

# 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

linkedin/simeon-kepp

s.kepp@ternlang.com Zabih Karimi IT & Network Engineering

linkedin/zabih-karimi

z.karimi@ternlang.com Nikoletta Csonka International Relations

linkedin/csonikoletta

csonikoletta@ternlang.com Lisa Scharler Social Technology & AI Ethics

linkedin/lisa-scharler

l.scharler@ternlang.com Louis Paul Ehrig Public Affairs & Dataset Curation

linkedin/louis-ehrig

l.ehrig@ternlang.com

**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  $512 \times 512$  elements yields 56.2% sparsity and a  $2.27\times$  reduction in multiply operations versus dense execution; a Compressed Sparse Column kernel with Rayon parallelism achieves up to  $122\times$  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 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  $\approx 8.5$ ) trained entirely on a 2013 consumer CPU with no GPU. Second, independently verified AVX2 SIMD hardware benchmarks confirm  $1.80\times$  throughput at 50% sparsity and  $3.0\times$  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 multiple sequential surgeries from 5L to 13L—the first documented multi-surgery trajectory in a fully ternary MoE—growing the model to  $\approx 63$ M parameters 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.

**Index Terms**—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

## I. 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 (§III).
- 2) The **BET ISA**: a formal 2-bit-encoded instruction set with opcode coverage from arithmetic through tensor operations and distributed agent control (§IV).
- 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 (§V).
- 4) A **Verilog-2001 hardware backend** with synthesisable sparse matmul array and full BET processor, plus a cycle-accurate RTL simulator in pure Rust (§VI).
- 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 (§IX).

- 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 (§X).
- 7) **Ecosystem bridges** connecting existing ternary projects to the BET VM as a common runtime (§XIV).
- 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`.

## II. BACKGROUND: BALANCED TERNARY

### A. 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 I  
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  $\text{neg}(0) = 0$ .*

### B. The 2-bit BET encoding

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

TABLE II  
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).*

### III. THE TERNLANG LANGUAGE

#### A. Design principles

Terlang 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.

#### B. Core language constructs

Listing 1. Ternary classifier with exhaustive match

```
1 fn classify(signal: trit) -> trit {
2   match signal {
3     -1 => conflict() // disagreement
4         confirmed
5     0 => hold()      // awaiting evidence
6     +1 => truth()    // assertion confirmed
7   }
8 }
```

Listing 2. Sparse matrix multiply with `@sparseskip`

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

Listing 3. Actor model for ternary message passing

```
1 agent Voter {
2   fn handle(msg: trit) -> trit {
3     consensus(msg, hold())
4   }
5 }
6
7 let v: agentref = spawn Voter;
8 send v truth();
9 let decision: trit = await v;
```

#### C. 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.

#### D. Ternary Error Propagation

Terlang 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:

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

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?`

```
1 ; expr already on stack: [val]
2 TDUP          ; [val, dup]
3 TJMP_NEG L1   ; consume dup; if -1 -> L1; else [
4   val] continues
5 L1: TRET      ; skip early return
6 L2: TRET      ; return -1 to caller
7 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

```
1 fn validate_signal(x: trit) -> trit {
2   let checked: trit = boundary_check(x)?; //
3   return range_check(checked)?;           //
4   propagate if conflict
5   propagate again
6 }
7
8 fn main() -> trit {
9   return validate_signal(1)?; // -1 if any check
10  fails
11 }
```

#### E. Cross-File Module System

Terlang’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

```

1 // agents/voter.tern
2 fn weighted_vote(a: trit, b: trit, c: trit) ->
   trit {
3   consensus(consensus(a, b), c)
4 }
5
6 // main.tern
7 fn main() -> trit {
8   use agents::voter;
9   return weighted_vote(1, 0, -1);
10 }

```

### F. 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.

## IV. THE BET INSTRUCTION SET ARCHITECTURE

### A. 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:  $27 = 3^3$ .

### B. Instruction encoding

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

TABLE III  
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 $addr$	$t \rightarrow (\text{jmp if } t = +1)$
0x06	TJMP_ZERO $addr$	$t \rightarrow (\text{jmp if } t = 0)$
0x07	TJMP_NEG $addr$	$t \rightarrow (\text{jmp if } t = -1)$
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 $N\ M$	$\rightarrow \text{ref}$
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	$\text{ref } i\ j \rightarrow t$
0x23	TSET	$\text{ref } i\ j\ t \rightarrow$
0x24	TSHAPE	$\text{ref} \rightarrow N\ M$
0x25	TSPARSITY	$\text{ref} \rightarrow \text{count}$
0x30	TSPAWN $id$	$\rightarrow \text{agentref}$
0x31	TSEND	$\text{agentref } m \rightarrow$
0x32	TAWAIT	$\text{agentref} \rightarrow t$

## V. SPARSE TERNARY INFERENCE

### A. 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  $\tau$ :

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

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  $\tau$  as defined in (2).

### B. 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:

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

In a dense  $N \times M$  matrix multiply, every element contributes  $N \cdot M$  multiplications. After ternary quantization with sparsity  $\rho$  (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

