Continuously-Learning Private AI: Self-Improvement and Privacy in the Hanzo Native Stack

Zach Kelling

Hanzo AI

June 2026

Abstract

A private continuously-learning model is three loops bolted together: an *inference* loop that serves a user and captures its own interactions; a *curation* loop that filters, redacts, and deduplicates that capture into safe training data; and a *training* loop that folds the data back into cheap, swappable per-user and per-task deltas. We design this system on the Hanzo native stack, which already owns the hard inference substrate—a ROCm decode matvec at the memory-bandwidth wall, at effective parity with llama.cpp (an 8B dense transformer at Q8_0, decode 24.5 tokens/s, 0.98× the 25.05 tokens/s of llama.cpp HIP on one gfx1151 / AMD 8060S)—and a real, tested reinforcement-learning trainer (a ported GRPO implementation with passing smoke tests). It does *not* yet own the curation loop (no span-level PII redactor exists) or a Rust federated control plane (it is Python-only in the `zen/gym` project). We are explicit throughout about which pieces are real, which are ported, and which are research. We then formalize the two privacy-and-stability invariants the design rests on: a *privacy invariant*—no personally-identifiable token enters the training set because a guard filter is applied to every datum before training—which we prove holds under the curation pipeline; and a *bounded-forgetting* guarantee for delta-personalization—a frozen quantized base implies zero base-capability drift, with all change localized to low-rank adapters—which we also prove. We are honest about the two failure modes that can sink the system: model collapse from self-training, and privacy leakage through the delta.

1 Introduction

The case for keeping a learning loop on the user's own machine is not ideological; it is the only architecture in which raw user data never has to move. That requires both local inference *and* local training to be fast enough to be invisible. This paper argues that the Hanzo native stack is unusually well positioned to host such a loop, designs the complete system, and is scrupulous about the boundary between what is built and what is aspirational.

Throughout, we are precise about the status of every claim, and we make the distinction in the prose rather than in shorthand: we say plainly when code exists today and was read from source, when a capability was ported from a named open-source project, when a piece is genuinely new Hanzo intellectual property, when something is buildable on the existing substrate with bounded engineering but is not yet built, and when a problem is open research. The reader should assume nothing here is built unless the text says so.

Inference is there. A sustained optimization campaign [11] drove the 8B model's Q8_0 decode from 8.5 to **24.5** tokens/s, i.e. 0.98× the llama.cpp HIP figure of 25.05 tokens/s—effective parity. The mechanism was a rewrite of the unified quantized matvec inner loop from a serial whole-warp-per-block scheme to a *lane-strided, warp-reduce* scheme, taking the dominant feed-forward shapes from roughly 45% to **97**% of the 217 GB/s memory roofline; the matvec is now bandwidth-wall-bound. Prefill is 1015 tokens/s, or 0.86× llama.cpp's 1176 tokens/s—the harder remaining axis, requiring a multi-wave WMMA flash-attention kernel. Mixture-of-experts (MoE) decode improved 2× (6.34 → 13.6 tokens/s) via the same unified core plus a resident-bank cache. These are honest numbers on a single gfx1151 (AMD 8060S / Strix Halo): decode is at parity, prefill trails, and beating llama.cpp on ROCm would require a PagedAttention keystone that has not yet paid off. The point for *this* paper is narrower: a model that serves the user at ∼25 tokens/s decode is fast enough to also be the data generator for its own training, and the same quantized weights it serves from are the base on which deltas sit.

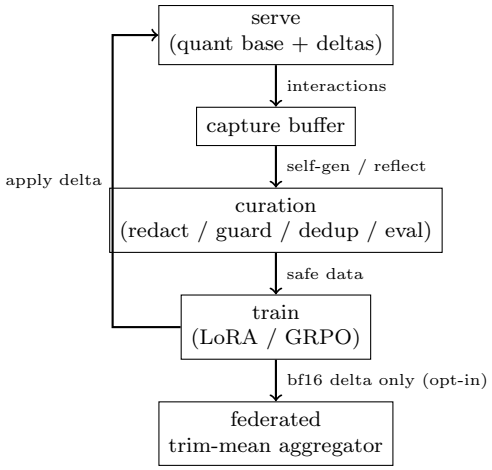

Figure 1: On-device continual-learning loop. Raw data never leaves the device; only an optional bf16 LoRA delta crosses the wire to the cross-device aggregator.

Training is partly there. As detailed in Section 4, the autograd engine, the optimizers, the checkpoint seam, and one full reinforcement-learning trainer are real and tested; a generic supervised fine-tuning (SFT) trainer that actually updates weights, and low-rank adapter injection, are not.

Contributions.

