Economical Serving and the hanzo.ai Platform: A Bandwidth-Optimal Unified Train-and-Serve Runtime

Zach Kelling

Hanzo AI

June 2026

Abstract

We present the serving economics, platform architecture, and on-device thesis that follow from a native Rust train-and-inference engine whose decode kernel runs at the memory-bandwidth wall. On a single AMD Ryzen AI Max 8060S (Strix Halo, gfx1151, RDNA3.5 iGPU) with an achievable read peak of ∼253 GB/s, the unified quant-matvec core reaches 245.8 GB/s after a lane-strided rewrite—97% of that peak, above the conservative 231 GB/s stream-copy figure. From this roofline we derive a cost law: for a memory-bound autoregressive decoder, tokens per second per dollar scales as achieved bandwidth divided by the product of per-parameter quantization width and active parameter count, making the quantization choice—not the purchase of larger accelerators—the primary cost lever. We prove this proposition and a companion multi-tenant throughput result under paged attention. The engine attains decode parity (0.98× llama.cpp) on an 8B dense transformer at Q8_0 and 2× throughput on a 35B-A3B mixture-of-experts model, both on consumer silicon, supports eleven sub-4-bit quantization formats bit-exactly against `ggml`, and shares one tensor type and one quantization core across training and serving. We describe the platform control plane and an on-device continuous-learning product direction, and we are explicit throughout about what exists today versus what remains roadmap. The honest claims are: AMD parity makes AMD a cost lever; a genuinely unified native train-and-serve runtime exists; and a structurally private on-device path is feasible—each with its gaps named.

1 Introduction

Autoregressive token generation (decode) on modern transformers is bound by memory bandwidth, not by arithmetic throughput. At batch size one, generating each token requires streaming the model's resident weights from memory once; the matrix-vector products that dominate decode are starved for operands long before they are starved for floating-point units. The roofline model [4] makes this concrete: a kernel whose arithmetic intensity falls below the machine's compute-to-bandwidth ratio is bandwidth-limited, and the per-token matrix-vector products sit far on the bandwidth-bound side of the ridge. This single fact, once measured rather than assumed, reorganizes the economics of model serving: the cost of a token is set by how many bytes must be read to produce it, and that byte count is governed by the on-disk quantization format.

This paper reports the consequences of building a native-Rust train-and-inference engine on top of that fact. The engine's quantization kernels were tuned on a single consumer AMD APU—the Ryzen AI Max 8060S (Strix Halo, gfx1151)—through a measurement-driven campaign that lifted the decode matrix-vector kernel to 97% of the chip's achievable read bandwidth and brought end-to-end decode to parity with the reference implementation, `llama.cpp`. We use that grounded artifact to argue three things, in decreasing order of present-day completeness:

1. **Economical per-quant serving** (Section 2): the quantization-to-cost law, a single quant-agnostic GPU core that removes the per-format kernel-maintenance burden, and the multi-tenant serving primitives (paged attention, continuous batching, model swap, load-time quantization) that exist in the tree today.

2. **The hanzo.ai platform** (Section 3): one control plane that takes a model from develop to train to personalize to serve, whose differentiator is a single native runtime shared by training and serving rather than an orchestrator of heterogeneous black-box engines. We are explicit that the runtime substrate is real while the orchestration, the generic supervised trainer, and the federation port are roadmap.

3. **hanzo Desktop** (Section 4): an on-device personal AI that runs, learns, and improves entirely on the user's machine. The runtime credibly fits consumer silicon today; the continuous-learning loop, the personal-data redactor, and the federation sync are designed but not yet assembled.

We close with a build-versus-buy cost model derived only from measured throughput (Section 5) and an honest accounting of limitations (Section 6). No claim in this paper invents a benchmark; every performance figure is cited from the measured campaign on the one 8060S.

