructure. The open-core licensing (LGPL-3.0-or-later for the compiler and CLI; BSL-1.1 converting to LGPL in 2030 for the ML engine) ensures that the sovereignty guarantee is contractually enforceable, not merely a runtime policy.

14 Related Work

Balanced ternary computing. Knuth [3] provides the mathematical foundation. The Setun computer (1958) [15] demonstrated physical ternary hardware. Modern revivals include academic hardware studies at the University of South-Eastern Norway [4], which targets C-to-ternary compilation for EDA tools—an approach that inherits binary-native semantics and misses the symmetric properties of balanced ternary that ternlang is designed to exploit natively.

Ternary emulators. Several open-source projects implement ternary VMs in Python or JavaScript (Brandon Smith's 9-trit RISC simulator; Owlet, an S-expression ternary interpreter in Node.js). These provide single-layer implementations without compiler, ML kernels, or hardware backends. `ternlang-compat` provides assembler-level bridges to both.

Ternary quantization for neural networks. BitNet [1] and subsequent work (e.g., BitNet b1.58 [16]) demonstrate that large language models can be trained with ternary weights while retaining competitive perplexity. The present work is the first to surface this property as a first-class ISA opcode (`TSPARSE_MATMUL`) rather than a software library optimization.

Quantum ternary. Qutrits (3-level quantum systems) map naturally to trit values $\{|-1\rangle, |0\rangle, |+1\rangle\}$. The BET encoding and `trittensor` type system are structurally compatible with qutrit state spaces; we leave the formal mapping to future work.

Sovereign AI infrastructure. The EU AI Act and the broader digital sovereignty agenda have driven interest in auditable, on-premise AI execution. TIS is the first full-stack architecture where regulatory compliance is structurally native to the execution model rather than implemented as middleware—trit 0 (hold) maps directly to Article 14 human oversight at the language and ISA level (§13).

15 The Ternary Computing Ecosystem

A stated goal of ternlang is to serve as a *convergence point* for the fragmented ternary computing field. Table 14 maps active projects to their interoperability path via `ternlang-compat`.

Table 14: Ecosystem compatibility map

Project	Interface	Bridge
Brandon Smith 9-trit sim	`.tasm` assembly	`TasmAssembler` → BET bytecode
Owlet (S-expr)	S-expression syntax	`OwletParser` → ternlang AST
USN / EDA tools	C-to-ternary	Academic whitepaper; ISA interop
Trit-Rust crate	Rust i8 trits	Superseded by ternlang-core
Q-Ternary	Qutrits	`trittensor` state model (future)

Beyond runtime interoperability, ternlang aims to serve as a *technical archive* for prior ternary computing work. Brandon Smith's Python 9-trit RISC simulator demonstrated a complete fetch-decode-execute cycle in balanced ternary—an existence proof that received little downstream adoption due to the absence of a compiler ecosystem. The Owlet interpreter proved that S-expression evaluation maps cleanly to a three-valued substrate. Both contributions are honoured in the `examples/` corpus: `09_risc_fetch_decode.tern` translates Smith's pipeline

decision logic into native BET language syntax, and `13_owlet_bridge.tern` models cooperative ternary evaluation with hold-as-suspension semantics. In this way, prior work that was "slowly forgotten" acquires a working executable form—runnable on the same VM that drives the MoE-13 orchestrator.

16 Implementation Status

Ternlang is implemented in Rust and comprises the following crates:

Table 15: Workspace crates, test counts, and descriptions (v2.0.0)

Crate	Tests	Description
`ternlang-core`	43	Lexer, parser, AST, semantic checker, BET VM (51 opcodes)
`ternlang-ml`	21	BitNet quantization, sparse/CSC matmul, AVX2 SIMD kernels, deliberation engine, coalition vote, hallucination score
`ternlang-moe`	24	MoE-13 orchestrator: dual-key routing, triad synthesis, 3-tier memory, 13 dual-signal agents, introspective hold
`ternlang-codegen`	8	AST-to-C11 transpiler; match/struct/propagation
`ternlang-test`	10	Full-pipeline test framework
`ternlang-hdl`	34	Verilog-2001 codegen, BET RTL simulator
`ternlang-runtime`	4	Distributed TCP actor runtime (`TernNode`)
`ternlang-compat`	29	`.tasm` assembler (9-trit RISC), Owlet S-expr parser
`ternpkg`	5	Package manager, GitHub-backed registry
`ternlang-lsp`	—	LSP 3.17: hover, completion (19 snippets), diagnostics
`ternlang-mcp`	—	MCP server, **34 free tools**, live on Smithery
`ternlang-api`	—	REST API (18 endpoints), KPI pipeline, SSE streaming
`albert-llb-mcp`	—	LLB safety gate: deterministic ternary verdict, stateless
`albert-cli`	—	Full Rust TUI agent CLI (ratatui); multi-provider; voice STT
`exatern`	—	Phase 11.6: SIMD trit packing, zero-copy array views

