UNIVERSITÉ DU QUÉBEC EN OUTAOUAIS

DÉPARTEMENT DES SCIENCES ADMINISTRATIVES

WORKING PAPER NO. 33

TITAN-MoE: Topology-Aware and Predictive Expert Orchestration for Distributed Mixture-of-Experts Inference on Apple Silicon

A Measured Systems Study on a Heterogeneous Mac Cluster

Simon-Pierre Boucher

Département des sciences administratives
Université du Québec en Outaouais

simon-pierre.boucher@uqo.ca

Gatineau – Pavillon Alexandre-Taché
283, boulevard Alexandre-Taché
Gatineau, Québec, Canada J9A 1L8

November 2026

Version 0.1

Abstract

Sparse mixture-of-experts (MoE) language models touch only a few experts per token, so a model whose weights exceed a machine's memory can still be served if the right experts are in the right place at the right time. This working paper reports the design, implementation and measurement of TITAN-MoE, an MLX-native inference runtime for Apple Silicon that manages expert residency with bounded per-layer slot tables, loads experts on demand from local NVMe, prefetches them with causal predictors, and (in its distributed mode) executes them on other Macs of a heterogeneous cluster connected by gigabit Ethernet. Every design choice is validated against mlx-lm on identical weights: with all experts resident the runtime is bit-identical in logits and within one percent of mlx-lm's decode throughput; with bounded residency the router's choices and the logits are unchanged by construction. We state five hypotheses (causal prefetch, topology-aware placement, hierarchical caching, fused Metal kernels, and adaptive orchestration on the latency–throughput–energy frontier) and test each one with a pre-registered protocol on a live inventory of a 21-node Mac cluster. This version (0.1) is a *preliminary* working paper whose results are mostly negative and fully attributed: on the small model, the best causal predictor (a per-layer logistic probe, recall@8 of 0.76, within 0.16 of the oracle hit rate) only ties with demand loading because installing a prefetched expert costs as much graph time as the read it saves (H1 not supported); kernel fusion is immaterial because decode at batch size one is launch-bound at 28 % of memory bandwidth (H4 not supported); and executing half the experts on a second Mac Studio over the gigabit Ethernet that actually connects the cluster costs 2.5 ms per layer, more than a cold-NVMe miss, so the remote tier is a capacity mechanism rather than a latency one (H2/H3 in this regime). The storage layout, by contrast, matters: an expert-contiguous record file raises cold-NVMe decoding from 8 to 36 tokens/s. Energy measurements and the adaptive scheduler are pending and labelled as such.

Keywords: mixture of experts, inference systems, Apple Silicon, expert offloading, predictive prefetching, distributed inference, energy efficiency

JEL Classification: C63, C88, L86, O33

1. Introduction

A sparse mixture-of-experts (MoE) transformer replaces the dense feed-forward block of each layer by a bank of E expert networks and a router that sends every token to $K \ll E$ of them (Rajbhandari et al., 2022). The consequence that matters for inference is arithmetic: a token activates K/E of the expert parameters, so the *working set* of a decode step is small even when the *model* is enormous. OLMoE-1B-7B activates 1.3 of its 6.9 billion parameters; in its 4-bit MLX conversion a decode step touches 128 experts of 3.4 MB each, 432 MiB out of 3.6 GB of expert weights (Section 8). Engines such as colibri (Fornaro, 2026) exploit exactly this to run models of hundreds of billions of parameters from an NVMe disk on a laptop; the price is that the disk, not the GPU, sets the speed.

Apple Silicon is an unusual target for this problem. Its unified memory removes the host-to-device copy that dominates GPU offloading systems, its NVMe storage is fast (we measure 8.5 GB/s for random 3.4 MB reads on an M5 Max), and small clusters of Macs are common in laboratories that cannot or do not want to buy datacenter accelerators. The MLX framework (Hannun et al., 2023) provides a quantized grouped matrix-multiplication kernel (`gather_qmm`) that executes a batch of expert products in one dispatch, and, since macOS 26.2, RDMA over Thunderbolt 5 between Mac Studios (Apple Machine Learning Research, 2026). What does not exist is a runtime that decides, per expert and per token, *where* an expert should reside (local memory, a peer's memory, or the local disk), *when* to move it (on demand or ahead of the router), and *which node* should execute it, using measured properties of the machines and links at hand.

This working paper describes TITAN-MoE (Topology-aware Intelligent exTert orchestrAtion for distributed MoE iNference), a runtime built to answer one question stated as five testable hypotheses (Section 1.1): can topology-aware placement, causal prefetching, hierarchical caching and specialised Metal kernels improve decode latency, prefill throughput, energy per token and feasible model capacity over strong existing baselines on Apple Silicon? We insist on three methodological commitments that shape the rest of the paper.

Placement decides speed, never semantics. The router's expert identities, gating weights and aggregation order are those of the reference implementation (mlx-lm). A resident, a prefetched and a remotely executed expert must produce the same bytes. Section 9.1 shows that TITAN-MoE with bounded residency reproduces mlx-lm's logits exactly ($\max|\Delta| = 0$ in float16) on the test prompts; cache-aware routing, which trades expert identity for hits (Skliar et al., 2025), is therefore out of scope by construction.

Measure before optimising. Every design decision in Section 5 is preceded by a microbenchmark recorded in the repository, from the cost of writing one expert into a stacked buffer (0.3 ms) to the

per-expert fetch latency of each tier (0.21 ms from the page cache, 0.43 ms from a cold NVMe, about 28 ms over the gigabit LAN that actually connects the cluster). The inventory of the cluster is read live and is reported as read, including the seven nodes that did not answer and the absence of any Thunderbolt link between the Mac Studios.

Report losses. Our first online experiment with one-layer-ahead prefetching *raised* the expert hit rate and *lowered* throughput (Section 9.4); the paper keeps that result and uses it to derive the condition under which prefetching can pay on this hardware.

1.1. Hypotheses

H1 (causal prefetch)
Predicting upcoming expert demand from information available at decision time and fetching asynchronously reduces expert-loading stalls net of predictor overhead.

H2 (topology-aware placement)
Placing experts on heterogeneous nodes according to measured compute, memory and link costs outperforms uniform or static sharding.

H3 (hierarchical caching)
Coordinating local memory, remote memory and local NVMe under a fixed memory budget improves throughput over any single second tier.

H4 (Metal kernels)
Selected fused Metal kernels improve end-to-end inference over stock MLX kernels for matching shapes and precision.

H5 (adaptive orchestration)
Adapting placement and prefetching online improves the Pareto frontier of latency, throughput and energy, not only isolated microbenchmarks.

1.2. Contributions and status of this version

Version 0.1 of this working paper contributes (i) a verified inventory and network characterisation of a 21-node Apple Silicon cluster and the audit of four reference code bases (colibri, MLX, mlx-lm, llama.cpp) with a prior-art matrix of 59 works; (ii) the TITAN-MoE single-node runtime: an expert registry over safetensors byte ranges, a `pread`-based NVMe tier, per-layer slot tables with LRU/LFU eviction, an exact execution block over `gather_qmm`, and an asynchronous prefetcher with four causal predictors; (iii) the correctness evidence (bit-identical logits, token-identical greedy decoding); (iv) a trace-driven simulator calibrated on measured constants and the first controlled single-node experiments on OLMoE-1B-7B; (v) a formal statement of the placement–prefetch

problem (Section 4) and the deterministic cost-aware heuristic that the distributed gates will use. (vi) the profiling-driven verdict on Metal kernels (Gate D), (vii) an authenticated TCP transport with a bit-exact remote expert worker and the two-node measurements on the cluster's gigabit Ethernet (Gate E), and (viii) the frequency-based placement experiment on the medium model (Gate F, reported to the extent that its logs exist). The energy Pareto study (H5) and the adaptive scheduler are pending and labelled as such.