- A complete three-loop architecture (serve–capture, curate, train, optionally federate) for on-device private continual learning, mapped onto concrete Hanzo source files with explicit reality tags.
- A formal *privacy invariant* (Definition 1) and a proof that the curation pipeline maintains it (Proposition 1).
- A formal *bounded-forgetting* guarantee for delta-personalization over a frozen quantized base (Proposition 2), with proof.
- An honest treatment of the two existential failure modes—model collapse and delta leakage—and a prioritized build order separating bounded engineering from genuine research.

2 The Loop

The system is three coupled loops that never let raw data leave the device. Figure 1 sketches the data path.

Stage A—serve and capture. The quantized base model (e.g. the 8B model at Q8_0), served through the real 460-LoC `engine/hanzo-engine/src/models/quantized_qwen3.rs` in the tree today, answers the user. Interactions—prompt, response, and any explicit or implicit feedback (thumbs, edits, acceptance of a code completion)—land in a local capture buffer. This buffer is plumbing that is buildable on the existing substrate but not yet built.

Stage B—self-generate / reflect. Beyond raw capture, the model manufactures training data: (i) *self-instruct*-style synthetic prompts in the user's domain; (ii) *self-critique / reflection*—generate an answer, then generate a critique-and-revise pass, keeping the revised answer as the target (a distillation-from-self signal). The honesty boundary here: the `zen/gym` reconnaissance confirmed it ships knowledge distillation (logit / top-k and online-teacher), but the reflection / self-critique data-manufacturing described here is open research—a recipe, not ported code.

Stage C—curation. The gate that makes the loop private and stable: PII redaction, a safety gate, dedup / diversity, and an eval gate (Sections 3 and 5). Nothing leaves this stage that contains personal data or that would degrade the model.

Stage D—train. Curated samples drive a parameter-efficient update—per-task / per-user LoRA deltas (cheap, swappable; Section 6)—via the native trainers (Section 4). For preference-shaped feedback (the user picked answer A over B), a DPO/KTO-style loss; for reward-shaped feedback (a score, a passing test), the real GRPO trainer. The update touches only the delta, never the frozen quantized base, which is what makes it cheap and forgetting-resistant (Section 6).

Stage E—optional federation. If the user opts in to contribute, only the bf16 LoRA delta crosses the wire to a trim-mean aggregator (Section 7). Raw data never does. This stage exists today only as a ported design, not in Rust: it is implemented in Python in `zen/gym` and is absent from the Hanzo `net/` crate.

3 Privacy: What the Guard Must Do, and What It Cannot

The privacy property is defense in depth on the data path between capture and training. An early framing described `zen-guard` as "the Rust filter family." The code says otherwise, and getting this right is load-bearing.

Ground truth on zen-guard. Read from the three `zen-guard` repositories: it is a *model family*—a generative, streaming safety-moderation classifier that emits per-turn safety and PII labels—with *zero Rust*, no `Cargo.toml`, no `.rs` file in any of them. `zen-guard-gen` (8B) is a generative classifier: it applies a chat template, generates, and regex-parses a verdict of the form `Safety: Safe|Unsafe|Controversial; Categories: ...; Refusal: Yes|No`. `zen-guard-stream` (4B) carries a per-token classification head (`stream_moderate_from_ids(token_id, role, stream_state)` returning a risk level in {Safe, Controversial, Unsafe}). There are nine categories, one of which is PII.

The critical nuance: **PII is a classification *label*, not a redaction *action***. The model answers "does this turn contain or solicit PII disclosure?" over a whole turn. It does *not* locate spans, emit offsets, or return a `<REDACTED>` string. Therefore:

- zen-guard is the right **gate / quarantine** mechanism: any self-generated or captured sample the classifier flags Unsafe or `Categories: PII` is dropped or quarantined before it enters the training buffer. This gate is buildable on the existing substrate once the guard is served in-process.
- zen-guard **cannot scrub**. For the keep-but-redact path one must build a span-level PII redactor that the existing guard does not provide. This is the one genuinely missing privacy

primitive, and it is new work relative to zen-guard (though general named-entity redactors such as Presidio [10] are prior art).

Defense in depth. The designed pipeline applies, to every captured or self-generated datum, in order:

1. **Deterministic redactor.** A cheap Rust pass (regex plus lightweight NER) that locates and replaces structured PII spans—emails, phone numbers, card/SSN-shaped tokens, API keys and secrets, file paths embedding usernames—with typed sentinel tokens. Deterministic, inline, cents-of-a-millisecond. This is buildable on the existing substrate but greenfield: a grep over `net/` and `ml/` confirms no redactor exists.
2. **Neural gate.** Run the distilled guard classifier over the *post-redaction* sample; if it still trips PII or Unsafe (catching free-text disclosures, medical or financial narrative that regex misses), drop the sample entirely.
3. **Secret-specific filters.** High-entropy-string and known-key-prefix detectors for credentials, which a general PII model under-weights.
4. **Egress containment.** Even after redaction, raw data never leaves the device; only the LoRA delta does (Section 7).