Honesty key. We annotate claims throughout: *measured* for figures from the campaign, *available* for capabilities in the tree today, *roadmap* for designed-but-unbuilt work, and *ported* for techniques adapted from `llama.cpp/ggml` or vLLM-style designs versus genuinely novel engine IP.

2 Economical Per-Quant Serving

2.1 The Roofline that Sets the Price

We first formalize the decode roofline and the cost law that follows. Throughout, let B denote achievable memory read bandwidth (bytes per second), and consider a memory-bound decoder generating tokens autoregressively at batch size one.

Definition 1 (Resident read footprint)**.** *The* resident read footprint W *of a decoder is the number of weight bytes that must be streamed from memory to generate one token. For a dense model, $W \approx q{\cdot}N$, where N is the parameter count and q is the average number of stored bytes per parameter under the chosen quantization format. For a mixture-of-experts (MoE) model that activates only a subset of experts per token, $W \approx q \cdot N_{\text{act}}$, where N_{act} is the number of* active *parameters per token.*

On the 8060S, the spec read bandwidth is 256 GB/s, but the achievable read bandwidth measured by a standalone `hipEvent` microbenchmark is 217 GB/s (measured). The decode matrix-vector kernel runs at that wall: after a lane-strided rewrite of the inner reduction loop, the unified `qmatvec_core` kernel reaches 245.8 GB/s on production decode shapes—97% of the ∼253 GB/s achievable read peak, and *above* the conservative 231 GB/s stream-copy figure (the 256 GB/s spec is not reached by any path)—effectively saturating the read wall (measured). Because the kernel is at the wall, the operator's lever is not the kernel but the footprint W. We first isolate the per-token timing fact.

Lemma 1 (Memory-bound per-token time)**.** *For a memory-bound decoder at batch size one achieving sustained read bandwidth B with resident read footprint W per token, the time to generate one token is $t = W/B$, and the token rate is $R = B/W$.*

Proof. Generating one token requires reading the resident footprint W from memory exactly once. By the memory-bound hypothesis the per-token wall-clock time is dominated by this transfer, with compute and launch overhead negligible, so it equals the time to move W bytes at sustained bandwidth B, namely $t = W/B$. The token rate is the reciprocal $R = 1/t = B/W$. □

Theorem 1 (Quantization is the primary cost lever)**.** *Let a memory-bound decoder achieve sustained read bandwidth B, and let it produce tokens at rate R (tokens per second) with resident read footprint W per token. Let hardware cost be C dollars per second. Then*

$$R \;=\; \frac{B}{W} \;=\; \frac{B}{q\,N_{\text{act}}}, \tag{1}$$

and the throughput-per-dollar is

$$\frac{R}{C} = \frac{B}{C\,q\,N_{\text{act}}}. \tag{2}$$

Consequently, for fixed silicon (B, C) and fixed model topology N_{act}, tokens per second per dollar is inversely proportional to the per-parameter quantization width q; the quantization choice is the dominant controllable cost lever.

Proof. By Lemma 1 the token rate is $R = B/W$. Substituting $W = q\,N_{\text{act}}$ from Definition 1 gives Equation (1). The monetary cost to produce R tokens in one second is C dollars, so the throughput per dollar is $R/C = B/(C\,q\,N_{\text{act}})$, which is Equation (2). For fixed B, C, and N_{act}, this quantity varies as $1/q$. Halving q (for example, moving from an 8-bit to a 4-bit weight format) doubles R/C on the same chip, whereas purchasing higher-bandwidth silicon improves R/C only linearly in B/C and typically raises C. Hence q is the dominant lever an operator controls without changing hardware. □

Two consequences for serving economics follow immediately from Theorem 1.

Right-size by quant, not by buying bigger accelerators. Halving bytes per token (for example, Q8_0 → Q4_K) roughly doubles the decode ceiling on the *same* silicon. Since the matrix-vector kernel is already at the bandwidth wall, the lever a serving operator actually controls is the on-disk quantization, not the kernel.

