One Native Stack for Private, Continuously-Learning AI: Inference at the Bandwidth Wall, On-Device QLoRA on the Same Quantized Weights, and a Unified Train-and-Serve Budget

Zach Kelling

Hanzo AI

June 2026

Abstract

Every production AI stack pays a *format-break tax*: a model is trained in one framework (PyTorch), exported to a second representation (ONNX/GGUF), re-quantized, and served on a *third* engine (vLLM/TensorRT/`llama.cpp`)—two engines, two quantization formats, two device backends, and a conversion step at every boundary, so the kernel that is served is never the kernel that was trained on. This paper is the north-star synthesis of five hanzo design papers and two patent memos, and it argues—and partly demonstrates on real silicon—that the tax can be removed by collapsing train and serve into *one* native-Rust engine that operates on the same tensor type, the same quantization compute core, and the same device backend, with zero conversion. The proven foundation is a native-ROCm inference engine that reaches effective decode parity (0.98×) and ≈ 0.86× prefill against `llama.cpp` on a consumer AMD APU (Ryzen AI Max+ 395 "Strix Halo", Radeon 8060S, `gfx1151`, native Windows, ROCm 7.1), built on a unified 1-bit-to-full quantization core whose decode matvec runs at 97% of the memory-bandwidth wall [1]. On top of that core, a Rust autograd (`hanzo-ml`, forked from HuggingFace `candle`) with real AdamW/SGD already trains two working algorithms (GRPO RL, an in-engine mixture-of-experts gate), and the QLoRA primitive is present in code: a dequantized quant weight is a detached constant, so a frozen quantized base plus a trainable bf16 adapter is the exact, correct QLoRA shape [2]. We formalize the central efficiency claim as a theorem: because a decode forward pass and a QLoRA training step over the *same* frozen quantized base both stream the base weights through the *same* bandwidth-bound core exactly once, serve-cost and train-cost share one bandwidth budget, and per-token training cost is a small constant multiple of per-token serving cost on the shared hardware. We give the honest scoreboard (decode 0.98×, prefill 0.86×, MoE decode 2× over hanzo's own prior, six native quant types plus eleven bit-exact sub-4-bit additions), a four-phase roadmap that respects the real dependency structure, and a frank separation of *ported* prior art from *candidate* hanzo IP. We claim no speed win over NVIDIA and no zero-leakage privacy guarantee; the defensible claims are a unified train-and-serve runtime, a structural delta-only egress contract, and a parity-enabled cost lever on cheaper silicon.

1 Introduction

The dominant pattern in applied machine learning is a pipeline of specialists. A model is trained with one toolchain, checkpointed in that toolchain's native format, exported and re-quantized into an inference format, and finally served by a separate, independently optimized engine. Each boundary is a *format break*: a lossy or lossless re-encoding of the weights, a re-quantization with its own rounding, and a hand-off to code that was never co-designed with the trainer. The consequence is operational and epistemic: the served kernel is not the trained kernel, so a numeric difference between training and serving is a routine, expected, and hard-to-attribute fact of life.

This paper takes the opposite bet. The thesis—developed across five companion papers [1, 2, 3, 4] and two patent memos—is that a single native-Rust engine can *train and serve a model on the same tensor type, the same quantization core, and the same device backend, with zero conversion*, and that doing so is not merely tidy but *efficient*: it places serving and on-device personalization under one bandwidth-bound budget on commodity hardware. We are deliberate about status throughout, distinguishing in the prose what runs in the tree today from what is buildable on the existing substrate with bounded engineering and what remains product thesis or open research. Every performance number is measured on *one* machine—an AMD Ryzen AI Max+ 395 "Strix Halo" (Radeon 8060S iGPU, `gfx1151`, native Windows, ROCm 7.1, single GPU)—against `llama.cpp` (HIP backend) on the *same* GPU, and is cited to the inference campaign [1]. We invent no numbers.

The one coherent story. The five layers of the single system are:

- **Inference** [1]: a native-ROCm engine that reaches decode parity (0.98×) and ≈ 0.86× prefill on a consumer APU, built on a unified 1-bit-to-full quantization compute core. This is the proven foundation; everything else rides on it.
- **Native training** [2]: a Rust autograd (`backprop.rs`; `hanzo-ml` forked from HuggingFace `candle`), real AdamW/SGD, and two working, tested trainers (GRPO [13] RL, an in-engine MoE gate). The load-bearing finding is that the QLoRA [12] primitive already exists: a dequantized quant weight is a detached constant, so a frozen quantized base plus trainable bf16 adapter variables is the exact, correct QLoRA shape.
- **Continuous learning and privacy** [3]: a serve → capture → self-generate → curate → train → optionally-federate loop, where personalization is a stack of cheap LoRA [11] deltas over the shared frozen quantized base, raw data never leaves the device, and only bf16 deltas optionally egress.
- **Platform and desktop** [4]: the same engine as both an economical per-quant cloud serving tier on cheaper AMD silicon and an on-device "learns-on-your-machine" desktop product, private by default.