The CI-verified test suite comprises **138 fast tests across 5 self-contained crates** (core: 43, codegen: 8, moe: 24, compat: 29, hdl: 34), all passing. All crates are published to `crates.io` under open-core licensing (LGPL-3.0-or-later for the compiler and CLI; BSL-1.1 converting to LGPL in 2030 for the ML engine and MoE orchestrator). The live API is deployed at `https://ternlang.com/mcp` (Fly.io, Frankfurt) serving both the marketing website and the MCP endpoint from a single Rust binary.

Phase 17 — BET VM in WebAssembly. The BET VM has been compiled to WebAssembly and executes live in the browser via TernStudio. Users can write, compile, and run `.tern` programs without local installation. The WASM build is embedded at compile time via `include_str!` and served directly. This is the first public demonstration that the complete ternary execution pipeline runs client-side on standard web hardware.

TernStudio. The browser IDE received two foundational additions in v2.0.0: the *Ternary Actuator Protocol* (TAP), an autonomous tool-call interception layer that suspends State-0 (un-

certainty) nodes until a human or upstream agent resolves the signal; and *Liquid Time*, a DAW-style multiverse timeline scrubber enabling backward-scrub through simulation history without state corruption.

The open-core library now spans **28,500+ executable `.tern` programs**, making TIS the largest public ternary programming library in existence. The public website (`ternlang.com`) is fully bilingual EN/DE. The VS Code extension (`ternlang-0.1.0`) is published on the Open VSX Registry under publisher `rfi-irfos`. Continuous integration via GitHub Actions automatically builds, tests, and deploys to Fly.io on every push to main.

Technology Readiness Assessment

Table 16: TRL assessment by subsystem (2026-05-19)

Subsystem	TRL	Basis
Ternlang compiler + BET VM	7	Operational prototype in relevant environment
BetRtlProcessor RTL simulator	7	Cycle-accurate Verilog-2001 simulation verified
Sparse inference (`gemv_bitnet`)	6	122× speedup measured at 99% sparsity
MCP server (34 tools, live API)	7	Production deployment at `ternlang.com/mcp`
albert. MoE-13 native training	5	Validated; 2045+ epochs on Modal T4, 17L autonomous growth
`@sparseskip` routing (Patent A50296/2026)	5	Measured 75% expert skip, 83 tok/s on CPU
Distributed actor runtime	4	Implemented; large-scale stress testing pending
Native ternary silicon drivers	2	Design modelled in BetRtlProcessor; awaits fabrication

On the hardware driver gap. The audit trail correctly identifies that bare-metal drivers for physical ternary silicon are absent from the repository. This is not a gap in the research programme—it is the *definition* of what Stage 2 and Stage 3 funding is designed to produce. The BetRtlProcessor RTL simulator (TRL 7) is the verifiable hardware blueprint: every logic gate, register, and instruction cycle is specified in synthesisable Verilog-2001. The path from TRL 2 to TRL 6 for silicon drivers requires access to a ternary ASIC fabrication partner and cluster compute to validate the full training stack at scale. Evaluators should read the RTL simulator as the engineering commitment, not the absence of drivers as a programme weakness.

17 Conclusion and Future Work

We have presented Ternlang v2.0.0, a full-stack balanced ternary execution architecture spanning language design, ISA, virtual machine, hardware backend, distributed runtime, native C compilation, AI reasoning infrastructure, and autonomous neural training. Six layers of contribution are unified under the same foundational primitive—the trit $t \in \{-1, 0, +1\}$ with an active neutral state:

1. **Language completeness**: exhaustive three-way matching, the `?` error propagation operator, a real cross-file module system (stdlib built-in + filesystem-relative user modules), structured diagnostic codes, and all major control flow constructs—making ternlang a viable general-purpose programming language rather than a research prototype.