MoE is the cheapest-per-quality tier. By Definition 1, a sparsely-gated mixture-of-experts model [6] reads only its active parameters per token. A 35B-A3B model reads only the ∼3B activated parameters, so a 35B-capacity model decodes like a ∼3B dense model. The campaign measured 13.6 tokens per second for a 35B-A3B MoE at Q4_K on the consumer APU (measured), a 2× improvement over the prior MoE path delivered by routing each expert through the same banked decode core.

Table 1 reports the measured decode figures and the bandwidth-implied ceilings on the single 8060S.

Table 1: Decode economics on one AMD 8060S (achievable read bandwidth 217 GB/s). Ceilings are computed from Equation (1); measured figures are from the ROCm campaign.

Model	**Quant**	**Resident**	**Decode (t/s)**
8B dense	Q8_0	∼8.7 GB	24.5 (0.98×)
8B dense	Q4_K	∼4.9 GB	kernel 181 GB/s
35B-A3B MoE	Q4_K	∼3 GB active	13.6 (2×)

The 8B-model Q8_0 figure of 24.5 tokens per second is 0.98× the reference `llama.cpp` decode rate of 25.05 on the same chip—effective parity, and the honest decode ceiling for this kernel-plus-fusion approach. The Q4_K kernel itself runs at 181 GB/s (79% of peak) at decode shapes and is already bandwidth-efficient; the 6–8 tokens per second observed on a 14B-Q4_K model is engine-side overhead amplified by depth, not a kernel deficiency—an honest gap discussed in Section 6.

2.2 One Quant-Agnostic Core, Not a Kernel-per-Quant Zoo

The campaign's central engineering artifact (novel) is a single quant-agnostic GPU core rather than a kernel-per-quant collection. The design mandate was explicit: support 1-bit through full precision through one templated core. The delivered structure is:

- A templated decode kernel `qmatvec_core<WTYPE,XT>` and a prefill kernel `qmmq_core<WTYPE>`, each with one body and a swappable per-type block decoder `qdec<WTYPE>::partial` plus a traits record `qdw_traits<WTYPE>` carrying block bytes, element count, symmetry, and scale-slot count. A macro emits the f16 and bf16 entry points.
- Table-driven dispatch through a `RocmQuantType` enum (with `from_ggml`, `decode_kernel`, `block_elems`, `block_bytes`), eliminating per-quant conditional ladders.
- On the CPU/format side (novel), a `quant_format!` macro and a `for_each_quant!` table single-source every block layout, with a byte-exact proof harness showing macro-generated code equals hand-written code for Q4_0, Q8_0, and IQ4_NL.

The operational payoff is a proven invariant: a new quantization format costs one block decoder, one traits record, one macro line, and one enum arm, with *zero* new kernels, and works across decode and MoE. We make this precise.

Proposition 1 (Constant per-format integration cost)**.** *Under a quant-agnostic core parameterized by a per-format block decoder and a traits record, adding support for a new quantization format requires* $O(1)$ *new code units—one decoder, one traits record, one table entry—and zero new compute kernels, while the decode, prefill (where unified), and MoE execution paths are reused unchanged.*

Proof. The compute kernels `qmatvec_core` and `qmmq_core` are written once and are polymorphic in the type parameter `WTYPE`; their control flow and memory-staging logic do not branch on the format. All format-specific behavior is confined to two artifacts: the block decoder `qdec<WTYPE>::partial`, which maps a stored block to dequantized values, and the traits record `qdw_traits<WTYPE>`, which supplies the byte and element geometry. Dispatch is a table lookup keyed by the format enum, not a hand-written conditional, so registering a format adds exactly one table arm. Therefore the marginal cost of a new format is the three constant-size artifacts (decoder, traits, table arm), independent of the number of formats already supported, and the kernel count is unchanged. Reuse across decode and MoE follows because the MoE path invokes the same `qmatvec_core` per routed expert. □