```

1 for i in 0..N:
2   for j in 0..M:

```

```

3   for k in 0..K:
4       w = W[k][j]
5       if w == HOLD: continue // skip
6       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.

### C. Benchmark results

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

TABLE IV  
SPARSE VS. DENSE TERNARY MATMUL ( $512 \times 512$ )

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

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

### D. 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 V  
AVX2 SIMD TERNARY INFERENCE — PRODUCTION HARDWARE THROUGHPUT

Sparsity	vs. Dense Ternary	vs. INT8
10%	1.02×	—
50%	<b>1.80×</b>	—
90%	1.63×	<b>3.0×</b>

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.

## VI. HARDWARE BACKEND

### A. 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

### B. 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

```

1 wire [1:0] w_ij = weight_reg[i][j];
2 wire skip = (w_ij == 2'b11); // hold = zero
   weight
3 wire [1:0] contrib = skip ?
   2'b11 : // hold if skipped
4   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.

### C. BET processor

The full `bet_processor` module wires together:

- `bet_regfile`: 27-register file ( $27 \times 2$  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.

## VII. 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.

### A. Transpilation Architecture

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

$$\text{.tern} \xrightarrow{\text{parse}} \text{AST} \xrightarrow{\text{CTranspiler}} \text{.c} \xrightarrow{\text{gcc/clang}} \text{native binary} \quad (4)$$

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()`.

### B. 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

```

1 // ternlang source
2 fn decide(x: trit) -> trit {
3     match x {
4         -1 => conflict()
5         0  => hold()
6         +1 => truth()
7     }
8 }
9
10 // generated C
11 trit decide(trit x) {
12     switch ((int)x) {
13         case -1: return trit_conflict(); break;
14         case 0:  return trit_hold();      break;
15         case 1: return trit_truth();      break;
16     }
17 }
```

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.

## VIII. ACTOR MODEL AND DISTRIBUTED RUNTIME

### A. 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.

### B. 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

```

1 let remote_voter: agentref =
2     spawn remote "192.168.1.42:7373" Voter;
3 send remote_voter truth();
4 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.

## IX. 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.

### A. Deliberation Engine

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

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

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.

### B. 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:

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

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

### C. 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:



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

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

#### D. 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.

### X. 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.

#### A. Six-Dimensional Competence Space

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

$$\mathbf{v} = [v_{\text{syntax}}, v_{\text{world}}, v_{\text{reason}}, v_{\text{tool}}, v_{\text{persona}}, v_{\text{safety}}] \quad (8)$$

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

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

#### B. 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:

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

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  $\sigma$ ; the router favours complementary coverage over redundant strength.

#### C. 1+1=3 Triad Synthesis

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

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

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.

#### D. 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:

$$\text{veto} \iff E_{k,\text{safety}} < \theta_s \quad (12)$$

where  $\theta_s$  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.

#### E. 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.

#### F. 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 ( $\text{risk} = \text{max\_pair}/\text{total}$ ).
- **Axis** (persistent): immutable audit log of safety vetoes with timestamp, expert identity, and query hash. Global priors over expert performance.

#### G. 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.

### H. 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:

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

where  $\theta^+$  and  $\theta^-$  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  $\geq 2$  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.

#### I. 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:

$$\text{stable\_hold} \iff |\text{affirm\_count} - \text{conflict\_count}| \leq 1 \wedge \text{engaged} \geq \text{trit\_action\_gate} \quad (14)$$

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

```
1 pub struct AggregateVerdict {
2   pub trit: i8,
3   pub confidence: f32,
4   pub verdicts: Vec<ExpertVerdict>,
5   pub is_stable_hold: bool,
6   pub hold_reason: Option<String>,
7   pub affirm_count: usize,
```

```
8   pub conflict_count: usize,
9   pub hold_count: usize,
10 }
```

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.