The unifying technical fact is the **quant core**: the kernel that serves a model at decode parity is the *same* kernel a QLoRA adapter trains over, is the *same* kernel that hosts any community GGUF at any quantization via a one-line table edit, is the *same* base that per-user deltas stack onto. One core, one tensor type, one backend, train and serve. The honest one-sentence posture: we are at parity with `llama.cpp` on decode and below it on prefill—we do not beat everyone—but we own a unified train-and-serve runtime that the orchestrate-black-box-engines competitors (Kubeflow/KServe/Ray) [15, 16, 17] structurally cannot.

Contributions.

- A precise statement and proof of the *central efficiency thesis* (Theorem 1): a decode step and a QLoRA training step over the same frozen quantized base both stream the base once through the same bandwidth-bound core, so serve-cost and train-cost share one bandwidth budget, with per-step training cost a small constant multiple of per-step serving cost (Corollary 1).
- A formal statement of the *no-format-break* invariant (Proposition 1) and its dependence on the unified-core bit-exactness lemma (Lemma 1): the served weights and the QLoRA base weights are the identical bytes read by the identical kernel.
- The honest, source-grounded scoreboard (Table 1) and a four-phase roadmap (Section 7, Table 2) ordered by value/effort and respecting the shared Var-backed model-construction dependency.
- A frank *ported-versus-novel* separation for a patent reader (Section 6), with the kernel IP rated patent-weak and the federation/privacy-loop combinations rated the stronger filings.

2 The Unified Quant Core

The architectural commitment—a standing user mandate during the inference campaign—was to support quantization from 1-bit to full *with no per-quant kernels*: one quant-agnostic GPU core

per stage, one decode function per format, reused across decode, prefill, and mixture-of-experts, such that adding a quantization format is one decode function plus one traits row plus one table entry, with *zero new kernels* [1]. Concretely the core is realized as a `WTYPE`-templated decode kernel `qmatvec_core<WTYPE>`, a matched prefill int8 GEMM core `qmmq_core<WTYPE>`, per-type device decode functions `qdec<WTYPE>::partial` / `decode_w_half<WTYPE>`, a `qdw_traits<WTYPE>` row, and `DEFINE_QMATVECU` / `quant_format!` code-generation macros. Six types are wired native today across all structural classes (symmetric, asymmetric super-block, signed 6-bit, LUT, ternary):

```
DEFINE_QMATVECU(q8_0,  DW_Q8_0)   // sym, 34B/32
DEFINE_QMATVECU(q4_0,  DW_Q4_0)   // sym, 18B/32
DEFINE_QMATVECU(q4k,   DW_Q4_K)   // ASYM, 144B/256
DEFINE_QMATVECU(q6k,   DW_Q6_K)   // signed 6b, 210B/256
DEFINE_QMATVECU(iq4xs, DW_IQ4_XS) // LUT, 136B/256
DEFINE_QMATVECU(tq2_0, DW_TQ2_0)  // ternary, 66B/256
```

Eleven additional sub-4-bit IQ/ternary/FP4 formats were added as load-and-dequant, bit-exact against `ggml` [5] (`nbad`= 0 / `max_err`= 0.0 on deterministic bytes), through a single `quant_format!` macro. The same decode and prefill cores drive single-token decode, batched prefill, and MoE expert evaluation; the MoE path dispatches each routed expert through `qmatvec_core<WTYPE>` with a resident-bank cache. The full inference results and their derivation are in the companion inference paper [1]; we restate only the load-bearing facts: the decode matvec sustains 245.8 GB/s (97% of the measured read-bandwidth wall), and the unified core is bit-exact against the CPU reference for every wired type. These two facts are the hinges of the efficiency argument below.

Why this is the through-line, not just an inference detail. For serving, the unified core turns "host any community GGUF at any quant" from a combinatorial maintenance burden into a table row. For training, the *same* core is what a QLoRA adapter differentiates over: the base weight is dequantized on the fly by `qdec<WTYPE>` into a detached value, and the adapter's bf16 variables are the only trainable parameters. For personalization, the same frozen base hosts a stack of per-user/per-task deltas. One core, four uses; the moat is the unified runtime, not any single format's math, which is `ggml`'s and is used as prior art (Section 6).