The empirical evidence for Proposition 1 is that eleven sub-4-bit IQ, ternary, and FP4 formats (IQ1_S/M, IQ2_XXS/XS/S, IQ3_XXS/S, TQ1_0/TQ2_0, NVFP4, Q1_0) were added as load-plus-dequantize paths and validated bit-exact against the `ggml` reference [3], with zero mismatching elements out of 512 and maximum error 0.0 for all eleven (measured). For a serving platform that intends to host any community GGUF at any quantization, this converts a combinatorial maintenance burden into a one-line table edit. The int8 matrix-core structure and the IQ codebook grids themselves are ported from `ggml/llama` [3]; the unified single-core abstraction over them is the engine IP.

2.3 Multi-Tenant Serving Primitives that Exist Today

The engine ships a real paged-attention serving path (available; ported, PagedAttention-style [1]) comprising a block-granular paged key-value (KV) cache, an iteration-level continuous-batching

scheduler with preemption [2], and prefix caching for shared system prompts. We formalize why paged attention raises multi-tenant throughput.

Definition 2 (Paged KV cache)**.** *A* paged KV cache *partitions device KV memory into fixed-size blocks of s tokens each. A sequence of length ℓ occupies $\lceil \ell/s \rceil$ blocks drawn from a shared pool of P blocks, rather than a contiguous per-request reservation sized to a maximum context length $L_{\max}$.*

Proposition 2 (Paged attention raises tenant concurrency)**.** *Let device KV memory hold P blocks of s tokens. Under contiguous per-request allocation sized to $L_{\max}$, the number of concurrently admissible sequences is $\lfloor Ps/L_{\max} \rfloor$. Under paged allocation, a set of sequences with lengths $\ell_1, \ldots, \ell_k$ is jointly admissible whenever*

$$\sum_{i=1}^{k} \left\lceil \frac{\ell_i}{s} \right\rceil \;\le\; P. \tag{3}$$

For any workload in which actual lengths ℓ_i are below $L_{\max}$, the paged bound admits at least as many sequences, and strictly more whenever some $\ell_i < L_{\max}$; the internal fragmentation per sequence is bounded by one block, i.e. at most $s-1$ wasted token slots.

Proof. Contiguous allocation reserves $L_{\max}$ token slots per sequence regardless of realized length, so the slot budget Ps admits at most $\lfloor Ps/L_{\max} \rfloor$ sequences. Paged allocation reserves only the blocks a sequence actually occupies, namely $\lceil \ell_i/s \rceil$ blocks for sequence i; the pool of P blocks therefore admits any set satisfying Equation (3). Because $\lceil \ell_i/s \rceil \le \lceil L_{\max}/s \rceil$ for all $\ell_i \le L_{\max}$, the paged constraint is no tighter than the contiguous one, and it is strictly looser whenever some realized length is below $L_{\max}$, since then $\lceil \ell_i/s \rceil < \lceil L_{\max}/s \rceil$. Finally, each sequence rounds its ℓ_i tokens up to a whole number of s-token blocks, so the only waste is the unfilled remainder of its last block, which is at most $s-1$ slots. Continuous batching admits new sequences into the running batch each decode step up to the budget of Equation (3), and under memory pressure the scheduler preempts the lowest-priority sequence by freeing its blocks and requeuing it, so the bound is maintained dynamically. □

The implementation realizes exactly this: a cache engine parameterized by block size and GPU/CPU block counts, a block pool and KV-cache manager for block-granular sharing, a paged scheduler with waiting and running queues and a preemption routine that frees blocks and requeues the lowest-priority sequence first, and block-hash-based prefix caching so that shared prefixes (system prompts, few-shot preambles) are computed once. A simpler first-come-first-served scheduler is the non-paged fallback.

