Deployment-Native Ternary Learning

Reproducible Training, Executable Model Streams,
and Continuously Curated Synthetic Corpora

Flyxion
Independent Researcher

2026

Abstract

This monograph develops a systems architecture for training compact ternary neural models under the same numerical and operational constraints used at inference time. It combines five ideas: quantization-aware learning over the alphabet $\{-1, 0, +1\}$; a versioned model capsule that binds weights, bytecode, tokenizer state, scales, and runtime requirements; message channels that transport executable model descriptions rather than passive records; reproducible, customer-controlled deployment through declarative environments; and a continuously operating data loop that produces silver-label question–answer examples under explicit validation and provenance rules. The central claim is limited but consequential: when the deployed forward operator is treated as part of the training specification, representation drift between optimization and execution becomes measurable and reducible. The resulting system is not automatically accurate, secure, private, or efficient. Those properties require separate tests. A reference architecture, formal model, failure analysis, security boundary, and experimental program are provided so that the proposal can be evaluated without relying on metaphor or promotional terminology.

Scope and status of claims

Established results on ternary-weight transformers, quantization-aware training, optimized low-bit inference, declarative software environments, and synthetic-data curation motivate the design. The integration proposed here is an engineering hypothesis. In particular, transporting a model capsule through a channel does not itself create a new computational primitive; its value must come from constrained representation, verifiable execution, isolation, and composability. Likewise, a self-generated corpus is useful only to the extent that its validation process predicts performance on independently collected data.

1 The deployment-gap problem

A conventional model pipeline often contains several non-identical machines. Optimization uses floating-point shadow weights and training kernels; export applies quantization and layout conversion; inference invokes a separate runtime; and deployment adds platform-specific tokenization, scaling, memory, and scheduling behavior. Every transformation can preserve average benchmark performance while changing individual outputs, numerical stability, latency, or failure behavior. The relevant object is therefore not merely a parameter tensor θ, but the complete deployed computation.

$$F = (G, Q, S, T, K, R, P)$$

Here G is the graph or bytecode, Q the weight alphabet and packing rule, S the scale metadata, T the tokenizer and preprocessing state, K the kernel semantics, R the runtime contract, and P the execution policy. Two artifacts with identical ternary weights can still implement different functions when any other component differs.

Deployment-native training does not require every training operation to be low precision. Gradient accumulation and optimizer state may remain floating point. It requires that the forward operator used to estimate the task loss faithfully reproduce the discrete weights, scales, rounding, saturation, activation treatment, and layer ordering that will be used after export.

1.1 Where drift accumulates

It is useful to separate deployment drift into components that arise at distinct stages of the pipeline, since each admits a different remedy. *Representation drift* arises when the forward operator used for loss estimation differs from the exported operator, most commonly through a mismatched straight-through surrogate or an export-time rounding rule not exercised in training. *Kernel drift* arises when the mathematically specified operator is realized by different low-level implementations across devices, for example a fused kernel that reorders accumulation and changes rounding under finite precision. *Preprocessing drift* arises when the tokenizer, normalization, or templating logic bound to the training data differs from the version bound to the runtime capsule, which can change token boundaries without changing any weight. *Scheduling drift* arises when batching, padding, or caching strategies at inference time alter numerical results relative to the unbatched or differently batched computation used during evaluation. These four sources are not mutually exclusive and can partially cancel or compound; a pipeline that reports only aggregate deployment disagreement without attributing it to a source cannot say which remedy would help, which is why Section 10 treats attribution as a required part of the evaluation record rather than an optional diagnostic.