Serving the gate efficiently. Running an 8B guard per sample is too heavy for bulk curation. The design serves a distilled 0.6B guard *in-process* through the engine's quantized transformer path (`quantized_qwen3.rs`) rather than a Python/vLLM sidecar—turning the gate into a local function call on already-fast inference. This is buildable on the existing substrate: the quantized path is real; the distilled 0.6B guard and the in-process wiring are the work that remains.

3.1 The Privacy Invariant, Formally

We now state the privacy guarantee precisely. Let $\mathcal{V}$ be the model vocabulary and let a *datum* x be a finite token sequence over $\mathcal{V}$. Let $\mathcal{P}(x) \subseteq \{0, \ldots, |x|-1\}$ denote the (ground-truth) set of token positions in x that constitute personally-identifiable information, and write $\mathsf{pii}(x) \equiv \mathcal{P}(x) \neq \varnothing$.

Let R be the deterministic redactor, a map on token sequences that replaces every position in a detected set $\hat{\mathcal{P}}(x) \subseteq \mathcal{P}(x)$ with a fixed, PII-free sentinel symbol $\perp \in \mathcal{V}$. Let F be the guard gate, a predicate $F(x) \in \{\mathsf{keep}, \mathsf{drop}\}$ with the convention that a sample is admitted iff $F(x) = \mathsf{keep}$. The composed curation filter is

$$\Phi(x) \;=\; \begin{cases} R(x), & F\big(R(x)\big) = \mathsf{keep}, \\ \varnothing, & \text{otherwise}, \end{cases}$$

and the training set after curating a capture stream X is $\mathcal{T} = \{\, \Phi(x) : x \in X,\ \Phi(x) \neq \varnothing \,\}$.

Definition 1 (Privacy invariant)**.** *A training set $\mathcal{T}$* satisfies the privacy invariant *if no admitted sample contains a PII token: for every $t \in \mathcal{T}$, $\mathsf{pii}(t) = \mathsf{false}$, i.e. $\mathcal{P}(t) = \varnothing$.*

To make the guarantee non-vacuous we state the exact assumptions under which it holds; honesty about residual risk (below) is then a statement about when these assumptions fail.

Definition 2 (Sound gate)**.** *The gate F is* sound for PII *if, for all x, $\mathsf{pii}(x) = \mathsf{true} \Rightarrow F(x) = \mathsf{drop}$. Equivalently, $F(x) = \mathsf{keep} \Rightarrow \mathcal{P}(x) = \varnothing$.*

Proposition 1 (Pipeline maintains the privacy invariant)**.** *If the redactor sentinel $\perp$ is PII-free and the gate F is sound for PII (Definition 2), then for every capture stream X the curated set $\mathcal{T} = \{\Phi(x) : x \in X,\ \Phi(x) \neq \varnothing\}$ satisfies the privacy invariant (Definition 1).*

Proof. Let $t \in \mathcal{T}$ be arbitrary. By construction of $\mathcal{T}$ there is an $x \in X$ with $t = \Phi(x)$ and $\Phi(x) \neq \varnothing$. By the definition of Φ, the second branch ($\varnothing$) is excluded, so the first branch was taken: $t = R(x)$ and $F\big(R(x)\big) = \mathsf{keep}$, i.e. $F(t) = \mathsf{keep}$.

Because F is sound for PII (Definition 2), $F(t) = \mathsf{keep}$ implies $\mathcal{P}(t) = \varnothing$. Hence $\mathsf{pii}(t) = \mathsf{false}$. Since t was an arbitrary element of $\mathcal{T}$, every admitted sample is PII-free, which is exactly Definition 1. □

Lemma 1 (Redaction never increases PII)**.** *If the sentinel $\perp$ is PII-free, then $\mathcal{P}(R(x)) \subseteq \mathcal{P}(x)$ for every x; in particular the redactor cannot create new PII.*

Proof. R acts position-wise: positions outside $\hat{\mathcal{P}}(x)$ are copied verbatim, and positions inside $\hat{\mathcal{P}}(x)$ are overwritten with $\perp$. A position i can be in $\mathcal{P}(R(x))$ only if $R(x)$ exposes PII at i. If $i \in \hat{\mathcal{P}}(x)$ then $R(x)_i = \perp$, which is PII-free by hypothesis, so $i \notin \mathcal{P}(R(x))$. If $i \notin \hat{\mathcal{P}}(x)$ then $R(x)_i = x_i$, so i can contribute PII to $R(x)$ only if it already did so in x, i.e. $i \in \mathcal{P}(x)$. Therefore $\mathcal{P}(R(x)) \subseteq \mathcal{P}(x) \setminus \hat{\mathcal{P}}(x) \subseteq \mathcal{P}(x)$. □

Lemma 1 clarifies the division of labor that the pipeline relies on: the redactor monotonically *reduces* the PII set (it removes $\hat{\mathcal{P}}(x)$ and adds nothing), while the gate F supplies the soundness that converts "reduced" into "empty." The redactor is what makes soundness of F achievable in practice—it removes the easy structured spans so the classifier only has to catch the residual free-text disclosures—but it is the gate, not the redactor, that the invariant proof leans on.

