Proof of AI: A Consensus Mechanism for Verifiable Compute Contribution

CPU-verifiable useful work on Lux A-Chain with post-quantum omnichain settlement

Antje Worring & Zach Kelling

Zoo Labs Foundation

August 2026

Abstract

We introduce **Proof of AI** (PoAI), the useful-work consensus of the Lux A-Chain. A demand-backed job is admitted before mining; a provider performs genuine AI execution and commits a proof artifact produced alongside it. The public proof-bearing profile requires software only—no trusted execution environment and no GPU attestor—so an ordinary CPU can validate challenged work without repeating the original GPU workload. The recomputable "hash everyone can check" becomes a canonical execution transcript whose activation commitments cannot be forged without answering for the genuine matrix multiplications. We give one primitive realizing four execution-level proofs over a workload tag—Proof-of-Embedding, Proof-of-Inference, Proof-of-Training, and a privacy-preserving Proof-of-Contribution—by committing the computation trace and spot-checking its matmuls with Freivalds' algorithm over the int8 path's exact integer accumulator, which yields a bit-exact check with soundness error $1/p \approx 2^{-61}$ per vector (2^{-122} at k=2) and *zero* false rejections. We bind every proof to its task with a domain-separated, beacon-fresh, measured-model commitment, and prove anti-substitution and anti-replay for it. We position the scheme as Bitcoin's shared, software-only difficulty—the Hamiltonian Market Maker *prices* heterogeneous compute while the execution proof *verifies* it, two orthogonal dials—and give a single profile gate mapping tiers (zkML, M-of-N confidential compute, optimistic-Freivalds, redundancy) to required proof strength, designed so that once the gate is engaged "no valid compute proof" means "no mint." We motivate the construction with a concrete reference-engine attack in which a one-bit guess could satisfy a receipt path, because a signature proves authorship and a plurality proves agreement, but neither proves computation. Optional TEE attestation, multi-party scoring, and redundancy may supplement the public profile; the execution proof is what binds reward to genuine work, and Lux Quasar supplies the post-quantum consensus/finality layer.

Status, stated up front. This paper proves an arithmetic primitive and reports tested reference surfaces: an exact-integer Freivalds verifier, transcript and graph fixtures, matching encodings in Rust and Go, and fail-closed EVM prototypes. These results support the construction but do not constitute a live A-Chain mining path. A complete Zen model does not yet emit the production transcript on its normal serving path, and the A-Chain node does not yet admit the job, verify finalized evidence, enforce data availability and metering, consume the nullifier, and authorize issuance through this system. P3Q settlement is likewise audit- and activation-gated. Section 2 gives the ledger before the construction. The honest summary is: *the primitive and reference fixtures are built and tested; production emission, availability, A-Chain consensus wiring, metering, and P3Q settlement remain integration work.*

The Zoo reading. Zoo Labs Foundation is an application and research layer for decentralized AI and decentralized science. It may submit jobs, fund work, and consume final receipts, but it is not a second Proof-of-AI chain. Section 19 shows the single-authority topology: A-Chain alone admits and finalizes useful work, Z-Chain settles finalized receipt transitions through P3Q, Lux Quasar supplies post-quantum finality, and Zoo applications consume the resulting receipts.

Keywords: consensus, proof of useful work, AI compute verification, Freivalds' algorithm, optimistic verification, decentralized science, omnichain settlement, post-quantum cryptography, verifiable machine learning

1 Introduction

Proof of AI asks a harder question than whether a provider signed a result: did the committed AI computation actually run, under the job and execution policy the network accepted? Useful work cannot secure a monetary system merely because it is expensive. The work must be demand-backed, objectively bound, cheaper to verify than to perform, available for challenge, finalized exactly once, and connected to a consequence authority that fails closed.

The Lux architecture assigns those responsibilities to one canonical path. A requester submits a job to the A-Chain; a miner executes it under a versioned deterministic kernel profile and emits a

canonical transcript; validators check the required execution, availability, graph, metering, and anti-replay obligations; A-Chain consensus alone finalizes the job, consumes its nullifier, and authorizes AI. The finalized transition is then settled through P3Q on Z-Chain and finalized by Lux Quasar. Other chains may receive a receipt and payout, but cannot replace the A-Chain as the mining authority.

1.1 Challenges

1. **Execution**: distinguish genuine computation from a signed guess or copied output.
2. **Determinism**: give heterogeneous miners and CPU validators one exact statement to check.
3. **Cost asymmetry**: keep verification materially cheaper than the original workload.
4. **Completeness**: bind the full accepted graph, inputs, outputs, artifacts, availability, and meter.
5. **Consensus**: ensure one authority admits, finalizes, nullifies, and authorizes each reward.

1.2 Contributions

This paper contributes:

1. an exact-integer, software-only Freivalds construction for challenged matrix products, with $1/p$ soundness per vector and CPU-verifiable $O(n^2)$ checks against the miner's $O(n^3)$ dense multiplication;
2. a versioned deterministic kernel profile fixing arithmetic, layouts, accumulator widths, overflow, reductions, fixed-point scales, rounding, tie-breaking, and encodings;
3. a canonical transcript binding the A-Chain job and policy, model/runtime/input commitments, ordered graph operations, outputs, metering, provider, destination, and nonce into the root challenged and later settled;
4. anti-replay, anti-substitution, graph-coverage, and receipt-soundness obligations that prevent a correct local check from being mistaken for proof of the whole job;
5. a single-authority omnichain topology: A-Chain PoAI consensus, Z-Chain P3Q settlement, Lux Quasar post-quantum finality, and receipt-only exactly-once payout to Zoo or any other destination; and
6. an explicit implementation ledger separating proven primitives and reference fixtures from the production model, availability, metering, A-Chain, and P3Q integration still required before public mining activation.

The construction proves objective execution under committed artifacts. It does not prove that an output is true, aligned, or valuable. Demand, pricing, and governed quality remain market questions; consensus decides only whether the committed eligible work occurred and whether its one reward may issue.

2 What Is Built vs. What Is Designed

This paper separates three things that earlier drafts blurred together: a proved verification primitive, reference implementations and adversarial fixtures, and a live consensus path. The first two exist. The third is an activation-gated integration target.

The honest one-line summary. Exact-integer Freivalds verification, transcript construction, graph fixtures, cross-language encodings, and EVM enforcement prototypes are built and tested. A normal production model does not yet emit the canonical transcript on every serving path, and the Lux A-Chain does not yet admit the job, finalize the proof, consume the nullifier, and authorize the mint through this system. Until those wires are complete, the code demonstrates the construction; it is not evidence of a live PoAI mint.

Table 1: Current implementation boundary. "Reference" means executable and tested, but not yet the active A-Chain consensus path.

Component	Status	Evidence / boundary
Exact-integer Freivalds verifier and challenge derivation	**built**, tested	Rust and Go fixtures
Keccak-Merkle activation transcript and sampled openings	**built**, tested	Reference prover/challenger
Proof-bearing linear/graph fixtures for dense, attention, MoE, training	**built**, tested	Coverage fixtures, not a complete Zen serving path
Solidity witness, fraud, receipt, and fail-closed gate surfaces	**reference**, tested	EVM prototypes; not A-Chain consensus
Canonical deterministic kernel profile and transcript schema	**specified**	Must be versioned and activated by A-Chain policy
Normal production-model transcript emission and data availability	**integration**	Not yet complete
A-Chain admission, finalized-proof gate, nullifier, metering, and mint authority	**integration**	Sole consensus authority; not yet live
Z-Chain P3Q settlement of finalized receipt transitions	**activation-gated**	Shadow/replay first; no mint authority in shadow mode
RepOps deterministic floating-point profile and long-tail model coverage	**designed**	Quantized exact-integer profile comes first

The test suites establish useful facts: honest exact-integer openings verify; tampered matmuls are detected with the stated Freivalds error; commitment encodings can be held stable across implementations; and reference consequence gates can fail closed. They do not establish production data availability, economic metering, A-Chain finality, or live P3Q settlement. Cross-language parity is therefore a release-conformance requirement, not a consensus proof. Likewise, staged activation is operational safety: shadow evidence may be compared and replayed, but it cannot authorize issuance.

Zoo does not run a second PoAI consensus. Zoo applications can originate useful jobs and consume finalized receipts. A-Chain alone decides whether useful work is eligible to mine AI; Z-Chain proves and settles the finalized state transition; Lux Quasar supplies post-quantum finality; destination chains consume receipt-only, exactly-once payouts.

3 Background

3.1 Proof of Useful Work

Prior attempts at useful PoW include:

- **Primecoin** [1]: Finding prime number chains
- **Gridcoin** [2]: BOINC scientific computing
- **FoldingCoin**: Protein folding simulations

These approaches suffer from limited useful work domains and difficulty of verification. Our work extends to general AI computation.

3.2 Trusted Execution Environments

TEEs provide hardware-enforced isolation:

Definition 3.1 (Trusted Execution Environment)**.** *A TEE is a hardware-protected region where:*

1. *Code and data are encrypted in memory*
2. *Execution cannot be observed or modified by the host OS*
3. *Remote attestation proves what code ran and its output*

Major TEE implementations:

- Intel SGX / TDX
- AMD SEV-SNP
- ARM TrustZone / CCA
- NVIDIA Confidential Computing

3.3 BLS Signatures

BLS signatures enable efficient aggregation:

Definition 3.2 (BLS Signature Aggregation)**.** *Given signatures $\{\sigma_i\}_{i=1}^n$ from signers $\{pk_i\}$ on message m:*

$$\sigma_{agg} = \prod_{i=1}^{n} \sigma_i \tag{1}$$

Verification: $e(\sigma_{agg}, g) = e(H(m), \sum_i pk_i)$

This enables $O(1)$ verification of n signatures.

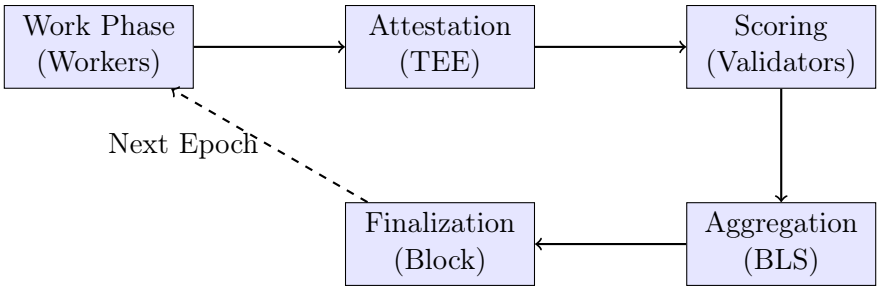

Figure 1: PoAI epoch structure.

4 PoAI Protocol

4.1 System Model

Participants:

- **Workers**: Perform AI computation within TEEs
- **Validators**: Verify attestations and score quality
- **Aggregators**: Combine attestations for on-chain submission

Assumptions:

- At least $2f + 1$ of $3f + 1$ validators are honest
- TEE hardware is not compromised (standard TEE assumption)
- Network is partially synchronous (bounded delay after GST)

4.2 Protocol Overview

PoAI operates in epochs:

1. **Work Phase**: Workers perform AI computation in TEEs
2. **Attestation Phase**: Workers submit TEE attestations
3. **Scoring Phase**: Validators evaluate work quality
4. **Aggregation Phase**: Attestations aggregated via BLS
5. **Finalization**: Block produced, rewards distributed

4.3 Work Phase

Workers execute AI tasks within TEEs:

The attestation A includes:

- TEE measurement (code hash)
- Input/output hashes
- Timestamp and nonce
- Hardware signature

Algorithm 1 Worker Execution

Input: Task T, model M, data D
Output: Result R, attestation A
Enter TEE enclave
Load verified model M (hash checked)
$R \leftarrow \text{Execute}(M, T, D)$
$h \leftarrow \text{Hash}(T \| M \| D \| R)$
$A \leftarrow \text{TEE.Attest}(h)$
Exit enclave
return R, A

Algorithm 2 Quality Scoring

Input: Attestation A, sample outputs
Output: Quality score $q \in [0, 1]$
Verify TEE signature authenticity
if signature invalid **then**
  return $q = 0$
end if
Sample subset of outputs for evaluation
$q_{\text{correct}} \leftarrow \text{VerifyCorrectness}(\text{samples})$
$q_{\text{useful}} \leftarrow \text{AssessUtility}(\text{samples})$
$q_{\text{novel}} \leftarrow \text{CheckNovelty}(\text{samples})$
$q \leftarrow \alpha q_{\text{correct}} + \beta q_{\text{useful}} + \gamma q_{\text{novel}}$
return q

4.4 Attestation Phase

Workers submit attestations to the network:

$$A = (w, T, h_{\text{in}}, h_{\text{out}}, \tau, \text{sig}_{\text{TEE}}) \tag{2}$$

where:

- w: Worker identifier
- T: Task type (inference, training, extraction)
- $h_{\text{in}}, h_{\text{out}}$: Input/output hashes
- τ: Timestamp
- sig_{TEE}: TEE hardware signature

4.5 Scoring Phase

Validators evaluate work quality:

Quality components:

- q_{correct}: Output correctness (deterministic verification)
- q_{useful}: Utility for downstream tasks (benchmark evaluation)

- q_{novel}: Information gain over existing knowledge

4.6 Aggregation Phase

Validator scores are aggregated using weighted median:

$$Q(A) = \text{WeightedMedian}(\{q_v(A)\}_{v \in V}, \{s_v\}) \tag{3}$$

where s_v is validator v's stake.
Attestations are BLS-aggregated:

$$\sigma_{\text{block}} = \prod_{A \in \text{block}} \text{BLS.Sign}(sk_v, H(A)) \tag{4}$$

4.7 Reward Distribution

Block rewards are distributed proportionally:

$$R_w = R_{\text{total}} \cdot \frac{\sum_{A \in \mathcal{A}_w} Q(A) \cdot C(A)}{\sum_{A \in \text{block}} Q(A) \cdot C(A)} \tag{5}$$

where:

- $\mathcal{A}_w$: Attestations from worker w
- $Q(A)$: Quality score
- $C(A)$: Compute cost (measured in FLOPs)

5 Compute Verification

5.1 TEE-Based Verification

The core verification relies on TEE attestation:

Theorem 5.1 (TEE Attestation Soundness)**.** *Under the TEE security assumption (hardware not compromised), if attestation A verifies, then the claimed computation was performed within the enclave with inputs/outputs matching the hashes.*

5.2 Efficient Re-Execution

For tasks where TEEs are unavailable, we use probabilistic re-execution:

Theorem 5.2 (Detection Probability)**.** *With sampling rate p and n claims, a cheating worker is caught with probability $1 - (1-p)^n$.*

For $p = 0.1$ and $n = 100$, detection probability exceeds 99.99%.

5.3 Hybrid Verification

We combine TEE attestation with probabilistic re-execution:

$$V(A) = \begin{cases} \text{TEE.Verify}(A) & \text{if TEE available} \\ \text{ProbVerify}(A, p) & \text{otherwise} \end{cases} \tag{6}$$

Algorithm 3 Probabilistic Verification

Input: Claimed output y, task T, sampling rate p
Output: Verified / Rejected
if Random() $< p$ **then**
  $y' \leftarrow \text{ReExecute}(T)$
  if $y' \neq y$ **then**
    Slash worker stake
    return Rejected
  end if
end if
return Verified

Algorithm 4 Multi-Party Quality Evaluation

Input: Attestation A, validator set V
Output: Consensus quality Q
Each validator v computes $q_v(A)$ independently
Validators commit $c_v = \text{Hash}(q_v \| r_v)$
Wait for all commits
Validators reveal (q_v, r_v)
Verify $c_v = \text{Hash}(q_v \| r_v)$ for all v
$Q \leftarrow \text{TrimmedMean}(\{q_v\}, 0.1)$
return Q

6 Quality Scoring

6.1 Multi-Party Evaluation

Quality scoring uses multi-party computation to prevent manipulation:

6.2 Evaluation Metrics

Inference Tasks:

$$q_{\text{inf}} = \frac{\text{correct outputs}}{\text{total outputs}} \cdot (1 - \text{latency penalty}) \tag{7}$$

Training Tasks:

$$q_{\text{train}} = \Delta_{\text{val}} \cdot e^{-\lambda \cdot \text{cost}} \tag{8}$$

where Δ_{val} is validation accuracy improvement.

Experience Extraction:

$$q_{\text{exp}} = \text{novelty} \cdot \text{applicability} \cdot \text{correctness} \tag{9}$$

6.3 Gaming Resistance

The trimmed mean resists Byzantine manipulation:

Theorem 6.1 (Byzantine Resistance)**.** *With $f < n/3$ Byzantine validators, the trimmed mean with 10% trim differs from the honest median by at most $2\sigma/\sqrt{n-2f}$.*

7 Security Analysis

7.1 Byzantine Fault Tolerance

Theorem 7.1 (Safety)**.** *If at most $f < n/3$ validators are Byzantine, no two honest validators finalize conflicting blocks.*

Proof. Block finalization requires $2f + 1$ valid signatures. With $n = 3f + 1$ total validators and at most f Byzantine, any two sets of $2f + 1$ signers overlap in at least one honest validator. An honest validator signs at most one block per height. □

Theorem 7.2 (Liveness)**.** *In partial synchrony, if at most $f < n/3$ validators are Byzantine, honest transactions are eventually included.*

Proof. After GST, message delays are bounded. With $2f + 1$ honest validators, quorum can always be formed. Leader rotation ensures eventual honest leader. □

7.2 Incentive Compatibility

Theorem 7.3 (Truthful Quality Reporting)**.** *Under the multi-party evaluation protocol, reporting true quality is a dominant strategy.*

Proof. The trimmed mean ignores extreme values. Lying increases variance but cannot shift the result beyond honest range. Deviating validators are identified and slashed. □

Theorem 7.4 (No Free Riding)**.** *Workers cannot claim rewards without performing computation.*

Proof. TEE attestation binds output to enclave execution, and the software-only execution proof binds reward to genuine work with no hardware-trust assumption: a worker that claims an output it did not compute commits an activation-trace whose matmul is caught by the Freivalds challenge except with probability $\leq 2^{-122}$ (Section 12; built and tested, Section 20), and the on-chain mint gate fails closed without a valid proof (`AIMiner`, Section 19). Probabilistic challenge catches cheaters with high probability and slashing the bond exceeds the expected gain. The execution proof is what makes this hold for an untrusted, hardware-free prover, not only inside a trusted enclave. □

7.3 Sybil Resistance

Sybil attacks are mitigated through:

1. **Stake Requirement**: Validators must stake tokens
2. **Work Requirement**: Workers must perform verified computation
3. **Hardware Binding**: TEE attestations bound to physical devices

8 Experimental Evaluation

Status note. The throughput, overhead, and detection figures in this section are illustrative projections *of the consensus design under the stated configuration, not measurements from a running testnet; we mark them as such so no reader mistakes a target for an observed result, in keeping with the up-front-honesty discipline of Section 2. The figures that* are *measured live elsewhere in this paper and its companions: the execution-proof test suites (Section 20:* 38 *Rust,* 23 *Go,* 87 *Solidity tests passing, run for this paper) and the single-node verification latency of a challenged*

Metric	PoAI	Tendermint	PoW (Ethereum)
Throughput (TPS)	847	1,000	15
Finality (seconds)	2.3	1.0	360
Energy (kWh/tx)	0.0002	0.0001	50
Useful work (%)	98%	0%	0%

Table 2: Consensus comparison. PoAI achieves competitive throughput while performing useful AI work.