3 Current-State Scoreboard

All numbers are measured on the one 8060S/`gfx1151` box (ROCm 7.1 [18]) against `llama.cpp` HIP on the same GPU running an 8B dense transformer [7] (parity targets, quiet GPU: pp512 = 1176.3, tg64 = 25.05). Every hanzo configuration in Table 1 is bit-exact (`nbad`= 0) against the CPU reference and produces coherent output [1].

Table 1: Current-state scoreboard (real numbers only), hanzo versus `llama.cpp` on the same GPU. Decode start "1.7 → 8.5" conflates the original one-block-per-row matvec (1.7) and the warp-per-row milestone (8.5); MoE "2×" is over hanzo's own prior (6.34 → 13.6), not over llama.

Axis	**Start**	**hanzo**	**llama**	**Ratio**
Decode (8B dense, Q8_0)	1.7–8.5	24.5	25.05	0.98×
Prefill (pp512)	39	1015	1176.3	0.86×
MoE decode (35B-A3B)	0.93	13.6	—	2× prior
Decode matvec (GB/s)	103.5	245.8	231 wall	97%
Prefill GEMM (tok/s, in-situ)	—	≈1414	1176 total	maxed

Training substrate. Autograd + AdamW/SGD + VarMap safetensors checkpointing is in the tree and working today; GRPO RL is algorithm-complete and tested (ported from `zen/gym`); an in-engine MoE gate trainer trains end-to-end today; the QLoRA autograd primitive (detached `dequantize()` + float-dtype optimizer filter) is present in code [2]. The trainable LoRA *layer*, the generic supervised fine-tuning (SFT) trainer (today a stub—`optimizer.rs:82` is a literal no-op), and DPO/KTO/ORPO are *not built*.

Quant coverage. Six quant types are wired native on the unified ROCm core (decode + MoE, all `nbad`= 0); eleven additional sub-4-bit IQ/ternary/FP4 types are added as load-and-dequant, bit-exact versus `ggml`. Native *prefill* is unified for Q8_0/Q4_0 only; Q4_K/Q6_K/IQ/TQ still dequant-to-f16 at prefill—an honest caveat and a named follow-on [1].

Serving. A real PagedAttention [9] path (paged KV, continuous batching with preemption, prefix caching) is in the tree today, ported from established designs; a concurrent multi-model map with on-demand unload/reload ships today; and load-time in-situ quantization across the full quant zoo ships today [4].

The one-line summary: *decode parity (*0.98×*) / prefill* 0.86× */ MoE* 2× */ unified quant core proven / native-training foundation exists*—with the SFT trainer, LoRA injection, the span-level PII redactor, and the Rust federation control plane all named as not-yet-built.

4 The Central Efficiency Thesis

We now make the efficiency claim precise. The intuition is simple: on a memory-bound decoder, the cost of a token is the cost of streaming the weights once [8]. A QLoRA training step over a frozen quantized base reads those *same* weights through the *same* core, plus a small amount of extra traffic for the low-rank adapter. Therefore serving and on-device training are governed by one bandwidth budget on the shared hardware. We formalize this in three steps: a bit-exactness lemma (the served weights *are* the QLoRA base weights), the central theorem (shared bandwidth budget), and a budget corollary (train cost is a small constant times serve cost).

Definition 1 (Shared weight-streaming model)**.** *Fix a transformer whose frozen base weights have total resident footprint W bytes in a quantization format `WTYPE`, read once per token through the unified core. Let B be the achieved (sustained) global-memory read bandwidth in bytes/second on the production path. A* QLoRA configuration *adds, per adapted projection of shape $d_{out} \times d_{in}$, two low-rank factors $A \in \mathbb{R}^{r \times d_{in}}$ and $B_A \in \mathbb{R}^{d_{out} \times r}$ in bf16 with rank $r \ll \min(d_{out}, d_{in})$; write P_{base} for the number of base parameters and $P_{lora} = \sum_\ell r(d^\ell_{out} + d^\ell_{in})$ for the adapter parameters, with adapter footprint $W_{lora} = 2\,P_{lora}$ bytes (bf16).*

Lemma 1 (QLoRA over the unified core is exact and base-weight-identical)**.** *Let $\hat{W} = \mathrm{DEQ}_{cpu}(W)$ be the reference CPU dequantization of the base to `f32`. For any activation x, a QLoRA forward through the unified core computes, per output row i,*

$$y_i \;=\; \underbrace{\textstyle\sum_j \hat{W}_{ij}\, x_j}_{\text{frozen base, via } \texttt{qmatvec\_core}} \;+\; \tfrac{\alpha}{r}\,\big(B_A\,(A\,x)\big)_i,$$

where the first term is bit-for-bit *the served decode output (the identical bytes W read by the identical kernel) and the second term depends only on the trainable adapter. Consequently the gradient of any loss $\mathcal{L}(y)$ with respect to the base is identically zero, $\partial\mathcal{L}/\partial W \equiv 0$, and the only trainable parameters are A, B_A.*