2. Background

2.1. Sparse MoE inference

Let a decoder have layers $l \in \mathscr{L} = \{1, \ldots, L\}$. In an MoE layer the token representation $h \in \mathbb{R}^d$ is routed by $g = \mathrm{softmax}(W_r h) \in \mathbb{R}^E$; the top-$K$ entries of g select experts $S(h) \subset \mathscr{E} = \{1, \ldots, E\}$, $|S(h)| = K$, and the layer output is $\sum_{e \in S(h)} \tilde{g}_e \, f_e(h)$ where f_e is a gated feed-forward network (SwiGLU in the models we use) and $\tilde{g}$ equals g restricted to $S(h)$, renormalised in some architectures. OLMoE (Muennighoff et al., 2024) and Qwen3-MoE (Yang et al., 2025) follow this template without a shared expert; DeepSeek-style models add shared experts and grouped routing, which our registry records but this version does not execute.

During prefill the T prompt tokens are processed together and a layer typically needs most of its experts (with $T = 20$ and $K = 8$ the union of selected experts already approaches $E = 64$ in OLMoE), so prefill is compute-bound and resembles dense inference. During decode one token at a time selects K experts per layer; the LK expert weight matrices must be read from wherever they reside. If every expert is resident in unified memory, the decode step is bound by memory bandwidth and kernel launch latency (Javat and Kazakov, 2026); if some are not, the step is bound by the latency of fetching the missing ones. The cache hit rate therefore translates almost directly into throughput, which is why expert caching and prediction dominate the literature of Section 3.

2.2. Apple Silicon as an inference platform

An Apple Silicon system-on-chip shares one physical memory between CPU, GPU and Neural Engine. For offloading this removes the PCIe copy but not the fetch: a weight on disk must still be read into memory, and a weight on another machine must cross a network. The MLX framework exposes lazy arrays evaluated as a graph; inserting an expert into a resident buffer is a slice assignment that MLX performs in place when the buffer is not otherwise referenced (we measure 0.3 ms for one OLMoE expert in Section 7). The `gather_qmm` primitive multiplies a batch of rows by a batch of quantized matrices selected by index, which is exactly the operation an MoE layer needs; mlx-lm uses it for all its 50-odd MoE architectures.

Three facts about our cluster matter for every later number. First, all nodes negotiate 1000BASE-T on their Ethernet port (the switch is gigabit even though the Mac Studios have 10 GbE ports); iperf3 measures 0.945 Gbit/s between any two Studios and ping 0.5–1.0 ms round trip. Second, no Thunderbolt link is active between any two nodes, so RDMA over Thunderbolt 5 and MLX's JACCL backend are unavailable until cables are installed. Third, the development laptop is not on the cluster LAN; all distributed experiments run between LAN nodes with a Mac Studio as coordinator.

2.3. Baselines

mlx-lm (Hannun et al., 2023) is the exact baseline: same weights, same quantization, same kernels. llama.cpp (Gerganov and ggml-org contributors, 2026) runs the same base checkpoints through its Metal backend but with its own K-quant formats, so its speed is reported with a precision flag and a perplexity check rather than as a like-for-like competitor. colibri (Fornaro, 2026) streams experts from disk in C; on macOS its OLMoE engine is CPU-only (its Metal backend covers other engines), so it serves as a reference for the design space rather than a same-hardware competitor; its cluster mode, which assigns experts to worker Macs by a deterministic weighted hash, is the closest existing analogue of our static placement.

3. Related Work

The prior-art table in the reproducibility appendix (and `research/prior_art.csv` in the repository) lists 59 works verified on their arXiv, publisher or repository pages on 2026-11-01; here we summarise what they establish and where TITAN-MoE differs.

Expert offloading and SSD streaming. Eliseev and Mazur (2023) introduced per-expert offloading with an LRU cache for Mixtral on consumer GPUs; MoE-Infinity (Xue et al., 2024) built sparsity-aware expert caches from activation traces; Fiddler (Kamahori et al., 2025), HybriMoE (Zhong et al., 2025) and DAOP (Zhang et al., 2025a) execute cold experts on the CPU rather than moving them; MoE-Lightning (Cao et al., 2025), fMoE (Yu et al., 2025), MoE-Gen (Xu et al., 2025) and Klotski (Fang et al., 2025a) optimise throughput under memory constraints with pipelined offloading. SSD-tier streaming has been demonstrated at trillion-parameter scale on x86 (Liang et al., 2026a); FlashMoE (Kim et al., 2026) learns a cache-replacement policy that beats LRU and LFU; colibri (Fornaro, 2026) combines a per-layer LRU, a pinned hot set learned across sessions and a one-layer router lookahead in a pure C engine. The principle that placement must not change the model's semantics is stated explicitly by colibri and adopted here.

Expert prediction. Pre-gated MoE (Hwang et al., 2024) and Read-ME (Cai et al., 2024) modify the architecture so that layer l's router is computed from layer $l-1$'s representation; Fate (Fang et al., 2025b), ProMoE (Song et al., 2024), ExpertFlow (He et al., 2026; Shen et al., 2025), LayerScope (Yu et al., 2026), DuoServe (Zhang et al., 2025b), SpecPrefetch (Kong et al., 2026) and SeqMoE (Wang et al., 2026) learn cross-layer or multi-step predictors on GPUs and NPUs; Liang et al. (2026b) show that routing locality varies across models and that a cache of about twice the active expert count is often enough. TITAN-MoE's predictors are deliberately simple (transition counts at depth d, a linear probe on router log-probabilities, a previous-token rank predictor) so that their cost can be measured against their benefit; the novelty we test is not a predictor but the *condition*, derived in Section 4, under which any predictor can pay on a given tier of a given machine.

Distributed inference on consumer hardware. Petals (Borzunov et al., 2023), exo (exo labs, 2025), prima.cpp (Li et al., 2025) and llama.cpp's RPC backend shard layers or tensors across devices; MLX's distributed backends (ring, MPI, JACCL) and Apple's own 4×M5 Ultra results (Apple Machine Learning Research, 2026) shard whole layers over a Thunderbolt mesh. Expert-level placement across nodes exists for datacenter GPUs (ExFlow (Yao et al., 2024), Lina (Li et al., 2023), DeepEP (Zhao et al., 2025)) and, in a simple form, in colibri's cluster mode, where experts are hashed to workers with weights proportional to disk speed. None of these models node heterogeneity, link latency or replication; H2 targets that gap on the only topology our cluster offers, gigabit Ethernet.

Energy and characterisation on Apple Silicon. GreenBench (Kannan et al., 2026) and Silicon Showdown (Javat and Kazakov, 2026) measure joules per token with `powermetrics` for dense models; Alfarizy et al. (2026) find that MoE energy on edge devices tracks total rather than active parameters; NPUMoE (Benazir and Lin, 2026) runs MoE on the Neural Engine with all experts resident. No prior work reports an energy–throughput frontier for *offloaded* or *distributed* MoE on Macs, which H5 addresses.

4. Mathematical Formulation

4.1. Objects

Let $\mathscr{E}$ be the set of (layer, expert) pairs, $|\mathscr{E}| = LE$, each of size b bytes (3.375 MiB for OLMoE at 4 bits; equal within an architecture). Let $\mathscr{N} = \{0, 1, \ldots, N\}$ be the nodes, node 0 being the coordinator that runs attention, the router and the KV cache, and let tier $\mathscr{T}_n = \{\text{ram}, \text{ssd}\}$ be the storage tiers of node n. Tokens are steps $t = 1, 2, \ldots$; the router of layer l at step t selects $S_{t,l} \subset \mathscr{E}_l$,

$|S_{t,l}| = K$. Decision variables are binary and indexed by step:

$$x_{e,n,t} \in \{0,1\} \quad \text{expert } e \text{ is resident in the memory of node } n \text{ at step } t, \tag{1}$$

$$z_{e,n,t} \in \{0,1\} \quad \text{expert } e \text{ is } \textit{executed} \text{ on node } n \text{ at step } t, \tag{2}$$

$$p_{e,n,t} \in \{0,1\} \quad \text{a prefetch of } e \text{ into node } n \text{ is issued during step } t. \tag{3}$$

Every expert exists on the local NVMe of each node that holds a copy of the checkpoint (the SSD tier is unbounded for our purposes and costs only time), so residency refers to memory.