Honest residual risk. No gate is perfect, so Definition 2 is an idealization, and Proposition 1 is exactly as strong as that assumption. A real distilled classifier has false negatives; the redactor will miss novel PII formats (so $\hat{\mathcal{P}}(x)$ may be a strict subset of $\mathcal{P}(x)$); and an adversarial prompt can smuggle PII into a self-generated sample. The mitigation is layering plus egress-only-delta containment, *not* a claim of zero leakage. Two consequences follow honestly. First, the invariant degrades gracefully: by Lemma 1 every admitted sample's PII set is bounded above by what the gate failed to catch *after* the redactor has already deleted $\hat{\mathcal{P}}(x)$, so even under an imperfect gate the residual exposure is no larger than the classifier's false-negative set on already-redacted text. Second, the delta itself can memorize and leak training content—addressed in Section 7; differential privacy on the delta is open research, not in code.

4 The Training Substrate: Real vs. Stub

Read from the `ml/` tree. We separate what updates weights today from what does not.

- **Autograd (real, in the tree).** `hanzo-ml/src/backprop.rs`: `GradStore` plus `Tensor::backward()` (candle's [1] reverse-mode engine, intact); `Var` threads through.
- **Optimizers (real in the right place, a stub in the wrong place).** The real ones live in `hanzo-nn/src/optim.rs`: an `Optimizer` trait with `step(&GradStore)` and `backward_step(&loss)`; a working `SGD` and a full decoupled-weight-decay `AdamW` (per-`Var` first/second moments, bias correction, eps, weight decay). The stub is `hanzo-training/src/optimizer.rs`: `OptimizerWrapper::step` is a verbatim no-op—its own comment reads *"Simplified optimizer step—in practice, this*

Table 1: Training-substrate gaps for the continual-learning loop.

Need	Status	Work
Generic SFT (updates weights)	Stub (no-op)	Swap in `AdamW`
LoRA injection	Config only	Adapter layer + merge
DPO / KTO / ORPO / SimPO	Absent	Preference losses
GRPO on real policy	Toy tested	`Policy` for the 8B model

would update parameters"—it only bumps a counter and recomputes the learning rate. The generic `hanzo-training::Trainer` is built on that stub, so the generic SFT path does not update weights yet. Replacing the stub with `hanzo_nn::AdamW` is the headline near-term task and is buildable on the existing substrate today (the real optimizer already exists one crate over).

- **Checkpoint seam (real, in the tree).** `hanzo-nn/src/var_map.rs`: `VarMap` with `save`/`load` over safetensors and `get`/`set`. This *is* the canonical bf16 delta seam the federation layer needs (Section 7); `save` calls `safetensors::serialize_to_file`.
- **Losses and layers (real, in the tree).** `hanzo-nn/src/loss.rs`: `nll`, `cross_entropy`, `mse`, `binary_cross_entropy_with_logit`, `huber`. Transformer building blocks are present (`moe`, `rotary_emb`, `kv_cache`, `linear`, `attention`).
- **GRPO (real, tested, and ported from `zen/gym`).** This is the proof that native training in Rust is not theoretical. `hanzo-training/src/grpo/` (8 files); the `mod.rs` docstring states it was *"ported from the `zen/gym` GRPO [3] trainer."* `GrpoTrainer<P: Policy, R: Policy>` runs the full five-phase step: rollout $\rightarrow$ reward $\rightarrow$ group-relative advantage $A_i = (r_i - \text{mean})/(\text{std} + \epsilon)$ $\rightarrow$ policy-gradient loss $-\text{mean}(A_i \cdot \log p)$ plus optional KL to a frozen reference policy $\rightarrow$ `AdamW.backward_step(&loss)`, with a non-finite-loss guard and per-step seed advance. The `Reward` trait is `Send + Sync` and pluggable. Two smoke tests train a toy policy through the real code path and assert that mean reward climbs toward the ceiling.

The gaps to close for the loop are summarized in Table 1; all are buildable on this substrate with bounded engineering. The reward-shaped path (a test passes, a score) is runnable today through GRPO—only a real `Policy` over the engine model is missing. The preference-shaped path (the user picked A over B) needs the DPO/KTO [5] losses (a straightforward port); the plain SFT path needs the stub replaced. None of this is research; it is bounded engineering on a substrate that already runs one RL recipe end to end.