2 Ternary parameterization

Let a layer maintain real-valued shadow parameters $W \in \mathbb{R}^{m \times n}$ while executing a ternary forward matrix $\hat{W}$. A simple symmetric quantizer uses a scale $\alpha > 0$ and threshold $\Delta \geq 0$:

$$\hat{W}_{ij} = \alpha \cdot q(W_{ij}/\alpha), \qquad q(x) \in \{-1, 0, +1\}$$

$$q(x) = \begin{cases} -1 & x < -\Delta \\ 0 & |x| \leq \Delta \\ +1 & x > \Delta \end{cases}$$

The ternary symbol requires $\log_2 3 \approx 1.585$ bits of information under a uniform code, although practical packing may use two physical bits per weight or block encodings with amortized overhead. Calling the representation strictly one-bit is therefore imprecise. Its computational appeal is that multiplication by a ternary weight reduces to signed accumulation and omission, subject to the cost of scales, activation arithmetic, packing, memory movement, and kernel dispatch.

Training commonly uses a straight-through approximation because q is piecewise constant. If L is the loss, the backward pass substitutes a surrogate derivative for $\partial q / \partial W$. This makes optimization possible but means the learned solution depends on the surrogate, clipping rule, scale

estimator, and update schedule. Reproducibility consequently requires recording these rules, not only the final packed weights.

2.1 Per-channel and asymmetric variants

The symmetric, per-tensor quantizer above is the simplest member of a larger family, and the family member chosen is itself part of the deployment specification F rather than an incidental training detail. A per-channel quantizer assigns an independent scale α_k to each output channel k, which typically reduces quantization error on layers whose channels have heterogeneous weight magnitude at the cost of storing and transporting a vector of scales instead of a scalar; the capsule manifest of Section 3 must then record the reduction axis and count, since a runtime that assumes a single scalar scale will silently misinterpret a per-channel capsule rather than failing loudly. An asymmetric variant replaces the zero-centered threshold with an independent offset β, applying the same symbol map $q(x) \in \{-1, 0, +1\}$ to a shifted argument, $\hat{W}_{ij} = \alpha \cdot q((W_{ij} - \beta)/\alpha) + \beta$. The quantization symbol $q(\cdot)$ is still drawn from the ternary alphabet, but the reconstructed weight $\hat{W}_{ij}$ now takes values in $\{\beta - \alpha,\ \beta,\ \beta + \alpha\}$, which is a three-level operator rather than a zero-centered ternary weight in the sense used elsewhere in this monograph. The distinction matters for the efficiency argument of Section 2: signed accumulation over $\{-1, 0, +1\}$ eliminates multiplication only when the reconstruction levels themselves are $\{-1, 0, +1\}$, which holds for the symmetric quantizer but not for the shifted one, since a nonzero β reintroduces an additive term into the accumulation step. A deployment that adopts the asymmetric variant should therefore report it as a distinct three-level operator with its own efficiency profile, and reserve the term "ternary weight" for the zero-centered, additive-free case. A third variant fixes Δ as a function of a calibration set rather than a free training parameter, trading adaptivity for a scale rule that can be audited independently of the optimizer trajectory. None of these variants dominates the others across all objectives; the falsifiability requirement of Section 11 applies separately to each variant a given deployment claims to support, since a result obtained under one quantizer family does not transfer by default to another.

2.2 Deployment equivalence

Let $f_{\text{train}}(x; \theta)$ denote the forward pass used while training and $f_{\text{run}}(x; C)$ the runtime evaluation of a serialized capsule C. For a test distribution D, define deployment disagreement as:

$$\delta_{\text{out}} = \Pr_{x \sim D} \left[\text{argmax}\, f_{\text{train}}(x; \theta) \neq \text{argmax}\, f_{\text{run}}(x; C) \right]$$

$$\delta_{\text{num}} = \mathbb{E}_{x \sim D} \left[\frac{\| f_{\text{train}}(x; \theta) - f_{\text{run}}(x; C) \|_2}{\| f_{\text{train}}(x; \theta) \|_2 + \varepsilon} \right]$$

Exact bitwise agreement may be unrealistic across heterogeneous devices, but token-level disagreement, normalized logit error, and task-level divergence are measurable. A pipeline should state which equivalence criterion it promises and on which platforms it has been tested.

3 The model capsule

The deployable unit should be a self-describing capsule rather than an unstructured weight file. The capsule binds executable semantics to model state and supports hashing, signature verification, compatibility checks, and deterministic reconstruction.