4.2. Constraints

$$\sum_{e \in \mathscr{E}} b\, x_{e,n,t} \le M_n \qquad \forall n,t \qquad \text{(memory budget of node } n\text{)} \tag{4}$$

$$\sum_{n \in \mathscr{N}} z_{e,n,t} = \mathbb{1}[e \in S_{t,l(e)}] \qquad \forall e,t \qquad \text{(every selected expert executed exactly once)} \tag{5}$$

$$z_{e,n,t} \le x_{e,n,t} + m_{e,n,t} \qquad \forall e,n,t \qquad \text{(execution needs residency or a demand miss } m\text{)} \tag{6}$$

$$x_{e,n,t+1} \le x_{e,n,t} + p_{e,n,t} + m_{e,n,t} \qquad \forall e,n,t \qquad \text{(residency only grows by prefetch or miss)} \tag{7}$$

where $m_{e,n,t} \in \{0,1\}$ is the demand-miss indicator, itself a consequence of the router and of x. Constraint (5) encodes the correctness invariant: the set of executed experts is the router's set, whatever the placement.

4.3. Costs

The per-step latency decomposes along the layer sequence, because layer l cannot start before layer $l-1$'s output exists:

$$\Lambda_t \;=\; \sum_{l=1}^{L} \Big[\underbrace{\tau_0^{\text{attn}}}_{\text{attention, node 0}} + \max_{n \in \mathscr{N}} \Big(\underbrace{c_n \cdot \textstyle\sum_{e \in S_{t,l}} z_{e,n,t}}_{\text{expert compute on } n} + \underbrace{\rho_{0n}\, \mathbb{1}[\exists e : z_{e,n,t} = 1, n \neq 0]}_{\text{activation round trip}} \Big) + \underbrace{\sigma_{t,l}}_{\text{stall}} \Big], \tag{8}$$

with c_n the measured per-expert compute time of node n, ρ_{0n} the measured round-trip time of a d-dimensional activation and its result between node 0 and n (on our LAN, 0.5–1.0 ms), and the stall

$$\sigma_{t,l} \;=\; \max\Big(0,\; \max_{e \in S_{t,l},\, m_{e,n,t}=1} \big(\lambda_{\text{tier}(e)} - \eta_{e,t}\big)\Big), \tag{9}$$

where λ_{tier} is the fetch latency of the tier the expert comes from ($\lambda_{\text{pc}} = 0.21$ ms page cache, $\lambda_{\text{ssd}} = 0.43$ ms cold NVMe, $\lambda_{\text{lan}} \approx 28$ ms for 3.4 MB over 0.945 Gbit/s, all measured on our

hardware) and $\eta_{e,t}$ is the lead time between the moment a prefetch of e was issued and the moment e is needed ($\eta = 0$ for a demand miss). Energy per step is $\Phi_t = \sum_n P_n \cdot \Lambda_t^{(n)}$ with P_n the package power of node n measured by `powermetrics` during the run, and communication cost $\Gamma_t = \sum_{n \neq 0} \beta_{0n}$ (activation bytes + weight bytes moved).

4.4. Problem

$$\min_{x,z,p} \mathbb{E}\Big[\sum_t \Lambda_t + \lambda\, \Phi_t + \mu\, \Gamma_t\Big] \quad \text{s.t. (4)–(7)}, \tag{10}$$

where the expectation is over the router's sequence $\{S_{t,l}\}$, which is revealed online: at step t, $S_{t,l}$ becomes known only after layer $l-1$'s output is computed. The problem is a stochastic online integer program; its offline relaxation with known S is the paging problem with multiple caches and non-uniform fetch costs, for which Belady's furthest-in-future rule is optimal per cache and which our simulator's *oracle* predictor approximates.

4.5. When can a prefetch pay?

Consider a predictor issuing, when layer $l-d$'s routing is known at step t, a set $\hat{S}_{t,l}$ of m experts for layer l, with recall $r = \mathbb{E}|\hat{S} \cap S|/K$ and precision $q = \mathbb{E}|\hat{S} \cap S|/m$. With per-layer decode time τ_ℓ the lead time is $\eta = d\,\tau_\ell$. Per layer, the expected stall without prefetch is $\sigma_0 = \lceil K(1-h)/w \rceil\, \lambda$ for hit rate h and w parallel loaders, and with prefetch

$$\sigma_d \;\approx\; \lceil K(1-h)(1-r')/w \rceil\, \lambda \;+\; \lceil K(1-h)\, r'/w \rceil\, \max(0, \lambda - d\,\tau_\ell), \tag{11}$$

where r' is the recall among the experts that would have missed. The prefetch also adds $m(1-q)$ useless fetches whose bandwidth and install cost κ each is paid on the critical path when the loader is shared with demand misses. A prefetch can only reduce latency if

$$r' \min(\lambda, d\,\tau_\ell) \;>\; m(1-q)\,\kappa \;/\; K(1-h), \tag{12}$$

and it can *remove* the stall only if $d\,\tau_\ell \geq \lambda$. On the laptop, $\tau_\ell = 0.128$ ms (489 tok/s over 16 layers), so a page-cache fetch ($\lambda = 0.21$ ms) needs $d \geq 2$ and a cold-NVMe fetch ($\lambda = 0.43$ ms) needs $d \geq 4$; a LAN fetch ($\lambda \approx 28$ ms) can never be hidden within a token on this model, which is why weights are not moved over the LAN during decode in TITAN-MoE and remote experts are instead executed where they reside. The one-layer online experiment of Section 9.4 violated (12) ($d = 1$, $q \approx 0.5$) and lost throughput, as predicted; the depth-d experiments test the condition directly.

Proposition 4.1 (Demand loading is $\mathcal{O}(1)$-competitive per layer only when the table exceeds the per-step working set)**.** *With per-layer tables of capacity $C \geq K$, LRU eviction, and no prefetch, the*

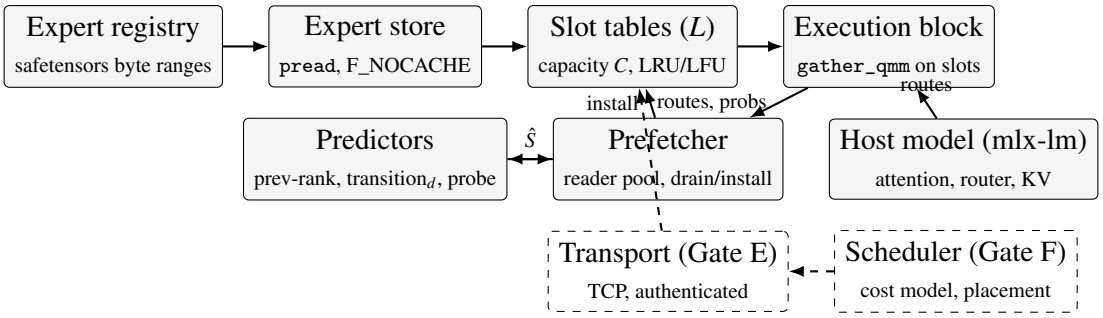


Figure 1. Components of TITAN-MoE on one node. Solid boxes are implemented and tested in version 0.1; dashed boxes are the distributed components whose status is given in Section 12.

number of demand misses at a decode step is at most K and at least $K - |S_{t,l} \cap S_{t-1,l}|$ when $C = K$; for $C \geq |\bigcup_{s \leq \Delta} S_{t-s,l}|$ the misses at step t are exactly the experts not used in the last Δ steps at that layer.

Proof. With $C = K$ the table after step $t-1$ holds exactly $S_{t-1,l}$, so every expert of $S_{t,l} \setminus S_{t-1,l}$ misses and no other does; misses are bounded by K because $|S_{t,l}| = K$. For the second claim, LRU keeps the most recently used C distinct experts; if C is at least the number of distinct experts used in the last Δ steps, all of them are resident, so a miss at step t is precisely an expert unused during those Δ steps. □

The proposition is elementary but useful: it says that the hit rate of LRU at capacity C is the fraction of a step's experts that recur within the C-recent window of that layer, a quantity we measure directly from traces (Section 9.3) and which, for OLMoE, is 0.57 at $C = 16$ and 0.80 at $C = 32$.

5. Architecture

Figure 1 shows the components of TITAN-MoE. The host model (embeddings, attention, norms, routers, KV cache) is mlx-lm's own module tree, loaded lazily so that the stacked expert tensors are never materialised; each layer's expert block is replaced by a TITAN-MoE block that owns a *slot table*. Everything else is orchestration around that table.