Model swap and load-time quantization. Beyond paged attention, the top-level engine holds a concurrent map of live engine instances and a map of unloaded-but-reloadable model states, plus model aliases; registration, blocking reload, and engine reboot bring an unloaded model back on demand (available). This per-model load/unload/reload machinery is the substrate for scaling a rarely used model to zero and cold-starting it on first request. Separately, in-situ quantization (ISQ) quantizes a model into any supported format at load time—Q4_0/Q5_0/Q8_0, the K-quants Q2_K through Q6_K, HQQ, FP8, AFQ, and MXFP4—so an operator can take one fp16 checkpoint and materialize the exact quantization the cost target demands, without a separate offline conversion step (available).

2.4 AMD/ROCm as a Cost Lever versus NVIDIA: Honest Framing

The campaign proved that the engine reaches decode parity (0.98× `llama.cpp`) on a consumer AMD APU, with prefill at ∼0.86× (1015 versus 1176 tokens per second), the prefill gap being attention not yet fused on matrix cores (measured). The economic argument is *not* that the engine beats NVIDIA; it is that a dollar-per-token-competitive serving tier can run on cheaper, more available AMD consumer hardware because the engine is bit-exact and bandwidth-optimal there. Surpassing the reference (true state-of-the-art, beyond parity) on ROCm would require a ROCm paged-attention keystone plus a multi-wave matrix-core flash-attention kernel; the first two stages of ROCm paged attention were built and validated on the development box, but the anticipated graph-replay speedup did not materialize because the engine's own earlier kernel-fusion work had already removed the launch-bound overhead the graph was meant to collapse—an honestly recorded dead end. The defensible claim is AMD parity, and therefore AMD as a cost lever; we do not market a ROCm speed advantage over NVIDIA.

3 The hanzo.ai Platform

3.1 What It Is, and What Actually Exists

The platform thesis (aspirational/roadmap) is one control plane that takes a model from develop to train (natively) to personalize to serve without leaving the stack, serving any model at any quantization, with the native Rust train-and-infer engine as the runtime. The unifying idea is that the same tensors (a candle-fork `hanzo-ml`), the same serving runtime (a `mistral.rs`-derived `hanzo-engine`), the same bf16 safetensors parameter store, the same quantization core, and the same paged serving path are used end-to-end, so there is no PyTorch-to-ONNX-to-TensorRT-to-vLLM handoff with a format break at every stage.

We are explicit about the gap between thesis and tree. Table 2 maps each layer to what is real today versus the gap to a finished platform.

So the platform is partly real (serve plus one reinforcement-learning trainer), partly scaffolding (generic train, personalize), and partly a port (federation). Claiming a finished Kubeflow competitor today would be false; claiming that the runtime substrate exists and is unusually unified is true.

3.2 The Moat: A Shared Native Runtime, Not the Orchestrator

Kubeflow, KServe, and Ray Serve are orchestrators of heterogeneous black-box runtimes: they schedule pods or actors, but each pod runs someone else's engine (vLLM, TGI, Triton, a PyTorch script), and the training stack (PyTorch) and the serving stack (TensorRT/vLLM) are entirely different binaries with a checkpoint-format seam between them. The engine's differentiator (novel as an integration thesis) is that the runtime is one native-Rust engine shared by training and serving:

- **One quantization core for train-time materialization and serve-time inference.** ISQ at load uses the same quantization code the serving matrix-vector reads, so "personalize then serve at Q4_K" has no conversion stage.
- **One tensor and checkpoint type across the loop.** The bf16 safetensors store is both the training parameter store and the federation wire format; a trained delta is serve-loadable directly.

Table 2: Platform layers: real today versus gap. Status keys: available, roadmap, separate runtime.