Proof. By the Bit-Exactness Lemma of the inference paper [1], `qmatvec_core<WTYPE>` evaluates $\sum_j \hat{W}_{ij} x_j$ accumulated in `f32` in the same per-block order as the CPU reference, for every supported `WTYPE`; this is the term served at decode, so the served bytes and the QLoRA base bytes are the same W read by the same kernel. In the autograd, the dequantization $\hat{W} = \mathrm{DEQ}(W)$ is materialized as a *detached* constant (`dequantize()` produces a value with no parent in the tape, and the optimizer filters to float-dtype variables only [2]); hence the computational graph contains W only as a constant leaf, and reverse-mode differentiation propagates no gradient into it: $\partial \mathcal{L}/\partial W \equiv 0$. The adapter path $\frac{\alpha}{r} B_A(Ax)$ is composed of ordinary bf16 variable operations, so A and B_A receive gradients in the usual way. This is exactly the QLoRA shape of Dettmers et al. [12]: a frozen quantized base plus a trainable low-rank adapter, with the base contributing a constant to the forward and nothing to the backward. □

Theorem 1 (Unified bandwidth budget for serve and train)**.** *Under Definition 1, let t_{serve} be the wall-clock time of one decode step and t_{train} the wall-clock time of one QLoRA step (forward + backward + optimizer) on the* same *hardware. Both are bounded below by the* same *base-streaming term, and their bandwidth-bound parts differ only by adapter traffic:*

$$t_{serve} \;\geq\; \frac{W}{B}, \qquad t_{train} \;\geq\; \frac{W + c\,W_{lora}}{B},$$

for a small constant c (the number of times the bf16 adapter is streamed per step, a low single-digit). Hence serve-cost and train-cost share one bandwidth budget dominated by the common term W/B, and

$$\frac{t_{train}}{t_{serve}} \;\geq\; 1 + c\,\frac{W_{lora}}{W}, \quad \textit{with} \quad \frac{W_{lora}}{W} \ll 1.$$

Proof. For serving, the decode step is memory-bound: by the Roofline Decode-Ceiling theorem [1, 8], the step must transfer at least the resident base footprint W at rate at most B, so $t_{\mathrm{serve}} \geq W/B$. For training, consider the three sub-steps. *Forward*: by Lemma 1 the base contribution is computed by the same `qmatvec_core`, streaming the same W bytes once; the adapter forward streams W_{lora} bytes and performs $O(r\,d)$ work per projection, which is negligible against the base read because $r \ll d$. *Backward*: by Lemma 1 no gradient flows into W, so the backward pass does *not* need to recompute or re-read the base for a weight gradient; it reads only adapter factors and the (small, $O(r)$-dimensional) intermediate Ax to form $\partial \mathcal{L}/\partial A$ and $\partial \mathcal{L}/\partial B_A$, i.e. $O(W_{\mathrm{lora}})$ traffic. (The base is read again only insofar as the activation gradient must traverse $\hat{W}^\top$; in the on-device personalization regime the adapted layers are few and shallow-bounded, and where the full base is re-read this contributes at most one additional W/B term, leaving the conclusion unchanged up to the constant on W/B.) *Optimizer*: AdamW updates only the P_{lora} adapter parameters and their two moment buffers, $O(W_{\mathrm{lora}})$ traffic and work. Summing, the mandatory traffic is W (base, once) plus a low single-digit multiple c of W_{lora} (adapter forward/backward/optimizer), at rate at most B, giving $t_{\mathrm{train}} \geq (W + c\,W_{\mathrm{lora}})/B$. Both bounds carry the identical W/B term—the shared budget—and dividing yields the stated ratio. Since $W_{\mathrm{lora}} = 2P_{\mathrm{lora}}$ with $P_{\mathrm{lora}} = \sum_\ell r(d^\ell_{\mathrm{out}} + d^\ell_{\mathrm{in}})$ and $r \ll \min(d_{\mathrm{out}}, d_{\mathrm{in}})$, we have $W_{\mathrm{lora}}/W \ll 1$, so $t_{\mathrm{train}}/t_{\mathrm{serve}} \to 1^+$. □