Component	Required contents	Reason
Manifest	format version, model identity, hashes, license, provenance	Defines the object and prevents ambiguous loading.
Program	typed bytecode or restricted graph	Specifies computation independently of host-language objects.
Parameters	packed ternary weights and scale blocks	Preserves the executed representation.
Text interface	tokenizer, normalization, vocabulary, templates	Avoids silent input mismatch.
Runtime contract	operators, numerical rules, memory limits, ABI	Makes compatibility testable.
Evidence	evaluation summary, calibration set hash, build attestation	Records what has actually been measured.

A capsule should not contain arbitrary native code. A restricted instruction set with bounded tensor shapes, declared memory, immutable parameter regions, and explicit operator versions is easier to validate and isolate. The signature authenticates the bytes and declared producer; it does not establish model quality or safety.

3.1 Versioning and compatibility semantics

A capsule format that changes without a disciplined versioning rule reintroduces the deployment gap it was built to close, since a runtime and a capsule produced under different silent assumptions can both parse successfully while disagreeing on semantics. A workable convention assigns three independent version fields to the manifest: a *format* version governing the container schema itself, a *contract* version governing the runtime operators, numerical rules, and ABI, and a *content* version identifying the specific trained artifact. A conservative runtime may require exact format-version equality, since a parser is not obligated to accept fields it does not recognize. A runtime claiming backward compatibility instead must specify which fields are optional, which unknown fields are ignorable, and which are critical; unknown critical fields require refusal. Contract versions should be compared under a declared compatibility relation, for example a runtime advertising contract version v accepts any capsule declaring a contract version in a closed range $[v_{\min}, v]$ that the runtime publishes, so that older capsules remain executable under runtimes that have only added operators and newer capsules are refused rather than silently misexecuted. Content versions carry no compatibility semantics of their own; they exist so that two capsules with identical format and contract versions can still be distinguished, hashed, and compared for provenance. This three-field scheme does not eliminate the need for the runtime version admissibility check in Section 4.1, but it gives that check a principled basis: refusal is warranted whenever the contract-version range test fails, and acceptance is not warranted merely because the capsule parses.

4 Channels as streams of executable models

Typed channels normally transport values between concurrent processes. If a channel value is a model capsule or a content-addressed reference to one, the receiving process can validate it,

allocate a fresh arena, bind inputs, and execute the declared graph. The channel then represents a stream of parameterized computations. This is best understood as mobile model description, not mobile authority: receipt of a capsule must not grant filesystem, network, process, or device access.

$$\text{send}(C) \to \text{receive}(C) \to \text{verify}(C) \to \text{allocate}(A) \to \text{execute}(C, A, x) \to y$$

Fresh arenas provide useful isolation between invocations. They eliminate accidental reuse of mutable intermediate state unless state is explicitly declared and transferred. They do not by themselves prevent denial of service, side channels, malformed-shape attacks, or vulnerabilities in the verifier and kernels.

4.1 Operational semantics

Let a receiver state be $\sigma = (A, M, V)$, where A is the available arena, M is the admitted capsule set, and V is the runtime version. The transition $\text{Exec}(C, x)$ is admissible only if the capsule signature and hashes verify, its declared operators are supported by V, its maximum memory and instruction budget fit A, and its input contract accepts x. Execution occurs in a new arena A' and returns either a typed result y with an execution record e or a typed refusal r.

$$(\sigma, C, x) \to (\sigma, y, e) \quad \text{or} \quad (\sigma, C, x) \to (\sigma, r)$$

Refusal is part of the protocol, not an exceptional accident. The execution record should include capsule hash, runtime hash, device class, input hash or privacy-preserving reference, start and end times, resource counters, and result status. Sensitive inputs and outputs need not be placed in the record.