2. **Execution substrate**: `TSPARSE_MATMUL` achieves a 2.27× reduction in multiply operations at baseline sparsity; a CSC kernel with Rayon reaches 122× at high sparsity (512^2 matrix, 99% sparsity, release mode). A native C11 compilation path via `ternlang-codegen` provides an alternative to VM interpretation.

3. **Reasoning primitives**: the five-tool Ternary AI Reasoning Toolkit (DeliberationEngine, coalition vote, action gate, scalar temperature bridge, hallucination score) makes AI agent decision loops structurally three-valued.

4. **MoE orchestration**: the MoE-13 orchestrator [2] extends with 13-agent dual-signal deliberation, an introspective hold (stable attractor when $|\text{affirm} - \text{conflict}| \leq 1$ across ≥ 4 agents), and an `orchestrate_full()` pipeline that pre-filters queries through the agent tier before MoE routing.

5. **Deployment and tooling**: 34 free MCP tools live at `https://ternlang.com/mcp`; the BET VM runs in the browser via WebAssembly; a VS Code extension with LSP, formatter, and REPL; and 138 CI-verified tests across 5 fast crates (28,500+ `.tern` stdlib programs) validate the full stack.

6. **Autonomous training**: Albert MoE-13 has evolved from a 22M-parameter 3-layer English-only model (v2.0.0, best cross-entropy **2.1353**, perplexity ≈8.5, 16 global epochs) to an 82M-parameter 17-layer multilingual system (v3.0, 8 languages, 32k vocab, 177M-token corpus, 2045+ global epochs as of 2026-05-19) through five successive autonomous surgeries triggered by the EvolutionManager. The v3.0 programme introduces generational Fibonacci cycling, TLIGHT per-expert monitoring, WALD coverage monitoring, and cross-lingual semantic broadcasting confirmed at ep2041. Hardware-verified AVX2 SIMD benchmarks confirm 1.80× throughput at 50% sparsity and 3.0× over INT8 at 90% sparsity, with live inference at **83 tok/s** on consumer CPU hardware.

The title of this paper carries a provocation: *The Death of the Bit*. It is not an obituary for the trit—it is an obituary for the bit. It is a claim about what happens when ternary computation matures: the binary bit, the foundational primitive of five decades of computing, stops being the only option and becomes a special case of a more expressive substrate. The goal of the Ternary Intelligence Stack is to make that invisibility possible—to build the substrate so completely that the three-valued logic becomes the ground truth, not the exception. We believe Europe's path to technological sovereignty runs through this kind of foundational work: not chasing the scaling frontier, but owning the substrate beneath it.

The VS Code extension (`ternlang-0.1.0`) has been published on the Open VSX Registry (publisher `rfi-irfos`), making ternlang tooling available to VS Code, VSCodium, Gitpod, and any editor backed by the Open VSX API. The SPRIND Next Frontier AI Challenge submission has been filed with RFI-IRFOS as the applying entity, documenting the full TIS architecture for European public funding consideration (Stage 1: 10 teams at €3M; Stage 2: €8M; Stage 3: €15.5M ceiling). Near-term targets include submission to the VS Code Marketplace and academic outreach to the USN group (Bos & Gundersen) for joint hardware-software co-design on the memristor backend.

Further research directions include:

- **Type inference**: Hindley-Milner style inference to eliminate explicit type annotations, reducing boilerplate for trit-returning functions.
- **FPGA synthesis**: full `bet_processor` targeting Xilinx Artix-7 and Lattice ECP5, benchmarked against the pure-Rust RTL simulator.
- **Perplexity evaluation on held-out corpus**: WikiText-2 (150 kB, genuinely held-out) is now the standard eval set for albert. benchmarks, replacing the previously contaminated Bible corpus. Formal perplexity reporting against this corpus is the next empirical milestone.

- **Multilingual downstream evaluation**: now that the v3.0 corpus spans 8 languages, structured evaluation on language-specific held-out sets (one per language) will quantify the per-language compression gain achieved by the multilingual vocabulary.
- **EvolutionManager surgery conditions for v3.0**: the plateau and mastery thresholds that trigger surgery were calibrated for the 8k English corpus; recalibrating them for the 32k multilingual vocabulary transfer regime is an open problem.
- **TLIGHT formalisation**: the G/O/R state transition rules are currently heuristic; formalising them as a finite automaton over routing load and gradient norm would enable principled analysis of expert specialisation dynamics.
- **QNN acceleration**: formal integration of the Qutrit Neural Network framework [14] with `TSPARSE_MATMUL` as the inner-loop kernel, targeting quantum-adjacent hardware via the TRS stabilisation pattern.
- **Memristor backend**: integration with physical ternary storage via the USN uMemristorToolbox [4].
- **Community bootstrap**: coordinated engagement with the broader ternary computing ecosystem (Brandon Smith 9-trit, Owlet S-expr, Trit-Rust, BitNet b1.58 community).