5 Self-Generated Data Quality: Not Poisoning the Well

The existential risk of a self-training loop is *model collapse*: recursive training on synthetic output narrows the distribution until the model forgets the tails and degenerates. This is a published failure mode [4], and any honest design must treat it as the default outcome to be actively prevented, not an edge case. The curation-stage mitigations are:

1. **Grounding beats pure self-gen.** The strongest data is real captured interactions with a real feedback signal (the user edited the answer; the test passed; the completion was accepted). Self-generated synthetic data is a supplement, weighted below grounded data, never the sole

source. Reflection / self-critique keeps a *verifiable* target (a revised answer that passes a check), not a free-floating one.

2. **Eval gate.** Maintain a frozen held-out probe set (a small fixed eval and an identity/behavior canary). After each candidate delta, evaluate; commit the delta *only* if the eval does not regress. This is the single most important guard against silent drift, and it is buildable on the existing substrate: `hanzo-training` already has an `evaluation.rs` and an `eval` binary.

3. **Dedup and diversity.** Near-duplicate self-gen samples amplify whatever bias produced them. De-duplicate (hash / embedding-similarity) and enforce a diversity floor. Buildable on the existing substrate but greenfield.

4. **Replay / anti-forgetting mixing.** Interleave a small fraction of general base data with the personalized data so the delta personalizes without erasing the base distribution—exactly what LoRA-over-frozen-base buys cheaply (Section 6).

5. **Provenance and reversibility.** Every committed delta is checkpointed (`VarMap::save`, which is real and in the tree) and tagged with the data window that produced it, so a bad update is one `load` away from rollback.

Honest framing: items 2–5 *reduce* collapse risk; they do not eliminate it. Long-horizon stability of a fully autonomous self-training loop is open research. The defensible near-term posture is bounded, gated, human-signal-anchored continual updates with a hard eval gate and one-click rollback—not an unsupervised model that retrains itself nightly on its own hallucinations.

6 Personalization: Delta Stacks over a Shared Quantized Base

The architecture that makes per-user / per-task continual learning cheap is a stack of small deltas over one frozen, quantized base:

- **One shared quantized base in VRAM.** The engine already serves quantized weights (Q8_0 at the bandwidth wall today; native decode also wired for Q4_K). The base is frozen and shared across every user and task on the device—paid for once.

- **Per-task / per-user LoRA deltas.** Each personalization is a low-rank delta [2] (a few MB), trained on that user's curated data. *Swappable*: load the delta for the active context, hot-swap on task change. *Mergeable*: a "LoRA soup" averages or trim-means a set of deltas into one (the same operation the federated aggregator performs, Section 7). This is buildable on the existing substrate once LoRA injection exists (Section 4); the config knobs are already declared.

Why deltas, specifically, for continual learning: they are *cheap* (train and store MB, not GB; many personas coexist over one base); *forgetting-resistant by construction* (the base is frozen, so general capability cannot be overwritten—the delta only adds); and *composable and reversible* (stack, merge, or drop deltas, and roll back a bad one without touching the base).

On BitDelta—an explicit honesty note. Earlier framing mentioned BitDelta as "ideal." **BitDelta does not exist anywhere in this stack.** Reconnaissance grep-confirmed: there is no BitDelta, no delta-compression, no model-soup/mergekit/TIES/DARE in `zen/gym`, and the only "DeltaSoup" is the federated trim-mean aggregator in `zen/gym` (Section 7)—which averages

full-precision bf16 LoRA deltas, not 1-bit BitDelta. So: LoRA-delta personalization is buildable on the existing substrate (pending injection code); BitDelta [6]-style 1-bit delta quantization is open research—attractive, and a natural fit for the existing unified `quant_format!` / `qmatvec_core<WTYPE>` machinery the ROCm work built, but not implemented and not to be claimed as present.

The unified quant abstraction is the genuinely novel Hanzo IP that makes the base side of this story strong: the ROCm work collapsed all formats, from 1-bit ternary to full precision, into one quant-agnostic GPU core—`qmatvec_core<WTYPE>` for decode, `qmmq_core<WTYPE>` for prefill, with a single `decode_block<WTYPE>` plus a traits row per format and zero new kernels per quant. That is the substrate that would host BitDelta deltas natively if and when built.

6.1 Bounded Forgetting, Formally

We now prove the structural forgetting-resistance claim. Let $W_0 \in \mathbb{R}^{m\times n}$ be a (frozen) base weight matrix and let the LoRA adapter be $\Delta W = \frac{\alpha}{r} BA$ with $B \in \mathbb{R}^{m\times r}$, $A \in \mathbb{R}^{r\times n}$, $r \ll \min(m,n)$. The effective weight is $W = W_0 + \Delta W$. Let $f_W(\cdot)$ denote the network's map under weights W, and let $\mathcal{C}_{\mathcal{B}}(W) = \mathbb{E}_{x\sim\mathcal{B}}\,[\,\ell(f_W(x))\,]$ be a base-capability functional measured on a frozen base distribution $\mathcal{B}$, for a fixed metric ℓ (e.g. a held-out base-eval score). Training adjusts only the adapter parameters (A, B); W_0 is never written.