4.2 Composability and capsule chaining

A single capsule executing a single graph is the base case of a more general pattern in which a receiver composes several verified capsules into a pipeline, for example a small retrieval-scoring capsule feeding a generation capsule under a shared input contract. Composition should be treated as a distinct admissibility question rather than an automatic consequence of each capsule individually verifying, because the output contract of one capsule and the input contract of the next may agree in type while disagreeing in the numerical assumptions carried in S, T, or K; a scale or tokenizer mismatch that neither capsule can detect on its own becomes detectable only at the interface. A composed transition $\text{Exec}(C_1, x) \to y_1$, $\text{Exec}(C_2, y_1) \to y_2$ is therefore admissible only if, in addition to each capsule's individual admissibility, the declared output contract of C_1 is accepted by the declared input contract of C_2, and the composed execution record links both capsule hashes and both individual records so that a downstream failure can be attributed to a specific stage rather than to the pipeline as an undifferentiated whole. This requirement is stricter than simple type compatibility and is the chaining analogue of the deployment-equivalence criterion in Section 2.2: composability without a stated equivalence criterion at the interface is not composability, only successful parsing.

4.3 Bounded stateful context kernels

A capsule that processes a message stream need not treat every invocation as history-free. The relevant addition is a small, explicit state machine bound into the capsule contract: a finite context

kernel with window length n, whose state at time t is the ordered tuple of the n most recently admitted inputs.

$$C_t = (x_{t-n+1}, \ldots, x_t), \qquad C_{t+1} = \text{shift}(C_t, x_{t+1})$$

A kernel of this kind is fully specified by the tuple $K = (X, S, U, R, s_0)$, where X is the input alphabet, S the finite state space, $U : S \times X \to S$ the update rule, $R : S \to Y$ the readout function, and s_0 the initialization state. Declaring K as part of the capsule contract lets a receiver reproduce or migrate context by transporting the state alongside the capsule, rather than relying on hidden process memory that the arena model of Section 4.1 does not otherwise account for.

Two readouts are useful and worth distinguishing precisely, since both are easy to describe loosely in ways that overstate what they compute. A window entry should not be assumed to encode a single trit merely because the alphabet is ternary; a packed implementation may store several balanced-ternary digits per cell, in which case the kernel should be documented as operating over a window of ternary cells or packed ternary tokens, and the packing width becomes part of X.

The first readout is a consensus operator over the window. A natural but non-obvious design choice makes indeterminacy absorbing: the operator returns $+1$ only when every cell in the window agrees positively, 0 when the window is empty or its cells disagree, and -1 when any cell is itself indeterminate, regardless of the value of the remaining cells. This is a stricter rule than ordinary three-valued conjunction. Under strong Kleene logic, false conjoined with unknown evaluates to false; here, an indeterminate cell propagates through the whole window regardless of what else is present. Table 2 states the rule explicitly so that it is not mistaken for a standard three-valued AND.

Table 2: Truth table for the strict-consensus (uncertainty-preserving conjunction) operator over a two-cell window, extended cell-by-cell to width n.

a	b	$\text{cand}(a, b)$
$+1$	$+1$	$+1$
$+1$	0	0
0	0	0
$+1$	-1	-1
0	-1	-1
-1	-1	-1

A rule of this shape is legitimate, but it should be named and specified as strict consensus or uncertainty-preserving conjunction, not presented as ternary AND without qualification, since a reader who assumes standard three-valued logic will draw the wrong conclusion about what a 0 result means. Table 2 is stated for a single scalar trit, and should be labelled as such: it is the special case of the operator that applies when each window entry carries exactly one trit.

When cells are packed, the general form is componentwise. Let $d(x)_j$ denote the j-th packed trit decoded from cell x. The consensus operator is then defined per component,

$$d(\text{cand}(a, b))_j = \begin{cases} -1, & d(a)_j = -1 \text{ or } d(b)_j = -1, \\ +1, & d(a)_j = d(b)_j = +1, \\ 0, & \text{otherwise}, \end{cases}$$

and the window readout is the componentwise fold of this operation across the occupied cells, returning a packed mixed cell rather than a single scalar in $\{-1, 0, +1\}$ whenever the window's constituent trits disagree across components. The scalar truth table of Table 2 recovers this definition exactly when each cell packs one trit; documenting the operator only at the scalar level would misdescribe its behavior on any window of multi-trit cells.