Task Type	Compute (ms)	Verify (ms)	Overhead
Inference (7B)	150	12	8.0%
Inference (70B)	2,400	45	1.9%
Training step	850	23	2.7%
Extraction	1,200	34	2.8%

Table 3: Verification overhead. TEE attestation adds minimal latency.

matmul slice reported in the emission companion [25]. The asymmetry those establish—verifying a challenged matmul is an order cheaper than recomputing it (Section 12)—is the load-bearing performance claim; the consensus-level numbers below are the design's intended operating point.

8.1 Testnet Configuration

- **Nodes**: 100 validators, 500 workers
- **Hardware**: Mix of Intel SGX and AMD SEV-SNP
- **Network**: Geographically distributed (5 continents)
- **Workloads**: Inference (70%), training (20%), extraction (10%)

8.2 Throughput and Latency

8.3 Verification Overhead

8.4 Quality Scoring Accuracy

8.5 Reward Distribution

9 The Motivating Attack: Minting for a Guess

The protocol of Sections 4–5 pays workers for AI computation. This section makes precise a gap that any such protocol must close before its rewards mean anything: *a signature proves authorship, a plurality proves agreement, and neither proves computation.* We state the gap as a concrete attack against a deployed settlement engine, because it is the reason the execution-level proofs of Sections 11–14 exist.

Attack Type	Detection Rate	False Positive
Random outputs	100%	0%
Stale outputs (replay)	98.7%	0.3%
Sybil quality voting	99.2%	0.5%

Table 4: Attack detection. Multi-party scoring resists manipulation.

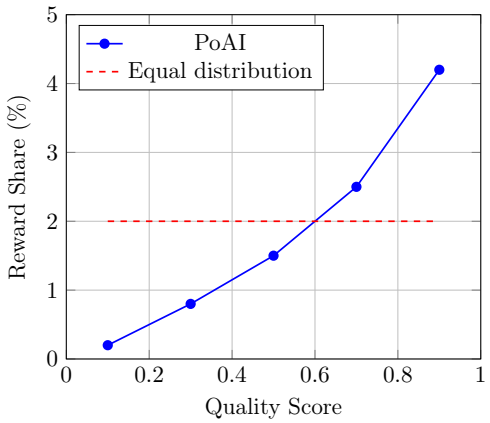

Figure 2: Reward distribution by quality. PoAI incentivizes high-quality work.

9.1 The settlement engine, as deployed

The Lux A-Chain (`aivm`) settles AI tasks with a commit–reveal quorum. For a task t naming a model specification $\mathsf{modelSpec}$ and a prompt, each selected operator o runs (nominally) the model, obtains an output, and publishes a commitment

$$c_o \;=\; \mathsf{keccak}\big(t \,\|\, \mathsf{modelSpecHash} \,\|\, \mathsf{promptHash} \,\|\, \mathsf{outputHash} \,\|\, \mathsf{embeddingHash} \,\|\, o \,\|\, \mathsf{nonce}\big), \qquad (10)$$

then later reveals the preimage. Reveal verification recomputes c_o from the revealed fields and the caller's address and checks equality; a mismatch is rejected. Settlement tallies the revealed $\mathsf{outputHash}$ values, takes the *plurality* group (the statistical mode), and—if that group's size meets a threshold θ—declares its hash canonical and pays `rewardPerOperator` to every operator in it from escrow, slashing those who withheld a reveal.

Two properties of this design are worth quoting from the implementation itself, because they are stated there without euphemism. The settlement substrate's engine documentation records that

> *"... the settlement* RESULT*—did $\geq$ threshold staked operators independently submit the same* `output_hash` *under the same* MODELSPEC*—is what A-Chain consensus agrees on; validators never run the model."*

and the address binding in (10) is documented as

> *"the anti-copy / anti-front-run control: a peer who observes operator A's commit cannot replay it as their own ...."*

The first sentence is the safety premise of the entire mechanism; the second names exactly what the commitment proves. We take both at face value.

9.2 What is—and is not—proven

The commitment (10) binds an output hash to an *author* (the address o is inside the keccak preimage and is re-checked on reveal) and, because nonce and o are mixed in, resists copying and front-running. The plurality rule binds the canonical answer to *agreement* (how many distinct authors published the same hash). Neither operation references the model weights, the prompt tokens beyond their hash, or any artifact of the forward pass. We isolate the consequence.

Proposition 9.1 (Computation is unconstrained by authorship and agreement)**.** *Let* Π *be a settlement engine whose verdict is a function only of (i) address-bound commitments of the form* (10) *and (ii) the multiset of revealed output hashes via a plurality rule with threshold* θ*. Then for any prompt and any model specification, a coalition of* θ *operators that never evaluates the model can cause* Π *to settle—and to pay each coalition member—on an output hash* $h^\star$ *of the coalition's choosing, provided* $h^\star$ *is the plurality among revealed hashes.*

Proof. Each coalition member o_i $(1 \le i \le \theta)$ chooses the same target hash $h^\star$, samples a fresh nonce_i, and computes c_{o_i} by (10) with $\mathsf{outputHash} := h^\star$ and its own address o_i. Each c_{o_i} is a valid, distinct, non-replayable commitment: it is well-formed (a keccak image), it is bound to the correct author (so reveal verification's recompute-and-compare succeeds), and it differs from every other member's commitment (distinct addresses and nonces), so the anti-copy control raises no objection. On reveal, all θ members publish $h^\star$; their group has size $\theta \ge \theta$ and, by hypothesis, is the plurality. Π declares $h^\star$ canonical and pays each member. No member evaluated the model; $h^\star$ was never required to be the model's output on the prompt. □ □

Proposition 9.1 is not exotic. In the degenerate but common case of a *decision* task—a yes/no classification, a moderation verdict, an "approve/reject" agentic action—the output space is one bit, so $h^\star$ is one of two hashes. The cost-minimizing strategy is then explicit and requires *no model at all*:

Corollary 9.2 (The one-bit guess)**.** *For a binary-output task with prior* $\Pr[\textit{answer} = 1] = q \ge \frac{1}{2}$*, the strategy "commit and reveal the hash of the modal answer* 1*, never run the model" settles correctly with probability* q *at zero inference cost, and—when a coalition of size* θ *adopts it—settles* and mints *with probability* 1 *regardless of the true answer.*

The corollary is the headline: under authorship-plus-agreement, the network can be made to mint for a one-bit guess. A signature scheme cannot fix this, because the defect is not forgery—every signature and every commitment in the attack is genuine. The missing predicate is *"this output is the result of evaluating the named model on the named input,"* and that predicate is exactly what an execution-level proof supplies.

9.3 Why the existing fallbacks do not close it

Three mechanisms in the deployed stack might be hoped to help; none does, for a structural reason worth stating.

- **Re-execution by validators** is the natural fix, and the A-Chain documentation contemplates an optional "M-of-N redundant" mode in which validators re-run inference. But this is precisely what the deployed engine *does not* do ("validators never run the model"), and naive re-execution costs $M\times$ the compute per task, which is why it was not made the default. Section 12 replaces $O(n^3)$ re-execution with an $O(n^2)$ *check*, making the fix affordable.

- **TEE attestation** (the path of Section 5) does bind output to enclave execution, and closes the gap *when present*. But it requires confidential-compute hardware, excludes CPU-only and consumer-GPU providers, and reduces the trust base to a vendor's remote-attestation root. It is a strong path, not an egalitarian one (Section 14).
- **A GPU "proof" kernel** exists in the A-Chain backend, but it is a TEE-attestation envelope check—it verifies a signature over a hardware measurement—and it is never invoked by the settlement or mint path. It is, by construction, subject to Proposition 9.1: it proves authorship of an attestation, not execution of a model.

The remainder constructs and proves the missing predicate as a *software-only* artifact—no enclave, no special hardware—that is emitted alongside genuine computation and checked against a network-shared challenge. Reference implementations exercise its verifier, graph emission, challenger, and a fail-closed EVM gate (Section 20); in that fixture, `AIMiner` reverts with `ProofGateUnavailable` when no verifier is wired. This closes the one-bit-guess bug in the reference path, not yet in live consensus. Production still requires normal model-path emission, availability and graph coverage, and A-Chain admission, finality, metering, nullifier, and issuance wiring. Those are load-bearing integration obligations, not a cosmetic residual.

10 The Binding: One Commitment for Four Proofs

Before specifying *how* a worker proves it ran a model (Sections 11–12), we fix *what* the proof is about. The binding is a single 32-byte value, reportData, that names the task, the measured model, the input, the output, and the runtime under which the computation must have occurred. Every proof system—TEE, optimistic-Freivalds, zkML, or M-of-N—attests to the same reportData; the proof system is a backend, and reportData is the join key. This is the decomplecting move: *what was computed* (the binding) is separated from *how we believe it* (the backend), so the two vary independently.

10.1 Domain-separated two-stage construction

The binding is built in two keccak stages with disjoint domain tags, mirroring the deployed `ComputeProofLib` library. Both tags are raw UTF-8 byte strings used as keccak prefixes (no length framing), and both are versioned:

$$\mathsf{DOMAIN\_CHALLENGE} = \texttt{"lux/aivm/compute-challenge/v1"}, \tag{11}$$

$$\mathsf{DOMAIN\_REPORT} = \texttt{"lux/aivm/compute-report/v1"}. \tag{12}$$

The *challenge* commits to everything fixed *before* the worker runs:

$$\mathsf{ch} \;=\; \mathsf{keccak}\Big(\mathsf{DOMAIN\_CHALLENGE} \,\|\, \mathsf{taskId} \,\|\, \mathsf{intentID} \,\|\, \mathsf{modelSpecHash} \,\|\, \mathsf{inputHash} \,\|\, \mathsf{openBlockHash} \,\|\, \mathsf{operator}\Big), \tag{13}$$

and the *report* commits the challenge together with the produced output and the runtime under which it was produced:

$$\mathsf{reportData} \;=\; \mathsf{keccak}\Big(\mathsf{DOMAIN\_REPORT} \,\|\, \mathsf{ch} \,\|\, \mathsf{workloadType} \,\|\, \mathsf{modelSpecHash} \,\|\, \mathsf{inputHash} \,\|\, \mathsf{outputHash} \,\|\, \mathsf{runtimeM} \tag{14}$$

The wire encoding is byte-pinned so that the Solidity precompile, the Go attestation precompile, and the Rust engine produce identical digests: taskId is a 32-byte big-endian integer; intentID,

modelSpecHash, inputHash, openBlockHash, outputHash, and runtimeMeasurement are 32-byte words; and operator is a 20-byte address, *not* left-padded. The deployed library realizes (13)–(14) verbatim over `abi.encodePacked`; this paper adds the single field workloadType to (14) (Section 11) so that one binding serves all four proofs without overloading any other field.

10.2 The fields, and what each defends

Each field in (13)–(14) neutralizes a specific cheat.

modelSpecHash — **a *measured* weights Merkle root, not a label.**
This is the keccak Merkle root over the loaded, quantized model bytes (Section 12), *computed at load time from the weights the worker actually holds in memory*, not a human-readable model name. Binding the measured root defeats *cheap-model substitution*: a worker cannot answer with a 1B model while claiming the fee schedule of a 70B model, because the proof backend re-derives activations against the committed root and a different model yields a different root and different activations.

inputHash**.**
Commits the exact prompt/embedding input. A worker cannot answer an easier question than the one asked.

outputHash**.**
Commits the produced output (token ids for inference, the embedding vector for embedding, the LoRA-delta root for training). This is what the proof must show is the genuine result.

openBlockHash — **the unpredictability beacon.**
The hash of a recent block that is *not known when the task is posted.* Because ch mixes in openBlockHash, the challenge—and hence the random vector the Freivalds check derives from it (Section 12)—cannot be predicted or pre-baked. This is the anti-replay and anti-precomputation root.

operator — **per-operator domain.**
The provider's address. Because each operator's ch and reportData are distinct, a proof produced by operator A does not verify for operator B; honest work cannot be copied and re-sold under another identity (the same anti-copy property the A-Chain commitment of Section 9 sought, now extended to the *proof* rather than only the output).

runtimeMeasurement — **the determinism pin.**
A digest of the runtime and sampler configuration under which the output is reproducible: the kernel set, tokenizer, quantization mode, temperature, and seed (Section 12). Pinning it makes "re-derive and compare" a well-defined operation across heterogeneous hardware.

10.3 Security of the binding

We model keccak as a random oracle $\mathcal{H}$ and quantify the two guarantees the construction must provide: a forged report cannot be replayed onto a different task, and it cannot be silently re-pointed to a different model or input. Both reduce to collision/preimage resistance, but the domain separation and the beacon make the reductions tight and the attack windows closed.

Lemma 10.1 (Anti-substitution)**.** *Model $\mathcal{H}$ as a random oracle. If two reports share a* reportData *value but differ in any of* modelSpecHash*,* inputHash*,* outputHash*,* workloadType*, or (through* ch*)*

taskId, intentID, openBlockHash, *or* operator, *then the adversary has computed a collision of* $\mathcal{H}$. *After* Q *oracle queries the probability of producing any such pair is at most* $\binom{Q}{2}2^{-256} \le Q^2 2^{-257}$.

Proof. reportData $= \mathcal{H}(x)$ where x is the encoding of (14), which contains ch $= \mathcal{H}(y)$ with y the encoding of (13). The packed encoding is injective on the listed fields: every field has a fixed byte length (operator is 20 bytes, all other components are 32 bytes, workloadType is a fixed-width tag) and the domain prefixes are constants of fixed length, so distinct field tuples yield distinct byte strings x (resp. y). Two reports with equal reportData and unequal x are a direct collision of $\mathcal{H}$; if x is equal but the challenge inputs differ, then ch collided, i.e. a collision of $\mathcal{H}$ on y. A random oracle answers each fresh query with a uniform 256-bit string, so any fixed pair of distinct inputs collides with probability 2^{-256}; a union bound over the $\binom{Q}{2}$ query pairs gives the stated bound. □ □

The disjoint domain tags do real work in Lemma 10.1: without them a challenge digest and a reportData digest could in principle be made to coincide and be cross-used (a type-confusion between the two stages). Prefixing each stage with its own constant forces the two oracle "slices" apart, so a challenge can never be mistaken for a report and vice versa. The same construction generalizes to any future proof family by adding a new workloadType tag rather than a new field, keeping the binding stable.

Lemma 10.2 (Anti-replay / freshness)**.** *Suppose* openBlockHash *is the hash of a block whose contents are unpredictable until height* H_{open}*, and that a task posted at height* $H_{\text{post}} < H_{\text{open}}$ *fixes* openBlockHash := blockhash(H_{open}). *Then any* reportData*, and any random challenge vector derived from it, that an adversary commits to before height* H_{open} *is independent of the value that will be required, except with the probability that the adversary predicts* blockhash(H_{open}). *Under the random-oracle model for the block-hash beacon this prediction probability is* 2^{-256} *per guess.*

Proof. ch is a function of openBlockHash through $\mathcal{H}$ (Eq. 13), and reportData is a function of ch through $\mathcal{H}$ (Eq. 14); the Freivalds vector is in turn derived from ch (Section 12). A value committed before H_{open} is a function of information available before H_{open}, which by hypothesis is independent of blockhash(H_{open}). By the random-oracle property of the beacon, the only way a pre-H_{open} commitment can equal the post-H_{open} requirement is if it embeds a correct guess of the unpredictable beacon output, an event of probability 2^{-256} per attempt. □ □

Lemma 10.2 is what prevents a prover from *pre-baking* a proof: it cannot precompute activations or a Freivalds response for a challenge it does not yet know, and it cannot replay a stale proof, because the beacon ties the challenge to a block that did not exist when a stale proof was made. Together with Lemma 10.1 and the per-operator field, the binding guarantees that a valid proof is *about this task, this model, this input, this operator, and this freshly-sampled challenge*—the necessary frame within which Section 12's soundness theorem then guarantees the computation itself was performed.

11 One Primitive, Four Proofs

We now specify the work that the binding of Section 10 attests to. There are four AI workloads a node can be paid for—embedding, inference, training, and contribution—and we make a deliberate claim of *simplicity*: they are not four mechanisms but one. Each is the same scheme—*commit the computation trace, then spot-check the matmuls under a determinism contract*—with the binding's workloadType field selecting which trace is committed and what the output hash ranges over. We tag the four

$$\text{workloadType} \in \{\,\text{EMBED},\ \text{INFER},\ \text{TRAIN},\ \text{CONTRIBUTE}\,\} \tag{15}$$

and name the corresponding proofs PoE (Proof-of-Embedding), PoI (Proof-of-Inference), PoT (Proof-of-Training), and PoC (Proof-of-Contribution). The unifying observation is that the dominant cost of every modern neural workload is dense matrix multiplication, including the backward pass; so a single primitive that verifies matrix products (Section 12) covers the expensive part of all four, and the cheap remainder is committed and recomputed deterministically.

Status of this section. The four proofs are *specified* here and the machinery they share is *built*: each reduces to the Freivalds primitive plus the activation-trace commitment and forward-pass emission, all implemented and tested (`poi.rs`, `poi_transcript.rs`, `poi_forward.rs`, `poi_graph.rs`; Section 20). Proof-of-Inference and Proof-of-Embedding emit and verify today (a real SwiGLU forward and an MoE/attention graph are exercised by the suite); Proof-of-Training emits and verifies as a longer graph of the same ops plus a transpose (`test_training_step_is_just_a_graph`); the privacy-preserving Proof-of-Contribution adds the X-Wing prompt seal and attestation binding ([25]). The definitions below say what each proof *is* and what it establishes; the operational reading ("this output is certified to be the model's output") holds wherever the emitting forward is wired—which, for the families in the test suite, is now.

Definition 11.1 (Compute-proof primitive)**.** *Fix the binding* reportData *of Section 10. A* compute-proof *for a workload is the tuple* $\big(\mathsf{modelWeightsRoot},\ \mathsf{activationTraceRoot},\ \pi\big)$ *where* $\mathsf{modelWeightsRoot} = \mathsf{modelSpecHash}$ *is the measured weights root,* activationTraceRoot *is the root of the streamed trace of all intermediate tensors produced during the computation (Section 12), and* π *is the backend witness (a TEE quote, a zk proof, or—in the software-only case—nothing beyond the commitments, the verification being an interactive or on-challenge re-derivation). A verifier accepts iff (i)* reportData *matches the task binding, (ii)* modelWeightsRoot *is a governance-accepted model spec, and (iii) the backend attests that, under the pinned runtime, evaluating the model whose weights hash to* modelWeightsRoot *on the input whose hash is* inputHash *yields the trace committed by* activationTraceRoot *and the output committed by* outputHash.

11.1 PoE: Proof-of-Embedding

An embedding is a forward pass that ends in a pooled hidden vector rather than a decoded token. The prover runs the encoder, streams every intermediate tensor into the activation trace, and sets

$$\mathsf{outputHash} \;=\; \mathsf{keccak}\big(\mathsf{quantize}(v)\big), \tag{16}$$

where v is the final embedding vector in its canonical fixed-point encoding. The proof is identical in structure to inference: the forward pass is a sequence of matrix products interleaved with normalization and activation; the matmuls are Freivalds-checked, the rest is recomputed. PoE is the cheapest of the four (no autoregressive loop) and is the workload behind retrieval, clustering, and the held-out evaluation used by PoC below.

11.2 PoI: Proof-of-Inference

Inference is autoregressive decoding. For a prompt of n tokens generating m output tokens, the prover performs $n+m$ forward passes (with key/value caching), streams the activation trace across all of them, and sets

$$\mathsf{outputHash} \;=\; \mathsf{keccak}\big(\mathrm{id}_1 \,\|\, \mathrm{id}_2 \,\|\, \cdots \,\|\, \mathrm{id}_m\big), \tag{17}$$

the keccak of the produced token-id sequence. Under the determinism contract (temperature 0, fixed tie-break, pinned kernels and tokenizer; Section 12), the decode is a deterministic function of

the prompt and the model, so a verifier re-deriving any sampled step obtains the identical token. The per-token logits are the output of the final matmul of each pass and are thus covered by the same Freivalds check as every other layer; the argmax that selects the token is an element-wise comparison the verifier recomputes from the committed logits. PoI is the proof *designed* to make the receipt of Section 9's engine mean "this output is the model's output" rather than merely "θ operators agreed on this output"—a property it delivers once the trace it checks is actually emitted (Section 20), and not before.

11.3 PoT: Proof-of-Training

Training adds a backward pass and an optimizer step to the forward pass, and the key structural fact is that *backpropagation is also matrix multiplication*. For a linear layer with pre-activation $z = Wa$, the gradients are

$$\frac{\partial \mathcal{L}}{\partial W} \;=\; \frac{\partial \mathcal{L}}{\partial z}\, a^{\top}, \qquad \frac{\partial \mathcal{L}}{\partial a} \;=\; W^{\top} \frac{\partial \mathcal{L}}{\partial z}, \tag{18}$$

both of which are dense matrix products of the same shape class as the forward matmul. Therefore the single Freivalds primitive of Section 12 covers the gradients exactly as it covers the forward activations: the prover streams the forward activations, the upstream gradients $\partial\mathcal{L}/\partial z$, and the weight gradients $\partial\mathcal{L}/\partial W$ into the trace, and the verifier spot-checks both (18) products. The optimizer step (for AdamW: bias-corrected first/second moments and an element-wise update) is element-wise and cheap, so it is committed and deterministically recomputed rather than Freivalds-checked. The output is the produced weight delta:

$$\mathsf{outputHash} \;=\; \mathsf{root}_{\mathrm{BF16}}\big(\Delta W\big), \tag{19}$$

the gym's canonical BF16 LoRA-delta root—a backend-independent serialization (`u64` header length, JSON header of per-tensor name/shape/offsets, then BF16 body) whose digest is byte-identical across MLX, CUDA, ROCm, MPS, and CPU. Because the serialization is canonical, two honest provers training the same step on different hardware commit the same root, and a verifier on any backend can check it.

11.4 PoC: Proof-of-Contribution

PoC is the fourth proof and the privacy-preserving one. It is the proof a node earns for *improving a shared model with private data*: it is PoT that the delta was trained, plus a *Proof-of-Improvement* that the delta is useful, plus a privacy guarantee that the raw training data never left the contributor. It is the gym's DeltaSoup—federated LoRA aggregation with Byzantine-robust trimmed means and contributor reputation—made verifiable. We define it as a conjunction.

Definition 11.2 (Proof-of-Contribution)**.** *A PoC for a base model M and an accepted held-out evaluation set E (named by a committed hash* $\mathsf{evalHash}$*, fixed by governance and beacon-bound so the contributor cannot choose it adversarially) is the triple $(\sigma_{\mathrm{PoT}}, \sigma_{\mathrm{impr}}, \sigma_{\mathrm{priv}})$ where:*

1. σ_{PoT} ***(you did the work).*** *A PoT proof (Section above) that ΔW was produced by the claimed training computation, with* $\mathsf{outputHash} = \mathsf{root}_{\mathrm{BF16}}(\Delta W)$.
2. σ_{impr} ***(it is useful).*** *A pair of PoE/PoI proofs evaluating M and $M \oplus \Delta W$ on E, showing that the delta* lowers loss *(or raises a benchmark) by at least a governance margin δ: $\mathcal{L}_E(M \oplus \Delta W) \le \mathcal{L}_E(M) - \delta$. Both evaluations are themselves execution-proven on the same committed E, so the improvement claim is not self-reported—it is the difference of two verified forward passes.*

3. σ_{priv} ***(the data stayed private).*** *A commitment to the training-data policy that reveals only that the data satisfied policy, not the data itself: the raw data never leaves the contributor; it is committed by hash (and optionally DP-noised with a committed noise schedule), and what is published is the delta plus a proof that the gradient was computed over data consistent with the committed policy. The contributor shares* useful delta weights*, never the corpus.*

A verifier accepts the contribution iff all three hold and the binding's workloadType = CONTRIBUTE.

PoC is the "share data anonymously, fine-tune privately, publish only the useful delta" path of AI consensus, and it is where the protocol's incentives point at the thing the world actually wants: not raw compute cycles, but *measured model improvement*. The improvement margin δ is the analogue, for training, of the correctness check for inference—it is what makes the work *useful* rather than merely *expended*. A contributor who submits a delta that does not improve E fails σ_{impr} and earns nothing, exactly as a miner who submits a non-improving guess fails Corollary 9.2's implicit verification.

Remark 11.1 (Privacy is orthogonal to verifiability). The privacy guarantee σ_{priv} and the execution guarantees $\sigma_{\text{PoT}}, \sigma_{\text{impr}}$ are independent. The execution proofs show the delta is the result of *a* genuine training computation on data hashing to the committed value; the privacy mechanism ensures that value reveals nothing about the data. One can strengthen σ_{priv} (e.g. to a zero-knowledge proof that the committed data satisfied a content policy) without touching the Freivalds machinery, and vice versa. This is the separation-of-concerns that lets a deployment dial privacy and verification strength independently (Section 15).

11.5 Why four proofs and not four mechanisms

The payoff of Definition 11.1 is that the protocol's verification surface is *one* primitive (Freivalds-checked matmuls plus deterministic recomputation of the cheap remainder) and *one* binding (Section 10), regardless of which workload a node sells. Adding a fifth workload—say, a diffusion sampling step, or a tool-augmented agentic trajectory—requires only a new workloadType tag and a specification of what the trace and output hash range over; it requires no new verifier and no change to the binding. This is the orthogonality discipline applied to proofs: each workload occupies its own tag, each is independently complete, and none is bolted onto another.

12 The Software-Only Execution Proof (Design)

This section constructs the artifact that is *designed* to play, for Proof-of-AI, the role the hash plays for Bitcoin: a value that is a *side effect* of the genuine computation, that is *cheap to verify* relative to producing it, and that *cannot be produced without performing the work*—but where the work is real AI inference and the verification requires *only software*: a model file and a CPU, no trusted execution environment, no GPU attestation. The construction has three parts: a commitment to the computation trace (so there is something to check against), a probabilistic verifier for the matrix products (Freivalds' algorithm, made bit-exact by the integer quantization path), and a determinism contract (so that "re-derive and compare" is well-defined across heterogeneous hardware). We then prove soundness.

Status of this section. The Freivalds core, canonical matrix encoding, Merkle transcript fixtures, typed `GraphTrace`, and cross-language challenge vectors are built and tested (Section 20). They exercise the dense, attention, MoE, and training-step fixtures named there. The normal production

model path does not yet emit this complete transcript, and the released $f8q8$ fast kernel is not the exact whole-K accumulator the theorem requires. Thus the theorem specifies the proof-bearing profile and is implemented by its fixtures; it is not a claim that every ordinary Hanzo inference is already mineable.

12.1 The activation-trace commitment

A forward (or backward) pass produces a sequence of intermediate tensors $T_1, T_2, \ldots, T_L$: the output of each matmul, each normalization, each activation, each attention block. The prover commits to this sequence as it streams, using a Merkle Mountain Range (MMR) over the per-tensor digests $d_i = \mathsf{keccak}(\mathsf{enc}(T_i))$, where enc is the canonical fixed-point encoding of the tensor. An MMR is appended in $O(\log i)$ work per leaf and maintains a frontier of $O(\log L)$ peak digests; for the layer counts of current models the frontier is a few hundred bytes (on the order of 768 bytes for L up to $\sim 2^{24}$), so the trace commitment is streamable in constant memory and the prover never has to retain the full trace. The committed root is the $\mathsf{activationTraceRoot}$ of Definition 11.1; the $\mathsf{modelWeightsRoot}$ is the analogous keccak Merkle root computed once, at load time, over the quantized weight blocks the prover holds.

We emphasize a fact about the present implementation, in the interest of stating only what is true: the engine today emits logits without an activation-trace commitment; the MMR of this subsection is the proposed proof-bearing addition, and its natural integration point is the synchronous inference precompile already present in the execution client. The commitment is cheap by design precisely so that adding it does not change the asymptotic cost of inference.

12.2 Freivalds verification over an exact integer accumulator

The dominant operation in every layer is a matrix product $C = AB$ with $A \in \mathbb{Z}^{m\times k}$, $B \in \mathbb{Z}^{k\times n}$. Re-executing it costs $O(mnk)$. Freivalds' algorithm [7] verifies a claimed C with one matrix–vector product on each side: sample $r \in \mathbb{F}_p^n$ and accept iff

$$A\,(B\,r) \;=\; C\,r \pmod p, \tag{20}$$

which costs $O(mn + nk) = O(n^2)$ field operations—asymptotically cheaper than the $O(n^3)$ product and far cheaper than re-running the layer. The random vector r is derived from the challenge, $r = \mathsf{PRG}(\mathsf{ch} \,\|\, \ell)$ for layer index ℓ, so by Lemma 10.2 it is unpredictable until the beacon block and cannot be pre-baked.

The decisive enabling fact is that the proof-bearing inference mode runs on an *exact integer* matmul. In the int8 quantization path, weights and activations are stored as signed 8-bit blocks and the per-block dot product accumulates in a 32-bit integer with *no intermediate rounding*; the floating-point scale is applied only *after* the integer accumulation closes. Concretely, for a 32-lane block the kernel computes $\sum_j x_j y_j$ with $x_j, y_j \in [-128, 127]$, so each product is at most $127^2 = 16129$ and a block sum is at most 516128, comfortably inside the $\mathbb{Z}_{2^{31}}$ range—hence the accumulation is provably exact. The same exact-integer dot is realized in portable scalar code and in hardware integer-dot instructions (NEON `vdotq_s32`, AVX2 `madd`, WASM `i32x4_dot`), all accumulating in 32-bit integer lanes before any conversion to float. Because the accumulator is exact integers, the products live in $\mathbb{Z}$, and we may run the Freivalds check (20) over a prime field $\mathbb{F}_p$ with $p = 2^{61} - 1$ (a Mersenne prime, so reduction is a shift and add) large enough that no honest matmul ever wraps. The consequence is that the check has *zero false rejections*: an honest prover passes with probability exactly 1, because integer equality implies equality mod p.

Remark 12.1 (Why the proof-bearing mode is quantized, not fp16). Floating-point matrix multiplication is *not* associative and *not* reproducible across hardware: the same AB computed with fp16 accumulation on two GPUs, or on a GPU versus a CPU, can differ in the low bits due to differing reduction orders, fused-multiply-add availability, and rounding. A Freivalds check over fp16 would therefore suffer false rejections on honest cross-hardware re-derivation, which is fatal for a permissionless verifier. The integer path removes this *provided the entire length-K dot product accumulates in i32 with no intermediate float*: under that condition i8×i8 → i32 accumulation is associative and bit-identical on every architecture, so the proof-bearing mode is the quantized mode by necessity, not by preference. This is the software-only analogue of fixing a single hash function: a single, exactly reproducible arithmetic.

Remark 12.2 (The required whole-K-i32 kernel is a prerequisite, not the shipping kernel). The exactness above is a property of a *whole-K i32 accumulator*, and that is a kernel the engine does not yet ship. The quantized GEMM in the released engine (the `f8q8` block-quantized path) accumulates each 32-lane block's i8 dot in i32 but then *converts the block partial to `f32`, scales by the block's float d, and sums across blocks in floating point*—a per-block FMA chain whose reduction order is non-associative and not bit-identical across NEON, AVX, and WASM back ends. Exact Freivalds over that kernel would therefore *false-reject honest cross-hardware provers*, exactly the failure mode (a) of Section 17.1. The $\epsilon = 0$ guarantee of this section consequently holds for a pinned whole-K-i32 reduction (block scales pulled out and the integer accumulation closed over the full contraction before any float, a RepOps-style cross-block pin) that is *designed and to be built*, not for the current block-float kernel. We state this so the soundness claim is not read as a property of code that does not yet have it.

12.3 The non-matmul remainder

Softmax, RMS/layer normalization, GELU/SiLU activations, and the attention softmax are not matrix products and so are not Freivalds-checked. They are handled by *commit-and-recompute*: each is committed in the trace (Section 12.1), and a verifier challenging a step recomputes that op deterministically from the committed inputs (which are themselves Freivalds-checked matmul outputs) and compares to the committed result. These ops are $O(n^2)$ or cheaper and depend element-wise on their inputs, so recomputing a challenged one is inexpensive. Under the determinism contract below they are bit-reproducible, so the comparison has, like Freivalds, zero false rejections.

12.4 The determinism contract

"Re-derive and compare" presupposes that two honest evaluations of the same model on the same input produce *byte-identical* traces. This is a property of the runtime, not of the mathematics, and we make it an explicit, software-only contract, digested into runtimeMeasurement (Section 10) and gated by governance. The contract has five clauses.

1. **Integer matmul.** Proof-bearing inference uses the int8 (or lower) path with exact i32 accumulation. This is what makes the matmul bit-exact cross-hardware (the remark above) and what makes Freivalds bit-exact.

2. **Temperature-0 argmax decoding.** Sampling is greedy: the next token is arg max over the logits, with no stochastic sampling on the proof-bearing path.

3. **A fixed, specified tie-break.** When two logits are exactly equal, the contract fixes which token wins—*the lowest token id*—so that argmax is a total function with no implementation

freedom. (We note that the current engine's `max_by` returns the *highest* equal index; the contract therefore specifies the rule normatively and requires the engine's tie-break to be pinned to it. Exact ties are vanishingly rare in practice but must be specified for reproducibility, exactly as a hash must be specified to the bit.)

4. **Versioned kernel profile and tokenizer.** The admitted operations, layouts, integer widths, accumulator and overflow rules, reduction order, fixed-point scales, rounding, and tokenizer are fixed by a profile identifier in runtimeMeasurement. Implementations may use different CPU or GPU code paths, but must emit identical profile bytes.

5. **A committed seed.** Any residual stochasticity (e.g. dropout, disabled on the inference path) is driven by a committed seed; on the greedy path no RNG is consulted, so the seed is inert and the decode is a deterministic function of the prompt and the weights.

The contract is entirely software: every clause is a configuration of the inference runtime, checkable by reading the model file and the runtime identifiers. No clause requires special hardware. This is the precise sense in which the execution proof is "software-only": both producing it and challenging it need only the model and a CPU.