Corollary 1 (Train cost is a small constant times serve cost)**.** *For a typical QLoRA configuration the adapter is a low-single-digit percent of base parameters; e.g. rank-16 adapters on the attention and FFN projections of the 8B model give $P_{lora}/P_{base} \approx 1\%$. Then by Theorem 1 the per-step training time is within a small constant factor of the per-step decode time on the same APU: $t_{train}/t_{serve} \lesssim 1 + c \cdot 0.01$ for the bandwidth-bound part. On the measured 8060S, where decode runs at the ≈ 25 tok/s bandwidth wall [1], an on-device QLoRA step's weight-streaming cost is therefore*

essentially one decode-step of base traffic plus an adapter-sized residual—placing personalized fine-tuning and serving in the same *cost regime on commodity hardware, which is the economic premise of the on-device product [3, 4].*

Proposition 1 (No format break)**.** *In the unified stack there exists no representation-conversion step between training and serving: the bytes W trained over (as the frozen QLoRA base) are the identical bytes served, read by the identical kernel* `qmatvec_core<WTYPE>`*, and an adapter trained on-device is served by adding its low-rank term to that same kernel's output. Equivalently, the train→serve map is the identity on the base and an addition on the adapter, with no re-quantization and no engine hand-off.*

Proof. Immediate from Lemma 1: the base term of the trained forward and the served forward are the same function of the same bytes computed by the same kernel, so the base is unchanged across train and serve (the identity map). The adapter is a separate bf16 tensor whose contribution at serve time is the same $\frac{\alpha}{r} B_A(Ax)$ added to the kernel output, optionally folded by the standard LoRA merge [11]; in neither case is the base re-encoded or the engine swapped. Contrast the conventional pipeline, where train and serve use different engines and the base is re-quantized at the export boundary, making the train→serve map a lossy re-encoding rather than the identity. □

Reading the result. Theorem 1 is the formal core of "one engine, one budget." It does not claim training is *free*; it claims training and serving are governed by the *same* bandwidth-bound term, so a machine that can serve a model at the wall can also personalize it at a comparable per-step cost—which is precisely what makes structurally private, on-device continual learning economically credible on a consumer APU. The honest scope: the theorem is about the bandwidth-bound *weight-streaming* cost of a step, which dominates on this memory-bound APU; it abstracts the optimizer's activation memory and the data pipeline, and it assumes the QLoRA shape of Lemma 1, which is real in code while the trainable-LoRA *layer* that would exercise it end-to-end is a Phase-1 build (Section 7).

5 The Top Three Differentiators

1. One native engine for train *and* serve—no format break. The same `hanzo-ml` (candle-fork) tensor, the same `VarMap` bf16 safetensors, the same quant core, and the same paged serving path run end-to-end. A QLoRA adapter is trained over the *exact* `qmatvec_core` kernel that serves it (Proposition 1); in-situ quantization at load uses the *same* quant code the matvec reads. Kubeflow/KServe/Ray [15, 16, 17] orchestrate *heterogeneous black-box* runtimes—each pod is someone else's engine—with a checkpoint-format seam between train and serve; hanzo owns the runtime and removes the seam. This is novel as an *integration thesis*, partly real today (serve + one RL trainer) and partly roadmap (generic SFT, orchestration); the moat is the unified runtime, not the scheduler.

2. The unified 1-bit-to-full quant core. One templated core per stage serves the whole quant zoo; adding a quant is a one-line table edit (eleven sub-4-bit types added bit-exact, zero new kernels) [1]. For serving this turns "host any community GGUF at any quant" into a table row; for training it means the personalization base can be *any* of those formats and the adapter trains over it natively (Lemma 1). Patent-weak (Section 6), but a real engineering and operational moat.

3. A structurally private, on-device continuous-learning path. This is an aspirational product direction built on a foundation that already ships. Because the full train+infer stack fits on a consumer APU (proven: decode parity, real GRPO) and training shares the serving budget (Theorem 1), personalization can happen entirely on the user's machine, and the egress contract is *structural*: if the loop only ever emits bf16 LoRA deltas, raw user data never crosses the wire by construction [3]. Per-user/per-task deltas over a shared frozen base are cheap, swappable, mergeable, and forgetting-resistant. The honest caveat: the loop, the span-level PII redactor, and the Rust federation are *not built yet*, and zero leakage is *not* claimed—differential privacy on the delta is research.