Expert registry. The registry parses `config.json` and the safetensors headers of a checkpoint directory and records, for every (layer, expert) and every matrix (gate, up, down) and component (packed weight, scales, biases), the absolute byte offset and length in the file, together with the routing semantics (softmax-then-top-k, renormalisation flag, shared experts) and the quantization (bits, group size). It accepts only verified formats (OLMoE and Qwen3-MoE in this version) and refuses the others rather than guessing.

Expert store (SSD tier). An expert is read with one `pread` per tensor at the registry's offsets, wrapped into MLX arrays without further copies on the Python side; bytes, read count and read time are accounted. The file descriptor can be opened with `F_NOCACHE` so that reads bypass the unified buffer cache, which is how the cold-NVMe tier is measured (after `purge`); without it the same code measures the page-cache tier.

Slot tables. Each layer owns C slots in stacked MLX buffers of the exact shapes mlx-lm uses for its stacked experts, so that the same `gather_qmm` kernel can run on them. A dictionary maps resident expert ids to slots; eviction is LRU by default (LFU available). Installing an expert is a slice assignment into the buffers, which MLX performs in place. Misses are read as a batch per layer and installed lazily so that the layer's kernel consumes the new buffers in the same graph: there is no per-expert synchronisation. The table records hits, misses, evictions, loads and stall time.

Execution block. Given the router's indices, the block materialises them on the host (the one synchronisation a demand-loading design cannot avoid), ensures residency, remaps expert ids to slot ids and runs `gather_qmm` for the up, gate and down projections with mlx-lm's compiled SwiGLU. When a layer's table holds every expert (the all-resident control) a device-side slot map is used and no synchronisation occurs, which recovers mlx-lm's throughput. Prefill rows whose union of experts exceeds C are chunked greedily by rows, which is exact because rows are independent.

Predictors and prefetcher. A predictor proposes up to m experts for a layer from information available at the decision point: the previous token's routing and router probabilities (token horizon), or the routing of layer $l-d$ at the current token (layer horizon, depth d). The prefetcher submits reads to a small thread pool, installs completed fetches when the target layer is reached, waits for fetches that are needed but still in flight (counted as *late*, with the residual wait charged as stall), and never changes which experts run.

Hierarchical memory and distributed execution. The remote tier does not move weights during decode (Section 4.5): a remote node that holds an expert executes it on the coordinator's activations and returns the d-dimensional result, exactly as colibri's cluster mode and as expert parallelism do. The transport (Gate E) is a persistent authenticated TCP connection per worker with a per-layer batched request, timeouts and local fallback; placement (Gate F) follows the cost model of Section 4 with measured c_n, M_n, ρ_{0n} and λ per node.

Algorithm 1 Decode step of one MoE layer in TITAN-MoE (single node)

Require: hidden rows X, router indices I ($R \times K$), slot table $\mathscr{S}_l$, prefetcher $\mathscr{P}$, depth d

if $\mathscr{S}_l$ holds every expert **then**
  return `gather_qmm`$(X, \mathscr{S}_l, \text{slotmap}[I])$ ▷ no host sync
end if
$U \leftarrow$ unique experts of I (host) ▷ the unavoidable synchronisation
$\mathscr{P}$.DRAIN(l, U) ▷ install finished prefetches; wait for in-flight experts in U (late)
$\hat{S} \leftarrow \mathscr{P}$.PREDICT$(l+d)$; $\mathscr{P}$.ISSUE$(l+d, \hat{S})$ ▷ before this layer's compute
$M \leftarrow U \setminus \text{resident}(\mathscr{S}_l)$; read M from the store as a batch; install into slots (lazy)
$J \leftarrow \text{slot}(I)$
return `gather_qmm`$(X, \mathscr{S}_l, J)$ ▷ same kernel, same bytes as mlx-lm

Algorithm 2 Deterministic cost-aware placement (Gate F heuristic)

Require: expert popularity π_e from fit traces, node costs (c_n, M_n, ρ_{0n}), expert size b

sort experts by π_e descending; $\text{free}_n \leftarrow M_n$
for each expert e in that order **do**
  $n^\star \leftarrow \arg\min_{n:\ \text{free}_n \geq b} \left(c_n + \mathbb{1}[n \neq 0]\, \rho_{0n}\right) \cdot \pi_e$ ▷ expected per-step cost
  place e on $n^\star$; $\text{free}_{n^\star} \leftarrow \text{free}_{n^\star} - b$
end for
replicate the ρ most popular experts on node 0 if free_0 allows (hot set)
experts placed nowhere fall back to node 0's slot table (SSD tier, demand loading)

6. Algorithms

Algorithm 1 is the per-layer path of Section 5; the prefetch for layer $l+d$ is issued *before* the current layer's kernels are dispatched so that the reader threads overlap the GPU work of the next d layers. Algorithm 2 is the first placement heuristic: greedy by popularity, cost-aware by node, with replication of a hot set on the coordinator. It runs in $\mathscr{O}(LE \log LE + LEN)$ time, negligible against inference, and will be the static baseline against which the adaptive scheduler (online re-estimation of π and of the node costs) is compared. Predictor fitting is $\mathscr{O}(TLK^2)$ for transition counts and one logistic regression per layer for the probe.

7. Implementation

TITAN-MoE is about 1 100 lines of Python over MLX 0.32.4 and mlx-lm 0.32.2 (Python 3.12), organised as `titan/{registry, store, slots, moe, prefetch, predictors, runtime}.py`, with benchmark drivers in `benchmarks/scripts/` and tests in `tests/`. Three implementation findings determined the design.

In-place slot updates. Writing the nine tensors of one OLMoE expert (3.4 MB) into stacked buffers of 64 slots (216 MB per matrix) costs 0.305 ms, which shows that MLX reuses the buffer rather than copying it; a per-expert `quantized_matmul` loop costs 0.43 ms per layer for one token against 0.26 ms for `gather_qmm`, and restacking eight experts per call 0.50 ms. The slot table with `gather_qmm` is therefore both the fastest and the most faithful option (`benchmarks/raw/gateB/design/`).

No per-miss synchronisation. The first version evaluated the table after every miss; removing that synchronisation and batching the reads of a layer's misses raised decode throughput at 16 slots from 28 to 51 tok/s (general prompts) and from 41 to 73 tok/s (code), with the stall halved (`benchmarks/raw/gateC/prefetch_online_laptop/`). The per-miss cost on the page-cache tier is now 0.21 ms: 0.13 ms of `pread`, 0.06 ms of array creation and the remainder in the lazy install.

Activation parity. An unfused float16 $\text{up} \cdot \sigma(\text{gate}) \cdot \text{gate}$ gave $\max|\Delta\text{logit}| = 0.0195$ against mlx-lm; using mlx-lm's compiled SwiGLU gives 0.0. Numerical parity in float16 is a property of the exact kernel sequence, not only of the weights, and the test suite pins it.

Threads and dtypes. Two defects were found by the exactness tests rather than by inspection. MLX arrays created in reader threads cannot be evaluated in another thread (streams are thread-local), so reader threads now return NumPy views and the consuming thread creates the MLX arrays. And the first transport cast activations to float16 on the wire, which is exact for OLMoE (float16) but not for Qwen3 (bfloat16): the first two-node sweep on Qwen3 showed $\max|\Delta\text{logit}| = 0.65$ and was discarded; activations and results now travel in the model's dtype and the Qwen3 two-node path is bit-identical again.

Reproducibility. Every benchmark writes JSON lines with the command, git commit, host, chip, memory, macOS, library versions, seed and timestamps; raw logs are append-only under `benchmarks/raw/` and tables and figures are generated from them by scripts. Model revisions are pinned by commit hash in `benchmarks/configs/models.yaml`; weights are never stored in the repository. A node is prepared with `scripts/cluster_setup.sh` (rsync of the code, pinned virtual environment, import check), and the sanitized inventory `artifacts/environment.json` is generated from raw probes that are kept out of version control.

Table 1. Machines used or available. GPU = GPU cores. Dev. laptop is not on the cluster LAN.