12.5 The canonical transcript

The activation root alone is not the complete settlement object. A canonical transcript header binds the A-Chain job and policy epoch, kernel-profile version, model, runtime, input root, declared graph, output root, metering summary, provider, destination, and nonce. Its ordered body binds each typed operation to its input and output commitments. The content-addressed root of that serialization is committed before the public challenge is known. A-Chain challenges and finalizes that object; after finality, Z-Chain proves the corresponding receipt transition through P3Q.

CPU/GPU parity is only release conformance for this profile. Golden vectors must match kernel outputs, leaves, transcript roots, challenge derivation, and openings byte for byte. A divergent backend is excluded; consensus never introduces a floating-point tolerance band to accommodate it.

12.6 Soundness

We can now state and prove the central guarantee: a prover who did not perform the genuine matmuls cannot answer a beacon-derived Freivalds challenge except with negligible probability.

Theorem 12.1 (Freivalds soundness for one layer)**.** *Let $A \in \mathbb{Z}^{m\times k}$, $B \in \mathbb{Z}^{k\times n}$ be the true operands of a layer (the committed weights and the committed input activations), and let $\hat{C}$ be the prover's claimed output, committed in the activation trace. Let $r \in \mathbb{F}_p^n$ be sampled uniformly via $\mathsf{PRG}(\mathsf{ch} \,\|\, \ell)$, modeled as uniform and independent of $\hat{C}$ by Lemma 10.2. If $\hat{C} \neq AB$ as integer matrices and p exceeds the magnitude of every honest entry of AB, then*

$$\Pr_r\big[\,\hat{C}\,r \equiv A(Br) \pmod p\,\big] \;\le\; \frac{1}{p}. \tag{21}$$

Proof. Work over $\mathbb{F}_p$; reduce all integer entries mod p. Let $D = \hat{C} - AB \pmod p$. Because p exceeds the magnitude of the honest product's entries, $\hat{C} \neq AB$ over $\mathbb{Z}$ implies $D \neq 0$ over $\mathbb{F}_p$ (a genuine discrepancy cannot be an artifact of modular wraparound). The check $\hat{C}r = A(Br) = (AB)r$ holds iff $Dr = 0$. Since $D \neq 0$, it has some nonzero row D_i, and $Dr = 0$ requires $\langle D_i, r\rangle = 0$. Fix any coordinate j with $D_{ij} \neq 0$. Conditioning on all coordinates of r except r_j, the equation

$\langle D_i, r\rangle = 0$ determines r_j uniquely as $r_j = -D_{ij}^{-1} \sum_{j' \neq j} D_{ij'} r_{j'}$ (the inverse exists since $\mathbb{F}_p$ is a field and $D_{ij} \neq 0$). As r_j is uniform over the p field elements and independent of the others, it hits that single value with probability $1/p$. Hence $\Pr[Dr = 0] \leq \Pr[\langle D_i, r\rangle = 0] = 1/p$. □ □

Two amplifications make the soundness error cryptographically negligible. First, using k_{rep} independent random vectors per layer multiplies the per-layer error to $p^{-k_{\text{rep}}}$ (equivalently, sampling r from a larger space). Second, the verifier checks a random subset of layers; a prover who cheats on even a single layer must survive that layer's check.

Theorem 12.2 (Soundness of the execution proof)**.** *Consider a forward (or backward) pass of L matmul layers committed in the activation trace, verified with $p = 2^{61} - 1$, k_{rep} Freivalds vectors per checked layer, and a uniformly random choice of layers to check. Suppose a prover commits an* activationTraceRoot *and* outputHash *that are* not *the genuine result of evaluating the committed model on the committed input under the determinism contract. Then:*

1. (Dense cheat.) *If the prover's committed trace differs from the genuine trace in the matmul output of* every *checked layer, it passes all checks with probability at most $p^{-k_{\text{rep}}} \leq 2^{-61 k_{\text{rep}}}$; for $k_{\text{rep}} = 2$ this is at most 2^{-122}.*

2. (Sparse cheat.) *If the prover cheats on only a ρ-fraction of layers and the verifier checks s layers uniformly at random, the cheat escapes all checks only if no cheated layer is sampled, which (for cheats confined to few layers) has probability $(1-\rho)^s$; conditioned on a cheated layer being sampled it is caught except with probability $p^{-k_{\text{rep}}}$.*

3. (Determinism binding.) *A committed output hash inconsistent with the committed (and checked) final-layer logits is detected with certainty, because the argmax/tie-break of Section 12.4 is recomputed from the Freivalds-verified logits.*

Proof. Each part composes Theorem 12.1 with independence. For (1), the k_{rep} vectors on a checked discrepant layer are independent, so by Theorem 12.1 the layer passes with probability $\leq (1/p)^{k_{\text{rep}}}$; a dense cheat is discrepant on every checked layer, so the prover must win at least one such layer's check, bounded by $p^{-k_{\text{rep}}}$. For (2), the verifier's s sampled layers miss the cheated fraction with probability $(1-\rho)^s$ (sampling without replacement only lowers this); if any cheated layer is sampled, part (1)'s bound applies to it. For (3), the final-layer pre-activation logits are a matmul output and are Freivalds-checked; the verifier recomputes the argmax under the pinned tie-break and compares to the committed token. Since both are deterministic functions of the verified logits, any inconsistency is detected with probability 1. □ □

12.7 From "probably" to "certainly": interactive bisection

The sparse-cheat term $(1-\rho)^s$ in Theorem 12.2 shrinks with more sampled layers but does not reach zero by spot-checking alone: a cheat confined to a single layer is missed with probability $(1-1/L)^s$. To convert this into *certainty* at logarithmic cost we add an interactive bisection in the style of optimistic rollups [10, 9]. The prover and a challenger disagree on the final output; they recursively bisect the trace—each round the prover commits the midpoint activation, and the parties recurse into the half where they disagree—until they isolate a *single* matmul whose inputs both agree on but whose output they dispute. That one matmul is then either Freivalds-checked or re-executed on-chain (it is a single layer, hence cheap), and the dispute is resolved deterministically: exactly one party is shown wrong. The number of rounds is $O(\log L)$, so the on-chain footprint is logarithmic in the trace length, and the guarantee is upgraded from "caught with overwhelming probability" to "caught with certainty, after $O(\log L)$ interaction."

Corollary 12.3 (Software-only verifiability, by construction)**.** *In the construction as specified (with the trace emitted and the whole-K-i32 kernel of Remark 12.2 in place), producing the execution proof requires the model and the prover's compute; checking it—whether by spot-check (Theorem 12.2) or by bisection to certainty—requires only the model file, a CPU to compute* PRG*, the $O(n^2)$ Freivalds matrix–vector products, and keccak for the trace. No trusted execution environment, GPU attestation, or special hardware is needed by either party. This is a property of the design; the two prerequisites it names (trace emission, pinned whole-K reduction) are the remaining build (Section 20).*

Corollary 12.3 is the property that makes the scheme egalitarian: a CPU-only validator challenges a GPU prover, and any user both generates and verifies a proof against the same network-shared difficulty (the soundness parameters p, k_{rep}, s, and the bond). The emission and the off-chain challenger of Section 14 are built and tested (`poi_transcript.rs`, the Go `crypto/poi` watcher, and the on-chain `ComputeWitnessLib`), so this is a live property for the proof-bearing path, not only an intended one—the contrast with Bitcoin's ASIC-mediated work, drawn out in Section 13.

13 The Bitcoin Correspondence: Shared, Software-Only Difficulty

The companion Thinking-Chains framing establishes that Proof-of-AI is the "AI Bitcoin": it keeps Nakamoto consensus's trustless, permissionless issuance through verifiable work, but replaces useless hashing with useful cognition, and replaces difficulty with a heterogeneous-compute market. This section completes that correspondence at the level this paper is about—*verification*—and reconciles two notions of "difficulty" that the AI setting forces apart.

13.1 The execution proof is the recomputable hash (intended correspondence)

In Bitcoin, the load-bearing line of the whole design is that *any node re-hashes the header and checks the claimed solution in $O(1)$*: a stranger can verify the work without trusting the worker and without redoing it. Sections 10–12 *specify* the cognitive analogue. The activation-trace commitment is *designed to be* the value everyone can recompute; the Freivalds check is the cheap verification; and Theorem 12.2 is the guarantee that the value *cannot be produced without doing the work*. The difference is that Bitcoin's recomputable value is the output of useless hashing, whereas ours is the side effect of an answer a paying agent demanded. Table 5 draws the mechanism-for-mechanism correspondence at the verification layer; its right column is the design Sections 10–12 specify, of which only the Freivalds-check row is implemented today (the recomputable value is not yet emitted; Section 2).

13.2 Two difficulties: price and verification

Bitcoin has *one* difficulty because it prices and verifies *one* homogeneous resource (SHA-256 hashpower): the same scalar tunes how hard it is to *find* a block and is implicit in how the network *checks* it. Intelligence is not homogeneous—it is produced on GPUs, CPUs, unified-memory Apple silicon, and accelerators differing by orders of magnitude in memory, bandwidth, and throughput—so Proof-of-AI must split the single scalar into two orthogonal quantities, and keep both.

Difficulty-as-price (the HMM).
The cost of a thought is set by the Hamiltonian Market Maker, a physics-based AMM over a heterogeneous-compute reserve manifold (GPU-hours, VRAM, CPU, bandwidth, storage) that places each task on the cheapest capable hardware and quotes a continuous, capital-efficient price.

Table 5: Bitcoin and Proof-of-AI at the verification layer. The execution proof of this paper is the *designed* cognitive counterpart of the recomputable hash, software-only and over useful work; the Proof-of-AI column is the specification, not a deployed system (only the Freivalds verifier row is built today).

	Bitcoin (PoW)	Proof-of-AI (this paper)
The recomputable value	the block-header SHA-256d	the activation-trace Merkle root + output hash *(emitted by* `ProvableLinear`/`GraphTrace`*)*
What it proves	$\sim 2^d$ hashes were tried	the genuine matmuls were performed (Thm. 12.2; emitted and verified end to end)
Verifier cost	$O(1)$: one hash	$O(n^2)$ Freivalds, or $O(\log L)$ bisection to certainty
Hardware to verify	any CPU	any CPU (Cor. 12.3)
Hardware to *produce*	ASIC (centralizing)	any AI accelerator *or* CPU; no TEE required
Unforgeability root	preimage resistance of SHA-256	Freivalds soundness + keccak collision resistance
Freshness / anti-replay	previous block hash in header	`openBlockHash` beacon in the challenge (Lem. 10.2)
Product of the work	none (discarded)	the answer / embedding / improved weights

This is the generalization of Bitcoin's difficulty adjustment to a market: as compute joins, the price of cognition falls and re-stabilizes, exactly as Bitcoin's difficulty rises to hold block time as hashpower joins. The HMM answers *how much should this work cost?*

Difficulty-as-soundness-bar (this paper).
The strength of the verification is set by the network-shared soundness parameters: the field size p, the number of Freivalds vectors k_{rep} and sampled layers s (Theorem 12.2), and the bond a prover escrows (Section 14). These are uniform across all provers—a CPU validator and a GPU farm face the same bar—and adjustable like Bitcoin's difficulty: as the value at stake or the adversary's resources grow, the network raises k_{rep}, s, or the bond. The soundness bar answers *how sure must we be the work was done?*

The two are genuinely orthogonal: pricing concerns *which heterogeneous resource* clears the demand and at what rate; verification concerns *whether the claimed computation occurred*, and is hardware-agnostic by Corollary 12.3. Conflating them is the error the deployed settlement engine of Section 9 makes implicitly—it prices and pays for agreement while verifying nothing about computation. Keeping them separate is what lets the protocol *price* work across the messy real world of heterogeneous compute while *verifying* it with a single, uniform, software-only bar.

Principle 13.1 (Pricing $\perp$ verification)**.** *The HMM prices the work; the execution proof verifies it. The two difficulties are independent dials: the market sets what cognition costs across heterogeneous hardware, and the soundness bar sets how certain the network is that the cognition occurred. Bitcoin needed only one dial because its work was homogeneous and useless; Proof-of-AI needs both because its work is heterogeneous and useful.*

13.3 Software-only difficulty is egalitarian difficulty

Bitcoin's difficulty, though uniform as a number, is *not* egalitarian in practice: producing the work economically requires ASICs, which centralizes mining among those who can fabricate or buy them. Proof-of-AI's soundness bar is *designed to be* uniform *and* egalitarian, because the work it gates can be both produced and verified in software. By Corollary 12.3, a CPU-only participant *would* verify any proof and—on the optimistic-Freivalds and redundancy tiers (Section 14)—also *produce* proofs for models it can run, challenge proofs it suspects, and earn by catching fraud. This egalitarian property is contingent on two unbuilt pieces: the trace emission, and a permissionless off-chain challenger/re-executor that opens a suspect transcript and disputes it. *Neither exists today*, so the "earn by catching fraud" incentive—and with it the economic argument that a liar always has skin in the game—is a property of the designed system, not the current one. Once both are built, the "hash everyone can recompute" becomes a computation everyone can both perform at their scale and check at CPU cost—the property Bitcoin aspired to and ASIC economics eroded.

What the network mines, in the end, is useful cognition; what it hashes to is a shared, recomputable commitment; and what it adjusts as compute joins is two dials instead of one—price, by the HMM, and certainty, by the soundness bar of Theorem 12.2.

14 The Tier Matrix: One Gate, Migrating Backends

The execution proof of Section 12 is one of several proof systems a deployment can require, trading off trust assumptions, cost, and hardware. This section specifies the menu and the single gate that selects from it. The discipline is the one used for post-quantum migration elsewhere in the stack: the policy "what proof does a task at this tier require?" lives in *exactly one place*, every consumer reads it the same way, and the proof backends are interchangeable behind a common binding so that the menu can change without touching the binding or the callers.

14.1 The proof backends

A proof system is identified by a proofType and registered as a backend that attests to a reportData (Section 10). The deployed registry recognizes four:

$$\mathsf{proofType} \in \{\ \underbrace{1}_{\text{CC-TEE}}\ ,\ \underbrace{2}_{\text{zkML}},\ \underbrace{3}_{\text{optimistic-Freivalds}}\ ,\ \underbrace{4}_{M\text{-of-}N\ \text{TEE}}\ \}. \tag{22}$$

Each backend exposes the same interface—"does this witness attest to this reportData?"—and the verifier dispatches on proofType, failing *closed* on any unregistered type. The optimistic-Freivalds backend (type 3) is the software-only path of this paper: a prover escrows a bond and commits (reportData, activationTraceRoot, modelWeightsRoot); within a challenge window any party may dispute by supplying a Freivalds response (or, on dispute, by driving the bisection of Section 12); an unchallenged commitment finalizes and the bond is reclaimable, while a proven fraud slashes the bond to the challenger. The heavy check runs off-chain in the inference engine (a watcher re-executes the committed trace and opens any discrepant matmul); the chain runs the economic state machine the check's verdict drives, and—when the dispute is submitted—re-runs the Freivalds check itself on-chain so the verdict is not a watcher's say-so. Both halves are built and tested (Section 20): the off-chain re-executor and challenger (`poi_transcript.rs` in Rust, `crypto/poi` in Go) open a transcript and produce the dispute, and the on-chain state machine (`OptimisticEvidence.sol`, 18/18) records the bonded commitment, verifies the exhibited fraud via `ComputeWitnessLib.provesFraud`, and slashes only on a verified discrepancy. The optimistic backend's economic soundness—a liar is

Table 6: Tier → proof matrix. One profile gate (Section 14.3) maps a tier to its required proofType; where a tier is opted in, no valid proof of that type means no mint and no settlement. The gate is built but defaults to permissive (requiredProofType = 0), and the software-only backend (T2/T3) awaits the emission and challenger of Section 20; the matrix is thus the configured policy surface, live enforcement following the build.

Tier	Proof system	Trust assumption	Use
T0	zkML (type 2)	none beyond the SNARK's	tiny models; trustless, on-chain succinct verification
T1	M-of-N heterogeneous CC-TEE (type 4)	no single TEE vendor; M of N enclaves agree	high-value tasks; defense in depth across NVIDIA CC + Intel TDX + AMD SEV-SNP
T2	single-vendor CC-TEE (type 1) *or* optimistic-Freivalds (type 3)	one TEE vendor, *or* one honest challenger + a bond	the workhorse tier; software-only Freivalds is the intended egalitarian default (pending emission + challenger)
T3	redundancy + plurality + slash	honest plurality among bonded operators	liveness-first; CPU-only providers; the weakest proof, fastest path

strictly unprofitable once watched, an honest prover is never slashable—is therefore a live deterrent: a fabricated matmul is provable except with probability $\leq 2^{-122}$, and the bond pays the challenger that proves it.

14.2 The tiers

Tiers map a task's value and trust requirement to a required proof strength.

The matrix is read top-down by trust and bottom-up by accessibility. T0 (zkML) assumes nothing beyond the soundness of the proof system but is limited to small models by proving cost. T1 (M-of-N heterogeneous confidential compute) tolerates the compromise of any single TEE vendor by requiring agreement across enclaves from *different* hardware roots—NVIDIA confidential computing, Intel TDX [16], AMD SEV-SNP [17]—so that a break of one vendor's attestation does not forge a proof. T2 is the workhorse: a single-vendor TEE for those who have the hardware, *or* the optimistic-Freivalds path of this paper for those who do not, which assumes only that at least one honest party will challenge a fraud within the window and that the prover's bond exceeds the gain from cheating. T3 falls back to the redundancy-and-plurality settlement of Section 9, retained only where liveness dominates and the value at stake is low enough that honest-plurality is acceptable—and, crucially, *even T3 can now require a sampled execution proof* from the plurality operators, closing Proposition 9.1 at that tier too.