#### J. 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.

#### K. 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.

## XI. ALBERT MOE-13: AUTONOMOUS TRAINING ON CONSUMER CPU HARDWARE

Albert MoE-13 is the reference implementation of the ideas developed in Sections V–X: 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 12-layer, 58M-parameter multilingual system (v3.0) trained on eight languages across a 177M-token corpus. 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.



## A. Architecture

TABLE VI

ALBERT MOE-13 ARCHITECTURE: V2.0.0 BASELINE AND V3.0 CURRENT STATE

Parameter	v2.0.0
Hidden size	256
Experts per layer	12
Active experts (Top- $k$ )	3
Expert skip rate (@sparseskip)	75%
Depth	5L
Context length	128 tokens
Total parameters	$\approx 22\text{M}$
Vocabulary	8,000 (EN)
Corpus tokens	—
Weight format	ternary
Inference throughput (CPU only)	<b>79.5–92.2 tok/s</b>
Perplexity (WikiText-2, held-out)	<b>2026.1</b>
Training hardware	HP ZBook 2013

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 V. 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.

### B. 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 12 layers as of Global Epoch 62:

TABLE VII  
ALBERT MOE-13 — AUTONOMOUS SURGERY HISTORY

Event	Architecture	Notes
Initialisation	$3L \cdot 8k \text{ vocab} \cdot \approx 22\text{M} \text{ params}$	v2.0.0 baseline
Surgery 1	$3L \rightarrow 5L \cdot 8k \text{ vocab}$	2026-05-06; checkpoint 91 MB
Vocab surgery	$5L \cdot 8k \rightarrow 32k \text{ vocab}$	ByteLevel BPE; 8 languages
Surgeries 2– $n$	$5L \rightarrow 12L \cdot 32k \text{ vocab}$	Sequential; $\approx 58\text{M} \text{ params}$

Each surgery preserves all prior weight state via Net2Net safe-copy initialisation [5]. The identity-preserving property ensures 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\_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:

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

The v2.0.0 threshold of 10.2 was calibrated for  $|V| = 8,000$  ( $\ln(8000) \approx 8.99$ , margin  $\approx 1.2$  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  $\approx 10.35$  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.

### C. 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.

a) *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:

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

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

The surgery cost  $C_{\text{surgery}}$  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.

*b) The intervention calculus.:* The surgery decision is therefore not a question of risk tolerance but of comparative cost. Denoting by  $\Delta_{\text{post}}$  the expected loss improvement unlocked by depth expansion:

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

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.

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

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

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.

*d) 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 Evolution-Manager 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.

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

The combined effect on useful optimiser throughput is approximately  $50\times$ .

#### E. 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.

#### F. 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:

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

ByteLevel BPE [6] 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 177,654,147 tokens distributed across 12 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:

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

Albert v3.0 currently operates at loss  $\approx 10.35$ —below the random baseline—confirming that multilingual structure is being extracted even in the early epochs of vocabulary transfer.

### G. 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  $\{G, O, R\}$ :

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

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:

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

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 12 layers.

**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 (§XI-G), 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

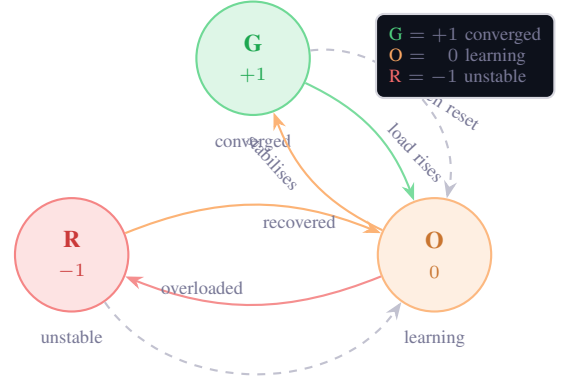


Fig. 1. TLIGHT per-expert state machine. Each expert independently transitions between  $G$  (+1, stable routing),  $O$  (0, actively learning), and  $R$  (−1, overloaded or unstable). Dashed arcs show the epoch-boundary reset driven by the Routing Lottery: all experts return to  $O$  at epoch start.