Alias	Chip	RAM (GB)	GPU	macOS	Role in this version
laptop	Apple M5 Max	48	40	27.0	single-node experiments (Gates B–C); NVMe 8.5
M3U96a	Apple M3 Ultra	96	60	26.6	coordinator; Qwen3-30B-A3B baselines
M3U96b	Apple M3 Ultra	96	60	26.4	worker (Gate E)
M2U64	Apple M2 Ultra	64	60	26.5	worker (Gate E)
M4M64a/b/c	Apple M4 Max	64	40	26.4–26.5	workers (Gate F)
M5M36	Apple M5 Max	36	32	27.0	heterogeneous worker (Gate F)
m4mm, m4mn	Apple M4 Pro	48	16	26.5	heterogeneous workers (Gate F)
others (12)	M1–M6, 8–32 GB				inventoried, unused

8. Experimental Setup

8.1. Hardware (live inventory, 2026-11-01)

Table 1 lists the machines whose specifications were read live (`sysctl`, `system_profiler`) on the day of the experiments; the sanitized inventory is `artifacts/environment.json`. Eight aliases of the cluster's nominal 29 did not answer SSH within 15 s and are excluded rather than assumed. All nodes negotiate 1000BASE-T; iperf3 measures 0.945–0.947 Gbit/s and ping 0.5–1.0 ms between the Studios; no Thunderbolt link is active between any two nodes.

8.2. Models

- **Small**: OLMoE-1B-7B-0125-Instruct (Muennighoff et al., 2024), mlx-community 4-bit conversion (affine, group 64), revision `e2384419`, Apache-2.0: 16 layers, 64 experts, top-8, no shared expert; 3.375 MiB per expert, 3.62 GB of experts, 0.27 GB dense; 128 experts (432 MiB) touched per decode token.
- **Medium**: Qwen3-30B-A3B-Instruct-2507 (Yang et al., 2025), mlx-community 4-bit (routers 8-bit), revision `e9675aa3`, Apache-2.0: 48 layers, 128 experts, top-8, renormalised gates; 2.53 MiB per expert, 16.3 GB of experts, 0.87 GB dense; 384 experts (0.97 GiB) per decode token. Stacked `switch_mlp` layout (per-expert byte ranges are slices of the first axis).
- **Large** (gpt-oss-120b, 62 GB): not downloaded in this version (authorization pending).

8.3. Workloads and metrics

Prompts are synthetic instructions written for this study in six families (general, code, reasoning, multilingual, long-context, domain drift), split into *fit* prompts (used to fit predictors) and *held-out* prompts (never used to fit anything); v1 has 18 prompts (Gate B), v2 has 65 (Gate C). Decoding is greedy. Metrics follow `research/protocol.md`: prefill and decode tok/s as reported by mlx-

lm's generator plus wall-clock inter-token latencies (p50/p95/p99) measured in the driver, time to first token, expert hit rate, misses, stall time, bytes read, peak memory; energy (`powermetrics`) in the distributed and Pareto experiments. Every run records its command line, git commit and environment; order is shuffled with a fixed seed; the first run of each configuration is a discarded warm-up; repeats are ≥ 2 in the smoke tests reported here and ≥ 5 in the final sweeps.

8.4. Tiers and constants used by the simulator

Measured on the laptop: page-cache fetch of one OLMoE expert 0.21 ms (batched demand path), cold NVMe 0.43 ms (F_NOCACHE after `purge`, single reader), 1 GbE transfer ≈ 28 ms; per-layer decode compute $\tau_\ell = 0.128$ ms (489 tok/s). The trace-driven simulator (`benchmarks/scripts/simulate_cache.` is exact for hits and misses and uses these constants for stall estimates.

9. Results

Every number below has a raw log in `benchmarks/raw/`; the path is given with each result. Results marked PENDING have a protocol but no log yet in this version.

9.1. Correctness

On four prompts (general, code, reasoning, French) TITAN-MoE reproduces mlx-lm's logits with $\max|\Delta| = 0.0$ (float16) at capacities 64, 32, 16 and 8 slots per layer, with identical argmax at every position and identical greedy continuations of 32 tokens at 16 slots (`benchmarks/raw/tests/numerical_*.json` `greedy_*.json`, `pytest_gateB_*.txt`, 7 tests passed). The registry's byte ranges read with `pread` are byte-equal to `mx.load` for sampled experts of both the per-expert layout (OLMoE) and the stacked layout (Qwen3-30B-A3B, on M3U96a). Qwen3-30B-A3B logit parity: PENDING (test launched on M3U96a; result to be inserted from its log).

9.2. All-resident control and demand loading

Table 2 summarises the laptop capacity sweep (64-token greedy generations, general and code prompts, page-cached checkpoint, `benchmarks/raw/gateB/capacity_sweep_laptop/sweep.jsonl` and `benchmarks/raw/gateC/prefetch_online_laptop/smoke_lazyinstall.jsonl`).

Two observations. The all-resident control matches mlx-lm within one percent only when every layer's table is complete; otherwise each layer pays a host synchronisation, which alone costs 2.4× in throughput on this model and is the floor of any demand-loading design, including colibri's. And the demand-loading rows are the H1 *control*: at 16 slots the model does 53 misses per token (Prop. 4.1), i.e. about 11 ms of fetch per 2 ms of compute.

Table 2. OLMoE-1B-7B 4-bit on the M5 Max laptop: decode throughput vs residency (2 repeats after warm-up; ranges over prompts).

Backend	Residency	decode tok/s	hit rate	stall / 64 tokens
mlx-lm	all	491–496	–	–
TITAN-MoE	64 slots, preloaded	489	1.00	0
TITAN-MoE	64 slots, not preloaded (sync path)	201–203	1.00	0
TITAN-MoE	32 slots, LRU, demand (v1 installs)	56–80	0.73–0.81	0.7–1.0 s
TITAN-MoE	16 slots, LRU, demand (v1 installs)	36–45	0.48–0.60	1.7–2.0 s
TITAN-MoE	16 slots, LRU, demand (lazy installs)	51–73	0.48–0.67	0.6–0.9 s
TITAN-MoE	16 slots, LRU + transition prefetch (d=1, m=8)	35–56	0.63–0.73	0.6–1.1 s

9.3. Routing locality

From the v1 traces (18 prompts, 3 686 tokens), per-layer LRU hit rates are 0.57 at 16 slots, 0.80 at 32 and 0.93 at 48 (LFU within 0.01), nearly identical on fit and held-out prompts (`benchmarks/processed/gateC/si` a one-layer-ahead transition predictor has recall@8 of 0.50 on held-out prompts. On the v2 traces (65 prompts, 6 families, `benchmarks/processed/gateC/`) the transition predictor keeps almost all of its accuracy when it looks further ahead (recall@8 0.534 / 0.525 / 0.517 at depth 1 / 2 / 3), and a per-layer logistic probe on the previous layer's router log-probabilities reaches recall@8 of **0.761** held-out (0.779 fit) with prefetch precision 0.74–0.83, lifting the simulated hit rate at 16 slots from 0.58 (LRU) to 0.81 (oracle 0.94). A first version of the probe scored 0.16 because of a label-alignment bug across traces; the fix is recorded in the log and the buggy rows are excluded from Table 3 and Figure 2. The rank-based previous-token predictor hurts at every capacity (precision 0.18–0.47, it evicts useful experts).

9.4. Prefetching (H1)

Page-cache tier, one-layer horizon. The online prefetcher raised the hit rate from 0.48 to 0.63 (general) and from 0.67 to 0.73 (code) but reduced throughput from 51 to 35 and from 73 to 56 tok/s (Table 2): the lead time $d\,\tau_\ell = 0.13$ ms is shorter than the 0.21 ms fetch, so predicted experts arrive late, and the 50 % of wrong predictions add traffic. This is the regime excluded by condition (12).

Cold-NVMe tier. The SSD tier itself had to be engineered before the question could be asked: in the published OLMoE checkpoint the nine tensors of one expert are scattered over a 1.5 GB span of the file (hash order of keys), so each miss cost nine random reads and 16-slot decoding ran at 8–10 tok/s; parallel demand reads raised it to 18 and the expert-contiguous layout of Section 5 (one 4 KiB-aligned record per expert, one `pread`) to 36 tok/s (`benchmarks/raw/gateC/cold_nvme_laptop/`). Table 5 then compares predictors on that tier with the page cache purged before every configuration:

Table 3. Trace-driven simulation (OLMoE, per-layer LRU tables, cold per prompt): hit rate and modelled stall per token by predictor, split and capacity C.

Predictor	split	$C = 16$		$C = 24$		$C = 32$		$C = 48$	
		hit	stall	hit	stall	hit	stall	hit	stall
none	fit	0.577	23.3	0.706	16.2	0.804	10.8	0.931	3.8
none	heldout	0.581	23.1	0.706	16.2	0.806	10.7	0.933	3.7
prev_rank	fit	0.545	25.0	0.665	18.4	0.770	12.7	0.919	4.5
prev_rank	heldout	0.556	24.4	0.670	18.1	0.771	12.6	0.920	4.4
transition	fit	0.680	17.3	0.770	12.5	0.839	8.8	0.940	3.3
transition	heldout	0.686	17.0	0.771	12.5	0.840	8.7	0.942	3.2
transition2	fit	0.663	18.0	0.758	13.0	0.832	9.1	0.938	3.4
transition2	heldout	0.672	17.6	0.761	12.9	0.833	9.0	0.940	3.3
transition3	fit	0.651	18.5	0.749	13.5	0.826	9.4	0.937	3.4
transition3	heldout	0.660	18.1	0.752	13.3	0.828	9.3	0.939	3.3
oracle	fit	0.938	3.0	0.959	2.0	0.973	1.4	0.991	0.5
oracle	heldout	0.940	2.9	0.959	2.0	0.972	1.4	0.991	0.5
probe	fit	0.813	9.8	0.868	7.0	0.908	4.9	0.965	1.9
probe	heldout	0.811	10.0	0.864	7.3	0.904	5.1	0.965	1.9

Source: `benchmarks/processed/gateC/sim_olmoe_v2_ssd043_m8.json`; t_read 0.43 ms, t_layer 0.128 ms, 1 loader(s); hit = per-layer hit rate, stall in ms per token (modelled).

at 16 slots the depth-2 transition predictor cuts the stall by a third (0.77 vs 1.15 s per 64 tokens) and ties with demand loading (35.0 vs 36.4 tok/s); every other predictor, including depth 1 and 3 and the rank-based one, loses; at 32 slots all of them lose to demand loading (41–42 vs 50.6 tok/s).

Attribution. Figure 3 and Table 4 split the host time per token into the wait for the GPU graph (which includes the slice writes that install experts), prefetch bookkeeping and late waits, and demand reads. The probe raises the hit rate to 0.78 and cuts demand reads from 19.9 to 12.8 ms per token, but the GPU wait grows from 14.7 to 19.7 ms and late waits add 3 ms: each installed expert costs nine lazy slice writes (about 0.13 ms of graph work on the critical path), and the prefetcher installs more experts than it saves reads for. Confidence thresholds and deeper horizons change the mix, not the sum. **H1 is therefore not supported on this model and tier**: the best predictor ties with no prefetch. The attribution is positive knowledge: the bottleneck is the install path, not prediction accuracy (the probe is already within 0.16 of the oracle's hit rate at 16 slots), so a single-copy install (one record per expert written in one operation) is the change that would make the probe's 7 ms of saved reads count, and it is the first item of version 0.2.

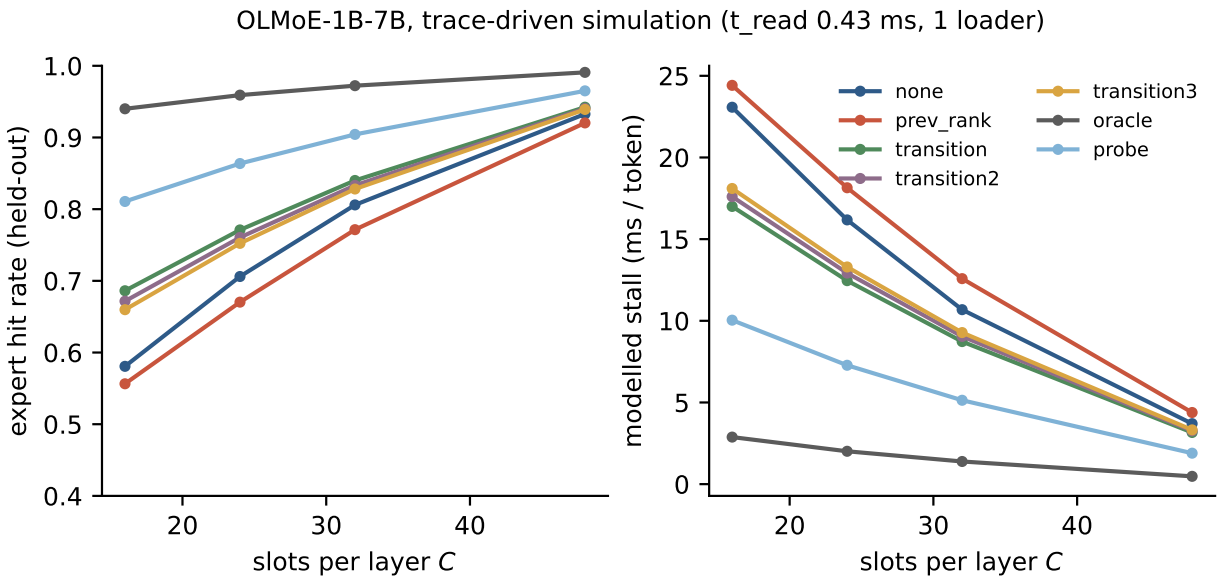


Figure 2. Trace-driven simulation on the v2 held-out prompts: per-layer hit rate (left) and modelled stall per token (right) by slot capacity and predictor. Generated by `scripts/make_figures.py` from `benchmarks/processed/gateC/`.

9.5. Medium model (Qwen3-30B-A3B)

On M3U96a (M3 Ultra, 96 GB) the stacked-layout registry reads byte-equal slices, logits are bit-identical to mlx-lm at 128, 64, 32 and 16 slots per layer and greedy decoding is token-identical at 32 slots (`benchmarks/raw/tests/m3u96a/`). Table 6 gives the baselines: mlx-lm 121.4 tok/s and TITAN-MoE all-resident 121.7 tok/s (medians of 3 repeats), i.e. parity on a model whose decode step touches 0.97 GiB of expert weights.

9.6. Two-node execution over gigabit Ethernet (H2, H3: first evidence)

With M3U96a as coordinator and M3U96b as worker (OLMoE, 32 slots, page-cached checkpoint on both, held-out general and code prompts, `benchmarks/raw/gateE/two_node_m3u96a_m3u96b.jsonl`), executing the odd-numbered experts of every layer on the worker keeps the logits bit-identical to mlx-lm but reduces decode throughput from 70 / 87 tok/s (local control) to 15 / 15 tok/s (Figure 5). The remote wait is 2.5 s per 64 tokens, i.e. about 2.5 ms per layer: a 0.5–0.65 ms TCP round trip plus the worker's array creation, kernel dispatch, evaluation and serialisation, paid 16 times per token. This is larger than a cold-NVMe demand miss at 16 slots (about 1.2 ms per layer) and twenty times the local compute per layer. On this topology the remote tier is therefore a capacity mechanism, not a latency one, exactly as Section 4.5 anticipates; H2 can only be tested in the regime where a node cannot hold the experts it needs locally.

The round trip was measured directly (`benchmarks/raw/gateE/remote_cost_breakdown_m3u96a_m3u96b`

Table 4. Predictor comparison with host-time attribution, cold NVMe, 16 slots, held-out general prompts (4 runs per row); see Figure 3 for the breakdown.

Backend	cfg	n	decode tok/s (median)	p95 ITL ms	hit	stall s	TTFT ms
titan	C=16, none, m=8, cold	4	33.0	39.2	0.492	1.24	361
titan	C=16, probe, m=16, cold	4	25.3	53.1	0.847	1.13	478
titan	C=16, probe, m=8, cold	4	31.8	39.3	0.775	0.89	382
titan	C=16, probe2, m=8, cold	4	31.3	40.3	0.740	0.98	415
titan	C=16, probe3, m=8, cold	4	30.7	41.2	0.720	1.03	467
titan	C=16, transition2, m=8, cold	4	27.1	45.5	0.638	1.20	376
titan	C=16, transition2t50, m=16, cold	4	28.6	42.7	0.654	1.07	346
titan	C=16, transition2t50, m=8, cold	4	27.9	42.9	0.630	1.13	358
titan	C=16, transition2t70, m=8, cold	4	27.8	43.4	0.583	1.19	365

Sources: `benchmarks/raw/gateC/cold_nvme_laptop/attribution.jsonl`.

application-level ping 0.74–1.17 ms; one layer's request (one row, eight pairs) 2.59 ms; eight rows with 64 pairs 7.3 ms; 32 rows with 256 pairs 19.1 ms, of which about 9 ms is the 1 MB reply on the wire. The worker therefore adds 1.2–1.7 ms per call to the network latency, which bounds what a faster transport could recover without also changing the worker's execution path.

9.7. Placement on the medium model (Gate F, H2)

The same two Studios ran Qwen3-30B-A3B with the coordinator's residency bounded to 32 or 64 of the 128 experts per layer and its page cache purged before every configuration, so that a locally executed miss is a cold-NVMe read (`benchmarks/raw/gateF/`, 16 runs per cell, 64 tokens, held-out general and code prompts, logits bit-identical to mlx-lm in every cell). Three placements were compared: everything local, the odd-numbered half of every layer's experts on the worker (static), and the least popular half on the worker with popularity estimated from the fit traces (the frequency-based placement of Algorithm 2 with one worker). Table 7 and Figure 6 give the result: local demand loading decodes at 8.8 tok/s with 32 slots (66 ms of stall per token) and 16.3 with 64; frequency-based placement beats static placement at both budgets (5.1 vs 4.2 and 6.0 vs 5.2 tok/s, 35 % fewer remote pairs, remote wait 67 vs 112 ms per token), but both lose to the local cold tier (-3.6 [-4.4, -2.9] and -10.2 [-13.6, -8.8] tok/s, Table 10). The ordering is the one the cost model predicts from the measured constants: a remote layer costs 1.5–3 ms over gigabit Ethernet, a cold miss on this NVMe about 0.9 ms per expert in parallel, and the medium model has 48 layers. **H2 is supported in its weak form** (cost-aware placement beats static placement) and **not in its strong form** (remote execution does not beat local NVMe) on this topology.

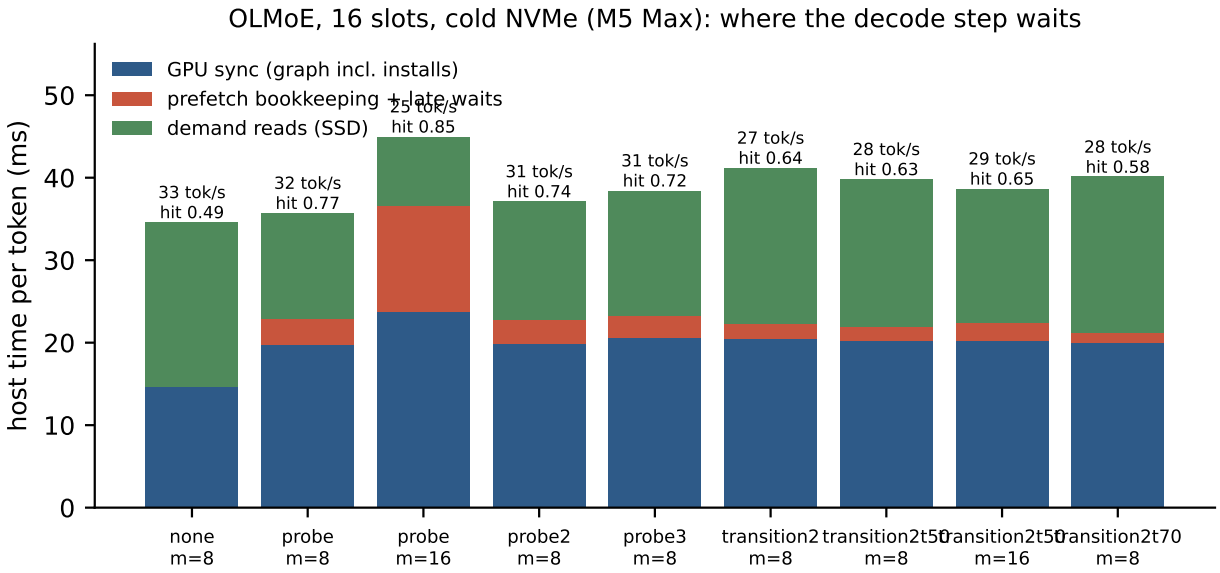


Figure 3. Host time per token on the cold-NVMe tier (OLMoE, 16 slots, held-out general prompts): GPU wait including installs, prefetch bookkeeping and late waits, demand reads. Labels give the median decode throughput and hit rate.

9.8. Metal kernels (H4)

Profiling on M4M64a (`benchmarks/raw/gateD/`) shows that a decode step with all experts resident takes 2.99 ms (334 tok/s synchronous) and moves 452 MB of expert weights, i.e. 151 GB/s or about 28 % of the chip's nominal bandwidth; a `gather_qmm` over 1 to 16 experts takes the same 0.17 ms (latency-bound; 250 GB/s only at 64 experts), and in-graph elementwise ops cost 5–7 μs each against a 0.16 ms synchronisation floor. The step is launch-bound, so the most a kernel fusion can save is a few dispatches per layer. We tested the cheapest one, a single `gather_qmm` over a stacked up+gate buffer (bit-identical logits, `tests/test_fused_upgate.py`): -0.2 % with all experts resident and $+2.6$ % on the cold tier, both within the run-to-run range (Table 8). **H4 is not supported** at batch size one on this model; the profile says that batching tokens, not kernels, is what would change the regime.

9.9. Energy per token (H5, single node)

Table 9 reports package power sampled at 1 s by `powermetrics` while four configurations decoded 256 tokens on the laptop (`benchmarks/raw/gateG/energy_laptop/`). All-resident decoding costs 0.101 J per token at 47 W whether mlx-lm or TITAN-MoE runs it; bounded residency on the cold tier costs five times more per token in gross terms (0.50 J at 16 W) because the machine waits on the disk most of the time; the probe prefetcher brings this to 0.44 J at 18 W. Net of the 13.2 W idle power the ordering reverses between the two cold configurations (0.086 vs 0.119 J),

Table 5. Cold-NVMe tier on the M5 Max laptop (purge + F_NOCACHE), OLMoE 4-bit, held-out general/code/reasoning prompts, 64 tokens, 12 runs per row.

Backend	cfg	n	decode tok/s (median)	p95 ITL ms	hit	stall s	TTFT ms
titan	C=16, none, m=8, cold	12	36.4	35.2	0.526	1.15	371
titan	C=16, prev_rank, m=16, cold	12	19.1	62.2	0.647	1.69	467
titan	C=16, transition, m=8, cold	12	27.9	45.2	0.670	1.24	446
titan	C=16, transition2, m=16, cold	12	25.9	50.8	0.736	1.21	423
titan	C=16, transition2, m=8, cold	12	35.0	35.7	0.654	0.77	339
titan	C=16, transition3, m=8, cold	12	29.1	43.4	0.639	1.17	393
titan	C=32, none, m=8, cold	12	50.6	25.9	0.788	0.62	236
titan	C=32, prev_rank, m=16, cold	12	28.4	45.7	0.785	0.91	263
titan	C=32, transition, m=8, cold	12	41.3	30.4	0.825	0.63	249
titan	C=32, transition2, m=16, cold	12	41.0	32.3	0.848	0.56	243
titan	C=32, transition2, m=8, cold	12	42.0	30.3	0.820	0.59	239
titan	C=32, transition3, m=8, cold	12	41.0	31.2	0.813	0.60	242

Sources: `benchmarks/raw/gateC/cold_nvme_laptop/sweep.jsonl`.

Table 6. Qwen3-30B-A3B-Instruct-2507 4-bit on M3U96a (M3 Ultra, 96 GB): mlx-lm vs TITAN all-resident (128 slots, preloaded), 64 tokens, 3 repeats.

Backend	cfg	n	decode tok/s (median)	p95 ITL ms	hit	stall s	TTFT ms
mlx_lm	–	18	121.7	8.4	–	–	50
titan	C=128, none, m=8	18	121.6	8.5	1.000	0.00	51

Sources: `benchmarks/raw/gateC/qwen3_m3u96a/baseline.jsonl`.

which is the usual ambiguity of attributing idle power. The same run also gives the first *positive* H1 evidence: over 256-token generations the probe reaches a hit rate of 0.84 and 42.0 tok/s against 31.9 for demand loading (+10.1 [+5.2, +16.9] tok/s, Table 10), whereas the 64-token runs of Section 9.4 showed a tie: the per-token install overhead is constant while the benefit of a warm, well-predicted table grows with sequence length. H1 is therefore *length-dependent* on this hardware, and the cluster-level Pareto study remains pending.

10. Ablations

The ablation grid of the protocol (predictor × depth × budget × policy × placement × transport × kernels × batch) is executed gate by gate. In this version only the single-node factors have logs:

- **Install path** (per-miss synchronisation vs lazy batched installs): +80 to +90 % decode throughput at 16 slots (Section 7).
- **Fast path** (device-side slot map when a layer is complete): 203 → 489 tok/s at 64 slots.
- **Eviction policy** (LRU vs LFU): within 0.01 hit rate at every capacity in simulation.

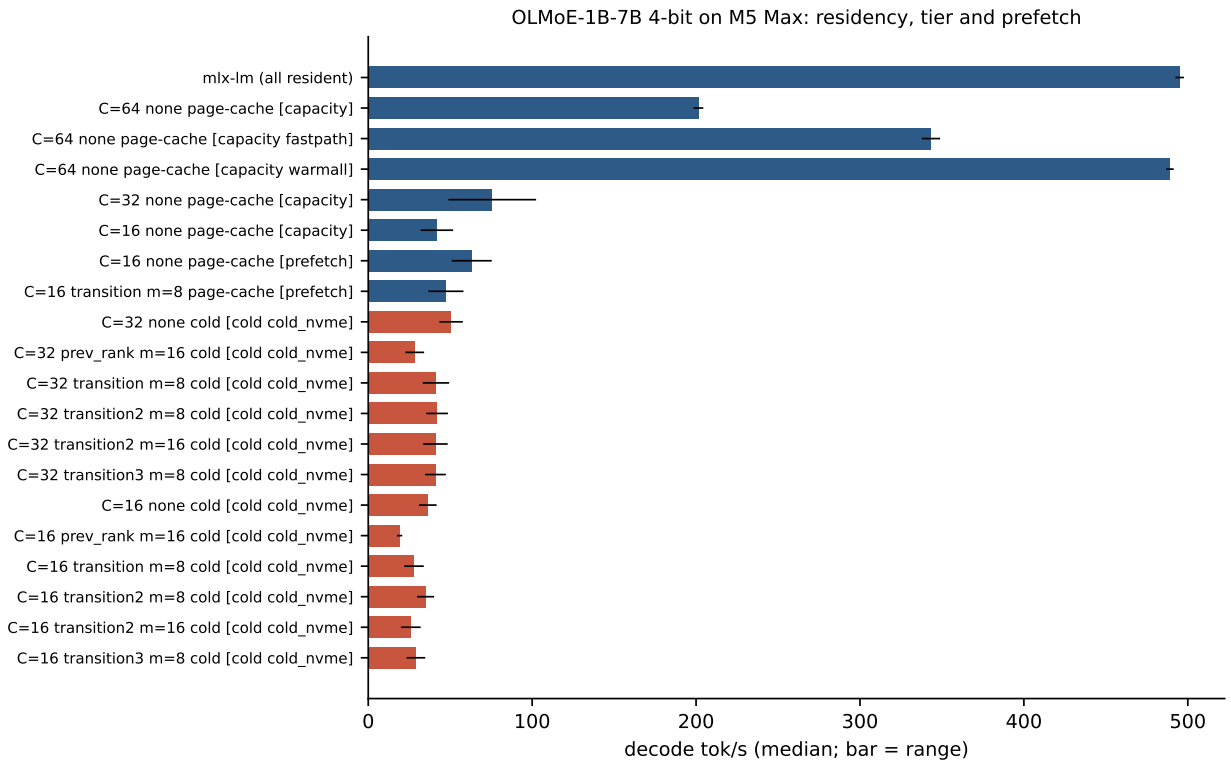


Figure 4. Decode throughput of every single-node configuration measured on the laptop (median over repeats, bar = range): mlx-lm, TITAN-MoE all-resident with and without the synchronisation-free path, bounded residency on the page-cache tier, and the cold-NVMe tier with and without prefetch. Source files are named in brackets.

- **Storage layout** (scattered safetensors vs expert-contiguous records) and **reader parallelism**: 8–10 $\rightarrow$ 18 $\rightarrow$ 36 tok/s on the cold tier at 16 slots.
- **Predictor** (none / prev-rank / transition$_d$ / probe$_d$ / oracle), **depth** and **prefetch width** m: Table 3 (simulation) and Tables 5–4 (online); the best online configuration ties with no prefetch.
- **Kernel fusion** (up+gate): no measurable effect (Table 8).
- **Placement** (local vs odd experts remote over 1 GbE): $-80\,\%$ throughput (Figure 5); frequency- and cost-aware placement are exercised in Gate F.

11. Robustness

Held-out prompt families give hit rates within 0.01 of the fit families at every capacity (v1 traces), and the transition predictor's recall drops from 0.53 (fit) to 0.50 (held-out), so the locality statistics are not an artefact of the fitting prompts. Throughput measurements on the laptop vary by less than 1 % across repeats for the all-resident configurations and by up to 40 % across prompt families for demand loading (code prompts route more repetitively than general prose, hence higher hit

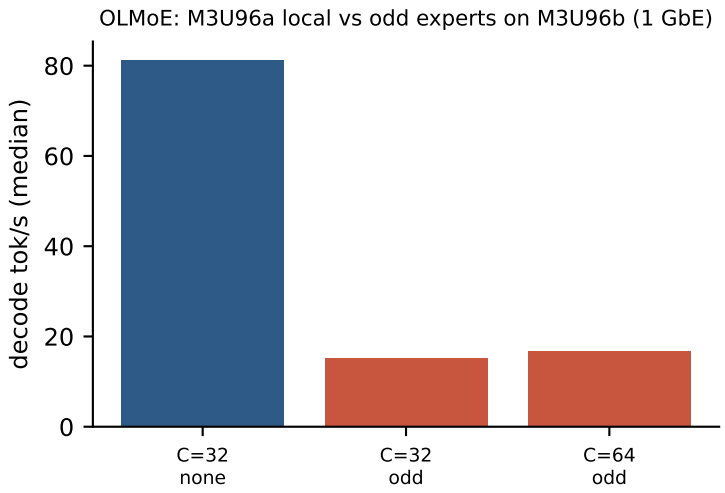


Figure 5. Two Mac Studios over 1 GbE: local bounded residency vs remote execution of half the experts (medians).

Table 7. Gate F: Qwen3-30B-A3B on M3U96a (bounded residency, cold NVMe) with M3U96b as expert worker over 1 GbE; placements: none (all local), odd (half remote, static), popular (least popular half remote, frequency-based); 64 tokens, held-out general/code, 8 runs per row.

Backend	cfg	*n*	decode tok/s (median)	p95 ITL ms	hit	stall s	TTFT m
titan-distributed	C=32, placement=none, cold	16	8.8	187.7	0.775	4.23	83
titan-distributed	C=32, placement=odd, cold	16	4.2	263.7	0.922	1.36	105
titan-distributed	C=32, placement=popular, cold	16	5.1	236.2	0.877	2.18	73
titan-distributed	C=64, placement=none, cold	16	16.3	100.2	0.922	1.37	47
titan-distributed	C=64, placement=odd, cold	16	5.2	224.1	1.000	0.00	92
titan-distributed	C=64, placement=popular, cold	16	6.0	200.9	1.000	0.00	71

Sources: `benchmarks/raw/gateF/two_node_qwen3_m3u96a_m3u96b.jsonl.tmp`.

rates). