General
Purpose AI
Policy Lab

Zero knowledge verification for frontier AI training is possible

Pierre Peigne-Lefebvre

Ky Nguyen

Paul Wang

Abstract

Frontier AI governance frameworks increasingly use cumulative training compute as the primary criterion for designating high-impact models, but enforcement rests on self-reporting because no technical verification primitive for training exists. Any future international agreement on frontier AI faces the same problem at higher stakes: coordinated regulation of technologies with significant externalities has historically rested on technical verification, without which agreements are declaratory. We propose a verification architecture for frontier dense pre-training that combines three anchors (a pre-committed training specification, inter-node network observations by a verifier-aligned anchor, and on-the-fly Merkle commitments of intermediate computation) verified through a generic zkVM with native BF16/FP32 precompiles. The proof checks the actual floating-point computation the GPU performed rather than a finite-field approximation. The protocol produces three proof types: a genesis proof at initialisation, in-training step proofs across the run, and ex-ante attestations that enforce policy-relevant claims (compute thresholds, procedure structure, data-content predicates) as running invariants, turning the training record into a governance-enforceable artefact. We estimate a deployable proof of concept within approximately 36 months, against a six-to-ten-year cycle for verification-grade custom silicon. Training-side overhead is single-digit percent and per-challenge proof size is approximately 200 KB. Thirteen open research and engineering problems are catalogued as a structured roadmap for external contribution.

Introduction

Training runs for frontier AI systems are increasingly at the centre of emerging governance frameworks that rely on claims the trainer makes about its run: what data was used, what procedure was followed, what compute was consumed. No technical verification primitive exists to check such claims today, and enforcement rests on trainer self-reporting. Any future international agreement that would govern the development of frontier AI faces the same problem at higher stakes (Baker et al. 2025; Scher et al. 2025). International coordination on technologies with significant externalities has historically relied on technical verification to move beyond declaratory commitments.[1] For the option of such an

[1] Established examples include the Montreal Protocol on ozone-depleting substances (verified through coordinated atmospheric monitoring and industrial declarations), the International Atomic Energy Agency safeguards framework under the Nuclear Non-Proliferation Treaty (verified through on-site inspections and material accountancy), and the Organisation for the Prohibition of Chemical Weapons regime (verified through

agreement to remain available to states on frontier AI, the verification primitive must exist before negotiations begin, not after. This paper proposes such a primitive.

A verification primitive for training must produce proof of faithful execution of the committed procedure, verifiable claims about the run (compute consumed, training regime, data-content filters), and privacy for the trainer, within a training-overhead budget low enough to be economically rational at frontier cost. Single-digit-percent overhead, not factor-of-two. Prior zero-knowledge machine-learning systems (Section 2) are confined to inference or LoRA fine-tuning, operate over finite-field proxies rather than native BF16/FP32 GPU execution, and plateau at $\sim$ 13B parameters. Alternative verification approaches based on dedicated hardware, such as TEE-attested accelerators, HBM-monitoring coprocessors (Petrie 2025), and NICs retrofitted with verification chiplets (Petrie and Aarne 2025), face two structural obstacles independent of cryptographic feasibility. Their silicon is proprietary to a handful of vendors, a liability for international verification where trust cannot rest on any single firm's supply chain. The research-and-development cycle for verification-grade custom silicon (specification, tape-out, formal RTL verification, manufacturing, certification, and cluster-wide deployment) is realistically six to ten years before the primitive becomes available to regulators.[2]

We propose a verification architecture that sidesteps these limitations and that we estimate can be deployed within approximately 36 months. It circumvents the finite-field arithmetic constraint of prior zero-knowledge ML systems by anchoring the proof in three complementary objects: a pre-committed training specification, inter-node network observations by a verifier-aligned anchor (physical TAP or attested SmartNIC), and on-the-fly Merkle commitments of intermediate computation. The anchors are verified through a generic zkVM equipped with native BF16/FP32 precompiles, so the proof checks the actual floating-point computation the GPU performed rather than a finite-field approximation of it. The protocol produces three proof types: a genesis proof at initialisation, in-training step proofs across the run, and ex-ante attestations enforcing policy-relevant claims as running invariants during training. The architecture is developed for frontier *dense pre-training* as a first target, because it is architecturally the simplest setting, accounts for the largest single block of frontier training compute, and is the root of trust for every post-trained or fine-tuned derivative.

---

declarations and on-demand challenge inspections). Each regime took its verification mechanism to be a prerequisite for the agreement, not an afterthought.

[2] Open-hardware alternatives such as flexible hardware-enabled guarantees (Petrie and Aarne 2025) address the vendor-concentration problem but push the deployment timeline further by requiring international coordination on the hardware design itself.

The architecture does not yet cover sparse mixture-of-experts, reinforcement-learning post-training, multi-datacentre training, or intra-node NVLink traffic. Each is an additive extension rather than a redesign. The full roadmap is thirteen open research and engineering problems, catalogued in Appendix A.

Three contributions follow. (i) An architectural blueprint showing that zero-knowledge verification of frontier dense pre-training is achievable at single-digit-percent training-side overhead, verifying the actual BF16/FP32 hardware execution. (ii) A characterisation of the ex-ante attestation as a governance-enforceable primitive, with compute-threshold enforcement as the immediate use case.[3] (iii) A structured roadmap of thirteen independently-attackable open problems delineating the remaining work to reach a fielded verification primitive.

Related work and current limitations

ZK for inference.

Recent work has applied zero-knowledge proofs to verify ML model inference. ZKML (Chen et al. 2024) builds an optimizing compiler from TensorFlow to halo2 ZK-SNARK circuits, proving inference for models up to GPT-2 (81M parameters) in approximately one hour. zkLLM (Sun, Li, and Zhang 2024) introduces specialized protocols for LLM inference (`tlookup` for non-arithmetic operations, `zkAttn` for the attention mechanism) with GPU-accelerated sumcheck proving, scaling to 13B parameters in under 15 minutes. South et al. (South et al. 2024) propose a general-purpose framework that converts any ONNX model into verifiable evaluation attestations via ezkl, demonstrating proofs across CNNs, LSTMs, and small transformers. These systems enable a model provider to prove that a committed set of weights produces a claimed output on a given input, without revealing the weights.

ZK for training and fine-tuning.

Extending ZK beyond inference to the training process is more challenging, as it requires proving backward passes and parameter updates. VeriLoRA (Liao et al. 2026) is the first framework to achieve this for LoRA fine-tuning, proving forward propagation, backward propagation, and parameter updates for a single mini-batch on models up to 13B parameters (OPT-13B, ~250 s proving time per step on A100). Earlier work on full training includes zkDL (Sun and Zhang 2024), which flattens CNN training into a single circuit

[3] The EU AI Act (Art. 51) sets a cumulative training-compute threshold around 10^{25} FLOPs for general-purpose AI models with systemic risk; this remains in force. The United States executive order 14110 used 10^{26} FLOPs on similar lines before its revocation in 2025. Both regimes rely on trainer self-reporting.

architecture (FAC4DNN) for models with tens of millions of parameters. Distributed settings such as Drynx and zkMLaaS combine MPC and interactive proofs to check aggregated updates, but remain limited to low-complexity learners.

Structural limitations.

All existing ZK-ML systems share two fundamental constraints that prevent their application to frontier training verification.

First, they operate over finite fields, where arithmetic is exact and associative. This is a structural requirement of SNARKs and sumcheck protocols, not a design choice. Every model must be converted to fixed-point arithmetic over a prime field before proving. The proof therefore verifies a different computation than what actually executes on GPU hardware, where training runs in BF16 or FP32 with non-associative floating-point arithmetic and hardware-specific rounding. For inference attestation this gap is acceptable: the quantized circuit is a reasonable proxy for the model's behavior. For training verification it is not, because the goal is to prove that a specific sequence of GPU operations was performed as claimed.

Second, proving cost makes monolithic proofs of training intractable. The fastest existing system (zkLLM) takes 15 minutes to prove a single forward pass at 13B parameters. A frontier training run involves millions of forward-backward steps at hundreds of billions of parameters (up to $\sim$ 1T total for sparse mixture-of-experts models), on distributed clusters with complex communication patterns (FSDP, TP, PP) that no existing system models.

Comparison of ZK-ML verification systems. Existing approaches prove computations over finite-field proxies of the model. Our approach verifies the actual BF16 GPU execution via deterministic training, physical observation, and targeted ZK proofs.

	ZKML	zkLLM	ezkl	VeriLoRA	**Ours**
Scope	Inference	Inference	Inference	Fine-tuning	**Pre-training**
Arithmetic	Finite field	Finite field	Finite field	Finite field	**Native BF16**
Max scale	81M	13B	250K	13B	**100-1000B (est.)**
Multi-step	Per-sample	Single pass	Per-sample	1 LoRA step	**Full run**
Distributed	No	No	No	No	Yes
Proves HW exec.	No	No	No	No	Yes

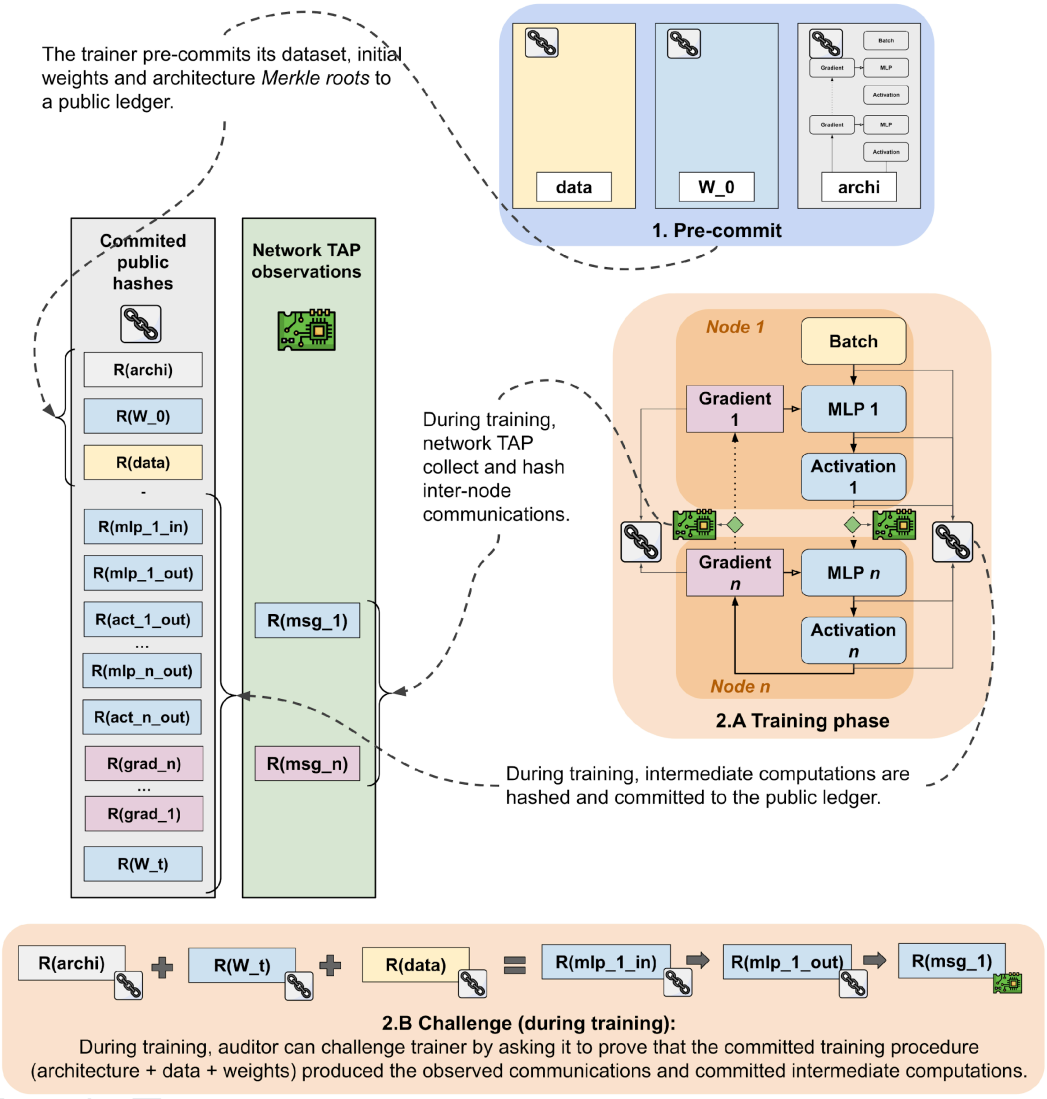
Proposed solution

Given the identified limits in Section 2, we propose a verification architecture that circumvents the intractability of monolithic ZK proofs of training by combining physical network measurements with lightweight cryptographic proofs. We first formalise the threat model, then present the general architecture and detail each component.

Scope.

The architecture developed in this paper targets *dense pre-training*. Today's frontier spans dense models at several hundred billion parameters (e.g. Llama 3.1 at 405B) and sparse mixture-of-experts architectures approaching one trillion total parameters with a small fraction active per token (e.g. DeepSeek-V3 at 671B, Kimi K2 at roughly 1T with $\sim$ 32B active per token). Two further regimes are increasingly load-bearing for frontier capability: sparse mixture-of-experts, and reinforcement-learning post-training (RLHF, RLAIF, rule-based RL for reasoning). Both relax the paper's implicit assumption that the computation graph is fully determined by the `arch_spec` ahead of time: MoE routing is data-dependent, and RL rollouts are stochastic and model-dependent. Each therefore requires protocol-level extensions beyond what is presented here. We treat these as distinct open problems (Appendix A, OP-10 for MoE and OP-11 for RL) rather than rolling them into the base architecture. We begin with dense pre-training because it is architecturally the simplest setting, because every post-trained model is rooted in a pre-training run (if pre-training is not verifiable, nothing downstream is), and because pre-training remains the largest single block of training compute even when the final deployed system uses MoE and RL.

General overview



Overview of the verification protocol. (Top) Pre-commit: the trainer publishes Merkle roots of the training specification, initial weights, and dataset to a public ledger. (Middle) During training, the trainer commits Merkle roots of intermediate tensors per node to a per-step hash chain, while a verifier-aligned network anchor independently hashes inter-node communications. (Bottom) Challenge: the auditor asks the trainer to prove that the committed procedure (architecture, weights, data) produced the committed intermediate computations and the observed network messages.

We propose a verification architecture that circumvents the intractability of monolithic ZK proofs of training by combining three complementary trust anchors:

- A pre-committed training procedure, model weights and dataset (3.2.2), bound to the physical cluster via a challenge-response genesis proof (3.4.1);
- External network TAPs providing public hashes of ongoing inter-node communications, anchoring the training to independently observable data (3.2.4);
- Merkle roots of selected intermediate computations, committed on-the-fly (3.2.3) and available for retrospective sampling challenges (3.4.2).

These anchors are verified through a zero-knowledge Virtual Machine (3.3.2) running a generic proof checker (3.3.3) against a committed training specification (3.3.1). The proof system operates in two phases: an *offline phase* performed once at initialisation (3.4.1), which binds the training procedure, initial weights and dataset to physical measurements via a verifier-supplied random challenge; and an *online phase* during training, where continuity proofs ensure that the committed procedure is the one actually being executed (3.4.2).

Modification of current AI training procedures

Enforcing determinism

Verification requires bit-exact determinism: the proof checker must reproduce the exact values committed during training. Standard GPU training is non-deterministic because floating-point reductions in cuBLAS, NCCL collectives, and attention kernels depend on thread scheduling, producing different bit-level results across runs.

Hardware-level determinism.

Achieving bit-exact determinism at acceptable cost requires per-operator kernel selection (FlashAttention-2 deterministic mode, pinned cuBLAS algorithms, NCCL NVLS reductions). Measured end-to-end overhead on Llama 7B FSDP (8 ×H100) is 1. 6–8. 2% depending on sequence length, dominated by the attention backward pass. Recent work on deterministic FlashAttention-3 scheduling (Qiang et al. 2026) reduces this attention overhead further, suggesting that the determinism tax on compute kernels is an engineering problem with a clear path to near-zero overhead. Kernel-configuration details and per-operator benchmarks are in Appendix C.

Network-level determinism beyond FSDP.

FSDP's reduce-scatter and allgather collectives are deterministic under NVLS at near-full bandwidth. Frontier training additionally uses tensor parallelism (TP), pipeline parallelism (PP), and expert parallelism (EP). PP and EP reduce to per-layer guarantees and to NVLS-deterministic all-to-all respectively. TP is the hard case: its intra-node allreduce over NVSwitch is *not* deterministic under NVLS above 128 MB, and the available deterministic

NCCL configurations incur 64–89% bandwidth loss. A custom deterministic allreduce kernel exploiting TP's fixed topology and message sizes is the proposed path; benchmarks and design are given in Appendix C.

Pre-committing public Merkle roots of the training procedure, model weights and dataset

Before training begins, the trainer publishes the commitment:

$$h_{commit} = H\Big(H(arch\_spec) \;\|\; MerkleRoot(dataset) \;\|\; MerkleRoot\big(w_0\big)\Big)$$

The dataset and initial weights use Merkle roots rather than flat hashes. This allows the prover to demonstrate membership of individual batches or per-layer weights via Merkle paths, without revealing the full dataset or model. The deterministic shuffling seed in the `arch_spec` ensures that, given a step number, the batch composition is fully determined and verifiable against the committed dataset.

The hash-of-hashes structure allows each component to be verified independently. The genesis proof checks all three. A continuity proof only needs Merkle paths for the relevant batch and layer weights.

On-the-fly Merkle root computation

During training, a GPU concurrent stream computes Merkle roots of selected intermediate tensors at every step. For each layer, six tensors are committed: layer input, layer output, Q, K, V, and attention output. The Merkle roots are computed using the same zkVM-native hash as precompile 3 (Poseidon in the current instantiation) and published in a per-step hash chain.

The per-step hash chain is the anchor for retrospective verification. When the verifier challenges a specific step, the prover must produce values consistent with the Merkle roots published at that step. The roots become immutable trust anchors once the per-step hash chain is frozen by a verifier-signed terminal anchor before the audit seed is opened; publication during training alone is not the soundness condition.

Overhead.

The hashing kernels run on a GPU concurrent stream alongside training. Measured bounds are a pure-HBM-read overhead of 0. 2% (bandwidth contention is negligible) and a 2-pass reduction-kernel overhead of 1. 6–3. 3% (dominated by SM contention). With an optimised kernel for the chosen hash, we estimate production overhead at 0. 5–1. 5%, between these two bounds. Combined with the determinism tax, the total trainer-side overhead is ~2–10% depending on sequence length; the full breakdown is in Table 9. The GPU concurrent stream is the default; the workload can also be offloaded to a SmartNIC or DPU for

network-visible tensors, or to the CPU for small commitments, at the cost of additional hardware or memory-bandwidth trade-offs (details in Appendix B).

Network anchoring

This component provides a record of the training run that is not generated by the trainer. Without it, the pre-commit (Section 3.2.2) and the on-the-fly Merkle roots (Section 3.2.3) are all trainer-produced, leaving open the fabrication of internally consistent commitments for computations that never physically occurred. An independent physical witness raises the cost of such phantom-training attacks to the point where executing the real workload is the cheapest remaining option.

We consider two deployment tiers. Both commit a Merkle tree over inter-node collective payloads using a hash function that can sustain cluster-scale throughput in silicon.[4] A tree structure is used rather than a flat hash so that the zkVM verifies a single path of depth $\log n$ at challenge time instead of rehashing the entire payload.

Tier 1: verifier-owned physical TAP.

A passive device installed by the verifier on the inter-node links. It captures frames at wire speed, strips volatile per-packet fields, canonicalises the payload, and hashes it into the Merkle tree. Reconciliation relies on the NCCL determinism established in Section 3.2.1: under a pinned configuration the wire byte sequence is a deterministic function of the step inputs, so the trainer reconstructs the same tree from a replay of the step (no separate wire-byte storage is required beyond the rolling window of Section 3.2.5). The trust root is physical: subversion requires tampering with hardware the verifier controls.

Tier 2: attested SmartNIC.

In clusters provisioned with programmable DPUs, the same hashing can run as signed firmware inside the NIC, exposing its root over an attested channel rooted in the DPU's secure boot and device identity key. The trust root is cryptographic, not physical: it requires trusting that the firmware was not replaced and that the device key has not been extracted. Tier 2 is weaker than Tier 1 against adversaries capable of firmware or supply-chain attacks but deploys on infrastructure already present in modern clusters.

Deployment trajectory.

The two tiers are stages of a single trajectory rather than equivalent alternatives. Tier 2 is the natural starting point for early adopters, industry-led auditing, and cooperative pilots: it

[4] Concretely a SHA-256 Merkle tree (precompile 8 in Table 3). Hashes with low in-circuit cost such as Poseidon cannot be computed at the required rates on commodity silicon; see Appendix D for the detailed comparison.

builds on infrastructure already in place and delivers a meaningful anchor against casual and software-level tampering. The long-term goal is Tier 1 deployed uniformly: any verification regime intended to hold against well-resourced actors with a material interest in subverting the record, including any framework grounded in international agreements, requires an anchor whose trust root does not depend on the audited party's own hardware. Tier 2's role is to carry the verification ecosystem through its first years while Tier 1 infrastructure is designed, standardised, and deployed.

The approach depends on a canonical mapping between logical tensors and NCCL's on-wire byte sequence, which is not a stable public interface today: we treat this as the central open engineering problem of network anchoring and discuss it with the other implementation limitations in Section 4.1 and in Appendix D.

Coverage.

Either tier observes only inter-node traffic; intra-node communication over NVLink is invisible, so hybrid FSDP+TP training is covered at the FSDP layer only. The ZK proofs still verify the full computation, but the physical witness stops at the node boundary. The candidate mechanism for intra-node coverage (GPU-resident trusted execution) is discussed alongside the other open problems in Appendix A.

Weight storage rolling window

The prover must be able to respond to challenges about recent or windowed post-training ranges. Recomputing from a distant checkpoint (replaying the forward pass from step 0 to step N) would be prohibitively expensive. Instead, the trainer stores a rolling window of model weights at every step on local storage (NVMe arrays or parallel file systems such as Lustre or GPFS, which are already provisioned for checkpointing on training clusters).

At 405B parameters in BF16, each snapshot is approximately 810 GB. A rolling window of 100 steps requires 81 TB. When challenged on step N, the prover loads the stored weights directly (a file read, not a replay) and runs a single forward pass to recompute the activations needed for the proof.

The key trade-off is storage space against challenge response time. Whole-run post-training sampling therefore needs either full retention or deterministic replay from durable checkpoints plus committed replay metadata: optimizer state, data-loader state, stochastic-component state, kernel/reduction-tree and precision metadata, checkpoint interval commitments, and the committed seed schedule. Rolling windows are valid for recent-or-windowed challenge policies only; they reduce storage but require replay for steps outside the window. At frontier scale, 81 TB of local storage is modest relative to the cluster's existing storage infrastructure.

Estimated overhead of the required modifications

The total training-side overhead, estimated at Llama 3.1 405B scale on a $100M training budget:[5]

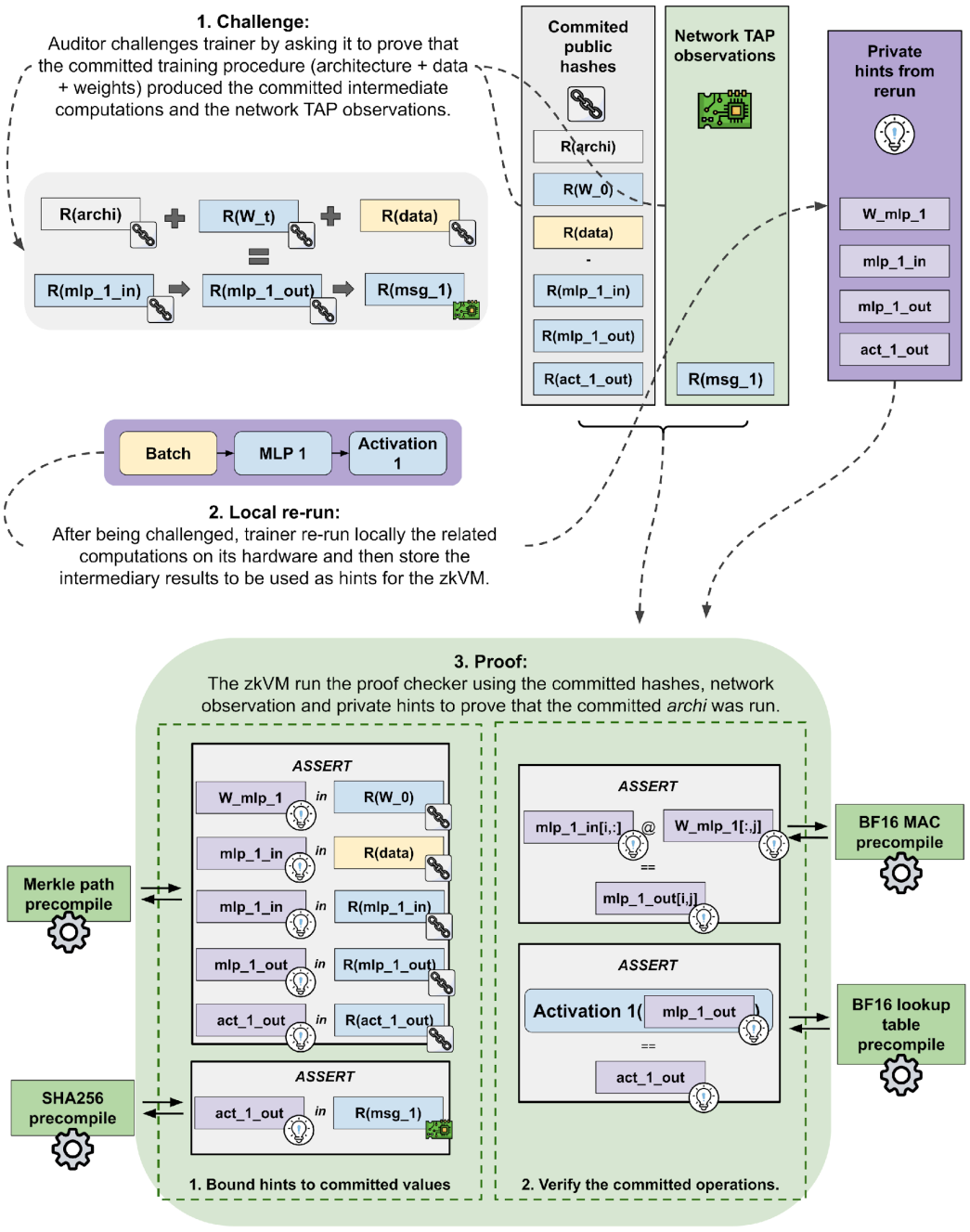
Estimated training-side overhead at Llama 3.1 405B scale. The dominant cost is determinism, driven by the attention backward pass. These figures assume FSDP-only parallelism. FSDP+TP setups will incur additional overhead from TP allreduce determinism until a custom kernel is deployed.

Component	Cost	% of training
Determinism (1.6–8.2%)	$1.6–8.2M	1.6–8.2%
Merkle computation (~ 0.5–1.5%)	$0.5–1.5M	0.5–1.5%
Weight storage (81 TB rolling)	negligible	<0.1%
Total (FSDP)	**$2–10M**	**~2–10%**

These figures cover modifications to the training pipeline only. Verification costs (genesis proof, sampling challenges) are borne by the proving system and are detailed in Section 3.4.

[5] Taken as a rough frontier-scale reference. Published estimates for Llama 3.1 405B-class training runs fall in the $60M–$120M range depending on whether cloud GPU pricing or amortised hardware-plus-energy costs are used. The Llama 3 technical report (Dubey et al. 2024) gives the compute budget (3.81025 FLOPs on 16,384 H100s) from which GPU-hour costs are derived; independent analyses by Epoch AI (Cottier et al. 2024) and SemiAnalysis (Patel and SemiAnalysis 2024) place the total training cost for models in this class within the same range.

Architecture of the proving system - Necessary components



Proof generation from an in-training challenge. (Top) The auditor challenges the trainer to prove that the committed procedure (architecture, weights, data) produced the committed intermediate computations and the observed network messages. (Middle) The trainer re-runs

the challenged operations locally on its own hardware and produces the intermediate tensors as private hints to the zkVM. (Bottom) The zkVM runs the proof checker in two phases. First, each private hint is bound to its committed Merkle root via a path verification (precompile 3 for trainer-side commitments, precompile 8 for the network-anchor commitment). Second, the declared operations are verified against the bound hints: matrix products by interactive sampling with the BF16 MAC precompile (precompile 1), element-wise activations with the BF16 lookup-table precompile (precompile 2). Each precompile manipulates integers following the IEEE 754 procedure for the given floating-point operation (sign-exponent-mantissa decomposition, mantissa arithmetic, RNE rounding), and encodes those integer operations as constraints over the prover's native prime field F_p. The proof therefore verifies the actual BF16/FP32 GPU computation rather than a finite-field approximation of it.

The proving system has four components: a training specification, a zkVM, a proof checker, and a set of precompiles. Its central design principle is that the zkVM never re-executes the training computation. When challenged on a specific training step, the prover loads the stored weights for that step, re-executes the forward and backward passes on GPU, and provides all intermediate tensor values as hints to the zkVM. The zkVM only verifies that these hints are consistent with the Merkle roots committed during training.

Two distinct categories of data enter the proof:

- **Public inputs** (Merkle roots): committed during training on a concurrent GPU stream and linked into a per-step hash chain. They become immutable trust anchors when the chain is frozen by a verifier-signed terminal anchor before the audit seed is opened.
- **Private hints** (full tensor values): produced by the host at challenge time by re-executing the training step from stored weights. These are unchecked claims that must be verified against the public Merkle roots.

Verification proceeds in two layers: first, each hinted value is bound to its committed Merkle root via a path check (preventing the host from fabricating hints); then, the computation on the bound values is verified using the appropriate precompile (preventing the host from committing incorrect values during training).

The training specification

The training specification (or `arch_spec`) is a structured, machine-readable description of one complete training step. It serves as the bridge between the trainer's GPU code and the proof checker: the trainer generates it from their training pipeline, and the proof checker reads it to determine which operations to verify and in what order.

The `arch_spec` declares: the model architecture as a sequence of typed operations per layer (GEMMs with dimensions, activations with function type, normalisations), the numerical precision (BF16 parameters, FP32 accumulators, IEEE 754 RNE rounding), the distributed communication pattern (allgather, reduce-scatter, their ordering), the data loading parameters (batch size, shuffling seed for deterministic batching, master RNG seed, and registered stochastic components), and the tensor boundaries at which Merkle roots are committed during training. This training RNG material is distinct from the verifier's audit seed: the former determines the claimed stochastic training execution, while the latter selects which committed positions are opened after the relevant stream segment has been frozen.

The `arch_spec` fully describes the training procedure. No separate commitment to the training source code is needed: the genesis proof (Section 3.4.1) validates that the trainer's actual GPU execution produces outputs consistent with the `arch_spec`. If the code diverges from the specification, the proof fails. The commitment structure is described in Section 3.2.2.

The zk Virtual Machine

The zkVM executes the proof checker inside a cryptographic environment that produces a succinct, non-interactive proof of correct execution. We require three capabilities: application-defined precompiles (native constraint circuits callable from the guest program), GPU-accelerated proof generation, and support for continuations and recursive proof composition.[6]

The zkVM operates on a *hint-and-verify* model. The host (the trainer's GPU) runs the full computation outside the proof and provides all intermediate values as hints to the guest. The guest (the proof checker, running inside the zkVM) does not re-execute the computation. It receives the hints and verifies their correctness by calling precompiles against Merkle-committed values. Since the host computation happens outside the proof, it generates zero constraints. Only the guest's verification logic contributes to the proof. This reduces the cost by approximately five orders of magnitude compared to full re-execution.[7] This hint-and-verify pattern is standard in zero-knowledge systems for machine learning (Kang et al. 2022; Chen et al. 2024; Sun, Li, and Zhang 2024), where re-execution of neural network operations inside the proof system has been shown to be impractical.

[6] We use RISC Zero as our reference implementation, as it satisfies all three requirements. The architecture is not coupled to this choice.

[7] The reduction factor is mn/kcsw/cpre, where mn is the GEMM output size, k the number of sampled entries, csw1,000 the cost of a software-emulated MAC in RISC-V cycles, and cpre90 the cost of a native MAC precompile in constraints. For typical configurations this ranges from 10^4 to 10^6.

The pattern has also been extended to training verification (Garg et al. 2023). We provide a complete walkthrough in Appendix E.

The same pattern extends to the backward pass, which involves transposed GEMMs (for gradient propagation), elementwise gradient computations (derivatives of activation functions), and optimizer state updates (Adam momentum and variance). The host GPU computes all gradients and updated parameters, provides them as hints, and the guest verifies sampled entries using the same precompiles: the MAC precompile handles gradient GEMMs, the BF16 lookup table handles activation derivatives (since BF16 GELU′(x) is also a 2^{16}-entry table), and the FP32 nonlinear precompiles handle the optimizer's sqrt and division operations. This is what makes it possible to verify training, not just inference.

The proof checker

The proof checker is a public, auditable program compiled to the zkVM's instruction set. It reads the `arch_spec`, iterates over the declared operations, and dispatches each to the appropriate precompile for verification. The same binary works for any model: only the `arch_spec` differs between a 10M-parameter proof of concept and a trillion-parameters frontier model.

The `arch_spec` is a *private* input to the proof: the verifier never sees it, only $H(arch\_spec)$ as part of h_{commit}. This preserves the trainer's intellectual property, since neither the model architecture nor its dimensions are revealed. The concrete structure of `arch_spec` (layer types, precision configuration, parallelism and communication schedule, etc.) is given in Appendix F. What the verifier actually obtains from a successful verification is: the verifier binary (public), the proof, and the public Merkle roots. What the verifier can nevertheless infer from public metadata is narrow and low-sensitivity: the number of layers (from the length of the per-step hash chain) and the approximate model scale (from the published weight-storage footprint or regulatory filings). Both quantities are already the subject of the regulatory regimes this work serves, so their disclosure is not a new information leak.

This genericity minimises the trust surface. The verifier does not need to trust the trainer's GPU code or training framework. It needs to audit the proof checker (a single, public binary), the precompile specifications, and the underlying zkVM's cryptographic soundness. This is a much smaller surface than trusting the trainer to report honestly.

The precompiles

Precompiles are native constraint circuits hardwired into the zkVM's proof system. They exist because verifying floating-point arithmetic through generic RISC-V instructions would be prohibitively expensive: each precompile implements a specific numerical operation as a compact set of constraints over the zkVM's native field.

This is a subtle but important distinction from prior ZK-ML systems. Prior approaches convert the *model's computation* into finite-field arithmetic: the GEMM itself is executed over F_p, using fixed-point representations of the weights and activations. The proof verifies this finite-field computation, which is a different computation than what the GPU performed. Our precompiles take the opposite approach: the GPU executes the actual BF16/FP32 computation, and the proof system verifies *individual operations* by encoding IEEE 754 semantics as constraints over F_p. Concretely, each floating-point value is decomposed into its sign, exponent, and mantissa (represented as integers in F_p), and the constraints enforce that the integer relationships between these components match the IEEE 754 specification for the given operation (multiplication, accumulation, rounding). The proof system operates entirely over F_p internally, but what it proves is a statement about the floating-point computation that actually ran on the GPU.

Eight precompiles cover the operations in a Transformer training step:

Precompile catalogue. The MAC chain (1) dominates the proof budget, accounting for over 99% of constraints at Llama 3.1 405B scale. The Merkle path (3) is invoked per sampled value and is structurally essential but contributes under 0.1% of the budget. SHA-256 (8) is invoked only when opening paths against the network anchor's commitment, on sampled wire-bound tensors. See Appendix B for the full breakdown.

#	Precompile	Constraints/op	Used for
1	BF16/FP32 MAC chain	~90	GEMM dot products
2	BF16 lookup table	~15	GELU, SiLU (2^{16} entries, exhaustive)
3	Merkle path (zkVM-native hash)	~1,500	Binding hints to trainer commitments
4	FP32 exp	~250	Softmax, cross-entropy loss
5	FP32 sqrt/rsqrt	~200	RMSNorm, Adam
6	FP32 div	~200	Normalisation, Adam
7	FP32 log	~250	Cross-entropy loss

#	Precompile	Constraints/op	Used for
8	SHA-256 compression	~7,500	Network-anchor reconciliation (Section 3.2.4)

BF16/FP32 MAC chain (precompile 1).

The dominant precompile, accounting for over 99% of all constraints at Llama 3.1 405B scale. It verifies a single multiply-accumulate: a BF16 multiply producing an FP32 intermediate, accumulated into an FP32 running sum. The circuit decomposes each BF16 input into sign, exponent, and mantissa, verifies the multiplication via integer arithmetic on the mantissa, and checks FP32 accumulation with IEEE 754 round-to-nearest- even rounding. Each MAC costs approximately 90 constraints; a full dot product of dimension d requires d sequential MACs. Used for all GEMM verification in both forward and backward passes.

Other precompiles.

Precompile 2 uses a pre-computed 2^{16}-entry exhaustive table for BF16 activations (GELU, SiLU, and derivatives) at ~15 constraints per element. Precompile 3 verifies Merkle-path membership using a zkVM-native hash (Poseidon today, ~1, 500 constraints per depth-30 path); every hinted value must pass a path verification to bind it to the committed root. Precompiles 4–7 encode FP32 nonlinearities (exp, log, sqrt/rsqrt, div) used sparsely in normalisation, softmax, optimiser, and loss, at 200–250 constraints per call for under 1% of the total proof budget. Precompile 8 (SHA-256 compression, ~7, 500 constraints per block) is used exclusively for network-anchor reconciliation (Section 3.2.4). Per-circuit details and the rationale for the Poseidon + SHA-256 dual-hash design are in Appendix B and Appendix D.

GEMM verification via interactive sampling.

GEMMs dominate the computational cost of a training step. Exact verification of a GEMM of dimensions $m \times d \times n$ at ~90 constraints per MAC costs $O(mnd)$ constraints, which at Llama 3.1 405B scale exceeds what current proof systems can handle by several orders of magnitude. Freivalds' algorithm reduces this to $O(mn + md + nd)$ over a finite field but does not apply to BF16/FP32 arithmetic because floating-point addition is not associative (details in Subsubappendix B.2.2). We use interactive per-entry sampling instead: the host provides the full result as a hint, the verifier selects k random output entries, and for each entry the proof checker recomputes the dot product via the MAC precompile and binds the inputs and output to their committed Merkle roots. With $k = 4,605$ samples, deviations affecting at least 1% of entries are detected with probability $1 - 10^{-20}$. Detection-rate analysis as a function of deviation magnitude and the per-step proof-cost breakdown are in

Appendix B. Recent work on approximate sumcheck (Bitan et al. 2025) suggests a path to recovering Freivalds-like efficiency for floating-point GEMMs; see Section 4.1.

Per-layer decomposition and fused operations.

Each layer is proved independently; soundness across layers is maintained by chaining Merkle roots (the committed root of layer L's output equals the committed root of layer $L + 1$'s input, verified inside each layer's proof). Per-layer sub-proofs run in parallel and are folded into a single ~200 KB aggregate via recursive composition (details and cost in Subappendix B.7). Fused GPU kernels (Flash Attention, fused activations) do not materialise intermediate tensors in HBM, so those tensors cannot be Merkle-committed individually; the proof treats each fused block as a unit verified end-to-end at its input and output commitments. For Flash Attention specifically, the commitments are Q, K, V (inputs) and the attention output, and recomputing a sampled attention-output entry requires a full softmax over the sequence dimension, which makes per-sample cost higher and motivates a reduced sample count for fused blocks (see Appendix B).

Protocol for AI training verification

The protocol produces three distinct proof types, each with its own role and firing pattern. The *genesis proof* runs once at initialisation and binds the committed training procedure to physical execution on the trainer's cluster. The *in-training step* proof is fired many times across a training run and verifies that sampled steps are consistent with the published hash chain and the network anchor. The *ex-ante attestation* binds policy-relevant claims (compute, training regime, data-content) at initialisation and enforces them as running invariants throughout training. All three share the commitments from Section 3.2.2, section 3.2.3, and section 3.2.4 and the precompile catalogue of Table 3. Full formal definitions are deferred to Section 3.4.4.

Genesis proof

The genesis proof binds the committed training procedure to actual GPU execution on the trainer's cluster, run once at initialisation before the hash chain begins. Without it, h_{commit} is a signed declaration of intent with no tie to physical work.

Protocol.

The verifier supplies a set of random sample indices $\{i_1, ..., i_b\}$ into the committed dataset. The trainer then:

1. constructs the batch $B = \left(data\left[i_1\right], ..., data\left[i_b\right]\right)$ in the committed canonical order;
2. executes one full training step from W_0 on B (forward, backward, optimiser update);

3. produces a zkVM proof that each sample in B is a committed element of $R(dataset)$ at its claimed index (one Merkle path per sample via precompile 3), that the step executes faithfully according to `arch_spec`, and that the resulting W_1 is the first link of the public hash chain.

Because the indices are chosen after h_{commit} is published, the trainer cannot precompute a fabricated step. The Merkle-path membership checks prevent substitution of a different batch. Running the full step exercises the entire GPU pipeline, so any divergence between claimed and executed code fails the proof. The cost is exactly one training step's worth of proof.

Scope.

The genesis proof establishes faithful execution of the committed procedure at step 0. It says nothing about learning-theoretic properties of the eventual trained model, nor about aggregate quantities such as total compute; those are the subject of the ex-ante attestation in Section 3.4.3.

In-training step proof

Training proceeds after genesis for many steps. The in-training step proof gives the verifier assurance that any sampled step is consistent with the published hash chain and the network anchor. It is fired many times across a training run and may be issued either live (while the run is ongoing) or post-hoc (after the run has concluded, against archived snapshots).

Protocol.

The verifier selects a step $t \in \{1, ..., T\}$, a tensor identifier (layer ℓ output, gradient at node i, an all-reduce payload, etc.), and k random entry indices. The trainer provides hints: the requested tensor entries, Merkle paths against the committed $R(\cdot)$ for step t, and, for wire-bound tensors, the corresponding SHA-256 Merkle path against $R\left(msg_*\right)$. The proof checker runs the precompile dispatch of Section 3.3: Merkle-path verification (precompile 3), operation verification (precompiles 1, 2, 4–7) and, where relevant, anchor consistency (precompile 8). At $k = 4,605$ samples per committed tensor, deviations affecting at least 1% of entries are detected with probability $1 - 10^{-20}$.

Cross-step binding.

Soundness across a training run composes from per-step soundness via two chaining relations. Within a step, layer boundaries are linked by the commitment identity $R\left(layer_{\ell}\_out\right) = R\left(layer_{\ell+1}\_in\right)$, verified inside each layer's proof. Across steps, the weight

root $R(W_{t+1})$ is a deterministic function of $R(W_t)$ and $R(grad_t)$ under the optimiser declared in `arch_spec`, verified as part of the step proof. Together these give a directed chain from $R(W_0)$ (genesis) through $R(W_T)$ (final).

Composition.

Across sampled steps, per-layer proofs and per-step proofs are aggregated via recursive STARK composition (Section 3.3) into a single artefact on the order of 200 KB, matching the aggregate figure of Appendix B. The challenge step t may depend adaptively on previously-observed public roots, since all roots are fixed before the challenge is issued.

Ex-ante attestation

The ex-ante attestation binds policy-relevant claims about the training run at initialisation and enforces them as running invariants throughout the online phase. Covered claim categories include *compute* (total FLOPs, tokens processed, GPU-hours), *procedure* (training regime, absence of a particular phase such as RL), and *data-content* (fraction of training samples flagged by a public classifier below a threshold). A trainer operating under an ex-ante attestation physically cannot exceed a committed bound without producing a publicly-failing proof, turning the training record into a governance-enforceable artefact rather than merely an audit trail.

Commitment extension.

At initialisation the trainer publishes a structured claim object ex_ante_claims containing the committed bounds, structural assertions, and references (e.g. `max_total_flops`, `max_steps`, `training_regime`, `data_filter_hash`). The genesis commitment is extended to

$$h_{commit} = H\big(H(arch\_spec) \;\|\; R(W_0) \;\|\; R(dataset) \;\|\; H(ex\_ante\_claims)\big).$$

The bound values themselves are public; only $arch\_spec$ stays private. Claim opening can happen upfront (the bounds are in the clear from the start) or on demand via selective disclosure against $H(ex\_ante\_claims)$.

Running enforcement.

Each in-training step proof carries an additional public running counter, e.g. $F_{cum}(t)$ for cumulative FLOPs. The step proof computes $F_{cum}(t) = F_{cum}(t-1) + F_{step}(arch\_spec, t)$ in-circuit and enforces $F_{cum}(t) \leq max\_total\_flops$. A violation at any step produces a failing step proof. The same pattern applies to step counts, per-regime step tallies, and any other monotone quantity declared in ex_ante_claims. Cost per step is a single addition and comparison, negligible relative to the MAC budget.

Claim structure.

All attestations share a common form: compute a public function f over committed state and check $f(\cdot) = y$ or $f(\cdot) \leq y$. The committed state varies with the sub-type: $arch\_spec$ and chain length for compute claims; $arch\_spec$ structure for procedure claims; sampled dataset entries plus a public classifier for data-content claims. Data-content attestations combine Merkle-path openings into $R(dataset)$ with in-circuit evaluation of the classifier and a concentration bound (Hoeffding) over the sampled fraction.

Attack surface for compute under-declaration.

Compute-threshold enforcement is our first target, and the adversary of interest under-declares FLOPs to stay below a regulatory tier. Three cheating modes are available: a larger model than declared, more training data than declared, or more training steps than declared. Each has a cheap detection path beyond the generic sampling argument. A wider or deeper model fails the first weight Merkle-path opening because commitment shapes do not match `arch_spec`. Extra training data fails input-Merkle openings at steps that use uncommitted samples: hiding a factor α of extra data forces $1 - 1/\alpha$ of openings to fail, which two challenges catch at 99% confidence for $\alpha = 10$. Extra training steps are visible to the network anchor, whose observed collective count exceeds the committed chain length by a single integer comparison. A compute-threshold attestation under these three detection mechanisms is cheaper to enforce than generic $arch\_spec$ verification against an adaptive adversary. Other attestation classes (for instance data-content filters targeting semantic smuggling, or procedural attestations on fine-grained regime compliance) lack this structural detection surface and require denser sampling. A case-by-case analysis is in Subsubappendix B.4.3.

Ex-post variant.

Emergent properties of the completed run that are not knowable at initialisation (final-weight norms, benchmark performance under a public evaluator) admit a natural ex-post variant of the same $f(X) = y$ structure, executed once against the completed hash chain and final weights rather than maintained as a running invariant. Ex-post attestations do not modify the online-phase protocol; they reuse the precompile toolkit on final commitments.

Protocol-aligned formalization of execution verification

In this section, we formalise the security, in a cryptographic sense, of our proposed solution in Section 3.2.2, section 3.2.3, and section 3.2.4. The architecture in Section 3.2.2, section 3.2.3, and section 3.2.4 ultimately verifies faithful execution of a committed training

procedure, thus we employ proof systems for NP relations to formalise our protocol[8]. We first recall the necessary notions, then provide the formal definitions of our desired properties in Section 3.4.5. Security statements and their analysis are deferred to Subappendix G.1.

An **NP relation** is a polynomial-time decidable predicate $R \subseteq X \times W$ on a public instance $x \in X$ and a private witness $w \in W$. Here x is the public training transcript: commitments, tags, public hyperparameters, the genesis challenge, verifier signatures, and the opened sampling seed after streaming has been frozen. The witness w is the private execution trace: architecture specification, compiled program, data, weights, batches, randomness, optimizer state, intermediate tensors, and traffic. We also make use of cryptographic **commitments**. On one hand, the prover's tensor and weight commitments are **Merkle-style commitments** to values that the prover later opens inside audited proofs. Their relevant property for soundness is binding: after a root has been published, the prover cannot open the same root to two different tensor values except with probability ε_{bind}. On the other hand, the verifier's **seed commitment** $Com(\sigma; \rho)$ has two distinct roles. Binding prevents the verifier from changing σ after the transcript is frozen, while hiding prevents the prover from learning σ during training. For soundness we use the prover-side prediction advantage $\varepsilon_{com,pred}$, not the verifier-side CZK hiding term $\varepsilon_{com,hide}$: for standard hiding commitments $\varepsilon_{com,pred}$ is bounded by the usual hiding advantage, but the games are conceptually different. Before training, during a **commit-then-reveal sampling** procedure, the verifier samples a seed σ and publishes $Com(\sigma; \rho)$. During training, the prover streams $\left(Com\left(w_t\right), h_t\right)$ and the public anchor chain binds these values in order. Only after the terminal anchor is signed by the verifier is (σ, ρ) opened. The step set S, audited layers, and audited entries are then derived from F_σ using fixed-length encodings and separate tags

$$SAMP/STEP, \qquad SAMP/LAYER, \qquad SAMP/ENTRY.$$

This ordering is load-bearing: if σ is known before the stream is frozen, an adaptive prover can place deviations entirely outside the audited positions. The terminal anchor freezes only the ordered public commitment stream. It does not certify that the prover still stores the private opening material for every old commitment, so availability of sampled openings is a separate operational precondition of the spot-check protocol; In terms of required

[8] A potential generalization is given in Subappendix G.3, for a PAC-style verifiability that encompasses learnability and consider distributions over AI models.

properties[9], *(i)* **knowledge soundness** requires that an accepting prover can be converted into an extractor that outputs a witness for the accepted statement. For the BCS-/FRI-style compiled STARK layer, the theorem-level accounting keeps the quadratic random-oracle contribution; outer commitment-chain extraction and wrapper extraction are separate losses (Ben-Sasson, Chiesa, and Spooner 2016; Ben-Sasson et al. 2018). We do not package these terms into a clean deployed-stack 2^{-128} theorem. *(ii)* **Auxiliary-input computational zero knowledge** requires that for every polynomial-time verifier with auxiliary input z, there is a simulator whose output is computationally indistinguishable from the real proof transcript, up to the public transcript and accept/reject bit (Goldreich 2001; Lindell 2015); *(iii)* Finally **completeness** requires that possessing a witness for an instance that satisfies the NP relation will allow the prover to convince the verifier with high probability.

Universal target relation and sampled verifier predicate

The paper should keep separate the relation the protocol is trying to certify from the predicate actually opened by the current sparse-auditing protocol. The target relation R_{train} is the universal execution relation over the full training run. The current spot-check verifier checks a sampled restriction, denoted $R_{spot}^{S,Q}$, and soundness for R_{train} pays the M2/M3 sampling terms below.

Let x_{run} be the public run instance containing: (i) the pre-commitment

$$R = H\Big(H(arch\_spec) \parallel MerkleRoot(dataset) \parallel MerkleRoot\big(w_0\big)\Big),$$

(ii) public hyperparameters θ, including T and the training schedule, (iii) the verifier public key vk_V, (iv) the genesis challenge batch and TAP tag, (v) streamed commitments $\{Com\big(w_t\big), h_t\}_{t=1}^{T}$, and (vi) the terminal anchor $anchor_T$ and verifier signature σ_{chain}.

The witness w contains the private execution trace: $arch\_spec$, the compiled zkVM program P_{prog}, the dataset, the weight trajectory $\{w_t\}_{t=0}^{T}$, the batches $\{B_t\}_{t=1}^{T}$, prover hints, intermediate tensors, optimizer state, registered stochastic-component states, and the raw network traffic $\{traffic_t\}_{t=1}^{T}$. In the base TAP model, the keyed-hash key K is a fixed run secret in the prover/TAP trust base and hidden from the verifier; the proof either treats K

[9] All properties in this paper are sequential and per-proof: it covers the proof objects in one audit session under the usual programmable-random-oracle modeling of Fiat–Shamir. It is not a generic concurrent-composition, GUC, or EUC claim.

as a private witness value shared with the TAP or treats TAP-tag correctness as an external functionality assumption.

We write $\left(x_{run}, w\right) \in R_{train}$ if all of the following hold:

1. **Commitment and anchor consistency.** The pre-commitment equation defining R holds; each $Com\left(w_t\right)$ is the Merkle apex of the declared committed tensors and optimizer state for step t; and the streamed values form the anchor chain

$$anchor_0 = H(ANCHOR/INIT \parallel R),$$

$$anchor_t = H\left(ANCHOR/LINK \parallel anchor_{t-1} \parallel Com\left(w_t\right) \parallel h_t \parallel t\right).$$

   The terminal signature verifies:

$$Verify\left(vk_V, ANCHOR/LINK \parallel R \parallel T \parallel anchor_T, \sigma_{chain}\right) = 1.$$

2. **Program/spec binding.** The zkVM program is the publicly specified deterministic compilation of the committed architecture:

$$P_{prog} = f_{compile}(arch\_spec).$$

3. **Batch and RNG derivation.** The batch schedule and every stochastic component output are derived from the committed dataset root, the master RNG seed, and the registered stochastic-component list in $arch\_spec$. Stateless randomness uses domain-separated keyed derivations such as $RNG/STEP$ and $RNG/ < COMP >$ with fixed-length encodings. Stateful stochastic rules, such as dynamic loss scaling, are updated by their declared deterministic transition functions.

4. **Genesis.** The genesis execution on the verifier's challenge batch is consistent with P_{prog}, w_0, and the TAP tag

$$h_{gen} = F_K\left(traffic_{gen}\right).$$

5. **Universal step consistency.** For every $t \in [T]$, the committed transition from $\left(B_t, w_{t-1}\right)$ to w_t is the transition prescribed by P_{prog}, and the produced traffic matches the TAP tag:

$$Exec\left(P_{prog}, B_t, w_{t-1}; h_t\right) = \left(w_t, traffic_t\right), \qquad h_t = F_K\left(traffic_t\right).$$

6. **Operation-level consistency.** Every committed operation value used in the transition is consistent with the relevant operation semantics. For GEMMs this

means bit-exact MAC-chain consistency against the committed operands; for nonlinear and lookup-heavy operations this means the corresponding precompile semantics for exp, sqrt/rsqrt, division, log, BF16 lookup, attention blocks, and normalization paths.

7. **Layer-boundary consistency.** Layer boundaries chain correctly: a layer's committed output root is the next layer's committed input root wherever the architecture declares such a boundary.

Let the spot-check transcript be

$$x_{spot} = \left(x_{run}, Com(\sigma), \sigma, \rho, S, Q\right).$$

Here Q denotes all sampled layer and entry positions. The predicate $R_{spot}^{S,Q}$ first checks

$$Open(Com(\sigma), \sigma, \rho) = 1,$$

and then checks that S, the audited layers, and the audited entries are exactly the outputs of F_σ under the fixed tags $SAMP/STEP$, $SAMP/LAYER$, and $SAMP/ENTRY$, using the protocol's fixed-length encoding convention. It checks (U1)–(U4) as public consistency conditions, and it checks (U5)–(U7) only at the sampled steps, layers, and entries specified by (S, Q).

Thus the current proof objects are arguments for $R_{spot}^{S,Q}$. They are arguments for the universal target relation R_{train} only up to the sampling miss terms in the soundness bound. In a V5-IVC architecture, the step restriction $t \in S$ is replaced by $t \in [T]$; this removes the M3 branch but replaces it with recursive-proof, commitment, and extractor costs. It does not by itself remove all operation-level sampling costs unless the per-step verifier is also strengthened.

Conclusion

We proposed a verification architecture for frontier dense pre-training that combines three complementary anchors (a committed training specification, inter-node network observations, and on-the-fly Merkle commitments of intermediate computation) verified through a generic zkVM with native BF16/FP32 precompiles. The architecture sidesteps the scale and arithmetic limits of prior zero-knowledge ML systems by verifying the actual hardware execution rather than a finite-field approximation of it. The protocol produces three proof types (genesis, in-training step, and ex-ante attestation), the last of which enforces policy-relevant claims as running invariants and turns the training record into a governance-enforceable artefact rather than an audit trail. At Llama 3.1 405B scale the

training-side overhead is single-digit percent, aggregate proof size after recursive composition is approximately 200 KB, and the end-to-end verification cost under a layered challenge mix (many single-commit challenges, fewer full-step audits) is a fraction of a percent of training budget. For the specific target of compute-threshold attestation the cost is lower still, because each of the three under-declaration modes (larger model, bigger data, more steps) admits a direct structural detection that does not require the generic sampling budget.

Limitations and open problems

The architecture of Section 3 leaves a number of open research and engineering problems. The most consequential are summarised below; full statements, success criteria, and references are in Appendix A, and a public-facing version of the roadmap is maintained at `gpaipolicylab.org/verification`.

Proof foundations.

The MAC precompile accounts for roughly 99% of the proof budget at frontier scale, so constraint-level improvements (OP-2) translate directly into proving cost. A more structural win would come from an algebraic reduction replacing per-element sampling (OP-1), whose feasibility for floating-point arithmetic is open; recent work on approximate sumcheck (Bitan et al. 2025) is a promising starting point. A full zero-knowledge proof of backpropagation at $\geq 10^6$ parameters (OP-3) is the single highest-risk milestone of the programme: it has never been demonstrated and a negative result would require redesigning the protocol.

Deterministic execution on hardware.

Attention backward is the dominant source of the determinism tax (23–82% overhead scaling with sequence length); a deterministic-by-construction kernel under 5% overhead (OP-4) is the most important single lever. Cross-architecture portability of the precompile catalogue (OP-5) remains hand-engineered today and would benefit from a parameterised precompile family instantiated from a formal hardware descriptor.

Hardware trust anchors.

The network anchor (Section 3.2.4) is the one component of the architecture that does not depend on trainer-generated data, and is therefore disproportionately load-bearing for the overall trust model. Three open problems shape its feasibility: an open-hardware TAP at line rate (OP-6), a canonical wire-to-tensor mapping for NCCL (OP-7, the central engineering problem of the approach), and intra-node coverage via GPU-resident trusted execution (OP-8). Distinguishing silent data corruption from adversarial deviation (OP-9) is a separate but related question affecting the interpretation of challenge failures.

Protocol extensions.

The base protocol covers dense pre-training on a single site. Three extensions, each a separate open problem, must follow: sparse MoE architectures with data-dependent routing (OP-10), reinforcement-learning post-training with stochastic rollouts and reference-model binding (OP-11), and multi-datacentre training with administratively partitioned links (OP-12).

Protocol tooling.

The protocol commits to `arch_spec` as the canonical description of the training computation, but trainers write PyTorch or JAX, not `arch_spec`. Bridging the two requires a converter pipeline (OP-13), a completeness concern rather than a soundness one: converter bugs produce failing proofs for honest trainers, not passing proofs for dishonest ones.

Outlook

Thirteen open problems separate this architecture from a fielded verification primitive. Their resolution is the remainder of the roadmap. The critical path passes through three items in particular: a deterministic-by-construction attention kernel (OP-4), a canonical wire-to-tensor mapping for NCCL (OP-7), and a zero-knowledge proof of backpropagation at non-trivial scale (OP-3). The first two are engineering problems with measurable success criteria. The third is the highest-risk milestone on the roadmap: a fundamental obstacle there would require rethinking the protocol, not just refining its implementation. Each item is independently scoped, so the architecture and its roadmap can advance in parallel across the cryptography, systems, hardware, and machine-learning communities. Whether the 36-month deployment envelope estimated here holds depends on coordinated progress along these fronts.

Open problems

This appendix catalogues the open research and engineering problems that remain after the architecture of Section 3. The list complements and expands the public roadmap maintained at `gpaipolicylab.org/verification`. Problems fall into four areas: proof foundations (questions that could shrink the proof budget), deterministic execution (questions that would reduce the training-side overhead), hardware trust anchors (questions that determine how independent the witness can be), and protocol extensions (questions about training regimes and deployment topologies beyond the base protocol). Each problem is phrased as a research question with a named target; they are independently attackable and invite contributions from cryptography, systems, hardware, and ML communities.

Proof foundations

OP-1. Algebraic reductions for floating-point GEMM verification.

Per-element sampling is the current verification strategy for BF16/FP32 GEMMs because Freivalds and sumcheck fail under non-associative floating-point rounding. *Question:* is per-element sampling provably optimal for floating-point matmul verification, or does an algebraic shortcut exist that exploits structure (e.g. the accumulation tree topology declared in the precompile)? Recent work on sumcheck with bounded approximation error (Bitan et al. 2025) suggests an approximate Freivalds test could absorb rounding discrepancy with graceful soundness degradation; the obstacles are (i) polynomial commitments that preserve soundness under approximation and (ii) a tight error bound δ distinguishing rounding noise from adversarial deviation. A tight lower bound would close the question; an algebraic reduction would cut the dominant proof cost by one to two orders of magnitude.

OP-2. Constraint minimisation for the MAC precompile.

The BF16/FP32 MAC precompile costs $\sim$ 90 constraints per operation over the Baby Bear field and accounts for roughly 99% of the proof budget at frontier scale. *Question:* can custom gates, lookup arguments, or alternative field representations bring the MAC cost down to 30–50 constraints per operation? Even a factor of two would halve end-to-end proving cost at modern frontier scale.

OP-3. Zero-knowledge proof of backpropagation.

No prior work has demonstrated a zero-knowledge proof of backpropagation for a full LLM in native floating-point arithmetic. Existing demonstrations cover LoRA fine-tuning on quantised models (Garg et al. 2023) or forward inference only. The backward pass reuses the same primitives (MAC, Merkle paths, lookup) but the complete forward+backward circuit has never been exercised at scale in a zkVM. *Question:* produce either (a) a functional proof for a transformer of at least 10^6 parameters with bit-exact gradient match to GPU execution, or (b) an identified fundamental obstacle. This is the highest-risk milestone on the roadmap: a negative result would require redesigning the verification protocol.

Deterministic execution on hardware

OP-4. Deterministic attention backward kernels with $<$ 5% overhead.

FlashAttention-2/3 backward incurs 23–82% overhead in deterministic mode, scaling with sequence length (Appendix C). DASH (Qiang et al. 2026) achieves 1.28× over baseline deterministic mode. *Question:* can attention backward be made deterministic *by construction* (fixed warp-level reduction trees, algebraic reformulation) with under 5% overhead for sequence lengths $\geq$ 8K? This is the dominant source of the determinism tax.

OP-5. Multi-architecture precompile specification.

Each GPU (H100, MI300X, Gaudi3) has distinct accumulation tree topology, rounding mode, and subnormal handling. Each currently requires a hand-engineered precompile. *Question:* can a parameterised precompile family be defined (instantiated automatically from a compact hardware descriptor) and validated against silicon? Sub-problems include a formal specification language for vendor numerics, automated constraint generation, and an empirical validation methodology (how many test vectors suffice to confirm a specification matches hardware?).

Hardware trust anchors

OP-6. Open-hardware network TAP at line rate.

Tier 1 of Section 3.2.4 requires a passive, auditable device that hashes inter-node traffic at 400+ Gbps. Commercial TAPs exist but an open-hardware, formally verified device would establish a new trust primitive for AI governance. *Question:* can streaming SHA-256 hashing at 400+ Gbps be implemented in FPGA fabric with formally verified RTL and tamper-evident physical packaging? Integration with commodity switching hardware is part of the target.

OP-7. Wire-to-tensor mapping for the network anchor.

The network anchor (Section 3.2.4) is useful only if wire observations can be reconciled against the trainer's tensor commitments. This requires a canonical mapping between logical tensors and NCCL's on-wire byte sequence, which is not a stable public interface today: NCCL's byte layout depends on runtime configuration (`NCCL_BUFFSIZE`, `NCCL_NTHREADS`, algorithm selection), ring-allreduce intermediate states on the wire are partial reductions rather than tensor slices, and the anchor must parse InfiniBand or RoCE framing to strip volatile fields (PSN, ICRC) before hashing. *Core question:* which resolution path delivers the best trade between verification assurance and ecosystem adoption? Three candidates are available:

1. **Patched NCCL with a documented canonical format.** A forked NCCL exposes a version-locked on-wire layout and enumerates which logical tensor bytes map to which wire offsets. The cleanest path, compatible with open-source verification, but requires ongoing maintenance as NCCL evolves.

2. **Canonicalisation shim at the NCCL-to-transport boundary.** A signed, auditable library interposed between NCCL and the NIC emits a canonicalised copy of each outgoing payload for the anchor to hash. Trades a small amount of additional memory traffic for independence from NCCL's internals.

3. **Vendor-supported "verifiable NCCL" mode.** A first-class configuration flag with documented wire semantics. The only path that does not require the verification ecosystem to maintain its own fork or shim; depends on vendor cooperation.

All three yield equivalent verification guarantees and differ in deployment effort, maintenance burden, and institutional leverage required.

Three subsidiary questions attach to this problem:

- **Queue-pair mapping auditability.** The NCCL-to-QP mapping is established at init by trainer-host software. How does the verifier audit this mapping without trusting the trainer's host? One candidate is to require the anchor (Tier 1) or the SmartNIC firmware (Tier 2) to publish its own independent view, which must match the trainer's or the run is rejected.
- **Tier 2 attestation protocol.** BlueField DOCA exposes a signing and attestation stack rooted in secure boot and a device identity key. The end-to-end protocol for continuous commitment streaming, key rotation, and revocation is unspecified and would benefit from a standardised reference design.
- **Chunk-boundary alignment.** Tensor elements (2 bytes in BF16) are small relative to Merkle leaves (tens of kilobytes). If the canonical wire format allows elements to straddle leaf boundaries, verifying a single element requires two path openings instead of one. Alignment guarantees must be specified in whichever resolution path is chosen.

OP-8. Intra-node physical witness.

Neither tier of the network anchor observes intra-node traffic over NVLink, so hybrid FSDP+TP training has no physical witness for the tensor-parallel layer. *Question:* can GPU-resident trusted execution (NVIDIA Confidential Compute or a successor mechanism) provide continuous, attested hashing of intra-node collective payloads at the bandwidth required? Current Confidential Compute does not expose the interface nor the throughput; identifying the minimal hardware-attestation primitives that would close this gap is an open design question.

OP-9. Distinguishing SDC from adversarial deviation.

Silent data corruption on frontier hardware produces bit-flips that are indistinguishable from intentional modification at the level of an individual event. Empirical SDC rates at hyperscale are on the order of one event per 10^4–10^5 GPU-hours (Dixit et al. 2021; Hochschild et al. 2021); for a 405B-class training run of $\sim 10^7$ GPU-hours this yields on the order of 10^2–10^3 SDC events (an optimistic anchor, $\sim$ 120 events at 1 *per* 10^5 GPU-hours). A sophisticated adversary could mask structured deviations as statistically natural SDC.

Question: can statistical tests be designed to distinguish SDC patterns (random, memoryless, correlated with known failure modes such as DRAM row faults or thermal events) from structured adversarial injection? Empirical characterisation of SDC distributions on current silicon is a prerequisite.

Protocol extensions

OP-10. Mixture-of-experts verification.

The base protocol assumes the computation graph is determined by the `arch_spec` ahead of time. MoE relaxes this: routing decisions are data-dependent, and which expert processes which token is decided at runtime by the router network. Extending the protocol to MoE introduces several distinct sub-problems:

- **Routing commitment.** A per-step Merkle root $R\left(routing_t\right)$ must join the hash chain, committing to which top-K experts were selected for each token. Without this, routing decisions are not bound to the proof.
- **Top-K selection verification.** Top-K over BF16 scores is a non-smooth, data-dependent operation. Either a dedicated precompile or a decomposition into pairwise comparisons is required; the latter is cheap per element but scales with $NlogK$ per token.
- **Deterministic tie-breaking.** Ties between expert scores (rare with BF16 but possible) must be resolved by a documented rule (e.g. expert ID) declared in `arch_spec`.
- **All-to-all canonicalisation.** Expert parallelism uses all-to-all to dispatch tokens to their selected experts. The wire byte layout depends on the committed routing, making canonicalisation a harder instance of OP-7: the mapping is data-dependent rather than configuration-dependent.
- **Load-balancing auxiliary losses.** Importance and load losses must be declared in `arch_spec` so they are bound by the proof; softmax, log, and division precompiles already cover the operations.

Framing for cost arguments. Per-step proof cost for MoE scales with per-token active compute, not with total parameters. For an MoE with K active experts per token, the per-step cost is approximately K times the cost of a dense model whose dimensions match an individual expert, plus the routing overhead listed above, because each sampled output entry requires verifying all K active experts. A 1T MoE with 32B active per token (e.g. Kimi K2, $K = 8$) therefore costs substantially less per step than a hypothetical 1T dense model, but more than a 32B dense model.

OP-11. Reinforcement-learning post-training.

RL post-training (PPO, GRPO, DPO, RLHF, RLAIF, rule-based RL for reasoning) relaxes a different assumption of the base protocol: rollouts are stochastic and model-dependent, so the inputs on which the model is trained are themselves outputs of the model. Extensions required:

- **Sampling seed commitment.** Per-rollout RNG seeds (and sampling hyperparameters) enter `arch_spec` so that rollouts are reproducible.
- **Rollout verification.** Given the committed policy and the committed seed, sampled completions are a deterministic function. The proof verifies that the stored rollout data matches this deterministic output.
- **Reference-model binding.** PPO/RLHF uses a frozen reference policy for KL regularisation. Its commitment (typically the pre-RL checkpoint, verifiable through the pre-training chain) is referenced by `arch_spec`.
- **Reward commitment.** If the reward is rule-based (deterministic function such as test-case pass/fail for code, or formal verification for math), it is specified in `arch_spec` directly. If the reward is a trained model, its training run has its own h_{commit}, referenced by the policy's `arch_spec`.
- **PPO clipping.** The clipped objective introduces data-dependent gradient masking (clipping depends on per-token probability ratios). Verifying the clipping condition per sampled token is an additional per-sample check, structurally similar to verifying routing.

Recommended first target. Rule-based RL for reasoning (e.g. math and code in the style of DeepSeek-R1) eliminates the reward-model chain and collapses to deterministic reward verification. This is the cleanest first RL target and should be where the extension is demonstrated before RLHF-style settings.

OP-12. Multi-site training verification.

Frontier training increasingly spans multiple datacentres connected over WAN links. The anchor model assumes a verifier-controlled observation point on every inter-node link; realistic deployments split link control across administrative boundaries. *Question:* how does the protocol adapt when some network segments are controlled by different parties? Sub-problems include consistency of Merkle commitments across multiple anchors, handling clock skew and latency heterogeneity, Byzantine-robust aggregation of per-site hashes, and incentive alignment when organisations share a training run.

Protocol tooling

OP-13. Converter from training code to `arch_spec`.

The protocol commits to `arch_spec` as the canonical description of the training computation, but real trainers write PyTorch (or JAX) code, not `arch_spec`. Bridging the two requires a converter pipeline: forward-graph capture (`torch.export`, `torch.fx`), backward-graph capture (`functorch`), mapping from ATen ops to declared `OpSpec` variants, distributed-annotation capture, optimiser-step capture, and serialisation. *Question:* can a converter be built that (a) covers the op set used by frontier training codebases, (b) produces a deterministic `arch_spec` from semantically equivalent PyTorch code, and (c) fails loudly on unsupported ops rather than silently dropping structure? The converter is a completeness concern (does every valid PyTorch training script produce a correct `arch_spec`?), not a soundness concern: the verifier only checks the proof against $Hash(arch\_spec)$, so a converter bug produces a failing proof for an honest trainer, never a passing proof for a dishonest one. ZKML (Chen et al. 2024) is a precedent for inference-only ONNX conversion; the training case adds backward capture and distributed annotations.

Proof cost and overhead estimation

This appendix supports the quantitative claims in Section 3 with detailed derivations. It covers (i) the zkVM cost model and per-operation constraint counts, (ii) why exact GEMM verification is infeasible over floats and why Freivalds' algorithm does not directly apply, (iii) the per-layer and per-step proof budget at Llama 3.1 405B scale under the commitment and fusion assumptions used in Section 3.3, (iv) the statistical properties of interactive sampling, (v) the MoE cost story, (vi) the three proof flows (genesis, in-training, ex-ante) with concrete per-flow costs, and (vii) a training-side overhead summary. Caveats and uncertainties are at the end.

Cost model

Mechanics: hint-and-verify

The zkVM operates as a verification machine rather than an execution machine. For each operation in the training step, the host (trainer's GPU) computes the result and provides it as an unchecked hint. The guest (proof checker running inside the zkVM) verifies the hint against committed Merkle roots and against declared operations via native constraint circuits called precompiles. Host execution is free from the proof's perspective; proving cost is determined entirely by the number of constraints the precompiles generate.

Per-operation constraint counts

Table 4 expands the precompile catalogue of Table 3 with the cost model at the primitive level.

Per-invocation constraint counts for the eight precompiles. The dominant primitive is the BF16/FP32 MAC; Merkle-path verifications are an additive per-sample cost; SHA-256 is invoked only for wire-bound sample reconciliation. Per-path totals assume a tree depth matched to the payload size (zkVM-native Merkle for tensors up to ~10^9 elements at depth 30; SHA-256 anchor paths at depth 20 for 1 GB collective payloads with 1 MB leaves).

#	Primitive cost	Per-invocation	Used for
1	BF16/FP32 MAC	~90 constraints	GEMM dot products (one per sampled entry times inner dim)
2	BF16 table lookup	~15 constraints	Activation functions (GELU, SiLU, their derivatives)
3	zkVM-native hash compression	~75 constraints	Merkle-path verification (~ 1,500 / path at depth 30)
4	FP32 exp	~250 constraints	Softmax, cross-entropy loss
5	FP32 sqrt/rsqrt	~200 constraints	RMSNorm, Adam
6	FP32 div	~200 constraints	Normalisation, Adam
7	FP32 log	~250 constraints	Cross-entropy loss
8	SHA-256 compression	~7,500 constraints	Anchor-path verification (~ 150,000 / path at depth 20)

The per-MAC count of ~90 is an estimate based on the IEEE 754 integer decomposition: BF16 multiply (~30–50 constraints for mantissa product and exponent addition), FP32 accumulation with RNE rounding (~40–60 constraints for exponent alignment, mantissa addition, guard/round/sticky bits). The 50–150 range this spans affects absolute proof-time numbers by up to $\sim 2\times$ but not comparative analysis between approaches; a direct measurement of the actual count is part of the planned empirical validation work.

Proving throughput

The cluster-side proving throughput assumed throughout is $\sim 10^6$ constraints per second per GPU (RISC Zero-class benchmark on A100/H100 for constraint-heavy guests), i.e. $\sim 10^9$ constraints per second aggregate on a 1,024-GPU cluster. Segment size is 2^{20} cycles, segment-level proving is embarrassingly parallel, and recursive composition folds per-segment proofs into a constant-size aggregate (~200 KB) independent of the underlying circuit size.

Exact GEMM verification is O(n^3); Freivalds does not apply

Why the naïve verification is cubic

A GEMM of dimensions $m \times d \times n$ computes $C = A \cdot B$ where each entry C_{ij} is a dot product of length d of BF16 inputs accumulated in FP32 with IEEE 754 RNE rounding. Verifying every entry exactly requires mn dot products, each consisting of d MACs, yielding mnd MAC verifications. At ~90 constraints per MAC this is $\sim 90mnd$ constraints. For a single Q-projection GEMM of Llama 3.1 405B (taking $m = 2,048$, $d = n = 16,384$), the unsampled cost is $\sim 5 \times 10^{13}$ constraints, which exceeds the capacity of current proof systems.

Why Freivalds' algorithm does not directly reduce this for floats

Freivalds' algorithm verifies $C = A \cdot B$ by drawing a random vector $r \in F_p^n$ and checking $A(Br) = Cr$. Over an exact field this reduces verification from $O(n^3)$ to $O(n^2)$ with error probability $\leq 1/|F_p|$. For native BF16/FP32 arithmetic the algorithm fails, because floating-point addition is not associative. The two computation paths $A(Br)$ and Cr involve different accumulation orders and therefore different rounding sequences, producing different bit-exact outputs even when C is the correct GPU output. The check rejects correct computations.

Converting floats to exact integers (for example, by scaling BF16 mantissa-exponent pairs to a common exponent and padding with zeros to a fixed bit-width) and applying Freivalds

over a large field is theoretically available, but introduces three obstacles: (i) the scaled integers exceed the prover's native Baby Bear field ($\sim 2^{31}$), requiring multi-limb arithmetic with $\sim$ 121 $\times$ overhead per operation; (ii) the exact integer product does not equal the GPU's rounded output, so an $O(n^2)$ rounding-verification pass must be added on top; (iii) the interaction between exact-integer Freivalds and the FP32 accumulation rounding chain that determines the GPU's actual output is an open research question. Recent theoretical work on sumcheck protocols that tolerate bounded approximation error (Bitan et al. 2025) points to a path for recovering Freivalds-like asymptotics over floats, discussed as an open problem (OP-1).

In the remainder of this appendix the baseline assumption is interactive per-element sampling; any future reduction is an optimisation on top.

Per-step cost at Llama 3.1 405B

Commitments and fusion assumptions

Per Section 3.2.3, the trainer commits six Merkle roots per layer during training: layer input, Q, K, V, attention output, and layer output. What is *not* committed during training has cost consequences, so we state this explicitly.

- **Flash Attention internals are truly fused.** The attention score matrix ($B \times heads \times s \times s$) and the softmax output never materialise in GPU memory; they are streamed through shared memory inside the Flash Attention kernel. These are uncommittable by kernel construction, not by choice. When the verifier samples attention-output entries, the host recomputes each sampled entry by running the full per-query attention over the sequence dimension. Per-sample cost is $\Theta(s \cdot d_{head})$ MACs, which is materially larger than a single GEMM dot product. To keep the attention contribution bounded, we use a reduced sample count $k_{attn} = 500$ for attention outputs, relying on the committed boundary roots (Q, K, V, attention output) for the primary consistency guarantee.
- **FFN-block internals are not truly fused.** In Llama-style SwiGLU blocks the gate and up projections materialise in HBM as separate tensors before the elementwise activation and multiply. We exploit this by committing $gate\_out$ and up_out Merkle roots *at challenge time* (rather than during training) when the trainer re-runs the step. The training-time hashing overhead therefore remains at six roots per layer, while the per-challenge proof samples each FFN GEMM independently at the cheaper per-GEMM rate. This is a deliberate design trade: training-time hashing stays cheap, challenge-time response grows by ~1 second per layer of hashing, and the per-sample cost of FFN verification drops by a factor of $\sim d_{ff}$.

Per-layer constraint budget

We compute the per-layer forward + backward cost at Llama 3.1 405B scale (d $_{model} = 16,384$, d $_{ff} = 53,248$, $n_{heads} = 128$, $n_{kv\_heads} = 8$ with GQA, $d_{head} = 128$, 126 layers, sequence length $s = 2,048$ per microbatch, $k = 4,605$ samples per committed GEMM, $k_{attn} = 500$ for attention outputs).

Each forward GEMM of shape $(m \times d) \times (d \times n)$ induces two backward GEMMs (input gradient and weight gradient) whose inner dimensions are n and m respectively. Per sampled output entry, the proof cost is inner-dimension × 90 constraints.

Per-layer forward + backward constraint budget at Llama 3.1 405B scale under the commitment and fusion assumptions of Subsubappendix B.3.1. Q/K/V/O and FFN GEMMs are sampled per-GEMM (trainer-side Merkle roots at layer boundaries and lazy intermediates bind the hints). Flash Attention uses reduced-sample recomputation; elementwise SiLU and multiply contribute per-FFN-sample. SHA-256 anchor-path verification (precompile 8) is zero here because no wire-bound tensors are verified per layer on an intra-node path; it appears only on sampled wire-bound tensors (Subsubappendix B.4.2).

Per-layer component	Inner-dim sum (fwd + 2 bwd)	Sampled cost (k = 4,605)	Layer contribution
Q projection	34,816	$90 \cdot k \cdot 34,816$	1.44×10^{10}
K projection (GQA, smaller n)	19,456	$90 \cdot k \cdot 19,456$	8.06×10^{9}
V projection (GQA)	19,456	$90 \cdot k \cdot 19,456$	8.06×10^{9}
O projection	34,816	$90 \cdot k \cdot 34,816$	1.44×10^{10}
FFN gate projection	71,680	$90 \cdot k \cdot 71,680$	2.97×10^{10}
FFN up projection	71,680	$90 \cdot k \cdot 71,680$	2.97×10^{10}
FFN down projection	71,680	$90 \cdot k \cdot 71,680$	2.97×10^{10}
Flash Attention (fused, $k_{attn} = 500$)	$\Theta(s \cdot d_{head})$ per sample	end-to-end recompute	$\sim 5 \times 10^{10}$
Elementwise SiLU + multiply	d_{ff} ops × (15 + 90)	lookup + BF16 mult	$\sim 2.6 \times 10^{10}$

Per-layer component (per FFN sample)	Inner-dim sum (fwd + 2 bwd)	Sampled cost (k = 4,605)	Layer contribution
Merkle paths (6 trainer-side per sample)	depth 30, zkVM-native hash	1,500 constraints each	$\sim 4 \times 10^7$
Per-layer total (fwd + bwd)			$\sim 2.1 \times 10^{11}$

Per-step total

Summing 126 layers contributes $126 \cdot 2.1 \times 10^{11} \approx 2.6 \times 10^{13}$ constraints. The LM-head contribution is bounded above by a single GEMM with inner-dim sum $d_{model} + vocab + m \approx 146,688$, giving $\sim 6 \times 10^{10}$ constraints (less than 0.3% of the total).

Per-step proof budget: $\sim 2.6 \times 10^{13}$ **constraints.** The MAC precompile accounts for over 99% of this budget; Merkle-path verification (precompile 3) contributes well under 0.1%. SHA-256 anchor-path verification (precompile 8) is invoked only for wire-bound samples, adding a bounded amount per challenged wire-bound tensor (see Subsubappendix B.4.2).

Proving time

At a cluster-wide throughput of $\sim 10^9$ constraints per second (1,024 GPUs at $\sim 10^6$ constraints per second each), a full-step proof of Llama 3.1 405B completes in $\sim$ 7 hours. Linear scaling of compute yields ~100 minutes on 4,096 GPUs, or ~25 minutes on 16,384 GPUs. Per-layer proofs are independent and compose via recursive STARK folding, so the critical path is the most expensive individual layer proof rather than the serial sum, which becomes relevant when proving is parallelised across dedicated proof clusters.

Comparison with the O(n^3) baseline

Summed over all per-layer GEMMs, the unsampled baseline at Llama 3.1 405B yields $\sim 5.5 \times 10^{18}$ constraints per step (~one-hundred-thousand-fold the sampled cost). The ~ 82,000× reduction from interactive sampling at $k = 4,605$ is what makes proof generation feasible at all at this scale.

Interactive sampling: statistical properties

Sample count for target security

For verifier-chosen samples drawn after Merkle roots have been committed (no grinding), the probability of missing a deviation affecting fraction f of entries is $P_{miss} = (1 - f)^k$. Solving for target miss probability τ:

$$k \geq \frac{ln(1/\tau)}{-ln(1-f)} \approx \frac{ln(1/\tau)}{f} \quad (for\ small\ f).$$

Sample counts for interactive sampling. The recommended default $k = 4,605$ achieves 10^{-20} miss probability per committed tensor at a 1% deviation fraction.

Target miss probability	$f = 10\%$	$f = 1\%$	$f = 0.1\%$
10^{-20}	$k = 437$	$k = 4,605$	$k = 46,052$
2^{-128} (match ZK soundness)	$k = 842$	$k = 8,840$	$k = 88,530$

The default $f = 1\%$ is a tuning choice rather than a fundamental threshold. Sampling detects deviations affecting a fraction f of entries with probability $1 - (1 - f)^k$, so any target miss probability is reachable by scaling k: smaller f simply requires more samples ($k \propto - ln(miss)/f$). The default $k = 4,605$ corresponds to 10^{-20} per-tensor miss probability at $f = 1\%$; the same guarantee at $f = 0.1\%$ requires $k = 46,052$. For wire-bound tensors, the network anchor (Section 3.2.4) provides an independent commitment that the proof binds against at every sampled entry; a mismatch between the trainer's tensor commitment and the anchor's wire commitment surfaces as a failed anchor-consistency check inside the zkVM proof. Formal sensitivity analysis bounding the smallest f a rational adversary can exploit is an open problem (OP-1).

Network-anchor bindings

Wire-bound tensors are committed twice: the trainer holds a zkVM-native Merkle root of the tensor value, and the network anchor holds an SHA-256 Merkle root of the wire payload that transported that tensor. When the verifier samples a wire-bound entry, the proof opens both paths and verifies that the element value is consistent with both commitments. Per sampled wire-bound element, the anchor consistency check costs $depth \cdot SHA - 256_{compress} + \lceil c/64 \rceil \cdot SHA - 256_{compress}$ for a chunk of c bytes (~2–8 million constraints at $c = 16$–64 KB leaves per Appendix D). Anchor checks are invoked

only for sampled wire-bound tensors, so their contribution to per-step budget is bounded: typically under 5% of the budget even when every inter-node tensor is sampled.

Defense-in-depth

The general-purpose security argument against an adaptive adversary composes three layers:

1. **Network anchor.** Catches any deviation that changes wire-bound content; deterministic (not statistical) detection on any sampled wire-bound tensor.
2. **On-the-fly Merkle commitments.** Each layer's output is committed during training. Any error in any intermediate that propagates to the layer output changes the committed root; the proof catches this at the layer boundary.
3. **Interactive sampling.** Adds 10^{-20} statistical assurance on intermediate GEMMs for deviations affecting $\geq$ 1% of entries.

Each layer is necessary: (1) alone cannot see intra-node computation; (2) alone does not rule out an adversary who produces internally consistent fake commitments; (3) alone has a detection floor at the deviation fraction. Together they provide complementary coverage.

Compute-threshold attestation: case-by-case analysis.

The three-layer argument above is the general case against an adversary cheating on arbitrary aspects of the training procedure. For the specific target of compute-threshold attestation (ex-ante binding of total FLOPs, Section 3.4.3), the adversary's options are narrower. The rational adversary under this target wants to *under*-declare compute: to run more than was claimed in order to stay below a regulatory threshold. Three cheating modes are available, each with a cheap and direct detection.

- **Model larger than declared (wider).** The adversary inflates d_{model}, d_{ff}, n_{heads}, or another dimension while publishing *arch_spec* for smaller values. The commitment $R(W_0)$ is a Merkle tree whose leaves and depth are determined by the actual weight shapes. An opening at any challenged weight position in the genesis proof must produce a path matching the *arch_spec*-declared shape. A wider actual weight matrix has leaves at different offsets; the path opening fails. Detection requires a single genesis weight-opening.
- **Model larger than declared (deeper).** The adversary runs $L_{actual} > L_{declared}$ layers. Three consequences: (i) the committed per-layer chain terminates at $L_{declared}$, so the input to the committed LM head is the declared-depth activation, not the actually-deeper activation from which the loss was computed; (ii) in the backward

pass, the gradient committed at declared layer $L_{declared}$ does not equal $\partial L/\partial layer_{L_{declared}}$ under $arch\_spec$ because the real backprop chain is longer; (iii) the network anchor observes inter-node communication volume proportional to L_{actual}. Any of these three discrepancies is caught by a single in-training or genesis challenge.

- **Data larger than declared.** The adversary trains on α-fold more data than committed in $R(dataset)$. The $arch\_spec$-declared shuffling seed determines which dataset indices appear in batch t. A fraction $1 - 1/\alpha$ of steps use indices absent from $R(dataset)$. Opening the layer-0 input commitment $R\left(mlp_0\_in\right)$ at committed positions fails for those steps. Detection probability with Q uniform random input-openings:

$$P(detect) = 1 - \alpha^{-Q}.$$

  Hiding $\alpha = 10$ data needs $Q = 2$ for 99% detection; $\alpha = 2$ needs $Q = 7$; $\alpha = 1.1$ needs $Q = 49$. Large-scale data smuggling is trivially caught; the detection budget scales with the inverse of the attempted inflation factor.

- **More training steps than declared.** The network anchor (Section 3.2.4) observes inter-node collectives continuously for the duration of the run. For training that requires inter-node communication at every step (FSDP, TP across nodes, PP), the anchor's observed collective count matches the actual step count. The trainer's published chain has one entry per declared step. Discrepancy is a single integer comparison and requires no cryptographic proof.

The four cases together imply a compute-threshold attestation needs far fewer challenges than general-purpose $arch\_spec$ verification. A conservative budget (genesis + a handful of input-data challenges at $Q \sim 10$ to rule out $\geq 2\times$ data smuggling + anchor-count check) costs on the order of $15–25K, rather than the $7–14M required for $Q \in \{500, 1,000\}$ random challenges in the generic worst case.

The overall cost-analysis tables retain the larger sampling budget as the default for two reasons. Formal sensitivity analysis (OP-1) could yet reveal corner cases that rescale the argument for the three detections above. Attestations beyond compute (for instance data-content filters targeting *semantic* smuggling within the committed dataset, or procedural attestations on fine-grained regime compliance) lack the structural detection surface of compute and operate closer to the 500-challenge regime against adaptive adversaries.

MoE verification cost

Per OP-10, per-step proof cost for MoE scales with per-token active compute, not with total parameters. For an MoE with K active experts per token of per-expert dimensions ($d_{model} \times d_{expert}$), the per-sample FFN cost is approximately K times the cost of a dense model with the same per-expert dimensions, plus a small routing overhead. Kimi K2 ($K = 8, d_{model} = 7,168, d_{expert} = 2,048$) costs substantially less per step than a hypothetical 1T dense model, but more than a 32B dense model because each sampled output requires verifying all K active experts.

Router verification adds a small per-token GEMM of inner dimension d_{model} plus a top-K selection step (either a dedicated precompile or a decomposition into pairwise comparisons, scaling as $NlogK$ for N total experts). Load-balancing auxiliary losses (importance, load) introduce per-token softmax and division, covered by existing precompiles 4 and 6 at negligible cost relative to the expert GEMMs. All-to-all canonicalisation for expert parallelism is a separate open problem (OP-7 and OP-10).

Concrete cost figures for MoE verification depend on sampling-scheme design decisions that are themselves open (OP-10); detailed per-architecture numbers are left to future work once OP-10 is resolved.

Proof-flow costs

The protocol produces three proof types (Section 3.4). Their costs differ.

Genesis proof

Executed once at initialisation on a verifier-chosen batch of b samples drawn from the committed dataset. The proof covers one full training step (forward + backward + optimiser update). Its cost is the per-step budget of Subsubappendix B.3.2 for the chosen b, plus b dataset-membership Merkle-path openings (one per sampled index). At $b \leq$ typical microbatch size this is dominated by the per-step cost and completes in the ~7-hour envelope on 1,024 GPUs. A reduced-b genesis (e.g. $b = 128$) lowers proof cost proportionally while still exercising all code paths, which may be appropriate when genesis latency is operationally important.

In-training step proof

Executed many times across a training run, each invocation on a sampled step t. Four challenge granularities are meaningful in practice; the verifier can mix them according to the threat model.

- **Single-commit challenge** (verify one committed root against an adjacent committed root via a single declared operation, for instance a GEMM verified by the MAC precompile, an activation verified by the BF16 lookup, or a LayerNorm verified by the FP32 nonlinear precompiles): up to $\sim 2 \times 10^{10}$ constraints, ~20 s on 1, 024 GPUs, ~$12 at $2/GPU-hour. GEMM-backed challenges set this headline cost; challenges that exercise cheaper precompiles (table lookups, FP32 nonlinear ops) are essentially free by comparison.
- **Layer-partial challenge** (verify one layer's forward pass against its input and output commitments, or one layer's backward pass against its gradient-boundary commitments; the two are structurally independent sub-proofs): $\sim 1 \times 10^{11}$ constraints each, ~1. 8 min on 1, 024 GPUs, ~$60.
- **Depth-N partial chain** (N consecutive forward layers, or N consecutive backward layers, chained via their committed boundary roots): N times the layer-partial cost. Useful when the verifier wants one proof that cross-checks multiple intermediate roots and their boundary consistency, for example an entire pipeline-parallel stage.
- **Full-step challenge** (all 126 layers of a step, forward plus backward; used for genesis and periodic comprehensive audits): $\sim 2.6 \times 10^{13}$ constraints, ~7 h on 1, 024 GPUs, ~$14K. Proving is embarrassingly parallel across segments, so on 16, 384 GPUs the same work completes in ~25 min for the same total GPU-hours.

A realistic challenge mix over a 100k-step run at Llama 3.1 405B scale might be 10, 000 single-commit challenges for broad coverage (~$120K), 500 layer-partial challenges for layer-boundary integrity (~$30K), 100 depth-6 partial chains for cross-checking PP-stage boundaries (~$36K), and 10 full-step challenges for comprehensive audits (~$140K). Total: ~$326K, or ~0. 33% of a $100M training budget. This "many cheap, few expensive" mix is driven by threat-model coverage rather than by any single security bound; verifiers targeting only compute-threshold attestation can run a much smaller budget (see Subsubappendix B.4.3).

Per-challenge response time exclusive of proof generation is dominated by the host-side re-run: loading the stored weights from the rolling window (seconds on NVMe) and re-executing the forward + backward pass for the challenged step on a partial replica (minutes to hours depending on challenge depth).

Ex-ante attestation running invariant

Ex-ante attestations enforce invariants (e.g. $F_{cum}(t) \leq max\_total\_flops$) at every step proof. The invariant check is a single addition and a single comparison per step, adding a

constant number of constraints (under 100) to each step proof. For any realistic invariant list the contribution is negligible relative to the per-step MAC budget.

Deeper proofs and proof composition

A depth-1 step proof trusts its input Merkle roots: if the trainer committed fake input roots, the proof still passes (the verifier catches the inconsistency downstream when the fake root does not match the preceding step's output). A depth-N proof starts from N layers earlier and verifies each intermediate commitment along the chain in-proof, at roughly N times the single-layer cost.

Proving time at Llama 3.1 405B scale on 1,024 GPUs. Within a proof, layer-level sub-proofs run in parallel on independent GPU subsets and are folded via recursive composition into a single ~200 KB aggregate. Linear scaling in cluster size: a full-step proof on 16,384 GPUs completes in ~25 minutes for the same total GPU-hours.

Challenge	Proving time (1,024 GPUs)	What it cross-checks
Single-commit (one declared op)	~20 s	One committed root against its adjacent committed root
Layer-partial (forward *or* backward of one layer)	~1. 8 min	That layer's input/output (or gradient) boundary
Depth-6 partial chain (forward *or* backward)	~11 min	One PP stage: intra-stage roots + one PP boundary
Full step (all 126 layers, forward + backward)	~7 h	Every intermediate root, every anchor observation

A realistic challenge mix combines shallow challenges for broad coverage (many, cheap) with occasional deep challenges that cross-check boundary commitments (few, expensive). The exact mix is an operational decision set by the verifier's risk budget.

Training-side hashing overhead

What must be hashed per step

Six tensors per layer are committed during training (Section 3.2.3): layer input, Q, K, V, attention output, layer output. At Llama 3.1 405B with $s = 2,048$, $m = 2,048$, activations are $m \cdot d_{model} \cdot 2\,B = 67$ MB each for the layer input/output and $m \cdot n_{heads} \cdot d_{head} \cdot 2 = 67$

MB for Q/O, $m \cdot n_{kv\_heads} \cdot d_{head} \cdot 2 = 4$ MB for K/V. Per layer, aggregate tensor bytes to hash are $\sim$ 275 MB. Over 126 layers per step the total is $\sim$ 35 GB.

The concurrent GPU stream delivers $\sim$ 80–120 GB/s effective throughput (CUDA hashing kernel, H100), putting per-step concurrent hashing time at $\sim$ 300–450 ms. At typical step times of ~1 s this is within the overlap window with $\sim$ 1–3% memory-bandwidth contention on the primary compute stream.

Wire-bound collective payloads are hashed by the network anchor (Tier 1 physical TAP or Tier 2 attested SmartNIC). Anchor hashing happens outside the GPU budget and does not contribute to trainer-side overhead.

DPU/SmartNIC hashing capabilities (Tier 2 context)

For Tier 2 deployments the anchor lives in the SmartNIC. Relevant hashing throughput figures:

Observed or projected SHA-256 throughput for network-anchor-side hashing hardware. BF3 software-only throughput is marginal for cluster-scale inter-node traffic; hardware-accelerated silicon (Marvell OCTEON, BF4, or a dedicated Tier-1 FPGA) is required to sustain line rate at frontier cluster fabrics.

Device	SHA-256 throughput	Mechanism	Notes
NVIDIA BF3	~10 GB/s	Software on 16× ARM A78 cores	Hardware SHA-256 removed vs BF2
NVIDIA BF4 (expected)	~40 GB/s	64× Neoverse V2 cores	Announced; throughput TBD
Marvell OCTEON 10	~12–25 GB/s	Hardware NITROX V	Commercially available
Dedicated FPGA (Tier 1)	~100+ GB/s	Pipelined SHA-256 IP	Per OP-6, design target

Total training-side overhead

Combining determinism and hashing costs, the total training-side overhead at Llama 3.1 405B is Table 9.

Training-side overhead at Llama 3.1 405B. Determinism is the dominant cost and is attention-backward-driven, scaling with sequence length. For a $100M training budget the end-to-end trainer-side verification overhead is ~$2–10M.

Component	Overhead	Confidence
Determinism (attention backward-dominated)	1. 6–8. 2%	Measured on Llama 7B H100; seq-len-dependent
Concurrent Merkle hashing (6 tensors/layer, GPU stream)	0. 5–1. 5%	Estimate from 2-pass kernel + optimised target
Weight rolling-window storage (81 TB)	< 0. 1%	Small vs cluster's existing checkpointing storage
Anchor-side hashing	~0% (trainer)	Moves to anchor hardware
Total trainer-side overhead	~2–10%	Sequence-length-dependent

Cost summary at Llama 3.1 405B

Verification overhead for a Llama 3.1 405B-scale pre-training run on a $100M training budget. The dominant cost is the determinism tax; zkVM proving, hashing, storage, and challenges combined add ~1–3%.

Component	Cost at $100M training budget	% of training
Determinism tax	$1.6–8.2M	1.6–8.2%
Concurrent Merkle hashing	$0.5–1.5M	0.5–1.5%
Genesis proof (one-time, reduced-batch)	~$1K	<0.01%
In-training challenges (layered mix: 10k single-commit + 500 layer-partial + 100 depth-6 + 10 full-step)	~$326K	~0.33%
Weight storage (81 TB rolling)	~$50K/year	<0.1%

Component	Cost at $100M training budget	% of training
Ex-ante attestation running cost	negligible	<0.001%
Total	**~$2–10M**	**~2–10%**

Caveats and open questions

1. **Proving throughput (~10^6 constraints/s/GPU)** is approximate and may be optimistic for constraint-heavy FP precompiles. A direct measurement is part of the planned empirical validation work. Relative comparisons between approaches are robust to this assumption.

2. **Per-MAC constraint count (~90)** has a range of 50–150 depending on implementation. Absolute proof times scale linearly with this; the relative structure of the breakdown does not.

3. **Non-linear lookup tables** (BF16 GELU, SiLU) must match the GPU kernel output at every input. Generation and validation of these tables is routine engineering work.

4. **Sampling security at** $f < 1\%$ is a defense-in-depth argument, not a hard theorem. Formal sensitivity analysis (OP-1) would close this.

5. **GPU-concurrent Merkle hashing overhead (~1–3%)** needs empirical validation on H100 with realistic training workloads.

6. **Fusion assumptions** (Flash Attention truly fused, FFN GEMMs separable with lazy commitment) are kernel-dependent. A training stack that fuses FFN end-to-end (e.g. a memory-bound variant) materially increases per-sample FFN verification cost, and the sample count for fused blocks must be reduced correspondingly.

7. **MoE cost numbers** depend on sampling-scheme design decisions open under OP-10. The order-of-magnitude framing ($K \times$ per-expert-dense) holds across reasonable choices; concrete per-architecture figures require OP-10 resolution.

8. **INT8 training with INT32 accumulation** (supported by H100/B200 Tensor Cores) would eliminate float non-associativity and make Freivalds directly applicable. No frontier model currently uses full INT8 pretraining, but the industry trend toward FP8 and FP4 is adjacent; verification cost at low-precision regimes is a worthwhile future study.

Determinism enforcement: detailed benchmarks and analysis

This appendix provides detailed measurements and analysis supporting the determinism discussion in Section 3.2.1.

Kernel-level determinism configuration

We enforce determinism through optimised kernel selection rather than PyTorch's blanket `torch.use_deterministic_algorithms(True)` flag. The configuration uses: FlashAttention-2 with its deterministic backward mode (`FLASH_ATTENTION_DETERMINISTIC=1`), which serializes CTA reductions at a measured cost; cuBLAS with pinned algorithm selection (`CUBLAS_WORKSPACE_CONFIG`); and NCCL with NVLS-based reduction (which provides fixed reduction order at near-zero overhead for FSDP allgather and reduce-scatter).

We measured the end-to-end overhead of this configuration on 8× H100 HBM3 with NVSwitch, training Llama 7B with FSDP:

Measured full-determinism overhead (Llama 7B, FSDP, 8×H100 HBM3). The overhead is attention-dominated: cuBLAS GEMMs contribute ~0.8% and NCCL communication (NVLS) contributes ~0%.

Seq. length	Non-det (ms/step)	Det (ms/step)	Overhead
2048	531.8	540.3	+1.6%
4096	999.1	1038.0	+3.9%
8192	1129.9	1222.0	+8.2%

Parallelism dimensions beyond FSDP

The measurements above apply to FSDP, which uses reduce-scatter and allgather (both deterministic under NVLS at near-full bandwidth). Frontier training runs typically combine multiple parallelism dimensions: tensor parallelism (TP) within a node, pipeline parallelism (PP) and FSDP across nodes, and expert parallelism (EP) for mixture-of-experts models.

Tensor parallelism.

TP is the most challenging dimension. It uses allreduce after each column-parallel and row-parallel GEMM. Our NCCL benchmarks show that intra-node NVSwitch allreduce is *not* deterministic under NVLS above 128 MB BF16. The only deterministic configurations are Tree+Simple (64% bandwidth loss) and Ring+1CTA (89% bandwidth loss, unusable). This

is a significant penalty: TP allreduce occurs at every layer, so even Tree+Simple could add 6–10% to total step time depending on the ratio of TP communication to compute.

Interestingly, inter-node allreduce (over TCP/RoCE) is deterministic under default NCCL configuration at all tested sizes (32–1024 MB BF16, 2-node setup). The determinism problem is specific to the intra-node NVSwitch code path.

Pipeline and expert parallelism.

PP is straightforward: it sends activations between pipeline stages in a fixed schedule, and determinism reduces to the same per-layer guarantees described above. EP introduces all-to-all communication for expert routing. Our benchmarks show NVLS all-to-all is deterministic at all tested sizes, and the routing decisions themselves are deterministic given a fixed gating function.

Towards a custom TP allreduce

The overhead of Tree+Simple for TP can likely be eliminated by a custom deterministic allreduce kernel optimised for the TP use case. TP operates under narrow, exploitable constraints: fixed topology (8 GPUs on NVSwitch), fixed message sizes (determined by model dimensions, constant across steps), and a fixed reduction order is all that is needed for bit-exactness. A custom kernel using NVLink direct writes with a hardcoded reduction tree for the specific topology and message size could match NCCL's default bandwidth while guaranteeing determinism by construction. Projects such as MSCCL (Microsoft's custom collective compiler) demonstrate that hand-written collectives can match or exceed NCCL for fixed topologies.

Importantly, PyTorch's process group architecture supports per-dimension configuration: TP, FSDP, and PP each use separate communicators. The custom kernel would replace NCCL only for TP allreduce calls, while FSDP and PP continue to use NVLS under default configuration. This avoids any performance regression on the already-solved FSDP and PP dimensions.

Our current measured overhead (1.6–8.2%) applies to FSDP-only training. For FSDP+TP setups, the E2E impact of the TP allreduce bandwidth loss is likely small in practice: TP communication and layer computation overlap, and for typical TP=8 intra-node configurations the compute time dominates. For example, a 100 MB allreduce at Tree+Simple bandwidth (169 GB/s) takes ~0.6 ms, which is well within the compute time of a single Transformer layer at frontier scale. The overhead would become significant only if TP communication becomes the bottleneck, for instance at very large TP degrees across nodes or with unusually small per-GPU compute. This remains to be validated end-to-end.

Network anchoring: hash choice, reconciliation, and open problems

This appendix supports Section 3.2.4 by (i) justifying the dual-hash design, (ii) describing the tree-structure and chunk-size tradeoffs, (iii) discussing composite precompile optimisations, (iv) developing the wire-to-tensor mapping problem in detail, and (v) listing the remaining open engineering problems.

Hash function choice: two regimes, two hashes

The architecture uses two distinct hash functions for its Merkle commitments: a zkVM-native hash (precompile 3, Poseidon today) for the trainer's on-the-fly tensor commitments, and a silicon-native hash (precompile 8, SHA-256) for the network anchor. This is not redundant: each hash is chosen for the bottleneck of the device that hashes at line rate.

The two hash families have opposite cost profiles. Poseidon operates in the prover's native prime field: its round function is built from $x^5 \ mod \ p$ S-boxes and MDS matrix multiplications over F_p. The same arithmetic that makes Poseidon inexpensive inside the proof ($\sim$ 75 constraints per compression, $\sim$ 1,500 constraints per path of depth 30) makes it expensive outside the proof: prime-field multiplication is not a native silicon primitive, and commercial FPGAs and DPUs achieve at best sub-GB/s throughput with significant area cost. SHA-256 inverts this. Its round function uses 32-bit bitwise operations (AND, XOR, rotation) and modular addition, which map directly to Boolean gates and integer ALUs: Intel SHA-NI sustains $\sim$ 1.9 cycles/byte, and FPGA IP cores pipeline at 5–20 Gbps per core and stack to hundreds of GB/s per device. The same simplicity is costly in-circuit, because bitwise operations over 32-bit words must be encoded via lookup tables or bit decomposition ($\sim$ 7,500 constraints per compression, $\sim$ 150,000 constraints per path of depth 20).

The asymmetry in cluster-scale throughput is therefore two to three orders of magnitude. Aggregate inter-node bandwidth on a modern training cluster reaches several hundred GB/s; Poseidon-family hashes cannot be computed at those rates on commodity silicon. The network anchor must use a hash with mature silicon support, which today means SHA-256. Conversely, the trainer's Merkle paths are opened thousands of times per sampled GEMM inside the zkVM; using SHA-256 for those paths would increase the proof budget for path verification by roughly 100 $\times$, whereas the GPU overhead saved on the trainer side is under 1%. Using one hash per regime is the only design that is efficient on both sides.

This split between an in-circuit-cheap hash and a silicon-cheap hash is a standard pattern whenever a system bridges two cost models with incompatible primitives; Ethereum rollups, for example, use Keccak at the L1 interface and zk-friendly hashes inside the rollup for the same reason.

Tree versus flat, and chunk size

Committing to wire payloads as a Merkle tree rather than a single flat hash trades a small amount of TAP-side work for a large reduction in challenge-time zkVM cost. Flat SHA-256 commits a payload with one digest; verifying consistency at challenge time then requires rehashing the entire payload inside the zkVM. For a 1 GB collective with the SHA-256 precompile this is roughly 16×10^6 blocks at $\sim 7,500$ constraints each, or $\sim 10^{11}$ constraints per challenge. Tree-structured commitments reduce this to $O(logn)$ path verification plus a local chunk rehash.

The tree structure costs almost nothing on the TAP side. For n leaves of size ℓ, the total hashing volume is $\sim 2n$ leaf-sized units, so the overhead over flat hashing is a small constant factor determined by ℓ/ℓ_{leaf}: with MB-sized leaves over a GB-sized payload it is at most $1.001 \times$.

The leaf size ℓ is a tuning parameter. Smaller leaves shrink the chunk rehash but deepen the tree, inflating path verification. Larger leaves invert the trade.

Constraint cost per wire-bound sample opening for a 1 GB collective as a function of Merkle-leaf size. Path and rehash costs assume the SHA-256 precompile at $\sim 7,500$ constraints per compression block.

Leaf size	Tree depth (1 GB)	Path cost	Chunk rehash cost	Total per opening
32 B (one element)	25	$\sim 190,000$	$\sim 7,500$	$\sim 200,000$
16 KB	16	$\sim 120,000$	$\sim 1,900,000$	$\sim 2,000,000$
64 KB	14	$\sim 105,000$	$\sim 7,500,000$	$\sim 7,600,000$
1 MB	10	$\sim 75,000$	120,000,000	120,000,000

The zkVM cost is minimised at small leaves, which places the sweet spot at tens of kilobytes when balanced against TAP-side chunk-boundary bookkeeping and packet-boundary alignment (discussed below). Values in the 16–64 KB range give opening costs in the 2–8 million constraint range per sample, bounded and much smaller than the MAC-chain budget of a GEMM.

Composite Merkle-path precompile

The precompile catalogue in Table 3 reports costs assuming per-compression precompile invocations composed in software (a loop over tree levels calling the SHA-256 or Poseidon primitive at each step). A composite precompile that absorbs path-traversal logic (sibling loading, direction bits, chained compressions) into a single dedicated circuit eliminates the

per-level RISC-V overhead and enables tighter constraint layouts across levels. Cairo's Merkle builtins and ongoing work in SP1 and Jolt follow this pattern. Estimated savings are 20–30% for SHA-256 paths (where the compressor is heavy and glue overhead relatively matters) and 15–25% for Poseidon paths. A further optimisation is a multi-path precompile that verifies several paths in one invocation, amortising transition cost across openings; this gives another 10–20% when the same challenge opens multiple paths (as is typical for GEMM sampling, which opens 3–5 paths per sampled entry).

These are implementation-level optimisations; they refine the proof-cost figures without changing the architectural claims.

Several open engineering problems attach to the network anchor: the hardware TAP itself, wire-to-tensor mapping (with queue-pair auditability, Tier 2 attestation protocol, and chunk-boundary alignment as subsidiary questions), and intra-node coverage. These are catalogued in Appendix A (OP-6 through OP-8) rather than duplicated here.

Toy example: end-to-end verification walkthrough

This appendix walks through the complete verification protocol on a minimal example: a two-layer MLP with ReLU activation, split across two nodes by pipeline parallelism. The example exercises precompiles 1 (MAC), 2 (BF16 lookup), 3 (Merkle path with the zkVM-native hash), and 8 (SHA-256 for network-anchor reconciliation). It does not exercise precompiles 4–7 (FP32 nonlinear operations for softmax, normalisation, Adam, or loss); those are triggered by additional operations in the full Transformer block and follow the same pattern.

Setup

The model is a two-layer MLP, $f(x) = W_2 \cdot ReLU\left(W_1 \cdot x\right)$, trained with pipeline parallelism over two nodes:

- **Node 1** holds the first linear layer $W_1 \in R^{d_h \times d_{in}}$.
- **Node 2** holds the second linear layer $W_2 \in R^{d_{out} \times d_h}$.
- Loss is $L = \frac{1}{2} \| y - target \|^2$ for simplicity of the backward pass.
- Optimiser is plain SGD ($W \leftarrow W - \eta \cdot grad$); no Adam state to track.
- Weights are BF16 with FP32 accumulators; rounding is RNE.

Each training step involves two inter-node messages: a forward activation passed from Node 1 to Node 2, and a backward gradient passed from Node 2 back to Node 1. The network anchor observes both.

Notation.

We write $R(\cdot)$ for a Merkle root over a tensor or byte sequence. Trainer-side Merkle roots use a zkVM-native hash (Poseidon in the current instantiation, verified by precompile 3). Network-anchor Merkle roots use a silicon-native hash (SHA-256, verified by precompile 8). Both support $logn$ membership proofs against the committed root; they differ only in per-path cost inside the proof. We write $Hash(\cdot)$ for the outer compound commitment.

Phase 1: pre-commit

Before training begins, the trainer publishes the initial commitment:

$$h_{commit} = Hash\Big(Hash(arch\_spec) \;\|\; R\big(W_0\big) \;\|\; R(dataset)\Big)$$

where $arch\_spec$ records the layer dimensions, activation type (ReLU), optimiser (SGD), precision (BF16/FP32, RNE), parallelism strategy (PP, world size 2), and shuffling seed. $R\big(W_0\big)$ is the Merkle root over the initial weight shards across both nodes; $R(dataset)$ is the Merkle root over the training data.

The auditor records h_{commit} and installs a network anchor on the inter-node link between Node 1 and Node 2.

Phase 2: one training step

We annotate a single step with every commitment published and every message observed. Node 1 processes batch shard x; arrows mark inter-node traffic.

The two nodes execute sequentially along the pipeline direction. Arrows mark inter-node messages observed by the network anchor.

Node 1 (holds W_1).

```
mlp_1_in  <- x                                        # batch shard
mlp_1_out <- W_1 @ mlp_1_in
act_1_out <- ReLU(mlp_1_out)
publish R(mlp_1_in), R(mlp_1_out), R(act_1_out)
send act_1_out  ---- forward msg_1 ---->  Node 2      (anchor: R(msg_1))
```

Node 2 (holds W_2).

```
recv act_1_out  from Node 1
mlp_2_in  <- act_1_out
mlp_2_out <- W_2 @ mlp_2_in
act_2_out <- ReLU(mlp_2_out)
publish R(mlp_2_in), R(mlp_2_out), R(act_2_out)
L <- 0.5 * || act_2_out - target ||^2
grad_act_2 <- act_2_out - target
grad_mlp_2 <- grad_act_2 * (mlp_2_out > 0)
grad_W_2   <- grad_mlp_2 @ mlp_2_in^T
grad_act_1 <- W_2^T @ grad_mlp_2
publish R(grad_2)
send grad_act_1  ---- backward msg_2 ---->  Node 1    (anchor: R(msg_2))
```

Node 1 (backward + optimiser).

```
recv grad_act_1  from Node 2
grad_mlp_1 <- grad_act_1 * (mlp_1_out > 0)
grad_W_1   <- grad_mlp_1 @ mlp_1_in^T
publish R(grad_1)
W_1 <- W_1 - eta * grad_W_1           # at Node 1
W_2 <- W_2 - eta * grad_W_2           # at Node 2
publish R(W_{t+1})                    # combined root
```

After the step, the trainer's public hash chain contains $R(archi)$, $R(W_t)$, $R(data)$ (from pre-commit), plus per-step $R(mlp_i\_in)$, $R(mlp_i\_out)$, $R(act_i\_out)$, $R(grad_i)$ for $i \in \{1, 2\}$, and $R(W_{t+1})$. The network anchor's record contains $R(msg_1)$ (forward activation payload) and $R(msg_2)$ (backward gradient payload).

Phase 3: challenge

The auditor selects a step t, a node i, and k random sample indices $(j_1, ..., j_k)$ for the output entries to challenge. Because the indices are chosen *after* the Merkle roots have been published to the hash chain, the trainer cannot predict which entries will be sampled.