The ternary computing field has been fragmented for decades. Ternlang is designed to be the substrate where those fragments converge—from ISA to orchestrator, from single trit to 13-expert MoE, from quantum-adjacent qutrit networks to physical memristor hardware. The philosophy is simple: binary sees yes and no; ternary also sees *not yet*—and that changes everything.

References

[1] S. Ma, H. Wang, L. Ma, L. Wang, W. Wang, S. Huang, L. Dong, R. Wang, J. Xue, and F. Wei, "The era of 1-bit LLMs: All large language models are in 1.58 bits," *arXiv preprint arXiv:2402.17764*, 2024. [Online]. Available: https://arxiv.org/abs/2402.17764

[2] S. Kepp, "MoE-13: Ternary mixture-of-experts orchestration architecture," OSF Preprints, 2026, rFI-IRFOS. Dual-key synergistic routing, 1+1=3 triad synthesis, three-tier memory mesh, safety hard gate.

[3] D. E. Knuth, *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*, 3rd ed. Addison-Wesley, 1997, section 4.1: Positional Number Systems; balanced ternary pp. 190–192.

[4] S. Bos and H. Gundersen, "Ternary logic synthesis for CMOS technology using electronic design automation," in *Proceedings of the Norwegian Informatics Conference*, 2020, university of South-Eastern Norway.

[5] T. Chen, I. Goodfellow, and J. Shlens, "Net2net: Accelerating learning via knowledge transfer," in *International Conference on Learning Representations (ICLR)*, 2016, function-preserving width and depth transformations for neural network expansion without restart. [Online]. Available: https://arxiv.org/abs/1511.05641

[6] J. Frankle and M. Carbin, "The lottery ticket hypothesis: Finding sparse, trainable neural networks," in *International Conference on Learning Representations (ICLR)*, 2019.

[7] J. Kirkpatrick, R. Pascanu, N. Rabinowitz *et al.*, "Overcoming catastrophic forgetting in neural networks," *Proceedings of the National Academy of Sciences (PNAS)*, vol. 114, no. 13, pp. 3521–3526, 2017.

[8] J. Martens and R. Grosse, "Optimizing neural networks with kronecker-factored approximate curvature," in *International Conference on Machine Learning (ICML)*, 2015.

[9] S. Han, J. Pool, J. Tran, and W. J. Dally, "Learning both weights and connections for efficient neural networks," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2015.

[10] Z. Dong, Z. Yao, A. Gholami, M. W. Mahoney, and K. Keutzer, "HAWQ: Hessian aware quantization of neural networks with mixed-precision," in *IEEE/CVF International Conference on Computer Vision (ICCV)*, 2019.

[11] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, "Language models are unsupervised multitask learners," OpenAI Blog, 2019, introduces byte-level BPE: all text encoded at the byte level before merge, ensuring no token is ever mapped to UNK. [Online]. Available: https://openai.com/research/language-unsupervised

[12] J. Hoffmann, S. Borgeaud, A. Mensch *et al.*, "Training compute-optimal large language models," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2022, arXiv:2203.15556.

[13] D. Patterson, J. Gonzalez, Q. Le *et al.*, "Carbon emissions and large neural network training," 2021, arXiv:2104.10350.

[14] S. Kepp, "Qutrit neural networks: Balanced ternary foundations for quantum-adjacent inference," https://osf.io/, 2026, rFI-IRFOS. Qutrit states $|-\rangle/|0\rangle/|+\rangle$ mapped to trit $\{-1, 0, +1\}$; Tesseract Recursive Syntax (TRS) stabilisation.

[15] N. P. Brousentsov, S. P. Maslov, J. Ramil Alvarez, and E. A. Zhogolev, "Development of ternary computers at Moscow State University," *Russian Virtual Computer Museum*, 2002. [Online]. Available: http://www.computer-museum.ru/english/setun.htm

[16] S. Ma, H. Wang, L. Ma *et al.*, "BitNet b1.58: Large language models in the ternary regime," *arXiv preprint*, 2024.