The second readout is an evidence accumulator, intended as a signed score over the window. Here a further precision is needed: summing the integer encodings of packed cells is not in general equivalent to summing their constituent trits, since balanced-ternary place values and carries can conflate distinct componentwise evidence patterns, and clamping the encoded sum can discard magnitude that the underlying trits still carry. The scientifically clean operator decodes first and accumulates per component,

$$s_j(C) = \sum_{i=1}^{m} d(x_i)_j,$$

and then either returns the vector $s(C)$ directly, applies componentwise clipping to it, or defines an explicit projection $\rho(s)$ down to a scalar with a stated rule. If an implementation instead sums the encoded integer values of the cells, that computation should be reported as an implementation-specific scalar score, not as a general componentwise evidence accumulator, since the two are equivalent only under packing schemes without cross-component carry.

Either form of the accumulator should be described as an evidence score, not as a pruner. It neither identifies an inconsistent token nor removes one; it aggregates. A readout that actually prunes would require an explicit per-element marginal-contribution criterion, for example

$$\iota_i = |\, \rho(s(C)) - \rho(s(C \setminus \{x_i\}))\, |,$$

together with a declared reference rule for what counts as inconsistent, since observing that removing a token changes the aggregate score does not by itself establish that the token was wrong.

These two readouts, together with insertion and circulation through the window and quantization performed before admission, are the mechanisms the kernel actually implements. They should not be described as instances of four independent operations named rewriting, pruning, consensus, and scoring unless a rewriting stage and an independent pruning stage are separately implemented; where they are not, the monograph should retain the two implemented mechanisms and drop the unimplemented role names rather than let the vocabulary imply a richer operator set than the kernel provides.

A further precision is required at initialization. A newly allocated n-cell window that is zero-filled makes an unwritten slot indistinguishable from a slot deliberately written with the zero token, which silently corrupts both readouts before the window has been filled once. The state should therefore carry an explicit occupancy count $m \leq n$ or a validity mask alongside C_t, and both the consensus operator and the evidence accumulator should be defined over the occupied subset of the window rather than over the full padded tuple.

Finally, a stateful kernel qualifies the fresh-arena isolation model of Section 4.1 rather than replacing it. If every received capsule executes in a freshly allocated arena, a kernel's context does not survive between invocations unless a policy says otherwise. Three explicit policies are available: reset the context on every invocation, serialize the context into the outgoing message so the next invocation carries it forward, or maintain a receiver-owned state object keyed by capsule identity outside the per-invocation arena. Which policy applies is a property of the deployment, not a default the runtime should assume, and it belongs in the capsule manifest alongside the other

contract fields of Section 3, so that context handling remains reproducible and so that state cannot leak between unrelated capsules or tenants under the isolation guarantees claimed in Section 9.

5 Continuously curated silver-label corpora

A continuously running data loop can convert system use, public source material, or synthetic prompts into candidate question–answer pairs. These pairs are silver labels: they have passed automated checks but not necessarily expert human review. The loop should be treated as a measurement and curation system, not as an autonomous source of truth.

$$S_t \to \text{propose}(q, a, p) \to \text{validate}(q, a, p) \to \text{deduplicate} \to \text{stratify} \to \text{publish } D_{t+1}$$

The provenance record p should identify source classes, generators, validators, prompt and model versions, transformation rules, timestamps, licenses, and hashes. Validation may combine schema checks, executable tests, retrieval-supported entailment, agreement among independent models, contradiction detection, and sampled human review. Generator–validator independence matters: a model grading its own output can reproduce its own errors with high confidence.