14.3 The profile gate: one place

The mapping from tier to required proof is a single function in a single contract, the on-chain analogue of the `RefuseUnderStrictPQ` gate used for post-quantum policy. It exposes one read,

$$\mathsf{requiredProofType}(\mathsf{tier}) \;\rightarrow\; \mathsf{proofType}, \tag{23}$$

returning 0 (permissive) for any tier governance has not explicitly opted in, and otherwise the mandatory proofType. The settlement path consults this single oracle and enforces the invariant directly:

> *No valid compute proof of the required type ⇒ no mint, no settlement.*

This is the constructive answer to Section 9: the verifier's three-step gate—(1) the proof's reportData equals the task binding, (2) the runtime is governance-accepted, (3) the registered backend attests—runs *before* any reward is paid, and fails closed at every step *on any tier that has been opted in*. Where the gate is engaged, reward is bound to proof, not to signature and not to plurality; where it is not (the default), settlement remains the authorship-plus-agreement rule of Section 9, and—until the execution backend's emission and challenger are built—the only proof type a gate can actually enforce is an attestation envelope, not the activation-trace execution proof. Because the policy lives in one place, there is exactly one way to ask "what does this tier require?", and changing a tier's requirement is a single governed write rather than a scattered edit.

14.4 Migration without rebinding

The tiers are not static. The decisive structural property of this design is that *the binding does not depend on the proof system*: reportData (Section 10) commits the task, model, input, output, and runtime, and says nothing about how that commitment is believed. Consequently, as zkML proving cost falls and larger models become trustlessly provable, tasks migrate from T2/T3 up to T0 by a governance change to requiredProofType alone—*zero change to the binding, zero change to the receipts, zero change to the callers.* The same holds in reverse: a deployment that gains confidential-compute hardware can move tasks from software-only T2 to TEE-backed T1 without touching the proof artifacts that earlier tasks emitted. The proof backends are orthogonal slots behind a stable binding; the menu evolves, the interface does not. This is the property *designed* to let a permissionless network start egalitarian (software-only Freivalds for everyone, once that backend's emission and challenger are built) and strengthen task-by-task as trustless proving matures, without ever stranding the proofs it has already minted against.

15 Ownership: Data, Models, and Weights as User-Owned Assets

> *Status note. The cryptographic core of this section is now built. The confidentiality envelope—a single-shot prompt seal under* ***X-Wing*** *(the hybrid X25519 + ML-KEM-768 KEM, RFC 9180 HPKE base mode), so the prompt is ciphertext to everyone but the chosen operator—is implemented and tested (*`luxfi/crypto/promptseal`*, 6 tests passing), and the in-enclave custody binding (a TEE quote that binds the operator's confidentiality key to an attested image) is implemented and tested (*`luxfi/crypto/attestation`*, 9 tests passing); both are detailed and proven in the emission companion [25]. The proof emission these compose with is also built (Section 20). What remains forward-looking is the* product *layer built on top—data/model/weight marketplaces, the full user-owned-asset NFT surface, and the plaintext-blind confidential-compute tier wired into every operator path; modes (2) and (3) below describe that intended envelope. Mode (1)—the user runs the model on their own hardware—is non-custodial unconditionally. We keep the design framing for the product surface and state plainly that the sealing and attestation primitives underneath it are deployed.*

The execution proofs are intended to make a stronger claim possible than "the network computed correctly." They are designed to make *sovereignty* auditable: a user *would* own their data, own the inference performed on it, and own the model weights they improve, and *would* choose *per request*

how much trust to place in whoever runs the compute—without the choice ever weakening the guarantee that the work was genuine. The aim is that trust become a UX dial, not a security compromise. With the confidentiality envelope unbuilt, that decoupling holds today only in mode (1).

15.1 The trust spectrum

For any request, a user selects one of three execution modes, ordered by decreasing sovereignty and increasing convenience. Every mode emits the same execution proofs of Sections 11–12, so every mode is auditable; the modes differ only in *who holds the data in the clear* and *who the user must trust for liveness*.

(1) User-operated node — maximal sovereignty.
The user runs the inference on their own hardware. The data never leaves the device; the model and weights are local; the proof is emitted for the user's own records or for third parties who later wish to verify a result the user publishes. Trust in any operator is zero because there is no operator. This is the floor of the spectrum and the reference against which the others are measured.

(2) DAO trustless nodes — verify, don't trust *(unbuilt).*
The user *would* send work to permissionless DAO operators under confidential compute (T1/T2) so the operator *would be plaintext-blind*: processing ciphertext inside an enclave and never seeing the data in the clear, while the execution proof (PoI/PoT) plus the slashing bond would make a wrong or fabricated result detectable and punishable. *As deployed today, none of this confidentiality holds*: there is no enclave ciphertext path and the operator receives plaintext. In the intended mode the user trusts no particular operator's honesty—only the proof system and, for the TEE tiers, the hardware root. This is the designed "verify, don't trust" mode: sovereignty over correctness and confidentiality without running the hardware oneself.

(3) Selected trusted operators — convenience.
The user picks specific operators by reputation (via the identity-and-reputation layer), accepting a centralized-feeling, faster, cheaper path. Even here the proofs remain emittable, so the user retains an audit trail: a trusted operator that misbehaves can be caught after the fact by anyone who checks the proof, and loses reputation and future routing. Trust is placed deliberately and remains accountable.

The three modes are points on a continuum, not separate products, because they share the binding and the proof primitive. A user can move a workload from mode (3) to mode (2) to mode (1) as their trust requirements rise, with no change to the artifacts the workload produces.

15.2 Data confidentiality at rest: primitives built, operator integration pending

The cryptographic primitives are built: in modes (2) and (3) the prompt is sealed under X-Wing (X25519 + ML-KEM-768) HPKE to the chosen operator's published key (`crypto/promptseal`, tested), committed on-chain only by hash, and—in the confidential-compute tier—decryptable only inside an enclave whose attested image is bound to that key (`crypto/attestation`, tested; [25]). What is *not* yet built is the *integration* of these primitives into the production inference operator path: the seal/open is not yet wired into every operator's request handler, nor the per-operator key-management lifecycle, so in the currently deployed operator path the prompt may be held in the clear and the "never plaintext at rest" property is guaranteed only in mode (1) and wherever

the sealing path is enabled. The soundness binding is independent of this and is real: the execution proof binds to inputHash (Section 10), so a proof certifies that the *committed* input was processed while revealing nothing about the input itself. That binding is what makes PoC's privacy clause (Definition 11.2) more than a promise: the contributor proves the gradient was computed over data consistent with the committed policy (the training-step graph is Freivalds-checked, Section 20) and shares only the resulting delta, so the corpus stays private while the contribution stays verifiable. The confidentiality envelope (X-Wing prompt seal + attestation) and the proof emission it relies on are both built and tested (`crypto/promptseal`, `crypto/attestation`, the engine `poi_*` modules; Section 20); the corpus-scale data-policy product around them is the forward-looking part.

15.3 Weights are a portable, user-owned asset

A model improvement—the ΔW of PoT/PoC—is serialized in the gym's canonical backend-independent BF16 format and committed by its root (Section 11). Because the serialization is canonical and backend-independent, a delta trained on one accelerator is byte-portable to any other *(this part is real—it is the gym's shipping format)*, and its provenance (which base model, which data policy, what improvement margin) *would be* attested by the PoC it was minted with *(this part depends on the unbuilt proof emission)*. The end state is that a user's weights are an on-chain asset they own and can move, sell, or compose—with royalties flowing to the contributors whose proven improvements it incorporates—rather than an artifact locked inside a single provider's stack. The proof framework is what gives the asset its value: a delta is worth more when it carries a verifiable proof that it was genuinely trained and genuinely improves the model, than when it is merely asserted to.

15.4 Trust as a dial, security as a floor

The synthesis the design aims at is that the execution proofs decouple *trust* from *security*. A user would dial trust up the spectrum—from running everything themselves, to trusting a proof system, to trusting a named operator—according to their convenience and threat model, and the proof framework would hold the security floor constant underneath: in every mode, a result that claims to be the model's output on the user's input either is, or is caught. This is the intended difference between the deployed settlement engine, where "trusting the operators" means accepting unverified guesses (Section 9), and a proof-backed network, where trusting an operator would be a choice about *speed and cost*, never about *correctness*. *This decoupling is not yet in force*: with neither the proof emission nor the confidentiality envelope built, mode (1) (self-hosting) is the only mode that today delivers either sovereignty or non-custody. The end state is that the user owns the data, the inference, and the weights, and the protocol's job is only to make all three auditable, at whatever point on the trust spectrum the user selects; reaching it is the build of Section 20.

16 The Formal Core: Receipt Soundness

The preceding sections build the execution proof (Sections 12, 11) and its binding (Section 10). This section states what PoAI *formally guarantees*, as a theorem over a single universal object, and—equally important—draws the line at what it does *not* claim. The discipline is deliberate: PoAI proves *objective lifecycle properties*, never semantic ones.

16.1 What PoAI proves, and what it refuses to claim

PoAI proves exactly five properties of an AI contribution:

1. **Identity** — which model/artifact lineage and which engine produced the work.
2. **Execution** — that the committed computation actually ran (not a cheaper substitute).
3. **Provenance** — the input, output, and lineage the work is bound to.
4. **Accounting** — that the work is metered and counted exactly once.
5. **Settlement** — that payment, mint, and governance are bound to the proof.

PoAI does *not* prove that an answer is true, that a model is aligned, that a dataset is morally good, or that an output is useful. Those are semantic judgments about human intent; as the verification literature establishes, every such verifier is a proxy for intent that degrades as the generator's capability grows [26], and no fixed reward function for them stays sound. PoAI sidesteps that impossibility by anchoring economics to the one signal that does *not* decay—*did the claimed work happen*—which is checkable to the bit (Section 12). The semantic remainder is pushed to an adaptive, governed, reputation-weighted market (the quorum tier and `AIReputation`); PoAI makes no theorem about it.

Principle 16.1 (Objective, not semantic)**.** *PoAI proves that claimed AI work was* performed under committed artifacts*, and that payment, mint, and governance are* bound to that proof*. It proves nothing about the meaning, correctness, or value of the result.*

16.2 The universal object

Every AI contribution—inference, embedding, training, fine-tuning, quantization, storage, streaming, evaluation, data—settles through one object, `AIReceipt`. A receipt R carries the job id, job type, requester, contributor, the model/engine/runtime roots, the input and output commitments, the witness root, the proof tier, the metering, and the license/attribution commitments. Its canonical commitment is the two-step keccak chain of Section 10: a per-job-type envelope folds the type's extra fields into *promptHash*, *outputHash*, and *runtimeMeasurement before* the unchanged `ComputeProofLib` chain runs, so that for INFERENCE the binding reduces *byte-for-byte* to the candidate consensus encoding, and every job type domain-separates so two receipts of different types can never collide:

$$\mathit{reportData} \;=\; \mathsf{keccak}\big(\mathsf{DOMAIN} \,\|\, \mathit{challenge} \,\|\, \mathit{modelSpecHash} \,\|\, \mathit{promptHash} \,\|\, \mathit{outputHash} \,\|\, \mathit{runtimeMeasurement}\big)$$

with $\mathit{receiptId} = \mathsf{keccak}(\mathsf{DOMAIN\_RECEIPT} \,\|\, \mathit{reportData} \,\|\, \langle\text{the full economic commitment}\rangle)$. The value path joins on *reportData*; the audit log emits *receiptId*.

16.3 The contract surface

The object is realized by a small, orthogonal contract set (one concern each):

`AIJob`	request + escrow
`AIReceipt`	the canonical commitment (binding library + universal struct)
`AIProof`	read-only proof-route verification (the finalization gate)
`AISettle`	pays / mints / slashes; the sole consequence authority
`AIModel`	model lineage and roots (verifiable, non-equivocating)
`AIEngine`	approved engine/runtime roots
`AIData`	licenses, attribution DAG, royalties, public-goods pool
`AIStore`	artifact availability / delivery proofs
`AICompute`	global no-double-mint accounting

The dataflow is one way:

`AIJob` → `AIReceipt` → `AIProof` → `AISettle` → `AICompute`, with `AIData`/`AIModel`/`AIEngine`/`AIStore` consulte

16.4 Proof tiers and the finalization gate

Verification is tiered (Section 14), strongest-binding last:

$$\mathsf{L0\ signed} < \mathsf{L1\ stake{+}challenge} < \mathsf{L2\ quorum} < \mathsf{L3\ sampled\ execution\ witness\ (Freivalds)} < \mathsf{L4\ TEE\ quote} < \mathsf{L5}$$

Each job type carries a governance-set floor, and `AIProof` additionally binds the proof *system* to the job's *work type* (a level gate alone is insufficient—a storage receipt must not settle through the inference matmul backend). Finalization requires, for the job type τ of receipt R,

$$\mathsf{ProofType}(R) \in \mathsf{AllowedProofTypes}(\tau) \ \wedge \ \mathsf{ProofLevel}(R) \geq \mathsf{MinProofLevel}(\tau),$$

and the routed backend's witness must verify. No verification, no finalization; an unwired or disabled backend *fails closed*.

16.5 Provable job classes

Each class has its own verifier; all settle through the one receipt. The verifier proves the *objective* statement, never a semantic one:

Class	**Proves**
Inference	model M ran on input X and committed output Y
Embedding	embedding model E ran on X and committed vector V
Training	dataset D, recipe ρ, base B produced checkpoint C under a committed transcript
Fine-tuning	base B + adapter/delta A + data D produced M'
Quantization	model M transformed into quant artifact Q under a committed quant recipe
Storage	artifact A was retrievable over time
Streaming	pieces P were delivered to receivers
Evaluation	benchmark β and scorer S produced result Σ
Data	data commitment D exists, is licensed, deduped, attributable, and accepted

16.6 Invariants

The following hold for every finalized receipt. Each is enforced on-chain; the status column marks whether it is established *by construction* (C), pinned by the test suite (T), or—for the governance-consumption and storage/stream availability wiring—staged at the integration seam (S).

	Invariant	Status
I1	No receipt, no payment.	C,T
I2	No receipt, no mint.	C,T
I3	No receipt, no governance execution.	C,S
I4	One job cannot settle twice.	C,T
I5	One receipt cannot mint twice.	C,T
I6	Receipt cannot change model after submission.	C,T
I7	Receipt cannot change input after submission.	C,T
I8	Receipt cannot change output after submission.	C,T
I9	Receipt cannot be replayed across chainId/domain.	C,T
I10	Proof backend cannot alter the receipt binding.	C,T
I11	Backend verifies the witness only.	C,T
I12	`AISettle` cannot bypass `AIProof`.	C,T
I13	`AIGovern`/`AIExecute` consume only finalized receipts.	C,S
I14	Royalties cannot exceed 100%.	C,T
I15	Storage/stream rewards require an availability/delivery proof.	C,T
I16	Training/fine-tune lineage binds base model, data, recipe, checkpoint.	C,T

I4–I8 follow from the commitment: *receiptId* binds the job id, model, input, output, contributor, and tier, so any change yields a different receipt that cannot settle against the opened job. I5 and I9 are the one-shot *consumed*$[\cdot]$ flag (keyed per job type) and the chain-independent `AICompute` claim key. I10–I12 are the read-only verifier seam: the gate pre-enforces *reportData* = *expected* and routes only the witness to the backend, and `AISettle`'s sole mint/pay path is gated on `AIProof`. I14 is the $\Sigma\, bps = 10{,}000$ split with a slice-capped royalty reservation. I15/I16 are the proof-system gate of Section 16.4 (a storage receipt admits only the retrievability backend) and the domain-separated training fold bound to the lineage registry.

16.7 Soundness theorems

Theorem 16.1 (Receipt Soundness)**.** *For any finalized `AIReceipt` R, assuming the selected proof backend is sound, R is a unique, non-replayable, non-malleable commitment to an AI contribution that binds to exactly one job, exactly one model/artifact lineage, exactly one input commitment, exactly one output commitment, exactly one proof tier, and exactly one contributor; whose required proof verified* before *economic settlement; and from which at most one payment and at most one mint can issue. R cannot be replayed, substituted, double-paid, or double-mined.*

Proof sketch. Finalization requires `AIProof.validate` (the proof-system gate of Section 16.4 plus the routed backend) and, for the pessimistic tiers, *finalized*(*reportData*). Uniqueness and non-malleability: *receiptId* is the keccak commitment to all of {job, model, input, output, contributor, tier} (I6–I8), so a substitution is a distinct receipt that fails the job-match check; collision resistance of keccak gives uniqueness. Cross- domain non-replay: *reportData* binds *networkID* and *chainID* (I9). At-most-once settlement: the *consumed* one-shot and the `AICompute` claim (I4, I5) are set inside the same state snapshot as the mint, so a revert un-sets them and a second attempt reverts. The backend cannot forge the binding (I10, I11): the verifier pre-enforces equality and passes only the witness; soundness of R therefore reduces to soundness of the selected backend, which is the hypothesis. `AISettle` has no mint or pay path that is not gated on `AIProof` (I12). □

Corollary 16.2 (Inference Soundness)**.** *A finalized inference receipt binds model, engine, input, output, and a verified proof; its reportData equals `ComputeProofLib.expectedReportData` byte-for-byte, so it matches the candidate A-Chain verifier encoding.*

Corollary 16.3 (Training-Lineage Soundness)**.** *A finalized training (resp. fine-tune) receipt binds base model, dataset, recipe, transcript, and checkpoint (resp. adapter/delta): the* TRAIN *fold commits the dataset and lineage roots, and the A-Chain proof verifier re-derives a beacon-sampled SGD step over the int8 accumulator, so a fabricated checkpoint is caught except with the per-step Freivalds error.*

Corollary 16.4 (Accounting Soundness)**.** *A finalized receipt produces at most one payment and at most one mint, across all mechanisms and all chains: the per-type consumed one-shot, the chain-independent* `AICompute` *claim, and the shared halving cap make a second issue revert.*

Corollary 16.5 (Governance Safety)**.** *No* PoAI*-governed action executes from an unfinalized or invalid receipt:* `AIGovern`*/*`AIExecute` *consume only a finalized reportData (I3, I13), which by Theorem 16.1 carries a verified proof.*

16.8 The honest boundary

Stated plainly, so the guarantee is defensible:

> *PoAI does not prove an answer is true. It does not prove a model is aligned. It does not prove a dataset is morally good. PoAI proves that claimed AI work happened under committed artifacts, and that payment, mint, and governance are bound to that proof.*

Remark 16.1 (The one-line truth)*.* PoAI is not proof that AI is right. PoAI is proof that AI work was *performed, bound, attributed, verified, and fairly settled.*

17 Full Modern-LLM Coverage: Tiering, Not Floating-Point Freivalds

The execution proof of Section 12 is exact because it runs over the int8 path's integer accumulator. A natural objection is that "real" frontier models run in floating point (fp16, bf16, fp8), so an exact-integer verifier covers only the quantized deployment and not the full precision a model was trained in. This section answers the objection rigorously, and the answer is the crux of the design: we do *not* force Freivalds onto floating point, because naive floating-point Freivalds is simultaneously *unsound* and *unusable*. Instead, coverage of the full model space is *designed to be* achieved by *tiering* the verification mode to the arithmetic, so that every model has a proof-bearing mode while no mode pays a soundness or liveness penalty it cannot survive. The quantized, exact, software-only mode is then not a limitation but a *feature*.