Layer	Real today	Gap to platform
Serve	`hanzo-server` (OpenAI HTTP + MCP), multi-model map, paged scheduler, ISQ, swap/reload	Multi-node placement and autoscaling control loop
Train	candle autograd, real AdamW/SGD, VarMap checkpoint, tested GRPO	Generic SFT `Trainer` is a stub; no LoRA injection; DPO/KTO/ORPO absent
Personalize	LoRA config declared; QLoRA-capable base	LoRA layer/merge code not written
Federate	Design in Python (canonical-bf16 delta seam, trim-mean aggregation)	No Rust federation; Python-only today
Distribute	Python P2P cluster (exo-fork)	Not the Rust engine; a separate runtime

- **In-process, not sidecar.** A safety or guard model of the same architecture can run through the engine's own quantized path in-process rather than as a separate sidecar; an orchestrator would deploy it as another pod.

The honest framing is that the moat is the unified native engine, not the orchestration layer. Placement, autoscaling, and multi-node routing are precisely where Kubeflow and Ray are mature, and are the part the platform must build or borrow. The bet is that owning the runtime end-to-end removes the format-break tax that those orchestrators institutionalize.

4 hanzo Desktop: On-Device Train, Infer, and Continuous-Learn

4.1 Product Thesis and Why It Is Credible

The product thesis (aspirational) is a personal AI that runs, learns, and improves entirely on the user's machine: inference is local, personalization is local, and a continuous-learning loop adapts the

model to the user, with no cloud required and raw data never leaving the device. The same native stack that serves the cloud tier also fits a laptop-class APU, so personal AI is the full train-and-infer engine running privately, not a stripped cloud client. There is no dedicated desktop application crate in the tree today; Section 4 is product direction grounded in what the runtime can already do locally, not a shipped app.

The technical credibility rests on three present-day facts. First, the entire performance campaign ran on a single 8060S, reaching decode parity on the 8B model at Q8_0 and 2× on a 35B-A3B MoE—desktop-class silicon, not a datacenter accelerator (measured). Second, the training primitives are native and local: autograd, AdamW and SGD, VarMap checkpointing, and a working, tested trainer for group relative policy optimization (GRPO) [8] all run in-process, so on-device reinforcement-learning personalization is not theoretical (available). Third, ISQ lets the desktop load one checkpoint and pick the quantization that fits local memory—a quantized low-rank-adaptation base in the manner of QLoRA [7]—trading quality for the device's bandwidth budget per the roofline of Section 2 (available).

4.2 Continuous Learning: Design and Honest Gaps

The continuous-learning loop is designed across the engine's research notes but not yet assembled in Rust (roadmap). The intended pipeline is: (1) the model's own outputs become candidate training data; (2) a personal-data and safety gate drops or quarantines samples containing personal information; (3) a native trainer applies a low-rank-adaptation (LoRA) [5] or GRPO update locally; (4) optionally, only a bf16 LoRA *delta*—never raw data—syncs to a federation mesh in the federated-averaging tradition [9] via the canonical-bf16 seam with Byzantine-robust trimmed-mean aggregation.

The honest gaps are specific. Step (2)'s guard model can *classify* "contains personal data" but is a gate, not a redactor: it returns a verdict, not scrubbed text or character spans. A span-level redactor does not exist and must be built. In step (3), GRPO is real but the generic supervised `Trainer` is a stub. Step (4)'s federation is designed in Python and not yet ported to Rust. What makes private-by-default the genuinely strong claim is that the egress contract is structural rather than promissory: if the loop only ever emits LoRA deltas over the canonical-bf16 seam, raw user data never crosses the wire by construction. We make the structural claim precise.