5.1 Feedback hazards

Repeated training on self-generated text can narrow linguistic and conceptual diversity, amplify systematic errors, leak evaluation material, and reward artifacts of the validator. A robust loop therefore keeps an immutable external evaluation set, retains real or independently sourced data, measures duplication and source concentration, records ancestry between examples, and limits the number of generations through which unsupported claims can propagate.

$$D_{t+1} = R_t \cup H_t \cup \{z \in G_t : V_t(z) = \text{accept}\}$$

R_t denotes independently sourced material, H_t human-reviewed examples, G_t generated candidates, and V_t the versioned validator. The decomposition makes it possible to report performance by data origin rather than obscuring the synthetic fraction in a single aggregate.

6 Public and private training as one protocol

The scientifically defensible version of an open-versus-private service uses the same training protocol, acceptance tests, and reporting schema in both modes. The variable being purchased is isolation and operational control, not a higher standard of evidence. Public runs may publish data, configuration, metrics, and artifacts. Private runs keep customer data and outputs inside a customer-controlled enclave or cloud account while emitting only the attestations and reports the customer authorizes.

Property	**Public run**	**Private run**
Compute owner	Operator or sponsor	Customer or isolated tenant
Dataset visibility	Publishable by default	Customer-controlled
Artifact visibility	Open unless restricted by license	Customer-controlled

Property	Public run	Private run
Protocol	Versioned and disclosed	Same versioned protocol
Acceptance criteria	Declared before training	Identical unless explicitly amended
Audit record	Public build and evaluation record	Private record under customer custody

Bring-your-own-cloud deployment reduces custody by allowing the customer to own accounts, encryption keys, logs, storage, and teardown. It does not eliminate trust: the customer must still assess the supplied code, binary provenance, infrastructure declarations, dependencies, and operator access paths.

7 Declarative and reproducible delivery

A declarative environment can specify compiler, libraries, runtime, scripts, model capsule, evaluation suite, and infrastructure adapters as a versioned dependency graph. Nix flakes are one implementation option; containers and locked language environments can complement them. Reproducibility should be stated in layers: source reproducibility, dependency closure, build reproducibility, environment recreation, deterministic data preparation, deterministic training where feasible, and statistical replication where bitwise identity is impossible.

A signed binary is evidence that particular bytes were endorsed by a signing key. A reproducible build connects those bytes to source and declared inputs. A software bill of materials records dependencies. None of these proves absence of vulnerabilities, data exfiltration, or hidden behavior. Strong delivery combines all three with isolated execution, minimal privileges, auditable network policy, and customer-verifiable teardown.

8 End-to-end reference architecture

The proposed system separates control, data, build, training, evaluation, and execution planes. The control plane admits jobs and records immutable specifications. The data plane stores versioned datasets and provenance graphs. The build plane constructs and signs the environment and capsule. The training plane optimizes shadow parameters while executing the deployment-faithful forward operator. The evaluation plane compares training and runtime behavior on held-out material. The execution plane receives verified capsules and runs them in fresh bounded arenas.

A minimal job specification contains a dataset manifest, data-policy declaration, model graph, quantizer definition, optimizer and schedule, random seeds, runtime target, resource budget, acceptance tests, disclosure policy, and output destinations. The system should refuse an underspecified job rather than silently filling consequential fields with mutable defaults.

9 Security and privacy model

The relevant adversaries include a malicious capsule producer, a compromised build dependency, an operator with excessive cloud privileges, a customer dataset containing secrets or prompt injections, and an external party observing timing or resource use. The trusted computing base should

therefore be as small as practical: parser, verifier, allocator, operator kernels, signature implementation, and the mechanism enforcing resource and I/O policy.

Threat	Control	Residual risk
Malformed capsule	Canonical parser, shape validation, fuzzing	Verifier or kernel vulnerability
Arbitrary code execution	Restricted bytecode; no native extensions	Bugs in permitted operators
Resource exhaustion	Static bounds plus runtime quotas	Adversarial worst-case scheduling
Training-data leakage	Access controls, minimization, memorization tests	Inference-time extraction remains possible
Supply-chain substitution	Pinned closure, hashes, signatures, reproducible builds	Compromised trusted source or key
Cross-job contamination	Fresh arenas and tenant isolation	Hardware and timing side channels

Privacy is a property of the complete workflow, not a synonym for running in a private account. It requires an explicit data-retention policy, logging policy, key ownership model, access review, egress controls, incident procedure, and deletion verification. Differential privacy or confidential-computing hardware may be added when their assumptions match the threat model, but neither should be implied merely by the word enclave.

9.1 Trusted computing base and verification cost

Shrinking the trusted computing base is only useful to the extent that the cost of exercising it stays bounded, so a deployment claim should report the verification overhead alongside the security boundary. Let c_v denote the wall-clock or resource cost of the parse-and-verify step for a capsule and c_x the cost of subsequently executing it once. The relative verification overhead $c_v/(c_v + c_x)$ is large and largely irrelevant for a capsule executed many times after a single verification, but it dominates the cost of a workload that streams many distinct small capsules through a receiver, verifying each once before a single execution; the two regimes should not be reported under one aggregate latency figure, since a system tuned for the first regime can perform poorly in the second without any change to the security boundary itself. A related and separable cost is re-verification triggered by contract-version changes under the compatibility rule of Section 3.1: a runtime upgrade that narrows its accepted contract-version range can force re-verification or outright refusal of a large previously admitted capsule population, which is a operational cost of the versioning discipline and should be planned for rather than discovered at upgrade time.

10 Experimental program

The architecture should be tested through preregistered comparisons rather than demonstrations alone. The primary baseline is a conventional floating-point training and post-training export

pipeline. The deployment-native condition uses the target quantizer and runtime-equivalent forward semantics during training. Both receive matched data, parameter counts, optimization budgets, and evaluation procedures.

Question	Primary measure	Necessary controls
Does native training reduce drift?	δ_{out} and δ_{num}	Same runtime, tokenizer, prompts, and seeds
Does ternary execution improve efficiency?	energy/token, latency, memory, throughput	Same device, batch, context, quality target
Do silver labels improve generalization?	held-out external task performance	Gold-only and unfiltered-synthetic ablations
Are capsules portable?	cross-platform conformance rate	Declared numerical tolerance and device matrix
Does isolation contain jobs?	red-team escape and leakage rate	Instrumented network, storage, and process boundaries

Quality and efficiency must be reported jointly. A speedup measured at a lower accuracy, shorter context, different sampler, or altered batch size is not a controlled comparison. Energy claims require wall-plug or device-level measurement with stated boundaries; arithmetic-operation counts alone do not capture memory traffic and system overhead.

10.1 Ablation sequence

A useful experimental sequence first verifies byte-level capsule stability and operator conformance on small networks. It then measures deployment disagreement for individual layers and complete models. Next it compares ternary-native training with post-training quantization. Only after those controls pass should the study add capsule streaming, multi-tenant execution, and the continuous data loop. This ordering prevents a complex integration from concealing a basic numerical mismatch.

10.2 Statistical methodology

Because δ_{out} and δ_{num} are estimated from a finite evaluation set, a reported reduction in either quantity should be accompanied by an interval estimate and a preregistered minimum detectable effect, not a point estimate alone. For δ_{out}, which is a proportion, a Wilson or Clopper–Pearson interval is preferable to a normal approximation when the disagreement rate is low, since low-rate proportions are exactly where normal intervals misbehave and where a pipeline is most likely to claim near-parity. Repeated measurement across seeds should be reported as a distribution rather than a single best run; a deployment-native condition that wins on a single seed and loses on most others is evidence of variance, not of the mechanism under test. When the same held-out set is reused across many ablations in the sequence of Section 10.1, the effective multiple-comparisons burden should be disclosed, since a family of ablations each tested at nominal significance against

a shared evaluation set inflates the chance of an apparent effect that does not replicate on the immutable external evaluation set required in Section 5.1. Where feasible, a preregistered analysis plan filed before training begins is stronger evidence than a plan constructed after results are observed, particularly for the falsifiable claims enumerated in Section 11.