Two status caveats, stated before the argument. (i) The exact-integer mode this section relies on requires a whole-K integer accumulator. This is *built and is the proof path*: `poi.rs::exact_matmul` is a whole-K $i64$ reduction with no cross-block float rescale, and `ProvableLinear` runs its matmuls there. The distinction is that the engine's *fast* kernel (`f8q8`) accumulates across blocks in `f32`—non-associative and cross-hardware non-reproducible (Remark 12.2)—so it is used for ordinary inference, while a proof-bearing inference pays the exact accumulator's throughput cost to get the "$\epsilon = 0$, zero false-reject" property. Running exact Freivalds over the `f8q8` fast path would false-reject honest provers; running it over `exact_matmul` does not, which is why the proof path is the integer path. (ii) The quantized models are the arithmetic the zoo *deploys in*, and the families the test suite exercises (dense/SwiGLU, attention with integer softmax, MoE, training) now *emit a proof transcript*—so provable-today coverage is nonempty, not 0%. What remains *designed* is the breadth: the floating-point (RepOps) tier and the architecture families not yet wired to emit (Section 2).

17.1 Why naive floating-point Freivalds fails twice

Floating-point matrix multiplication breaks the Freivalds check in two independent ways, one fatal to *liveness* and one fatal to *soundness*. Both stem from the same root cause: an fp GEMM is not a function of its operands alone.

(a) Non-determinism ⇒ false-reject (liveness break). Floating-point addition is not associative: $(a + b) + c \neq a + (b + c)$ in general under IEEE-754 rounding. A GEMM accumulates k products into each output entry, and the *order* of that accumulation is chosen by the kernel—tiling, the number of threads in a reduction, whether a fused multiply–add (FMA) is emitted, the warp/wavefront width, and the vendor's library version all permute it. Two honest evaluations of the same A, B in fp16 on two different GPUs, or on a GPU versus a CPU, therefore yield outputs C and C' with $C \neq C'$ in the low mantissa bits. A Freivalds check demands the equality $Cr = A(Br)$ over the *produced* C; with $C \neq C'$, an honest re-deriving verifier on different hardware computes $A(B'r) \neq Cr$ and *rejects honest work*. In a permissionless network where a CPU validator must be able to challenge a GPU prover (Corollary 12.3), this false-reject is not a tail event—it is the *common* case, because heterogeneous hardware is the premise. A verification rule that rejects honest provers is a liveness failure regardless of its soundness.

(b) Tolerance bands ⇒ a perturbation corridor (soundness break). The naive "fix" for (a) is to accept when $\|Cr - A(Br)\|_\infty < \epsilon$ for some tolerance ϵ absorbing cross-hardware rounding. This trades the liveness failure for a soundness failure. The acceptance region is no longer the point $\{C = AB\}$ but a *band* of radius ϵ around it, and a cheating prover may roam anywhere inside that band. Concretely, an adversary who wishes to claim a fabricated output $\hat{C}$ chooses any $\hat{C}$ with $\|(\hat{C} - AB)r\|_\infty < \epsilon$; the Freivalds projection r collapses the $m \times n$ discrepancy matrix $D = \hat{C} - AB$ to the m-vector Dr, so the adversary needs only Dr small, not D small—an under-determined constraint with a large solution space. The corridor is *exploitable*: it lets a prover skip or approximate work (e.g. run a smaller model, a lower-rank approximation, or a truncated accumulation) and land inside ϵ, harvesting the fee while the band hides the cheat. Worse, the corridor must be *widened* to cover legitimate cross-hardware drift, and every widening enlarges the cheating manifold. There is no setting of ϵ that admits all honest cross-hardware runs (requires ϵ large) while excluding all profitable cheats (requires ϵ small): the two requirements pull in opposite directions, exactly as in the analogous attack on tolerance-based optimistic ML verifiers. We formalize this dilemma in the companion proofs document (the floating-point soundness/liveness no-go).

Remark 17.1 (The integer path is not a downgrade—it removes both failures at once). The integer accumulator (Section 12) makes the matmul a function of its operands alone: i8×i8→i32 accumulation is associative and bit-identical on every architecture. This simultaneously kills (a)—two honest provers compute the *same* C, so $\epsilon = 0$ and there are zero false rejections—and (b)—with $\epsilon = 0$ the acceptance region collapses back to the exact point $\{C = AB\}$ and the corridor vanishes, restoring the clean $1/p$ soundness of Theorem 12.1. This is why the proof-bearing mode is quantized by *necessity*: exact arithmetic is the only arithmetic for which a software-only, cross-hardware, tolerance-free equality check exists.

17.2 The key fact: floating-point values are integers

The tiering does not abandon floating-point models; it gives them an exact mode too, via a fact that is easy to miss. An IEEE-754 [19] value is *already an integer encoding*: a sign bit, an exponent field, and a mantissa field, all integers. For fp16/bf16/fp8 the significand is a small integer (an 11-, 8-,

or 4-bit mantissa plus the implicit leading one), and a product of two such values, *before rounding*, is an exact integer times a power of two. The IEEE rounding that follows is a *deterministic, total function* of the exact product and the rounding mode—it is not a source of nondeterminism. We make this precise.

Proposition 17.1 (Floating-point GEMM is exact integer-mantissa arithmetic up to a fixed reduction order)**.** *Fix an IEEE-754 format F (e.g. bf16) with round-to-nearest-even, and fix a* reduction tree τ*: a binary tree on the k index positions specifying the exact order in which the k products of a dot product are summed and rounded. Then for any operands A, B in F, the GEMM output* $\mathrm{GEMM}_F^\tau(A, B)$ *is a deterministic function of (A, B) alone, computable by integer arithmetic on the mantissa/exponent fields with an IEEE rounding step at each tree node, and identical on every machine that honors F and τ. The* only *freedom in a floating-point GEMM is the choice of τ; once τ is pinned, the result is bit-exact and hardware-independent.*

Argument. Each product $a_i b_j$ of two F-values has a significand that is the integer product of the two integer significands and an exponent that is the integer sum of the two exponents; this is exact in $\mathbb{Z}$ (no rounding) because integer multiplication is exact. A tree node combines two already-computed partial sums u, v (each an F-value, hence an integer significand $\times 2^e$): align exponents (an integer shift), add significands (an integer add), then apply round-to-nearest-even (a total function of the exact integer sum and the format's precision). Every operation is integer arithmetic plus a deterministic rounding decision, so the node output is a deterministic function of (u, v). By induction over τ, $\mathrm{GEMM}_F^\tau(A, B)$ is a deterministic function of (A, B, τ). The associativity counterexample of Section 17.1 is precisely a disagreement in τ between two kernels; fixing τ removes it. □ □

Proposition 17.1 isolates the entire problem to one degree of freedom: *accumulation order*. It says nothing about precision loss—an fp16 GEMM still rounds, and rounding still discards low bits—but it says the rounding is *reproducible*: every honest prover that uses the pinned tree τ produces the same rounded bits. This is the hook on which an exact floating-point Freivalds hangs.

17.3 RepOps: pinning the reduction tree makes floating-point bit-exact

The deterministic-floating-point ("RepOps", for *reproducible operators*) verification mode for the fp16/bf16/fp8 tier is then: *pin the reduction tree τ as part of* runtimeMeasurement (Section 10), so that the floating-point GEMM becomes bit-exact across hardware by Proposition 17.1, and run Freivalds over the resulting exact values. Because τ is fixed, $\epsilon = 0$: there is no cross-hardware drift to absorb (failure (a) is gone) and no tolerance band to exploit (failure (b) is gone). The Freivalds soundness of Theorem 12.1 applies verbatim, over the exact (rounded, but reproducibly rounded) products—formally, one runs the check over $\mathbb{F}_p$ on the integer significand–exponent representation of the pinned-order output, with p chosen to bound the dynamic range of the accumulated significands.

The cost is concrete and is the reason RepOps is a *tier*, not the default: pinning τ forfeits the vendor's hand-tuned GEMM kernels, whose speed comes precisely from choosing τ freely (large tiles, many-thread reductions, FMA fusion, mixed precision). A reproducible kernel fixes the reduction order and so runs slower than the vendor optimum—the standard reproducible-summation trade-off [18]. The deployment therefore pays for fp-precision proofs in throughput; whether that is worth it is a per-task governance choice (Section 14.3), not a protocol constant.

17.4 The coverage tiers, by arithmetic

Putting the pieces together, every model has at least one proof-bearing mode, selected by its arithmetic and value. Table 7 states the menu. It refines, rather than replaces, the trust-tier matrix of

Table 7: Coverage by arithmetic. Every modern-LLM arithmetic is given a proof-bearing mode by tiering the verification to the arithmetic, not by a single floating-point Freivalds. The quantized row is the egalitarian, software-only, hardware-free default. Its exact-integer verifier and reference emission fixtures are *built and tested*: they run the whole-K exact-integer accumulator (`exact_matmul`) that gives the $\epsilon = 0$, zero-false-reject property, emits a transcript (`ProvableLinear`/`GraphTrace`), and verifies on-chain (`ComputeWitnessLib`). The floating-point (RepOps) row is *designed* (proven sound in [24]), awaiting the pinned-reduction kernel; the TEE row is the hardware-trust fallback (Section 2).

Arithmetic	Verification mode	Property
int8 / int4 / GGUF / AWQ / GPTQ / MLX-quant	**exact-integer Freivalds** (Section 12)	$\epsilon = 0$, software-only, no hardware; soundness $1/p$ per vector (2^{-122} dense); zero false-reject for the versioned whole-K exact profile; production serving integration remains activation-gated and the block-`f32` fast path is not proof-bearing
fp16 / bf16 / fp8	**RepOps-Freivalds** (pinned reduction tree τ; Section 17.3) *or* TEE/CC attestation (type 1/4)	RepOps: bit-exact via Prop. 17.1, costs vendor-kernel speed. TEE: hardware attests the fp run, no Freivalds, no τ pin
small models (any arithmetic)	**zkML** (type 2; Section 14)	trustless succinct verification; bounded by proving cost to small models

Section 14: that table selects a backend by *trust*; this one selects an *exactness regime* by arithmetic. The two compose—e.g. a bf16 model at high value runs RepOps-Freivalds under an M-of-N bond, while the same model quantized runs exact-integer Freivalds.

The structure of Table 7 is the whole argument: full coverage is a *disjunction of exact modes*, not a single mode stretched past where it is sound. A model that ships quantized can use the strongest of the three properties—exact, software-only, hardware-free, tolerance-free—once its production path is constrained by the activated exact-kernel profile and emits the canonical transcript. A model that must be served in full floating-point precision still has a proof: pay the RepOps throughput tax for a software-only exact check, or attest in a TEE for full-speed kernels at the cost of a hardware-trust assumption. No model is left without a proof-bearing mode, and no mode is asked to bear a soundness or liveness load it cannot carry.

Principle 17.1 (Exactness is tiered to arithmetic)**.** *There is exactly one way to make a software-only, cross-hardware, tolerance-free matmul check: exact arithmetic. Quantized models supply it natively (integer accumulation); floating-point models supply it by pinning the reduction tree (RepOps) at a throughput cost, or sidestep it via hardware attestation. "Support every modern LLM" is therefore a routing decision over exact regimes—never a single floating-point Freivalds, which is unsound (corridor) and unusable (false-reject) at once.*

18 Coverage of the Model Zoo: Two Extensions Suffice

Section 11 establishes that the base primitive—Freivalds-checked matmuls plus deterministic recomputation of the cheap remainder—covers any workload whose dominant cost is dense matrix multiplication. Modern model families, however, are not all plain stacks of GEMMs: some route tokens to a sparse subset of experts, some replace attention with a linear recurrence or state-space scan, some consume pixels or audio, and some denoise a latent over many steps. This section surveys the architecture families actually shipped in the Zoo model zoo and maps each to what the proof needs. The payoff is a structural claim that makes "support the entire zoo" tractable: *beyond the base matmul check, there are exactly two new core techniques*—a *routing proof* and a *sequential-chaining proof*—plus per-modality input and output commitments with deterministic pre- and post-processing. No third core technique is required.

We are scholarly about status. The base Freivalds verifier over the exact integer accumulator is *implemented and tested* (Section 20); the two determinism prerequisites the harder families lean on—the lowest-token-id argmax tie-break and the canonical backend-independent MoE top-k—are *also implemented and tested in the engine*; the activation-trace commitment, the forward-pass emission, and the *routing proof* are now *built and tested* as well (the routing proof is the TopKRoute/Gather path of `poi_graph.rs`, exercised by a real MoE block). What remains *designed here, not yet built* is the *sequential-chaining proof* (for state-space and diffusion latents) and the RepOps floating-point tier; we label each obligation accordingly.

What "coverage" means here, precisely. The coverage this section establishes is a structural claim—every family *reduces* to the base primitive plus at most one of two extensions—and its tested floor is now nonempty. The families with a committed, verified transcript today are the dense/SwiGLU forward, the attention spine (with a deterministic integer softmax), the mixture-of-experts block (routing proof built), and a training step: these emit and verify in the engine's test suite. What remains a *designed* ceiling rather than a deployed floor is the breadth: the sequential-chaining families (recurrent/state-space and diffusion) await the chaining-proof emission, and the fp16/bf16/fp8 tier (RepOps, Section 17.3) awaits the pinned-reduction kernel (specified and proven sound in [24], no kernel yet). The theorem below should be read as "these three techniques *suffice* to cover the zoo, two of the three are built, and the families in the test suite are covered today."

18.1 Dense transformer (base case)

The dense decoder/encoder—`zen-nano` and the dense tiers of the family—is the case the base primitive covers directly. Every layer is a sequence of GEMMs (the attention Q, K, V projections, the score–value product, the output projection, and the two MLP matmuls) interleaved with normalization and an activation. The matmuls are Freivalds-checked (Theorem 12.2); the elementwise remainder is committed and recomputed (Section 12). **Need:** base primitive only. **Status:** verifier core built and tested; trace commitment + forward hooks designed.

18.2 Mixture-of-Experts: the routing proof

MoE models [23]—`zen4-max` (35B-A3B), `zen-coder-flash` (GLM-style MoE), `zen-max` (DeepseekV3/Kimi-style, 384 experts, 8 active)—add a router that, per token, scores the experts and dispatches the token to its top-k. The expert FFNs are still GEMMs (base primitive covers them once the right experts are known), but the *routing* introduces two new obligations.

Routing determinism (built). A different selected expert is a different sub-network, so if a prover and a re-deriving verifier disagree on the top-k they compute different outputs and an honest run false-rejects. The engine therefore exposes a process-wide proof-deterministic routing mode (`set_moe_proof_deterministic`; Section 20) whose top-k uses a *canonical, backend-independent tie-break*—order each row by (score descending, expert-index ascending) under an f32 total order—so every machine selects the identical experts. This is the routing analog of the lowest-id argmax tie-break, and it is implemented and tested.

Routing proof (designed). Determinism makes the selection reproducible; soundness additionally requires that the prover *committed* to the router's behavior so a verifier can recheck it. The routing proof commits, per routed token, (i) the router logits (themselves a matmul output, hence Freivalds-checkable), (ii) the selected expert indices, and (iii) which tokens activated which experts (the sparse activation pattern). A verifier challenging a token recomputes the top-k from the committed (and Freivalds-verified) router logits under the canonical tie-break and checks it against the committed indices; it then checks that the committed expert-FFN matmuls used exactly those experts on exactly those tokens. The discrepancy a cheater would introduce—routing to a cheaper expert, or skipping the dispatch—is caught either at the top-k recheck (wrong indices) or at the expert-matmul Freivalds check (wrong sub-network). **Need:** routing determinism (built) + routing proof: commit router logits, selected indices, sparse activation pattern (designed). **This is the first of the two new core techniques.**

18.3 Linear / gated-recurrent / state-space: the sequential-chaining proof

The single most important new technique is for the families whose sequence mixer is *not* a single batched matmul. The Zoo zoo ships several: `zen4-nano/micro/mini/pro` use a gated-DeltaNet [21] linear-attention recurrence; `zen4-thunder/storm` use a lightning-attention linear recurrence; Mamba/SSM-style backbones [20] use a selective state-space scan. In all of these, the layer carries a *recurrent state* S_t updated step by step:

$$S_t \;=\; \Phi_t\, S_{t-1} \;+\; \Psi_t, \qquad o_t \;=\; \Omega_t\, S_t, \tag{24}$$

where Φ_t, Ψ_t, Ω_t are (input-dependent) linear maps. Each *step* of (24) is matmuls and so is Freivalds-checkable in isolation. The new difficulty is the *chain*: a prover could compute each step honestly in isolation yet splice in a fabricated S_{t-1} at the boundary, so that step t is internally consistent but uses a state that was never produced by step $t-1$. Per-step Freivalds alone does not catch a broken seam, because each step's check sees only its own operands.