Proposition 3 (Structural egress bound)**.** *Suppose the only network egress channel from the on-device loop is the delta-sync transport, and suppose that transport accepts as payload only a bf16 LoRA delta tensor Δ of fixed shape, where $\Delta = W' - W_0$ is the difference between the locally adapted weights W' and the public base weights W_0. Then no raw user datum is transmitted, and the bytes observable to the mesh are a function of Δ alone. If, in addition, calibrated noise is added so that the released delta is $\tilde{\Delta} = \Delta + \eta$ under an (ϵ, δ)-differentially-private mechanism, then the released payload satisfies the corresponding (ϵ, δ) guarantee with respect to any single training sample.*

Proof. By hypothesis the loop has exactly one egress channel, and that channel's type signature admits only the tensor Δ of fixed shape; any raw datum, having a different type, cannot be placed on the channel, so it cannot be transmitted. The mesh therefore observes a function of Δ only, establishing the first claim by construction rather than by policy. For the second claim, differential privacy is preserved under post-processing and is established by the releasing mechanism: if $\tilde{\Delta} = \Delta + \eta$ is produced by an (ϵ, δ)-DP mechanism calibrated to the sensitivity of the delta with respect to one sample, then by definition the distribution of $\tilde{\Delta}$ on neighboring datasets differing in one sample satisfies the (ϵ, δ) inequality, and any downstream aggregation is post-processing that cannot weaken the guarantee. □

Proposition 3 states what the architecture buys and what it does not. The first half—raw data never crossing the wire—is structural and holds today the moment the delta-only channel is the sole egress. The second half is conditional: the differential-privacy noise η [10] is *not* in the code and is a recommended addition. Without it, the released delta still reveals only a function of the training data, but that function is not formally private; we therefore claim structural data containment now and formal privacy only upon implementing the noise mechanism.

5 Build-versus-Buy and Private-Hosting Cost Model

This section's arithmetic is illustrative and derived only from the measured roofline; it is not a price quote (roadmap). From Theorem 1, decode cost per million tokens is

$$\$/1\text{M tokens} \;=\; \frac{C_{\text{hr}}}{3600} \cdot \frac{10^6}{R}, \tag{4}$$

where C_{hr} is hardware dollars per hour and R is decode tokens per second, itself set by quantization and bandwidth through Equation (1). Two levers, both owned by the operator, appear: choose cheaper hardware (an AMD APU versus an NVIDIA datacenter accelerator) and choose a smaller quantization (Q4_K versus Q8_0, roughly 2× throughput on the same chip).

Worked from measured throughput on the 8060S for the 8B model: at Q8_0 and 24.5 tokens per second, one million tokens is about 11.3 GPU-hours of decode by Equation (4); at Q4_K with a kernel-bound ceiling near 44 tokens per second, about 6.3 GPU-hours—provided the engine tail is closed to realize it, since a 14B-Q4_K model is engine-limited to 6–8 tokens per second today (an honest gap, not the kernel). The shape of the model dominates: a 35B-A3B MoE at Q4_K decodes about 2× faster than a dense 8B at Q8_0 while carrying far more capacity, so MoE-at-low-quant is the cheapest quality tier.

The break-even argument (Table 3): managed-serving markups make self-hosting on cheap AMD silicon attractive once volume is steady, and the engine's AMD parity is what makes that hardware viable. The desktop tier pushes marginal serving cost toward zero (the user's machine) for personal and privacy use cases—the strongest cost-plus-privacy combination, gated on building the continuous-learning loop.

The private-hosting foundation exists (available): `hanzo-server` (OpenAI HTTP plus MCP) with the paged scheduler, multi-model map, and ISQ on a pool of AMD APUs. To become a real private cloud (roadmap) requires a multi-node placement and autoscaling control loop (scale-to-zero via the existing unload/reload machinery, cold-start on first request), the federation control plane ported from Python to Rust, and multi-node KV routing. These are named, bounded gaps, not hand-waves.

6 Limitations and Future Work

We summarize the honest state of each pillar.

Serving (Section 2). Real and strong: a bandwidth-optimal unified quant core (decode at 97% of the bandwidth wall, parity with the reference on AMD), real paged multi-tenant serving, real model swap, and load-time quantization. The quantization-to-cost law of Theorem 1 is the product's economic engine. The remaining decode gap to the reference is about 1.6 tokens per second of non-matrix-vector overhead (decode attention, KV append, RoPE, normalization, sampling); decode attention is not a winnable lever at batch one because the workload under-occupies a 40-CU GPU,

Table 3: Build-versus-buy. Readiness keys: available, roadmap.