Definition 3 (Base-capability drift)**.** *The* base-capability drift *induced by an adapter ΔW is $D(\Delta W) = \mathcal{C}_{\mathcal{B}}(W_0 + \Delta W) - \mathcal{C}_{\mathcal{B}}(W_0)$.*

Proposition 2 (Bounded forgetting under a frozen base)**.** *Suppose training writes only the adapter parameters (A, B) and leaves W_0 byte-for-byte unchanged. Then:*

(i) ***Zero drift at initialization.*** *Under the standard LoRA initialization $B = 0$ (with A arbitrary), the adapter contributes $\Delta W = 0$, so $W = W_0$ and $D(\Delta W) = 0$ exactly. The model at the start of every personalization episode is bitwise the base on the entire base distribution.*

(ii) ***Localized, reversible change.*** *For any trained adapter, the change to the effective weight is confined to the rank-$\le r$ subspace* $\mathrm{col}(B)$*: $\mathrm{rank}(\Delta W) \le r$ and $\Delta W\, v = 0$ for every $v \in \ker(A)$, a subspace of dimension at least $n - r$. Setting $(A, B) \leftarrow (A, 0)$ restores W_0 exactly, so any drift $D(\Delta W)$ is fully reversible without touching the base.*

(iii) ***Bounded drift.*** *If $\mathcal{C}_{\mathcal{B}}$ is L-Lipschitz in the weights under the spectral norm, then $|D(\Delta W)| \le L\,\|\Delta W\|_2 \le L\,\frac{\alpha}{r}\,\|B\|_2\,\|A\|_2$. In particular, with the standard practice of mixing a replay fraction of base data and gating on the base eval (Section 5), the optimizer is steered toward small $\|\Delta W\|_2$ and hence small drift.*

Proof. (i) With $B = 0$ we have $\Delta W = \frac{\alpha}{r} BA = 0$ regardless of A, hence $W = W_0 + 0 = W_0$. Therefore $\mathcal{C}_{\mathcal{B}}(W_0 + \Delta W) = \mathcal{C}_{\mathcal{B}}(W_0)$ and $D(\Delta W) = 0$ by Definition 3. Since every layer's effective weight equals its base weight and no other parameters are modified, $f_W \equiv f_{W_0}$ pointwise, so the equality holds on the entire support of $\mathcal{B}$ (indeed on all inputs), not merely in expectation.

(ii) $\Delta W = \frac{\alpha}{r} BA$ is a product whose left factor B has at most r columns, so $\mathrm{rank}(\Delta W) \le \mathrm{rank}(B) \le r$. For $v \in \ker(A)$ we have $Av = 0$, hence $\Delta W\, v = \frac{\alpha}{r} B(Av) = 0$; by rank–nullity $\dim\ker(A) \ge n - \mathrm{rank}(A) \ge n - r$. Because W_0 was never written, the base weights are still present in memory unmodified; assigning $B \leftarrow 0$ yields $\Delta W = 0$ and restores $W = W_0$ exactly, so the map returns to f_{W_0} and the drift returns to 0.

(iii) By L-Lipschitzness of $\mathcal{C}_{\mathcal{B}}$ under the spectral norm,

$$|D(\Delta W)| = \left|\mathcal{C}_{\mathcal{B}}(W_0 + \Delta W) - \mathcal{C}_{\mathcal{B}}(W_0)\right| \leq L \, \|\Delta W\|_2.$$

Submultiplicativity of the spectral norm gives $\|\Delta W\|_2 = \frac{\alpha}{r}\|BA\|_2 \leq \frac{\alpha}{r}\|B\|_2\|A\|_2$, which yields the stated bound. The drift is thus controlled by the adapter norm; the bound is tight in the trivial direction by part (i) ($\|\Delta W\|_2 = 0 \Rightarrow D = 0$). □

The contrast with full fine-tuning is the point of Proposition 2. If training were allowed to write W_0, then part (i) fails (there is no fixed reference to return to), part (ii) fails (the change need not be low-rank and is not reversible by zeroing an adapter), and catastrophic forgetting of the base is possible. Freezing the base converts forgetting-resistance from a hope into a structural property: drift is exactly zero at the start of each episode, is confined to a rank-$\leq r$ subspace, is reversible by construction, and is norm-bounded thereafter. The adapter can still over-specialize *within* that subspace—which is precisely what the replay mixing and eval gate of Section 5 are for—but it cannot overwrite the base.