The challenge demands a proof of the following statement, phrased at Node 1 for concreteness (Figure in Section 3):

$$R(archi),\ R(W_t),\ R(data)\ \vdash\ R(mlp_1\_in)\ \rightarrow\ R(mlp_1\_out)\ \rightarrow\ R(msg_1)$$

That is: executing the committed architecture with the committed weights on the committed data produces intermediate tensors whose Merkle roots match the committed

$R(mlp_1\_in)$ and $R(mlp_1\_out)$, and the forward message serialisation is consistent with the network anchor's observation $R(msg_1)$.

Phase 4: verification inside the zkVM

The proof checker is a public, auditable program. It reads the *arch_spec* as a private input, verifies it matches h_{commit}, then, for each sampled output entry, runs the following checks. We illustrate one entry at Node 1: the output of the first GEMM at row r, column j.

```
# ---- bind the private arch_spec to the public commitment ----
arch_spec = read_private_input()
assert Hash(arch_spec) == h_commit.arch_spec_hash

# ---- for each sampled (r, j) at Node 1 ----
for (r, j) in challenge_indices:

    # (a) input row: open against R(mlp_1_in)
    x_row    = read_hint()                # private: mlp_1_in[r, :]
    path_in  = read_hint()
    MerklePath(x_row, path_in, R(mlp_1_in))         # precompile 3

    # (b) weight column: open against R(W_t)
    W1_col   = read_hint()                # private: W_1[:, j]
    path_W   = read_hint()
    MerklePath(W1_col, path_W, R(W_t))              # precompile 3

    # (c) GEMM entry: recompute via MAC chain and bind to output root
    mlp_out_rj = read_hint()              # private: mlp_1_out[r, j]
    MAC_chain(x_row, W1_col, mlp_out_rj)            # precompile 1
    path_out = read_hint()
    MerklePath(mlp_out_rj, path_out, R(mlp_1_out))  # precompile 3

    # (d) ReLU output: verify via BF16 lookup, bind to activation root
    act_rj   = read_hint()                # private: act_1_out[r, j]
    Lookup(mlp_out_rj, ReLU_TABLE, act_rj)          # precompile 2
    path_act = read_hint()
    MerklePath(act_rj, path_act, R(act_1_out))      # precompile 3

    # (e) anchor consistency: the same act_rj must appear at the
    #     computed byte offset in the forward message
    offset   = compute_msg_offset(arch_spec, r, j)
    chunk    = read_hint()                # bytes of the msg chunk
    path_msg = read_hint()
    MerklePath_SHA256(SHA256(chunk), path_msg, R(msg_1))  # precompile 8
```

```
    assert chunk[offset : offset + bf16_size] == act_rj

emit "PASS"
```

Step (e) is the link between the trainer's tensor commitment and the network anchor's wire commitment: the same BF16 value of $act_1\_out[r, j]$ must be present both under the trainer-committed $R(act_1\_out)$ (via Poseidon path) and under the anchor-committed $R(msg_1)$ (via SHA-256 path at the offset dictated by the serialisation defined in $arch\_spec$).

The guest emits as public outputs: h_{commit}, the committed roots $R(mlp_1\_in)$, $R(mlp_1\_out)$, $R(act_1\_out)$, the anchor root $R(msg_1)$, and a "PASS" bit. The auditor STARK-verifies the proof (milliseconds on a laptop), checks that the public outputs match the published hash chain and the anchor record, and accepts.

What each party sees

	Trainer (host)	Proof checker (guest)	Auditor
$arch\_spec$	full	private input, bound to h_{commit}	only $Hash(arch\_spec)$
Weights W_1, W_2	full	hints, bound to $R(W_t)$	never
Activations, gradients	full	hints, bound via MAC + lookup	never
Merkle roots $R(\cdot)$	computes	public verification anchors	reads from hash chain
Anchor root $R(msg_k)$	not involved	public verification anchor	reads from anchor
STARK proof	not involved	produces it	verifies it

Proof cost

For illustration, take $d_{in} = d_h = d_{out} = 1024$ and $k = 100$ sampled output entries per committed tensor at each of the two nodes (a small value chosen for readability; the main-text analysis uses $k \approx 4,605$ for 10^{-20} soundness error at 1% deviation). The approximate constraint budget for one step:

Verification work	Precompile	Constraints (order of magnitude)
First GEMM ($d_h \cdot 90$ per sample $\times\ k$)	MAC (1)	$\sim 9 \times 10^6$
ReLU lookup (k per committed element)	BF16 lookup (2)	$\sim 1.5 \times 10^3$
Second GEMM (same magnitude)	MAC (1)	$\sim 9 \times 10^6$
Merkle paths, trainer side (~6 per sample $\times\ k$)	Poseidon path (3)	$\sim 9 \times 10^5$
Anchor consistency (1 path + 1 chunk rehash per sample)	SHA-256 (8)	$\sim 6 \times 10^6$

Total: roughly 2.5×10^7 constraints per step, which is well inside the capacity of current zkVMs (STARK proof size $\sim$ 200 KB after recursive composition). At production scale the MAC chain dominates because d and k both grow by two to three orders of magnitude; see Appendix B for the full breakdown.

Training specification (`arch_spec`) schema

This appendix gives the concrete structure of the `arch_spec` committed during pre-training (Section 3.2.2) and verified by the proof checker (Section 3.3). The `arch_spec` is a private input to the proof; the verifier sees only $Hash(arch\_spec)$ as part of h_{commit}. The schema below is the reference definition against which implementations are written. It is presented in Rust-style record syntax for concreteness; any structured serialisation with an equivalent set of fields is acceptable.

Top-level `arch_spec`

```
struct ArchSpec {
    // Model architecture
    layers:       Vec<LayerSpec>,
    embedding:    EmbeddingSpec,
    output_head:  OutputHeadSpec,

    // Training configuration
    optimizer:    OptimizerSpec,        // Adam, AdamW, SGD, ...
    data_loading: DataLoadingSpec,      // batch derivation from committed
```

```
dataset
    precision:    PrecisionSpec,          // BF16/FP32, rounding, hardware

    // Distributed training
    communication: CommSpec,              // allreduce, allgather, send/recv
    parallelism:   ParallelismSpec,       // TP, PP, DP, FSDP configuration

    // Verification
    merkle_points: Vec<MerklePoint>,      // tensors committed on-the-fly
    rng_spec:      RNGSpec,               // data-shuffling and dropout seeds
}
```

Every operation that takes place during a training step must be expressible as a composition of the elements declared by `arch_spec`. Any operation outside this vocabulary is rejected by the proof checker.

Layer specification

```
struct LayerSpec {
    forward_ops:   Vec<OpSpec>,
    backward_ops:  Vec<OpSpec>,           // gradient computations mirroring
forward
    optimizer_ops: Vec<OpSpec>,           // parameter update for this layer
}

enum OpSpec {
    // Linear algebra
    GEMM { m: usize, k: usize, n: usize },

    // Attention (fused; internal score and softmax tensors never
materialise;
    // verified end-to-end at Q/K/V/attention-output boundaries, see Sec.
3.4.5)
    FusedAttention {
        n_heads:     usize,
        n_kv_heads:  usize,               // GQA: < n_heads ; MHA: == n_heads
        d_head:      usize,
        seq_len:     usize,
        causal:      bool,
        positional:  PositionalEncoding, // RoPE, Learned, ALiBi, None
    },

    // Non-linear activations (BF16, verified via precompile 2)
    Activation { kind: ActivationType },  // GELU, SiLU, SwiGLU, ReLU

    // Normalisation (RMSNorm used by Llama family)
```

```
    LayerNorm { dim: usize },
    RMSNorm   { dim: usize },

    // Structural
    Residual,

    // Mixture-of-experts block (out of scope for base protocol; see OP-10)
    MoEBlock {
        router:   RouterSpec,
        experts:  Vec<Vec<OpSpec>>,
        combine:  CombineSpec,
    },
}
```

A forward `GEMM` with inputs A and B produces two backward `GEMM` operations: one for the gradient with respect to the input ($grad_A = grad_C \cdot B^\top$) and one for the gradient with respect to the weight ($grad_B = A^\top \cdot grad_C$). The `backward_ops` list enumerates these explicitly.

Fused operations carry both a forward and a backward variant; both are sampled end-to-end at their input and output commitment boundaries (Subsubappendix B.3.1).

Precision specification

```
struct PrecisionSpec {
    compute:          Precision,        // BF16 for parameters/activations
    accumulate:       Precision,        // FP32 for GEMM accumulators
    rounding:         RoundingMode,     // RoundToNearestEven
    accum_order:      AccumOrder,       // LinearLeftToRight, ...
    hardware:         HardwareTarget,   // H100, MI300X, ...
    cublas_algo:      Option<String>,   // pinned cuBLAS algorithm ID
    flash_attn_version: String,         // e.g.
"FlashAttention-3-deterministic"
}
```

The `accum_order` field is load-bearing for determinism: given non-associative floating-point addition, two distinct orders produce two distinct bit-exact results. The committed order is what the proof checker verifies.

Communication specification

`CommSpec` governs operations that perform floating-point reductions (`AllReduce`, `ReduceScatter`). The reduction order is explicit so the proof can verify the arithmetic. Data-movement operations that do not perform reductions (pipeline-parallel send/recv,

FSDP allgather) are verified through Merkle-root matching at endpoints plus network-anchor consistency, without reduction-order specification.

```
struct CommSpec {
    // Global NCCL configuration pinned for determinism (Sec. 3.3.1)
    nccl_algo:     NCCLAlgo,              // Ring, Tree, NVLS
    nccl_proto:    NCCLProto,             // Simple
    nccl_channels: usize,

    ops: Vec<CommOp>,
}

enum CommOp {
    AllReduce {
        tensor_shape:    Vec<usize>,
        dtype:           Dtype,
        reduction_order: Vec<usize>,     // explicit FP addition order
        group:           ProcessGroup,
    },
    AllGather {
        shard_shape:  Vec<usize>,
        dtype:        Dtype,
        gather_order: Vec<usize>,        // rank-order (0, 1, 2, ...)
        group:        ProcessGroup,
    },
    ReduceScatter {
        tensor_shape:    Vec<usize>,
        dtype:           Dtype,
        reduction_order: Vec<usize>,
        group:           ProcessGroup,
    },
    Send { tensor_shape: Vec<usize>, src: usize, dst: usize },
    Recv { tensor_shape: Vec<usize>, src: usize, dst: usize },
}
```

Parallelism specification

```
struct ParallelismSpec {
    tp_degree:        usize,             // tensor parallelism (typically 8)
    pp_stages:        usize,             // pipeline parallelism stages
    dp_degree:        usize,             // data parallelism replicas
    fsdp_enabled:     bool,
    fsdp_shard_degree: usize,            // ranks sharing each parameter
shard

    // Expert parallelism (MoE models only; out of scope for base protocol)
```

```
    ep: Option<ExpertParallelismSpec>,

    // Process-group topology
    tp_groups: Vec<Vec<usize>>,
    pp_groups: Vec<Vec<usize>>,
    dp_groups: Vec<Vec<usize>>,
}
```

Commitment structure

The outer public commitment, re-stated here for reference, combines the `arch_spec` hash with the Merkle roots of initial weights and dataset:

$$h_{commit} = Hash\Big(Hash(arch\_spec) \;\|\; R\big(W_0\big) \;\|\; R(dataset)\Big).$$

Any ex-ante attestation claims (Section 3.4.3) extend this structure by appending $Hash(ex\_ante\_claims)$.

Converter pipeline

Generating the `arch_spec` from the trainer's actual PyTorch (or JAX) training code requires a converter pipeline; this is treated as its own open problem (see OP-13 in Appendix A).

References

Baker, Mauricio, Gabriel Kulp, Oliver Marks, Miles Brundage, and Lennart Heim. 2025. "Verifying International Agreements on AI: Six Layers of Verification for Rules on Large-Scale AI Development and Deployment." https://doi.org/10.48550/arXiv.2507.15916.

Ben-Sasson, Eli, Iddo Bentov, Yinon Horesh, and Michael Riabzev. 2018. "Scalable, Transparent, and Post-Quantum Secure Computational Integrity." *Cryptology ePrint Archive, Report 2018/046*.

Ben-Sasson, Eli, Alessandro Chiesa, and Nicholas Spooner. 2016. "Interactive Oracle Proofs." In *Theory of Cryptography Conference (TCC 2016-b)*.

Bitan, Dor, Zachary DeStefano, Shafi Goldwasser, Yuval Ishai, Yael Tauman Kalai, and Justin Thaler. 2025. "Sum-Check Protocol for Approximate Computations." *Cryptology ePrint Archive, Report 2025/2152*.

Chen, Bing-Jyue, Suppakit Waiwitlikhit, Ion Stoica, and Daniel Kang. 2024. "ZKML: An Optimizing System for ML Inference in Zero-Knowledge Proofs." In *Proceedings of the Nineteenth European Conference on Computer Systems (EuroSys)*.

Cottier, Ben, Robi Rahman, Loredana Fattorini, Nestor Maslej, Tamay Besiroglu, and David Owen. 2024. "The Rising Costs of Training Frontier AI Models." *arXiv Preprint arXiv:2405.21015*.

Dixit, Harish Dattatraya, Sneha Pendharkar, Matt Beadon, Chris Mason, Tejasvi Chakravarthy, Bharath Muthiah, and Sriram Sankar. 2021. "Silent Data Corruptions at Scale." *arXiv Preprint arXiv:2102.11245*.

Dubey, Abhimanyu, Abhinav Jauhri, Abhinav Pandey, et al. 2024. "The Llama 3 Herd of Models." *arXiv Preprint arXiv:2407.21783*.

Garg, Sanjam, Aarushi Guo, Omer Reingold, and Ron Roth. 2023. "Experimenting with Zero-Knowledge Proofs of Training." *arXiv Preprint arXiv:2310.02421*.

Goldreich, Oded. 2001. *Foundations of Cryptography: Basic Tools*. Cambridge University Press.

Hochschild, Peter H., Paul Turner, Jeffrey C. Mogul, Rama Govindaraju, Parthasarathy Ranganathan, David E. Culler, and Amin Vahdat. 2021. "Cores That Don't Count." In *Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS)*.

Kang, Daniel, Tatsunori Hashimoto, Ion Stoica, and Yi Sun. 2022. "Scaling up Trustless DNN Inference with Zero-Knowledge Proofs." In *arXiv Preprint arXiv:2210.08674*.

Liao, Guofu, Taotao Wang, Shengli Zhang, Jiqun Zhang, Long Shi, and Dacheng Tao. 2026. "VeriLoRA: Fine-Tuning Large Language Models with Verifiable Security via Zero-Knowledge Proofs." In *Network and Distributed System Security Symposium (NDSS)*.

Lindell, Yehuda. 2015. "An Efficient Transform from Sigma Protocols to NIZK with a CRS and Non-Programmable Random Oracle." In *Advances in Cryptology – EUROCRYPT 2015*.

Patel, Dylan, and SemiAnalysis. 2024. "Llama 3 Training Economics and Cost Breakdown."

Petrie, James. 2025. "Guaranteeable Memory: An HBM-Based Chiplet for Verifiable AI Workloads." In *Workshop on Technical AI Governance (TAIG) at ICML 2025*. Vancouver, Canada. https://openreview.net/pdf?id=uc79kOv0MV.

Petrie, James, and Onni Aarne. 2025. "Flexible Hardware-Enabled Guarantees: Part II: Technical Options for Flexible Hardware-Enabled Guarantees." ARIA. https://arxiv.org/abs/2506.03409.

Qiang, Xinwei, Hongmin Chen, Shixuan Sun, Jingwen Leng, Xin Liu, and Minyi Guo. 2026. "DASH: Deterministic Attention Scheduling for High-Throughput Reproducible LLM Training." In *International Conference on Learning Representations (ICLR)*.

Scher, Aaron, David Abecassis, Peter Barnett, and Brian Abeyta. 2025. "An International Agreement to Prevent the Premature Creation of Artificial Superintelligence." https://doi.org/10.48550/arXiv.2511.10783.

South, Tobin, Alexander Camuto, Shrey Jain, Shayla Nguyen, Robert Mahari, Christian Paquin, Jason Morton, and Alex Pentland. 2024. "Verifiable Evaluations of Machine Learning Models Using zkSNARKs." *arXiv Preprint arXiv:2402.02675*.

Sun, Haochen, Jason Li, and Hongyang Zhang. 2024. "zkLLM: Zero Knowledge Proofs for Large Language Models." In *Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security (CCS)*.

Sun, Haochen, and Hongyang Zhang. 2024. "zkDL: Efficient Zero-Knowledge Proofs of Deep Learning Training." *arXiv Preprint arXiv:2307.16273*.