Option	**When it wins**	**Readiness**
Buy API (frontier)	Spiky/low volume, no privacy constraint	n/a
Buy managed serving	Steady volume, OSS models, no infra team	OpenAI-API compatible ⇒ drop-in self-host
Build: self-host on AMD APUs	Privacy-required, steady, cost-sensitive	parity decode + paged multi-tenant serving
Build: on-device	Per-user privacy, personalization, offline	infer available; continuous-learn roadmap

making 24.5 tokens per second the honest decode ceiling for this approach. Prefill at $\sim$0.86$\times$ needs a multi-wave matrix-core flash-attention kernel and further GEMM work; surpassing the reference requires the ROCm paged-attention keystone.

Platform (Section 3). Mixed. The runtime substrate is real and unusually unified (one engine, one tensor type, one quantization core for train and serve). The orchestration layer, the generic supervised trainer plus LoRA injection, and the federation port are roadmap. The moat is the native runtime, not the scheduler.

Desktop (Section 4). Aspirational on an available foundation. The runtime credibly fits a consumer APU (proven), and on-device training primitives exist (GRPO real). The continuous-learning loop, the span-level personal-data redactor, and the federation sync are not yet built. Private-by-default is structurally sound (Proposition 3) if delta-only egress is implemented; formal privacy additionally requires the differential-privacy noise mechanism, which is not yet in the code.

Cost model (Section 5). The arithmetic is real, derived from measured tokens per second; the multi-node private-cloud control plane is roadmap.

7 Conclusion

We have shown that once decode is recognized as memory-bound and the matrix-vector kernel is driven to the bandwidth wall—97% of achievable read bandwidth on a consumer AMD APU—the economics of serving follow from a single law: tokens per second per dollar scales inversely with the

per-parameter quantization width times the active parameter count (Theorem 1). The quantization choice, not accelerator purchase, is the primary cost lever; MoE at low quantization is the cheapest quality tier; and paged attention raises tenant concurrency to the realized-length bound rather than the worst-case bound (Proposition 2). A single quant-agnostic core makes per-format support an $O(1)$ table edit (Proposition 1), validated bit-exact across eleven sub-4-bit formats. The platform's moat is a native runtime shared by training and serving, and the desktop's privacy is structural by an egress bound (Proposition 3). We make no claim of beating everyone: the defensible claims are AMD parity as a cost lever, a genuinely unified native train-and-serve runtime, and a structurally private on-device path, each with its gaps named.

References

[1] W. Kwon, Z. Li, S. Zhuang, et al. *Efficient Memory Management for Large Language Model Serving with PagedAttention*. ACM SOSP, 2023.

[2] G. Yu, J. Jeong, G. Kim, S. Kim, and B. Chun. *Orca: A Distributed Serving System for Transformer-Based Generative Models*. USENIX OSDI, 2022.

[3] G. Gerganov et al. *ggml / llama.cpp: Tensor library and LLM inference in C/C++*. Open-source project, 2023.

[4] 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.

[5] E. Hu, Y. Shen, P. Wallis, et al. *LoRA: Low-Rank Adaptation of Large Language Models*. ICLR, 2022.

[6] N. Shazeer, A. Mirhoseini, K. Maziarz, et al. *Outrageously Large Neural Networks: The Sparsely-Gated Mixture-of-Experts Layer*. ICLR, 2017.

[7] T. Dettmers, A. Pagnoni, A. Holtzman, and L. Zettlemoyer. *QLoRA: Efficient Finetuning of Quantized LLMs*. NeurIPS, 2023.

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

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

[10] C. Dwork and A. Roth. *The Algorithmic Foundations of Differential Privacy*. Foundations and Trends in Theoretical Computer Science, 9(3–4):211–407, 2014.