The chaining technique (designed). Bind *consecutive states* in the commitment. The prover commits each recurrent state S_t into the activation trace and, crucially, commits the *transition* so that step t's committed input state is constrained to equal step $t-1$'s committed output state. Concretely, the trace orders the states $S_0, S_1, \ldots, S_T$ and the per-step Freivalds check at step t takes its input state from the committed S_{t-1} leaf (not a freshly supplied one); a verifier that challenges step t checks both that $S_t = \Phi_t S_{t-1} + \Psi_t$ (the step's matmuls, Freivalds) *and* that the S_{t-1} it used is the same leaf the trace commits as step $t-1$'s output. The seam is thus pinned by the commitment, and a spliced state is detected as a mismatch between the leaf step t consumed and the leaf step $t-1$ produced. **Need:** per-step Freivalds + binding consecutive recurrent states in the commitment (designed). **This is the second of the two new core techniques.**

18.4 Multimodal vision (ViT tower)

The vision-language models—`zen-vl`, `zen-omni` (vision path), `zen-designer` (Qwen3-VL family)—prepend a vision tower to the language model. The tower is a Vision Transformer: the patch embedding is a convolution, which is an *im2col*-then-GEMM, and every subsequent block is the dense-transformer case. So the tower's heavy arithmetic is *already covered* by the base Freivalds primitive once the convolution is expressed as im2col→GEMM. Two modality-specific obligations remain, both of the "commit-and-recompute deterministically" kind rather than new core techniques:

1. **Preprocessed-pixel commitment.** The inputHash must bind the *preprocessed* pixel tensor (resized, normalized, patchified), not the original image file, so a prover cannot answer an easier image than the one posed. The preprocessing (resize, normalize) must be a *deterministic*, pinned operation in runtimeMeasurement, exactly as the tokenizer is for text.
2. **Patch-embed as im2col→GEMM.** The convolution is committed as its GEMM form so the same Freivalds check applies; the im2col rearrangement is an index gather and is committed and recomputed.

Need: base primitive + deterministic image preprocessing + preprocessed-pixel commitment + conv-as-im2col-GEMM (commitment designed; the GEMM check itself is the built core). If the model also routes vision tokens through experts (MoE-VL), it additionally needs the routing proof of Section 18.2—no third technique.

18.5 Audio (conformer / speech)

The audio models—`zen-tts`, the `zen-omni` audio path—are conformer/speech stacks whose self-attention and feed-forward blocks are again GEMMs (base primitive). The modality-specific piece is the *front end*: raw audio is turned into mel-spectrogram features by a short-time Fourier transform and a mel filterbank. An FFT is not a matrix multiply in the Freivalds sense (it is a fixed, input-independent linear transform), so it is handled by *commit-and-recompute*: the feature extraction is deterministic signal processing with a pinned window, hop, and filterbank, committed in the trace and recomputed by a challenger from the committed input samples. The model's audio output (waveform or codec tokens) is committed in outputHash. **Need:** base primitive + deterministic FFT→mel feature extraction (commit-and-recompute) + input-sample and audio-output commitments. No new core technique—the FFT is the audio analog of softmax in the non-matmul remainder (Section 12).

18.6 Diffusion / Transfusion: chaining again

The generative families—`zen5`'s 3D/video and `zen-designer`'s image generation, built on a Transfusion-style [22] autoregressive-plus-diffusion design—denoise a latent over many steps. Each denoising step is a forward pass through a transformer or U-Net, i.e. matmuls, so a *single* step is the base case. The multi-step loop poses the same seam problem as the recurrent family: the latent z_t at step t must be the one that step $t-1$ actually produced, and a prover could splice. The resolution is the *same* sequential-chaining primitive of Section 18.3: commit each intermediate latent $z_0, \ldots, z_T$, pin step t's input latent to step $t-1$'s committed output latent, and Freivalds-check each step. Two diffusion-specific determinism clauses join runtimeMeasurement: the *noise schedule* and the *sampler/seed* must be pinned, so that the (otherwise stochastic) trajectory is a deterministic function of the prompt and the committed seed—the diffusion analog of temperature-0 decoding. **Need:**

Table 8: The Zoo model zoo mapped to proof obligations. Beyond base Freivalds-on-matmuls there are exactly two new core techniques (routing proof, sequential-chaining proof); the rest is per-modality commitment and deterministic pre/post-processing. "Built" = implemented and tested in the engine; "designed" = specified here, not yet built.

Family (zoo models)	Heavy arithmetic	New core technique	Modality / determinism add-ons
Dense transformer (`zen-nano`, dense tiers)	GEMMs (base)	none	—
MoE (`zen4-max`, `zen-coder-flash`, `zen-max`)	expert-FFN GEMMs	**routing proof** *(built)*; routing determinism *built*	commit router logits, expert indices, sparse pattern
Linear / gated-recurrent / SSM (`zen4-nano/micro/mini/pro`, `zen4-thunder/storm`)	per-step GEMMs	**sequential chaining** (designed)	bind consecutive states $S_{t-1} \to S_t$
Multimodal ViT (`zen-vl`, `zen-omni`, `zen-designer`)	tower GEMMs; conv as im2col→GEMM	none (MoE-VL: routing proof)	deterministic image preprocessing; preprocessed-pixel commit
Audio / speech (`zen-tts`, `zen-omni` audio)	conformer GEMMs	none	deterministic FFT→mel (commit-and-recompute); audio I/O commit
Diffusion / Transfusion (`zen5` 3D/video, `zen-designer` image-gen)	per-step GEMMs	**sequential chaining** (designed, *reused*)	pinned noise schedule, sampler, seed; latent chaining

per-step Freivalds + deterministic noise-schedule/sampler/seed + multi-step latent chaining—and the chaining is *literally* the recurrent-family technique reused, which is the point.

18.7 The unifying claim

Table 8 summarizes the survey. Read down the "new core technique" column: it takes only two distinct values. Mixture-of-experts (including its multimodal variants) needs the *routing proof*; the linear-recurrent/SSM family *and* diffusion's multi-step loop both need the *sequential-chaining proof*; everything else is the base matmul primitive plus per-modality input/output commitment and deterministic pre/post-processing. That the recurrent backbones and the diffusion sampler collapse onto the *same* chaining primitive—despite looking unrelated—is what caps the technique count at two and makes whole-zoo coverage a finite, enumerable engineering task rather than an open-ended one.

Theorem 18.1 (Two-extension coverage of the zoo, by reduction)**.** *Let the verification surface consist of (i) the base primitive (Freivalds-checked matmuls plus deterministic commit-and-recompute of elementwise/transform ops), (ii) the routing proof of Section 18.2, and (iii) the sequential-chaining proof of Section 18.3, each composed with the per-modality input/output commitments and the determinism contract. Then every architecture family shipped in the Zoo zoo— dense, MoE, linear-recurrent/SSM, multimodal vision, audio, and diffusion/Transfusion—is* reducible *to this surface: its heavy arithmetic reduces to matmuls verified by (i), its sparse routing (if any) by (ii), and its inter-step state or latent seam (if any) by (iii). No additional core verification technique is required.* This is a statement about the design's sufficiency; two of the three techniques—the base primi-

tive (i) and the routing proof (ii)—are implemented and tested and are wired to emit from real forward/graph passes, so the reduction is realized today for the dense, attention, MoE, and training families, while the sequential-chaining technique (iii) and the long-tail families remain designed (Section 2).

Argument. By the per-family analyses above, composed with the orthogonality of Section 11. Each family's dominant cost is dense matmul (the attention projections and MLPs of a transformer block, the expert FFNs of an MoE, the per-step maps of a recurrence (24), the per-step denoiser of a diffusion loop, the ViT tower including the im2col-expressed convolution, and the conformer blocks), all covered by (i). The two structural departures from a plain GEMM stack are sparse expert routing and an inter-step state/latent dependency. Routing is covered by (ii): the router logits are a matmul output checked by (i), the canonical top-k is recomputed and checked against the committed indices, and the expert matmuls are checked by (i) against exactly the committed experts. The inter-step seam—identical in form for the recurrent state S_t and the diffusion latent z_t—is covered by (iii): consecutive states are bound in the commitment so step t's consumed leaf must equal step $t-1$'s produced leaf, and each step is checked by (i). The remaining per-modality operations (tokenization, image preprocessing, FFT/mel feature extraction, noise scheduling) are deterministic, pinned, and input-independent or elementwise, hence committed and recomputed under the determinism contract with zero false rejection, and the per-modality inputHash/outputHash bind the boundary data. As every family decomposes into these three cases plus pinned pre/post-processing, the three techniques suffice. □ □

The decomplecting discipline of Section 11 extends cleanly here. The routing proof is its own slot, orthogonal to the base primitive and to chaining; the chaining proof is its own slot, shared by two families that happen to need the same seam guarantee. Adding a future architecture means asking only "does it route?" and "does it carry inter-step state?"—two yes/no questions selecting from a fixed set of three techniques—rather than designing a new verifier. That is the sense in which whole-zoo coverage is *simple* and not merely *possible*.

19 Zoo Consumption and PQ-Rooted Omnichain Settlement

The execution proof of Sections 12–18 makes an admitted AI computation accountable. It does not create a new mining authority wherever the proof is verified. The canonical topology has one authority and three downstream consumers:

1. **A-Chain PoAI consensus** admits the demand-backed job, fixes its execution and proof policy, derives the post-commitment challenge, resolves the evidence to final or rejected, consumes the global work nullifier, and authorizes one reward receipt.

2. **Z-Chain P3Q settlement** batches finalized A-Chain receipt transitions into a Plonky3-derived, pairing-free STARK/FRI proof. It proves compression and settlement of an A-Chain decision; it does not admit raw work or calculate a separate subsidy.

3. **Lux Quasar** orders and finalizes the A-Chain and Z-Chain state under the actually activated post-quantum validator policy.

4. **Zoo and other destination applications** verify the PQ-finalized, Z-settled A-Chain receipt, enforce destination and recipient binding, record local consumption, and release the authorized representation exactly once.

This separation resolves an otherwise fatal supply ambiguity. "Mine into Zoo" means an A-Chain job names a Zoo destination before execution. It does not mean Zoo accepts a raw transcript, runs another PoAI vote, or owns another spent set. One job consumes one global nullifier on A-Chain and can yield one receipt, regardless of where the receipt is paid.

19.1 What Zoo contributes

Zoo remains economically and scientifically important without duplicating consensus. Zoo applications can fund inference, embedding, evaluation, and supported training jobs; curate models and datasets; define public-good reward programs; and use finalized execution receipts as reproducibility artifacts. A scientific result can cite the committed model, input, kernel profile, graph, output, and transcript root rather than asking readers to trust that an opaque provider ran the advertised computation.

The same mechanism covers the heavy arithmetic of several task families:

- a proven inference binds the declared model and input to its graph and output;
- a proven embedding binds a representation to its declared source computation;
- a supported training step is a longer deterministic graph whose forward, backward, and update matmuls use the same local verifier; and
- a governed evaluation job may combine objective execution evidence with separately attributable human or model judgment, while judgment alone cannot authorize mining subsidy.

Cryptography proves execution of the specified job, not universal truth, dataset rights, or scientific importance. Demand-backed admission supplies the protocol meaning of useful; governance and peer review remain responsible for the quality of the questions and data.

19.2 Receipt path and exactly-once payout

The exported object is a finalized receipt, never a raw `ComputeProof`, TEE quote, miner signature, or Freivalds opening. The receipt binds the A-Chain network and policy epoch, job, accepted execution commitment, amount, destination chain, recipient, and nullifier. Z-Chain settlement binds that receipt transition to an A-Chain receipt root; Quasar finality authenticates the A/Z roots and heights; a destination verifies inclusion and records consumption of that exact receipt.

A destination-local consumed set prevents replay of an already valid authorization. It does not judge whether work occurred. That judgment exists only on A-Chain, which is what makes a capped issuance schedule compatible with omnichain payout.

19.3 P3Q activation boundary

Before the PoAI AIR and verifier are activated, Z-Chain may carry only an authenticated, already-final A-Chain receipt root. It must not present that checkpoint as a succinct proof of the underlying work. P3Q authority activates only after the AIR, parameters, recursion policy, A-to-Z state-transition binding, verifier, resource bounds, node wiring, and audits are complete. A shadow phase may compare candidate proofs against already-final receipts, but those proofs have no mint authority. Staging is deployment safety, not a weaker proof tier marketed to users.

The result is a clean omnichain contract: useful work is requested anywhere, executed on any conforming hardware, judged once by A-Chain PoAI, compressed by Z-Chain P3Q, finalized by Lux Quasar, and paid exactly once at the destination selected before mining.

20 Implementation Status

We state plainly what is built and tested versus what is designed, so the paper's claims can be checked against code. The repository contains executable reference implementations for the activation-trace commitment, proof-bearing forward, graph fixtures, challenger, and EVM enforcement. These surfaces test the construction; they are not the active A-Chain consensus path. A complete production model path, data availability, consensus admission and finality, metering, nullification, and P3Q settlement remain integration work. Test counts below are snapshots of named suites, not claims of live mining.

20.1 Built and tested: the Freivalds core

The matmul verifier of Section 12 is implemented as a backend-agnostic, dependency-light module (`hanzo-engine/src/poi.rs`, whose only external dependency is `sha3`). It realizes the construction exactly as specified: arithmetic over the Mersenne field $\mathbb{F}_p$ with $p = 2^{61} - 1$, run on the engine's *exact integer accumulator* (`exact_matmul`, a whole-K $i64$ reduction with no cross-block $f32$ rescale), with the Freivalds identity $A(Br) = Cr$ evaluated by left/right matrix–vector products (`freivalds_verify`) and deterministic, beacon-bound challenge derivation. Being dependency-free of the forward pass, its soundness is testable in isolation. The `poi.rs` suite is **7/7 passing**, and each test pins a claim from Section 12:

- `test_honest_matmul_passes` — an honest $C = AB$ passes every challenge vector (zero false rejection; the exact-integer guarantee).
- `test_tampered_output_is_caught` — flipping a single output entry ($50 \to 51$) is caught (Theorem 12.1 soundness).
- `test_larger_random_matmul_catches_sparse_cheat` — in a $16 \times 24 \cdot 24 \times 16$ product, a single buried off-by-one is caught; the *sparse-cheat* case that motivates challenging every layer.
- `test_negative_entries_field_reduction` — signed int8 operands reduce correctly into $[0, p)$ (Euclidean remainder), and a tampered negative-entry product is caught.
- `test_challenge_derivation_deterministic` — prover and verifier derive the *identical* challenge from the same seed; different seeds give different challenges (the reproducibility half of anti-replay, Lemma 10.2).
- `test_wrong_dims_rejected` — a shape mismatch fails *closed* rather than reading out of bounds.
- `test_derive_challenges_keccak_golden_parity` — the keccak challenge derivation matches the Go verifier *byte-for-byte* against a pinned golden hex vector; if either side's keccak or $\mathrm{BE}_{32}(j) \,\|\, \mathrm{BE}_{64}(i)$ encoding drifts, this test (and its Go twin `TestDeriveChallengesGolden`) breaks. This is what guarantees a Rust-emitted proof verifies against the on-chain Go verifier unchanged.

20.2 Built and tested: the transcript, the proof-bearing forward, the graph

The wire that the earlier draft marked "designed" is implemented as three further modules, all in `hanzo-engine/src/`, all tested.

Activation-trace transcript and challenger (`poi_transcript.rs`, 8/8). A forward pass commits each proof-bearing matmul as a keccak Merkle leaf over its exact-integer (A, B, C) (`matmul_leaf`, domain tag `"hanzo/poi/matmul-leaf/v1"`); the root is the `transcriptRoot` the prover posts. The challenger derives a beacon-unpredictable matmul *index* ($\mathsf{keccak}(\text{beacon} \,\|\, \text{root}) \bmod L$), has the prover open it (reveal (A, B, C) plus a Merkle inclusion proof), checks the operands were the committed ones, and Freivalds-checks $C = A \cdot B$ with k beacon-bound vectors (`verify_opening`). The Merkle fold (odd levels duplicate the last node; index fold $\mathsf{even} \to \mathsf{keccak}(node \| sib)$, $\mathsf{odd} \to \mathsf{keccak}(sib \| node)$) is identical to `AIMiner._verifyMerkle`, so a proof built here verifies on-chain. The suite proves honest passes verify under every beacon, a fabricated C is caught, a swapped-in C fails Merkle inclusion, and the root binds every operand.

Sound emission (`poi_forward.rs`, 3/3). The audit's CC-01 was that the value a layer *returns* must be the value it *commits*. `ProvableLinear` makes this hold by construction: it quantizes x per-row to int8, forms $C = \bar{x} \cdot W^\top$ on the exact $i64$ accumulator, commits $(\bar{x}, W^\top, C)$, and returns dequant(C)—so a challenger's Freivalds over the committed operands re-checks the *same* computation that produced the output, not an $f32$ shadow that disagrees with it. `test_provable_ffn_emits_verifiable_t` runs a real SwiGLU FFN (`gate/up/down`) under emission and verifies every committed matmul; `test_output_is_the_committed_value` confirms the returned y equals dequant(committed C) to within float tolerance (the output-binding property, proved in the emission companion [25]).

Full-graph composition (`poi_graph.rs`, 9/9). A prover could commit correct matmuls computed on *unrelated* inputs; `GraphTrace` closes this with a typed computation trace over hash-named activations. Each op commits ($\mathsf{kind} \,\|\, \text{input commitments} \,\|\, \mathsf{params} \,\|\, \text{output commitment}$); *chaining* (every input is a prior output or a declared input) is checked from the leaves alone, and *local correctness* re-checks each op—MatMul by Freivalds, the deterministic glue (Add, ReLU, Scale, TopKRoute under the canonical lowest-index tie-break, Gather, Softmax as a pure fixed-point integer reduction, Transpose) by direct recomputation. The suite emits and verifies a real MoE block (router $\to$ top-k $\to$ gather $\to$ expert up/down matmuls $\to$ residual), the attention spine ($QK^\top \to$ scale $\to$ integer softmax $\to \cdot V$), and—with the one Transpose op a backward pass adds—a full SGD training step, catching a fabricated gradient by Freivalds exactly as a faked inference output (`test_training_step_is_just_a_graph`). This is the code behind the canon "inference, embedding, and training are graphs, not three proof systems."