7 Federation: Delta-Only Egress

If a user opts to contribute their learning to a shared model *without sharing their data*, the mechanism is: train locally, export only the bf16 LoRA delta, aggregate across contributors with a robust mean, pull the consensus delta back, and apply it.

Ground truth. In `zen/gym` (Python-only), the load-bearing invariant is a canonical BF16 LoRA-delta safetensors blob as the cross-vendor seam: `backend/base.py`'s `export_lora_delta` / `import_lora_delta` let MLX, CUDA, ROCm, and CPU federate with no cross-vendor NCCL—`distributed/__init__.py` states, *"sync at LoRA-delta granularity over HTTP. No cross-vendor NCCL."* The control plane: `topology.py` (capacity_score $= \sqrt{\text{mem}} \cdot \sqrt{\text{tflops}}$), `scheduler.py` (capacity-weighted data sharding plus best-fit-decreasing MoE expert pinning), `transport.py` (vendor-neutral HTTP, HMAC-SHA256 over `method|path|ts|sha256(body)`), `coordinator.py` (aggregation), and `worker.py` (a model-agnostic loop: train N steps → export delta → push → pull aggregate → apply). The aggregator `coordinator.py:aggregate()` is Byzantine-robust trimmed-mean [9] (drop top and bottom 1 if $N \geq 4$), or median, or mean, over the bf16→f32 view; this is federated averaging [8] restricted to the LoRA-delta seam, and its docstring calls it *"DeltaSoup-style ... without the full reputation system."*

Two honesty boundaries.

1. **This is not in Rust.** A grep over Hanzo `net/src` for `federat|deltasoup|lora.delta|byzantine|aggregate.*delta` returns only a false-positive CSS file. The whole coordinator/worker/transport/DeltaSoup stack is a ported design whose code is not yet ported—Python in `zen/gym`, absent in Hanzo. Porting it is buildable on the existing substrate and small: the seam is just safetensors bf16 plus a uint16↔f32 bit-shift, and `VarMap` (Section 4) already gives that; a Rust coordinator/worker over `axum`/`reqwest` with HMAC is well-specified. The `worker.py` loop maps cleanly onto the native trainer: `step_fn` = `GrpoTrainer::step` (or an SFT step), `params_iter` = the LoRA `Vars`, `apply_fn` = add the consensus delta into those `Vars`.

Table 2: Measured inference performance, Hanzo vs. llama.cpp HIP, single gfx1151 (AMD 8060S).

Workload	**Hanzo**	**llama.cpp**	**Ratio**
Q8_0 decode (t/s)	24.5	25.05	0.98×
Q8_0 prefill (t/s)	1015	1176	0.86×
MoE decode (t/s)	13.6	—	2.0× †
FFN roofline (%)	97	—	—

†MoE decode improvement is 2× over the prior Hanzo baseline (6.34 → 13.6 t/s), not over llama.cpp.

Table 3: Reality status of each loop component.

Component	**Status**
Quantized inference (decode parity)	Shipping
Autograd, optimizers, checkpoint	Shipping
GRPO trainer (toy policy)	Shipping, tested, ported
Unified quant core (`qmatvec_core`)	Shipping, novel
Generic SFT (weight update)	Stub (no-op)
LoRA injection	Config only
Span-level PII redactor	Novel, not built
In-process distilled guard	Planned (buildable)
Eval gate / dedup / replay	Planned (buildable)
Preference losses (DPO/KTO/...)	Absent
Rust federation control plane	Ported design (Python only)
DP-on-delta; BitDelta; stability	Research

2. **The delta still leaks.** "Only the delta crosses the wire" bounds leakage; it does not zero it—a LoRA delta can memorize training content and is invertible-ish under attack. The clean mitigation is differential-privacy noise on the delta before egress [7], which is open research and not in code (grep-confirmed: no DP / Gaussian / Laplace noise in `zen/gym`).

Why this is the strongest patent angle. The heterogeneous-vendor federation via the canonical-bf16 delta seam plus capacity scheduler is the strongest filing candidate: code substantially exists (Python), and the non-obvious combination is the *backend-identity-erasing bf16-delta as the sole interop seam* enabling truly mixed-vendor federation with no shared collective. The plain trimmed-mean algorithm itself is federated-learning prior art and must not be claimed alone. A second candidate is the privacy continual-learning loop (self-gen PII gate plus delta-only egress plus trim-mean, with DP-on-delta as the novelty hook to clear generic-FL prior art)—to be filed *before* building.

8 Results and Status Summary

Table 2 collects the measured performance numbers from the ROCm campaign (single gfx1151 / AMD 8060S, 217 GB/s memory roofline); Table 3 summarizes the reality tags for each component of the loop. We report parity, not superiority: decode is at 0.98× llama.cpp, prefill trails at 0.86×.

9 Limitations and Future Work

We are explicit about what is not solved.