11 Falsifiable claims

The proposal should be rejected or revised if deployment-native training does not reduce output disagreement relative to a carefully tuned export baseline; if claimed efficiency disappears at matched task quality; if the silver-label loop improves its internal validator while degrading external evaluation; if capsule verification cannot bound resource use; or if private operation depends on undeclared operator access. Negative results would still be informative because they would locate the cost of representation alignment and identify which integration layers provide no measurable benefit.

12 Discussion

The strongest idea in this architecture is not that ternary cells are miniature intelligent agents or that channels become fundamentally new objects. It is that a deployed neural computation can be packaged as a constrained, typed, verifiable program whose discrete representation is present during optimization, evaluation, transport, and execution. This changes the unit of reproducibility from a weight file to a complete computational contract.

The continuous corpus loop supplies a second contribution when it is governed as an evidence pipeline. Generation, validation, ancestry, licensing, deduplication, and external evaluation become explicit stages. Public operation can expose these stages for inspection, while private operation can preserve the same protocol within customer-controlled infrastructure. In that form, openness and privacy are deployment modes of one method rather than different levels of methodological rigor.

The approach is particularly plausible for small, specialized models whose graphs and memory bounds are tractable enough to verify. Whether the same capsule semantics scale efficiently to large general-purpose language models is an empirical question. The immediate research opportunity lies in miniature models that can be streamed, composed, benchmarked, and refused safely.

Conclusion

Deployment-native ternary learning integrates low-bit optimization with reproducible systems engineering. Its scientific content is a set of testable commitments: the inference representation is exercised during training; the deployable object binds program semantics to parameters and preprocessing state; receivers verify and bound executable model descriptions; public and private training use the same declared protocol; synthetic examples carry provenance and are evaluated outside their generating loop; and every claim of equivalence, efficiency, privacy, or quality is attached to a measurement. These commitments are sufficient to motivate a serious research program without metaphysical language or claims that exceed the evidence.

References

[1] Ma, S., Wang, H., Ma, L., Wang, L., Wang, W., Huang, S., Dong, L., Wang, R., Xue, J., and Wei, F. (2024). The Era of 1-bit LLMs: All Large Language Models are in 1.58 Bits. *arXiv:2402.17764*.

[2] Wang, J., Zhou, H., Song, T., Cao, S., Xia, Y., Cao, T., Wei, J., Ma, S., Wang, H., and Wei, F. (2025). Bitnet.cpp: Efficient Edge Inference for Ternary LLMs. *Proceedings of ACL 2025*, 9256–9270.

[3] Wang, H., Ma, S., Dong, L., Huang, S., Wang, H., Ma, L., Yang, F., Wang, R., and Wei, F. (2023). BitNet: Scaling 1-bit Transformers for Large Language Models. *arXiv:2310.11453*.

[4] Guix, X., Fontich, E., Garcia, R., and others (2015). Nix Based Fully Automated Workflows and Ecosystem to Guarantee Scientific Result Reproducibility Across Software Environments and Systems. *Proceedings of the 1st International Workshop on Software Engineering for High Performance Computing in Computational Science and Engineering*.

[5] Shumailov, I., Shumaylov, Z., Zhao, Y., Gal, Y., Papernot, N., and Anderson, R. (2024). AI Models Collapse When Trained on Recursively Generated Data. *Nature* 631, 755–759.

[6] Bai, Y., Kadavath, S., Kundu, S., et al. (2022). Constitutional AI: Harmlessness from AI Feedback. *arXiv:2212.08073*.

[7] Beyer, B., Jones, C., Petoff, J., and Murphy, N. R., eds. (2016). *Site Reliability Engineering: How Google Runs Production Systems*. O'Reilly Media.

[8] The Reproducible Builds Project. *Reproducible Builds: A Set of Software Development Practices That Create an Independently Verifiable Path from Source to Binary Code*. reproducible-builds.org.