6 Ported Versus Novel

The user's explicit directive during the kernel campaign was "copy llama," and the papers honor it. The line a patent attorney must trust:

Ported (prior art, do not claim). The int8 MMQ GEMM design (quantize activations once to q8_1, dequant weights to int8 in shared memory, int32 accumulate then scale)—ported line-by-line from `llama.cpp`'s `mmq.cuh` [6]; the 128×128 tile, de-fused activation quant, bank-conflict pad, 16-byte vectorized staging, WMMA/flash-attention generally; *every* quantization format and IQ codebook grid (Q4_K, Q6_K, IQ1_S–IQ4_XS, TQ, MXFP4) and their dequant math, auto-extracted verbatim from `ggml` [5]; CUDA-graph decode over PagedAttention [9]; the lane-strided matvec reduction (textbook GPU pattern); GRPO [13], LoRA/QLoRA/DoRA [11, 12], DPO/KTO/ORPO/SimPO, trimmed-mean federated aggregation, and a generative/streaming safety-moderation classifier; the serving/runtime scaffold (engine crates, `AnyMoE` gate training, paged attention, ISQ/UQFF, the model-zoo definitions) is a rename-fork of EricLBuehler's `mistral.rs` (`mistralrs-*`→`hanzo-*`); and the tensor/autograd substrate `hanzo-ml` is a fork of HuggingFace `candle`—all public. These forks supply the *scaffold* only; the ROCm/HIP/Metal/CUDA quant-kernel layer and the unified quant core built on them (∼73K LoC in `hanzo-ml`, ∼130K in the engine—the measured decode-parity work) are hanzo-original, present in neither upstream.

Novel (candidate IP, rated skeptically). (i) The *unified quant-agnostic compute core*—`qmatvec_core<WTYPE>` + `qmmq_core<WTYPE>` + per-type decode + traits row + generation macros, such that one decode core and one prefill core serve the entire 1-bit-to-full zoo and the same cores drive decode, prefill, and MoE with a one-line add-a-quant contract. Patent verdict: *weak-to-moderate; better as trade secret*—llama's shared-MMA-loop/swappable-front-end design and CUTLASS-style typed GEMM substantially anticipate it. (ii) *Heterogeneous-vendor federation via a canonical-bf16 LoRA-delta seam* (MLX/CUDA/ROCm/CPU federate with no cross-vendor collective) plus a capacity scheduler—code exists in Python (`zen/gym`), not Rust. Verdict: *strongest candidate (medium-high)*; file on the system combination, not on federated learning generally. (iii) The *on-device privacy continual-learning loop* (PII gate on *self-generated* data + delta-only egress + robust merge, with differential-privacy-on-delta as the novelty hook)—not yet assembled; file before building. (iv) A *span-level PII redactor feeding a distilled in-process quantized guard*—the thing per-turn classifiers do not do; greenfield, fold into the privacy filing.

Honest bottom line. File nothing as a strong standalone patent from the kernel campaign; the unified core is at best a narrow defensive filing or a trade secret; the federation + privacy-loop

combinations are the better filings and need counsel plus a formal prior-art search. There is no "we beat everyone" claim anywhere to support.

7 The Phased Roadmap

The perf campaign is the proven foundation—Phase 0 is done. The ordering below is by value/effort with the companion papers' risk estimates, and it respects the hard dependency that the Var-backed model-construction path is shared by SFT, the trainable LoRA layer, and a real GRPO policy [2].

Table 2: Phased roadmap. Phase 0 is shipped; risks and efforts are from the companion papers [1, 2, 3, 4].

Phase	Item	Risk	Status
0	Inference parity + unified core + substrate	—	Shipping
1	Trainable LoRA/QLoRA layer over frozen base	LOW	Planned (∼1 wk)
1	Real SFT loop (replace no-op stub)	MED	Planned (1–2 wk)
1	Real-transformer GRPO policy	MED	Planned (1–2 wk)
2	Span-level PII redactor + in-proc guard	—	Planned (greenfield)
2	Capture/self-gen/replay + eval gate	—	Planned
2	Preference losses (DPO/KTO/ORPO/SimPO)	LOW	Planned (days)
2	Per-user/per-task LoRA delta stacks	LOW	Planned (after 1.1)
3	Port `zen/gym` federation control plane to Rust	—	Planned (ported design)
3	Multi-node placement / autoscaling loop	—	Planned (build or borrow)
4	Multi-wave WMMA flash-attn prefill (parity)	MED-HIGH	Research
4	DP-on-delta; BitDelta; collapse stability	—	Research

Phase 0 — Foundation (done, shipping today). Native-ROCm inference at decode parity / 0.86× prefill / 2× MoE; the unified quant core (six native + eleven bit-exact); real PagedAttention serving; in-situ quantization; the autograd + AdamW + VarMap training substrate; GRPO + MoE-gate as proof the substrate trains. This is the measurement-driven base that de-risks everything after it: the kernels are at the roofline, the numbers are honest, the abstraction is proven.