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 ( $7 \times \text{burst\_count} \bmod N$ , coprime to  $N = 12$ , ensuring full expert coverage over successive bursts). Burst state is reinforced by direct EMA imprinting: Green experts receive  $\text{EMA} \leftarrow 0.4 \times \text{GREEN\_THRESH}$  and Red experts  $\text{EMA} \leftarrow 1.6 \times \text{RED\_THRESH}$ , extending differentiation  $\approx 20$  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  $5 \times$  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.

### H. 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

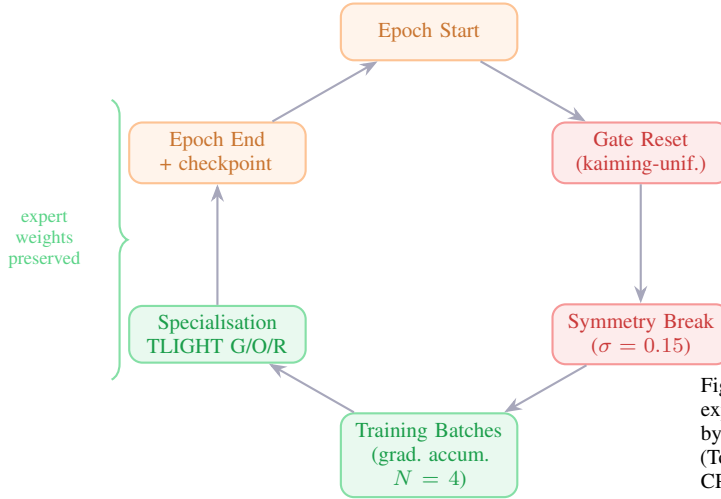


Fig. 2. Epoch-Gated Routing Lottery lifecycle. All gate weights are reset to kaiming-uniform at every epoch boundary; Gaussian noise ( $\sigma = 0.15$ ) is applied to all 576 expert tensors to prevent dead-expert inertia. Expert weights are preserved throughout—only the routing map is redrawn. Gradient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

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.

### I. @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 (§V) realised at the neural routing level:

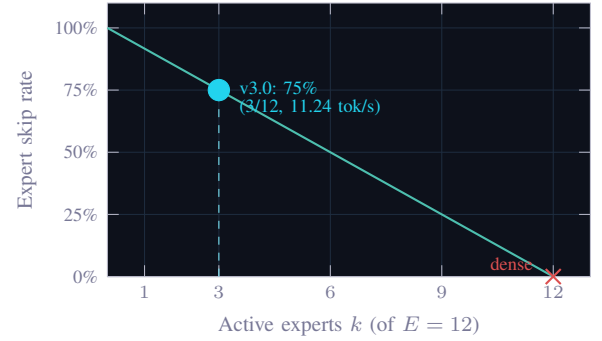


Fig. 3. @sparseskip expert skip rate  $= 1 - k/E$  as a function of active experts  $k$  for  $E = 12$ . The curve is exact—no approximation is introduced by skipping, since  $\text{mul}(a, 0) = 0 \forall a \in T$  (Eq. 3). The v3.0 operating point (Top-3, cyan marker) achieves 75% skip at 11.24 tok/s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

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

TABLE VIII  
@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	<b>75%</b>	<b>75%</b>
Inference throughput	<b>92.2 tok/s</b>	<b>79.5 tok/s</b>
Latency	10.8 ms/tok	12.6 ms/tok
Perplexity (WikiText-2)	<b>2026.2</b>	<b>2026.1</b>
Context length	128 tokens	128 tokens
GPU	None	None

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 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.

### J. Gradient Accumulation as Ternary Hold

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

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

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:

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

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.

## K. Training Results

TABLE IX  
ALBERT MOE-13 v2.0.0 — 16-EPOCH TRAINING SUMMARY (ENGLISH CORPUS)

Metric	Value
Global epochs completed	16
Total batches logged	1384
Epoch 1 average loss	8.284
Epoch 15 average loss	8.521
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 · <b>no GPU</b>

The v2.0.0 all-time best of 2.1353 (Epoch 9, single batch, perplexity  $\approx 8.5$ ) 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.