The two failure modes that can sink the system. *Model collapse* is the default failure of self-training; Section 5's eval gate, grounded-data weighting, dedup, and replay *bound* it but do not eliminate it. Long-horizon autonomous stability is open research, so one must not ship an unsupervised nightly self-retrainer. *Privacy leakage* is bounded by the redactor → gate → secret-filter → delta-only-egress chain of Sections 3 and 7; but the redactor is greenfield, DP-on-delta is research, and Proposition 1 is exactly as strong as the gate-soundness assumption it rests on—so there is *no* "zero leakage" claim.

Three lesser risks. Per-user deltas can over-specialize despite a frozen base (Proposition 2 bounds base drift, not within-subspace specialization; mitigated by replay and eval, not eliminated). zen-guard miscategorizes and labels PII per turn, not per span—so the redactor must do the span work the classifier cannot. And the eval probe set itself can be gamed by a loop that overfits to it; rotate probes and keep a canary.

Build order. The dividing line is sharp. Items 1–6 are bounded engineering buildable on the existing substrate—which already runs one RL recipe end to end and serves quantized inference at decode parity with llama.cpp—while item 7 is genuine open research and the cleanest patent trigger.

1. Replace the `OptimizerWrapper` stub with `hanzo_nn::AdamW`—a generic SFT trainer that actually updates weights.
2. LoRA injection layer plus merge—enables the swappable per-user delta stack.
3. Span-level PII redactor in Rust plus a distilled 0.6B guard served in-process—the curation gate, and the missing privacy primitive.
4. Eval gate, dedup/diversity, and capture/replay buffer—the collapse defenses and the capture/self-gen plumbing.
5. Preference losses (DPO/KTO/ORPO/SimPO) plus a real `Policy` over the engine model—both feedback-shaped training paths.
6. Port the `zen/gym` federation control plane to Rust over the `VarMap` bf16 seam—opt-in delta-only federation.
7. *(Research, not engineering.)* DP-noise-on-delta; BitDelta 1-bit delta quantization over the unified `qmatvec_core<WTYPE>` core; long-horizon collapse stability.

10 Conclusion

A private continuously-learning model is realizable on the Hanzo native stack because the hardest substrate—fast on-device quantized inference—is already at parity with llama.cpp on ROCm, and one full RL trainer already runs end to end. The privacy and stability properties the design rests on are not hand-waved: the curation pipeline maintains a precise privacy invariant under a sound gate (Proposition 1), and delta-personalization over a frozen base guarantees zero base-capability drift at episode start with norm-bounded, reversible, low-rank change thereafter (Proposition 2). Everything we claim as built was read from source; every performance figure was cited from the measured ROCm campaign; and the two pieces that can sink the system—model collapse and delta leakage—are named, bounded, and left honestly open.

References

[1] The Candle Authors. *Candle: Minimalist ML Framework for Rust*. Hugging Face, 2023.

[2] E. Hu, Y. Shen, P. Wallis, Z. Allen-Zhu, Y. Li, S. Wang, L. Wang, and W. Chen. *LoRA: Low-Rank Adaptation of Large Language Models*. ICLR, 2022.

[3] Z. Shao, P. Wang, Q. Zhu, et al. *DeepSeekMath: Pushing the Limits of Mathematical Reasoning in Open Language Models*. arXiv:2402.03300, 2024.

[4] I. Shumailov, Z. Shumaylov, Y. Zhao, Y. Gal, N. Papernot, and R. Anderson. *The Curse of Recursion: Training on Generated Data Makes Models Forget*. arXiv:2305.17493, 2023.

[5] R. Rafailov, A. Sharma, E. Mitchell, S. Ermon, C. Manning, and C. Finn. *Direct Preference Optimization: Your Language Model is Secretly a Reward Model*. NeurIPS, 2023.

[6] J. Liu, G. Xiao, K. Li, J. D. Lee, S. Han, T. Dao, and T. Cai. *BitDelta: Your Fine-Tune May Only Be Worth One Bit*. NeurIPS, 2024.

[7] M. Abadi, A. Chu, I. Goodfellow, H. B. McMahan, I. Mironov, K. Talwar, and L. Zhang. *Deep Learning with Differential Privacy*. ACM CCS, 2016.

[8] H. B. McMahan, E. Moore, D. Ramage, S. Hampson, and B. A. y Arcas. *Communication-Efficient Learning of Deep Networks from Decentralized Data*. AISTATS, 2017.

[9] D. Yin, Y. Chen, K. Ramchandran, and P. Bartlett. *Byzantine-Robust Distributed Learning: Towards Optimal Statistical Rates*. ICML, 2018.

[10] Microsoft. *Presidio: Data Protection and De-identification SDK*. 2020.

[11] Hanzo AI Research. *Native ROCm Quantized Inference on gfx1151: A Bandwidth-Wall Optimization Campaign*. Hanzo AI, 2026.