Phase 1 — Make training real (the highest-value unlock). (1) A trainable LoRA/QLoRA adapter layer over a frozen base (QLoRA over `QTensor` via `qmatvec_core`); risk LOW—the AD primitive is proven (Lemma 1), the MoE-gate trainer proves train-over-frozen, and the adapter math exists in `loralinear.rs`; ∼1 week. *This is the single highest-value build*: it turns the train-and-serve-one-engine thesis from latent to demonstrated and exercises Theorem 1 end-to-end. (2) Replace the stub `Trainer` with a real SFT loop (`hanzo_nn::AdamW` + teacher-forced cross-entropy + tokenized dataset + VarMap save); needs a Var-backed model-construction path; risk MEDIUM (plumbing; substrate proven); 1–2 weeks. (3) A real-transformer GRPO policy (implement the three trait methods against a hanzo-engine model; today only a toy policy exists); shares the Var-backed path with (2); risk MEDIUM; 1–2 weeks.

Phase 2 — The continuous-learning loop + privacy. (4) A span-level PII redactor (Rust) + a distilled-0.6B guard served in-process via the quantized transformer path; greenfield, the missing privacy primitive and a patent sub-claim. (5) Capture/self-generate/replay buffer + eval gate + dedup/diversity—the collapse defenses; the eval gate (commit a delta only if a frozen probe

set does not regress) is the single most important guard against drift. (6) Preference losses (DPO/KTO/ORPO/SimPO) over the sequence log-probability the GRPO policy provides; risk LOW, days each. (7) Per-user/per-task LoRA delta stacks over the shared frozen quant base (swappable, mergeable "LoRA soup"), enabled by Phase 1.1.

Phase 3 — Platform + federation. (8) Port the `zen/gym` federation control plane to Rust over the VarMap bf16 seam (coordinator/worker/transport + HMAC + trimmed-mean); the `worker.py` loop maps cleanly onto the native trainer; the strongest patent angle—file before/at build. (9) A multi-node placement/autoscaling control loop (scale-to-zero via the existing unload/reload machinery); this is where Kubeflow/Ray are mature and hanzo must build or borrow.

Phase 4 — Beating llama, and research (aspirational). Prefill parity via a multi-wave cooperative WMMA flash-attention prefill kernel [10] (the scalar and single-wave attempts are documented as reverted/insufficient; even a perfect attention kernel lands $\approx 4\%$ short, so true parity also needs porting llama's register-lean `mma.cuh` WMMA abstraction to push the GEMM further); surpassing llama needs the ROCm-PagedAttention keystone (now built) plus split-KV flash-decode and speculative decode/MTP; research includes differential-privacy noise on the delta, BitDelta 1-bit delta quantization over the unified core, long-horizon model-collapse stability [14], and CubeCL backend portability (verdict not-yet—its HIP compiler cannot emit the int8 `iu8` intrinsic).

8 Honest Risks

(1) The prefill gap is real, and parity may be the ceiling. The remaining $0.86\times \rightarrow 1.0\times$ is a multi-wave WMMA flash-attention kernel plus deeper GEMM work, and the documented arithmetic shows even perfect attention lands $\approx 4\%$ short [1]. Do not promise SOTA on AMD; AMD is honestly a *cost lever* (parity $\rightarrow$ cheaper/more-available silicon), not a speed win over NVIDIA.

(2) The training stack is mostly potential. Two real trainers prove the substrate, but the generic SFT trainer is a literal no-op stub, there is no trainable-LoRA layer in the tree, and the GRPO policy is implemented only for a toy [2]. The "train and serve one engine" story is credible and well-scoped but currently demonstrated only by GRPO + the MoE gate, not by a general fine-tune. Phase 1 is make-or-break.

(3) Model collapse in the self-training loop. Recursive training on synthetic output is a published failure mode [14] and the default outcome to be actively prevented. The eval gate + grounded-data weighting + dedup + replay *reduce* it; long-horizon autonomous stability is research. Do not ship an unsupervised nightly self-retrainer [3].

(4) Privacy is defense-in-depth, not a guarantee. Per-turn classifiers cannot redact spans, so the redactor is greenfield; the delta itself can memorize and leak; differential privacy on the delta is not in code. The defensible claim is structural egress containment (delta-only), not zero leakage.

(5) The strongest IP is Python-only and the kernel IP is patent-weak. The federation seam (the best filing) exists only in `zen/gym` Python; the unified core (the best engineering moat) is more defensible as a trade secret than a patent. Filings need counsel plus a formal prior-art search (Section 6).