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

### L. Experimental Training Methodology: A Phase-by-Phase Account

1) *Phase 1 — Hardware Transition and GPU Migration:* Training of `albert.v3.0` began on a 2013 HP ZBook (Intel

TABLE X  
ALBERT MOE-13 v3.0 — CURRENT STATE (MULTILINGUAL, 32K VOCAB, AS OF 2026-05-12)

Metric	
Architecture	12L · 256H · 12E · Top-3
Vocabulary	32,000 tokens (ByteLevel BPE, 8)
Corpus	177,654,147 tokens (chaos)
Total parameters	$\approx 58$
Global epochs at v3.0 launch	
Global epochs completed	143+ (Modal.com)
Best epoch-average loss	<b>10.326</b>
Best single-batch loss	<b>10.3023</b>
Random baseline $\ln(32000)$	
Sustained loss rate (Ep83–143)	$-0.001$
Gradient norm unblocked (Ep111)	$7 \times 10^{-6}$
Expert skip rate	<b>75%</b> (Top-3 of 12, con)
Inference throughput (CPU, @sparseskip)	<b>83 tok/s</b> (5L demo) · <b>11–22</b>
Training hardware	Modal.com T4 GPU ( $\approx \$0$ )
CPU validation hardware	HP ZBook 2013 · Intel i7-4800MQ
Net2Net surgery gate (next milestone)	loss $\leq 9.8$

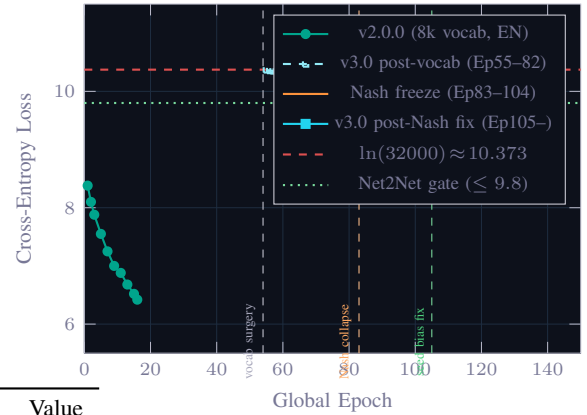


Fig. 4. Full loss trajectory for Albert MoE-13 across v2.0.0 and v3.0. Three distinct phases are visible: (1) post-vocabulary-surgery plateau near  $\ln(32000)$  (Ep55–82); (2) Nash equilibrium freeze at Ep83–104 where expert weights became identical and gradient flow through expert MLPs ceased; (3) post-Nash descent from Ep105 onward after introduction of expert skip biases. The dotted green line marks the Net2Net surgery gate (loss  $\leq 9.8$ ). Best epoch-average: 10.3262 (Ep115); best single batch: 10.3023 (Ep139).

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  $\approx 30$  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  $\approx 400$  ms—compared to 2–4 seconds per batch on CPU—yielding an epoch duration of  $\approx 2$  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 en-

tirely 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.

#### 2) 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`  $\geq 5$ . Gradient norms for expert MLP parameters dropped from  $7 \times 10^{-5}$  to  $7 \times 10^{-6}$ —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  $\mathcal{L}_{LB}$  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.

3) 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:

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

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  $7 \times 10^{-6}$  to  $7 \times 10^{-5}$  (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:  $2.445 \rightarrow 2.445 \rightarrow 2.476$ , 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  $\leq 9.8$ .

4) 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`  $\geq 5$ ), 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.

5) *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 ( $\text{loss\_avg} \leq 9.8$ , requiring a further  $\approx 0.53$  nats improvement from the Ep143 level of 10.334) is approximately 500 epochs away. Acceleration of this trajectory—via diversity loss reactivation, entropy-floor gate regularisation, or learning rate warm-up after surgery—is the primary engineering target for the post-Stage-1 programme.

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

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

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.

#### M. Inference: Hardware-Portable @sparseskip Throughput

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

TABLE XI  
ALBERT MOE-13 @SPARSESKIP INFERENCE — CPU HARDWARE VALIDATION

Metric	
Hardware	HP ZBook 2013 · Intel i7-4800MQ · 7.1 GB RAM
Model version at measurement	v2.0.0 · 5L · 256H · 128CTK · 8K program
Sequence length	128 tokens
Batch size	128
Expert skip rate	75%
Inference throughput	83.4 tok/s
Per-token latency	12.0 ms
Total efficiency (TIS compute chain)	152.8× over dense F22 baseline
Current 12L model (CPU)	11–22 tok/s (heavier depth, same skip rate)

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.

#### N. 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.

## XII. QUTRIT NEURAL NETWORKS AND TERNLANG

The Qutrit Neural Network (QNN) framework [8] 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:

$$|-\rangle \leftrightarrow -1, \quad |0\rangle \leftrightarrow 0, \quad |+\rangle \leftrightarrow +1 \quad (29)$$

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.