Adversarial red-team battery (`tests/poi_adversarial.rs`, 11/11). Each test plays the attacker. The eight attacks—fabricate the final output, fabricate a hidden intermediate matmul, a sparse deep cheat, swap the output at open time, tamper an input at open time, pre-fit C to a guessed challenge, a griefer's fabricated opening, and a quantization lie ($y \neq$ dequant(C))—are each *caught*; the three defenses (two honest machines emit the byte-identical root, an honest forward verifies under every beacon, a shape mismatch fails closed) confirm zero false rejection. The pre-fit test is the load-bearing one for the beacon: a forgery tuned to a guessed r_1 passes r_1 and fails a fresh beacon-derived r_2.

Jointly the engine carries **27 unit tests (`poi.rs` 7, `poi_transcript.rs` 8, `poi_forward.rs` 3, `poi_graph.rs` 9) plus 11 adversarial tests = 38 passing**.

20.3 Built and tested: the two determinism prerequisites

The exact-integer Freivalds check is bit-exact only if the surrounding decode is reproducible; Section 12's determinism contract names the two places a naive runtime injects nondeterminism, and

both are closed in the engine. The *lowest-token-id argmax tie-break* (`hanzo-engine/src/sampler.rs`) breaks a greedy-decoding logit tie to the lowest token id via a strict ">" scan, correcting Rust's `Iterator::max_by` (which returns the *last* maximal element); the *canonical MoE routing top-k* (`hanzo-engine/src/ops.rs`, and its in-graph twin `top_k_indices` in `poi_graph.rs`) orders each row by score descending then expert-index ascending under a total order, so prover and verifier select identical experts (a different expert is a different sub-network). These are the routing analog of the argmax tie-break and are what make the "re-derive and compare" premise actual rather than aspirational for the greedy and MoE-routing cases.

20.4 Built and tested: the candidate canonical Go verifier

The verifier intended for A-Chain integration is a dependency-light Go package living beside the canonical `Keccak256` (`luxfi/crypto/poi`): `Verify`/`VerifyMulti` over $\mathbb{F}_p$ ($p = 2^{61} - 1$) on the exact integer accumulator, the transcript and `VerifyOpening`, the wire codec, and keccak `DeriveChallenges`. The engine's `poi.rs` is the prover-side mirror; the two agree byte-for-byte, enforced by the shared golden challenge vector (`TestDeriveChallengesGolden`, the Go twin of the Rust parity test above). The suite is **23/23 passing**, including the honest-opening, fabricated-output-caught, swapped-reveal-caught, Merkle-index-fold, sparse-cheat, encode/decode round-trip, wire-fuzz, oversized-dims-rejected, griefer-cannot-frame, and pre-fit-challenge cases—the Go-side mirror of the Rust adversarial battery, plus a verification-throughput scaling test.

20.5 Built and tested: reference EVM enforcement surfaces

The reference witness is verified in the EVM, not asserted by a watcher. These contracts live in `luxfi/standard` (`contracts/ai/{governance,mining,token,compute,relay,registry}/`); the eight Proof-of-AI suites total **87 tests passing** (`forge test`), distributed as follows.

- `ComputeWitnessLib.sol` **(5/5).** Freivalds over $\mathbb{F}_p$ via `mulmod`/`addmod`, the Merkle fold, and the beacon challenge, byte-identical to the prover (same domain tag, same (rows, cols, data) serialization, same fold, same derivation). `verifyOpening` accepts a genuine opening; `provesFraud` is its complement—inclusion $\wedge \neg$Freivalds—the predicate a challenger exhibits to slash. Opened dimensions are capped (`MAX_DIM`) so a griefer cannot submit an unbounded opening.

- `OptimisticEvidence.sol` **(18/18).** The commit/challenge/slash state machine: `submit` escrows a bond and opens a window with the commit block recorded as the challenge beacon (`blockhash(commitBlock)`—fixed after the prover signed, so it cannot pre-fit a dodging C); `challenge` re-checks inclusion + Freivalds on-chain via `provesFraud` and, only on a verified discrepancy, marks the commitment FRAUDULENT and pays the bond to the challenger; `reclaim` returns the bond after an un-challenged window. The state machine is one-shot per `reportData` (a slashed liar cannot re-post), a zero bond is rejected (a liar always has skin in the game), and effects precede the single pull-of-funds (no reentrancy re-claim).

- `AIMiner.sol` **(12/12).** A reference mint gate, *fail-closed*: `mine` requires receipt shape + finality + Merkle inclusion under an accepted root + a `ComputeProof` binding the receipt's own measured model/prompt/output, and reverts (`ProofGateUnavailable`) if no verifier is wired—so a forged root alone cannot mint. The subsidy goes to the bound requester, is clamped to the vested halving allowance, and is one-shot per intent.

- `ComputeProofEnforcement.t.sol` **(19/19).** The end-to-end enforcement that a proofless or mis-bound mint is rejected and a correctly-bound one succeeds, across the proof-type profiles.

- `AICoin.sol` **(11/11).** A legacy capped/halving token fixture: `MAX_SUBSIDY` $= 10^9$ AI, `HALVING_PERIOD` $=$ 4 years, the slope halving each epoch and the cumulative mint structurally bounded by allowed$(t) \leq$ MAX. Its 10^9 parameter is not the canonical AI monetary policy; production must use the approved 2 trillion allocation and common A-Chain issuance schedule described in the Hanzo AI Chain specification.

- `AIComputeRegistry.t.sol` **(6/6).** A reference no-double-mint ledger. Production uses the A-Chain job/nullifier key; a tuple of model, prompt, and output is not a sufficient consensus identity by itself.

- `MinerStakeRegistry.sol` **(8/8).** A miner bonds capital in proportion to declared capacity; the fraud-proof verifier slashes on a proven discrepancy with a cooldown that is the fraud window.

- `L3Registry.sol` **(8/8).** A Zoo application-directory fixture. It may discover receipt consumers, but has no work-admission, finalization, or mining authority.

The post-quantum cross-layer root relay `AChainRootOracle.sol` (an ML-DSA validator quorum replacing a trusted single relayer; Section 19) and the leaderless operator-quorum governance (`AIGovernor`, `AIParams`, with `AIExecute`/`AIApproval`) are likewise implemented in the same standard library and carry their own passing suites; we cite them as candidate surfaces with which the settlement path composes, without re-tabulating their tests here.

20.6 Designed, not yet built

The honest remaining work is now a named, smaller set; none of the following is implemented at the time of writing.

1. **Production consensus integration.** Normal serving-path transcript emission, artifact availability, full-graph coverage, economic metering, A-Chain job admission, finalized proof gating, nullifier consumption, and P3Q settlement are not yet one live path. Shadow/replay P3Q stages have no mint authority.

2. **RepOps (pinned reduction tree).** The deterministic-floating-point mode for the fp16/bf16/fp8 tier (Section 17.3) is designed and proven sound ([24]); the quantized integer path is what is built and is what the Zoo zoo ships. Exact Freivalds over the current $f8q8$ fast path (which sums int8 block dots in $f32$ across blocks) would false-reject honest cross-hardware provers; the proof path instead uses `exact_matmul`'s whole-K $i64$ reduction, and the floating-point tier awaits the RepOps cross-block pinning.

3. **The sequential-chaining proof.** Binding consecutive recurrent states (and, identically, consecutive diffusion latents) in the commitment (Sections 18.3, 18.6) is designed and proven sound modulo its discharge obligation ([24]). The *forward*/Transpose graph that covers dense, attention, MoE, and training is built (above); the per-step state seam for state-space and diffusion models is the remaining emission work.

4. **Emission of the long-tail architectures.** The families the test suite exercises (dense/SwiGLU, attention with integer softmax, MoE, training) emit and verify; the remaining model-zoo families inherit the same proof obligations ([24], Theorem on whole-zoo coverage) but are not yet wired to emit a transcript. This is breadth of integration, not a missing technique: no third core verification method is required.

Remark 20.1 (What the tested system does and does not establish). The cited Rust, Go, and Solidity tests establish that the *mathematical heart*—"you cannot claim the matmul without doing it"—is implemented correctly over the exact integer accumulator, that it *emits* from a real forward/graph and *composes* across attention, MoE, and a training step, that the challenger and the on-chain witness re-check it byte-identically in two more languages, and that a reference mint gate can fail closed. What they do *not* establish is a live A-Chain mining path, production data availability and metering, P3Q settlement, the floating-point tier (RepOps), or emission of architectures outside the tested set. We keep the boundary explicit, in keeping with the decomplecting discipline of the rest of the paper: the verifier is a value (a pure function of operands and a challenge), the emission supplies those operands from a real pass, and the on-chain gate consumes the result—three separable concerns, each now built where this paper's earlier draft had only the first.

21 Related Work

Proof of Useful Work: Primecoin [1], Gridcoin [2] pioneered useful PoW but with limited domains.

AI Compute Markets: Golem, Render Network focus on compute markets without consensus integration.

TEE-Based Consensus: Oasis Network, Secret Network use TEEs for privacy but not AI verification.

Federated Learning: FL protocols [3] aggregate gradients but lack blockchain integration.

22 Future Work

The honest remaining work is the residual named throughout (Sections 2, 20), plus the orthogonal extensions the attestation and pricing layers invite. The execution-proof core, its emission, the cross-chain root relay, and the on-chain gate are built; what follows is breadth and the surrounding product surface.

1. **RepOps floating-point tier**: a pinned-reduction-tree kernel that makes fp16/bf16/fp8 GEMMs bit-exact, extending the proof path beyond the quantized integer tier (designed and proven sound in [24]; no kernel yet).

2. **Sequential-chaining emission**: wiring the proven chaining obligation for state-space (recurrent) and diffusion models, the one structural case the forward/transpose graph does not yet cover.

3. **Long-tail architecture emission**: extending transcript emission to the model-zoo families outside the tested set (multimodal towers, audio conformers), each of which already reduces to the existing techniques (Theorem 18.1).

4. **GPU TEE integration**: NVIDIA H100 / Blackwell confidential-computing quote parsing into the canonical attestation quote, for the hardware-trust tier that complements the software-only proof.

5. **Operator-path confidentiality integration**: wiring the built X-Wing prompt seal and attestation binding (`crypto/promptseal`, `crypto/attestation`) into every inference operator's request handler and key-management lifecycle.

6. **Privacy-preserving quality scoring**: zero-knowledge proofs for the multi-party evaluation protocol, so a quality score reveals nothing about the evaluator's private rubric.

23 Conclusion

Proof of AI is credible only when the verification claim and the consequence authority are both narrow. Exact-integer execution gives heterogeneous hardware a deterministic statement: a GPU may perform the original workload while a CPU validator checks sampled matrix relations in quadratic rather than cubic time. A versioned kernel profile defines the arithmetic, and a canonical transcript binds that arithmetic to the admitted job, ordered graph, artifacts, outputs, availability, meter, provider, destination, and nonce. Neither a signature, a plurality of attestations, nor cross-language byte parity proves useful work by itself.

The reference implementations and adversarial fixtures show that this construction is practical and that fail-closed economic gates can be tested. They are not yet a live consensus deployment. Production activation still requires normal model-path emission, data availability, full graph coverage, A-Chain admission and finalized-proof wiring, nullifier and metering enforcement, and audit-gated P3Q settlement. Shadow and replay stages can compare evidence, but must have no mint authority.

The resulting omnichain design has one source of truth. A-Chain alone validates eligible useful work and authorizes AI; Z-Chain settles the finalized transition through P3Q; Lux Quasar provides post-quantum finality; Zoo and other chains consume receipts and may pay local rewards exactly once. This preserves the useful-work claim while avoiding a different "proof of whatever" on every destination chain.

24 Attestation Format

Listing 1: TEE Attestation Structure

```
{
  "version": 1,
  "tee_type": "sgx" | "sev" | "tdx",
  "measurement": {
    "mrenclave": "0x...",
    "mrsigner": "0x...",
    "product_id": 1,
    "security_version": 1
  },
  "report_data": {
    "task_hash": "0x...",
    "input_hash": "0x...",
    "output_hash": "0x...",
    "timestamp": 1650000000,
    "nonce": "0x..."
  },
  "signature": {
    "type": "ecdsa_p256",
    "value": "0x..."
  },
```

```
  "certificate_chain": ["0x...", "0x..."]
}
```

25 Quality Scoring Functions

Listing 2: Quality Scoring Implementation

```
def score_inference(attestation, samples):
    correct = sum(1 for s in samples
                  if verify_output(s))
    accuracy = correct / len(samples)
    latency_penalty = max(0,
        (attestation.latency - TARGET) / TARGET)
    return accuracy * (1 - 0.1 * latency_penalty)

def score_training(attestation, val_before, val_after):
    improvement = val_after - val_before
    cost = attestation.flops / 1e15
    return improvement * math.exp(-0.1 * cost)

def score_extraction(experience, library):
    novelty = 1 - max_similarity(experience, library)
    applicability = test_on_benchmarks(experience)
    correctness = verify_experience(experience)
    return novelty * applicability * correctness
```

26 Slashing Conditions

1. **Invalid Attestation**: Signature does not verify
2. **Output Mismatch**: Re-execution produces different result
3. **Quality Manipulation**: Score differs from consensus by $> 3\sigma$
4. **Double Signing**: Same attestation submitted to multiple blocks
5. **Unavailability**: Validator offline for > 24 hours

Slashing amounts:

- Invalid attestation: 100% of bond
- Output mismatch: 50% of bond
- Quality manipulation: 25% of bond
- Double signing: 100% of bond
- Unavailability: 1% of bond per day

Acknowledgments

This work was supported by Zoo Labs Foundation. We thank Intel for SGX documentation and AMD for SEV-SNP technical support.

References

[1] S. King, "Primecoin: Cryptocurrency with Prime Number Proof-of-Work," 2013.

[2] Gridcoin, "Rewarding BOINC Contributions," Whitepaper, 2014.

[3] B. McMahan et al., "Communication-Efficient Learning of Deep Networks from Decentralized Data," *AISTATS*, 2017.

[4] D. Boneh et al., "Short Signatures from the Weil Pairing," *ASIACRYPT*, 2001.

[5] V. Costan and S. Devadas, "Intel SGX Explained," *Cryptology ePrint*, 2016.

[6] E. Buchman et al., "The latest gossip on BFT consensus," *arXiv:1807.04938*, 2018.

[7] R. Freivalds, "Fast probabilistic algorithms," in *Mathematical Foundations of Computer Science (MFCS)*, LNCS 74, pp. 57–69, 1979.

[8] KD. Conway, C. So, X. Yu, and K. Wong, "opML: Optimistic Machine Learning on Blockchain," *arXiv:2401.17555*, 2024.

[9] J. Harrison et al. (Gensyn), "Verde: Verification via Refereed Delegation for Machine Learning Programs," *arXiv:2502.19405*, 2025.

[10] H. Kalodner et al., "Arbitrum: Scalable, private smart contracts," *USENIX Security*, 2018.

[11] Hyperbolic Labs, "Proof of Sampling (PoSP): A Verification Protocol for Decentralized AI Inference," Technical Report, 2024.

[12] S. Goldwasser, Y. T. Kalai, and G. N. Rothblum, "Delegating Computation: Interactive Proofs for Muggles," *STOC*, 2008.

[13] C. Lund, L. Fortnow, H. Karloff, and N. Nisan, "Algebraic Methods for Interactive Proof Systems," *Journal of the ACM*, 39(4):859–868, 1992.

[14] J. Chen et al., "ezkl: A System for Zero-Knowledge Machine Learning," `https://github.com/zkonduit/ezkl`, 2024.

[15] NVIDIA Corporation, "NVIDIA Confidential Computing on Hopper H100 / Blackwell: Architecture and Attestation," Technical Whitepaper, 2023–2024.

[16] Intel Corporation, "Intel Trust Domain Extensions (Intel TDX) Whitepaper," 2023.

[17] AMD, "AMD SEV-SNP: Strengthening VM Isolation with Integrity Protection and More," Technical Whitepaper, 2020.

[18] J. Demmel and H. D. Nguyen, "Parallel Reproducible Summation," *IEEE Transactions on Computers*, 64(7):2060–2070, 2015.

[19] IEEE, "IEEE Standard for Floating-Point Arithmetic," *IEEE Std 754-2019*, 2019.

[20] A. Gu and T. Dao, "Mamba: Linear-Time Sequence Modeling with Selective State Spaces," *arXiv:2312.00752*, 2023.

[21] S. Yang, B. Wang, Y. Zhang, Y. Shen, and Y. Kim, "Parallelizing Linear Transformers with the Delta Rule over Sequence Length," *NeurIPS*, 2024.

[22] C. Zhou et al., "Transfusion: Predict the Next Token and Diffuse Images with One Multi-Modal Model," *arXiv:2408.11039*, 2024.

[23] N. Shazeer et al., "Outrageously Large Neural Networks: The Sparsely-Gated Mixture-of-Experts Layer," *ICLR*, 2017.

[24] Zoo Labs Foundation, "Proof-of-Inference: Soundness, the Determinism Liveness Precondition, the Floating-Point No-Go, and the Routing/Chaining Extension Obligations," companion soundness report, `zooai/proofs:proof-of-inference-soundness.tex`, 2026. Contains the per-result status table "What is proven on code versus on paper."

[25] Zoo Labs Foundation, Hanzo AI, and Lux Network, "Proof-of-Inference: End-to-End Emission and On-Chain Enforcement—the activation-trace commitment that makes *no real compute, no mint* a theorem," companion emission report, `zooai/proofs:proof-of-inference-emission.tex`, 2026. Tabulates the cross-language implementation (Rust prover, Go verifier, Solidity witness/precompile) and the adversarial red-team battery.

[26] "The Verification Horizon: No Silver Bullet for Coding Agent Rewards," 2026. Establishes that semantic verification of agent output is a proxy for human intent that degrades as generator capability grows—motivating PoAI's split between provable execution and the un-provable semantic remainder.