(6) Single-machine, single-vendor validation. Every number is from one 8060S box; the CUDA fast-MMQ path is wired but unvalidated (no NVIDIA hardware); the Metal graph code is Mac-unvalidated. Cross-vendor and multi-node claims are design, not measurement.

Denominator hygiene. The companion papers carry three figures for the achievable read bandwidth (217 conservative / 231 stream-read / ≈ 253 spec); the 97% matvec claim is computed against the 231 GB/s read wall, and the ≈ 25 tok/s decode ceiling against the conservative 217 [1]. The headline prefill number is 1015 (≈ 0.86×) with 937–985 as the typical band; the in-situ GEMM figure of ≈ 1414 tok/s is kernel-isolated and does not deliver end-to-end. None of these affect the conclusions; they are labeling hygiene to settle before external publication.

9 Conclusion

The bottom line is deliberately unembellished. On a consumer RDNA3.5 APU with no vendor matmul library, a measurement-driven campaign reached effective decode parity (0.98×) and ≈ 0.86× prefill against `llama.cpp` on the same silicon, with a unified 1-bit-to-full quant core keeping every format bit-exact [1]. On that core sits a `candle`-derived Rust autograd that already trains two algorithms and contains the exact QLoRA primitive [2]. The synthesis claim of this paper is that these are one system, not two: by Lemma 1 and Proposition 1 the served weights *are* the trained base, read by the same kernel with no format break, and by Theorem 1 and Corollary 1 serving and on-device QLoRA training share one bandwidth-bound budget, so a machine that serves a model at the wall can personalize it at a comparable per-step cost. That is the moat—a unified train-and-serve runtime that orchestrate-black-box-engine platforms structurally cannot match [15, 16, 17]—and the premise of structurally private, on-device continual learning [3, 4]. We do not beat llama here, we do not claim a speed win over NVIDIA, and we do not claim zero leakage; the honest, defensible claims are the unified runtime, the structural delta-only egress contract, and a parity-enabled cost lever on cheaper silicon. Phase 1—the trainable LoRA layer and a real SFT loop—is the make-or-break that turns the thesis from formally true to operationally demonstrated.

References

[1] Hanzo AI Research. *Native ROCm Inference on a Consumer RDNA3.5 APU: Reaching llama.cpp Decode Parity via a Unified 1-bit-to-Full Quantization Compute Core*. Hanzo Industries, 2026.

[2] Hanzo AI Research. *Native Training in hanzo-ml/hanzo-engine: One Engine, One Quant Format, One Backend*. Hanzo Industries, 2026.

[3] Hanzo AI Research. *Continuously-Learning Private AI: Self-Improvement and Privacy in the Hanzo Native Stack*. Hanzo Industries, 2026.

[4] Hanzo AI Research. *Economical Serving and the hanzo.ai Platform: A Bandwidth-Optimal Unified Train-and-Serve Runtime*. Hanzo Industries, 2026.

[5] G. Gerganov et al. *ggml: Tensor library for machine learning*. 2023.

[6] G. Gerganov et al. *llama.cpp: LLM inference in C/C++*. 2023.

[7] Qwen Team. *Qwen3 Technical Report*. Alibaba, 2025.

[8] S. Williams, A. Waterman, and D. Patterson. Roofline: An Insightful Visual Performance Model for Multicore Architectures. *Communications of the ACM*, 52(4):65–76, 2009.

[9] W. Kwon et al. Efficient Memory Management for Large Language Model Serving with PagedAttention. *SOSP*, 2023.

[10] T. Dao et al. FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness. *NeurIPS*, 2022.

[11] E. Hu et al. LoRA: Low-Rank Adaptation of Large Language Models. *ICLR*, 2022.

[12] T. Dettmers et al. QLoRA: Efficient Finetuning of Quantized LLMs. *NeurIPS*, 2023.

[13] Z. Shao et al. DeepSeekMath: Pushing the Limits of Mathematical Reasoning in Open Language Models. 2024. (Group Relative Policy Optimization.)

[14] I. Shumailov et al. AI models collapse when trained on recursively generated data. *Nature*, 631:755–759, 2024.

[15] The Kubeflow Authors. *Kubeflow: The Machine Learning Toolkit for Kubernetes*. 2018.

[16] The KServe Authors. *KServe: Standardized Serverless ML Inference on Kubernetes*. 2021.

[17] P. Moritz et al. Ray: A Distributed Framework for Emerging AI Applications. *OSDI*, 2018.

[18] AMD. *ROCm: Open Software Platform for GPU Compute*. Version 7.1, 2026.