#### A. 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 X-I): when affirmation and conflict signals are balanced, the system stabilises at trit 0 rather than forcing a premature commitment.

#### B. 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-products
- Qutrit8 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 $\times$  speedup over dense computation observed in classical BitNet workloads.

### XIII. RELATED WORK

**Balanced ternary computing.** Knuth [3] provides the mathematical foundation. The Setun computer (1958) [9] 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 [10]) 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.

### XIV. THE TERNARY COMPUTING ECOSYSTEM

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

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

TABLE XII  
ECOSYSTEM COMPATIBILITY MAP

Project	Interface	Bridge
Brandon Smith 9-trit sim	<code>.tasm</code> assembly	<code>TasmAssembler</code> $\rightarrow$ BET bytecode
Owlet (S-expr)	S-expression syntax	<code>OwletParser</code> $\rightarrow$ 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	<code>trittensor</code> state model (future)

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.

### XV. IMPLEMENTATION STATUS

Ternlang is implemented in Rust and comprises the following crates:

TABLE XIII  
WORKSPACE CRATES, TEST COUNTS, AND DESCRIPTIONS (v2.0.0)

Crate	Tests	Description
<code>ternlang-core</code>	43	Lexer, parser, AST, semantic checker, BET VM (51 opcodes)
<code>ternlang-ml</code>	21	BitNet quantization, sparse/CSC matmul, AVX2 SIMD kernels, deliberation engine, coalition vote, hallucination score
<code>ternlang-moe</code>	24	MoE-13 orchestrator: dual-key routing, triad synthesis, 3-tier memory, 13 dual-signal agents, introspective hold
<code>ternlang-codegen8</code>	—	AST-to-C11 transpiler; match/struct/propagation
<code>ternlang-test</code>	10	Full-pipeline test framework
<code>ternlang-hdl</code>	34	Verilog-2001 codegen, BET RTL simulator
<code>ternlang-runtime4</code>	—	Distributed TCP actor runtime ( <code>TernNode</code> )
<code>ternlang-compat</code>	29	<code>.tasm</code> assembler (9-trit RISC), Owlet S-expr parser
<code>ternpkg</code>	5	Package manager, GitHub-backed registry
<code>ternlang-lsp</code>	—	LSP 3.17: hover, completion (19 snippets), diagnostics
<code>ternlang-mcp</code>	—	MCP server, <b>34 free tools</b> , live on Smithery
<code>ternlang-api</code>	—	REST API (18 endpoints), KPI pipeline, SSE streaming
<code>albert-llb-mcp</code>	—	LLB safety gate: deterministic ternary verdict, stateless
<code>albert-cli</code>	—	Full Rust TUI agent CLI ( <code>ratatui</code> ); multi-provider; voice STT
<code>exatern</code>	—	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 (uncertainty) 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](https://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.

## XVI. 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  $2.27\times$  reduction in multiply operations at baseline sparsity; a CSC kernel with Rayon reaches  $122\times$  at high sparsity (512<sup>2</sup> 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  $\geq 4$  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  $\approx 8.5$ , 16 global epochs) to a 58M-parameter 12-layer multilingual system (v3.0, 8 languages, 32k vocab, 177M-token corpus, 62+ global epochs as of 2026-05-11) through successive autonomous surgeries triggered by the EvolutionManager—all running on a 2013 consumer CPU with no GPU. The v3.0 programme introduces the Epoch-Gated Routing Lottery, TLIGHT per-expert monitoring, and gradient accumulation as ternary hold, with live inference confirming **75%** expert skip at **11.24 tok/s** on the training hardware. Hardware-verified AVX2 SIMD benchmarks confirm  $1.80\times$  throughput at 50% sparsity and  $3.0\times$  over INT8 at 90% sparsity.

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 (deadline 2026-05-15) documents the full TIS architecture for European public funding consideration. 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 [8] 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, Owllet 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] 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>
- [7] S. Merity, C. Xiong, J. Bradbury, and R. Socher, “Pointer sentinel mixture models,” 2016, introduces the WikiText-2 and WikiText-103 language modeling benchmarks, now standard held-out evaluation corpora for LMs. [Online]. Available: <https://arxiv.org/abs/1609.07843>
- [8] 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.

- [9] 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>
- [10] S. Ma, H. Wang, L. Ma *et al.*, “BitNet b1.58: Large language models in the ternary regime,” *arXiv preprint*, 2024.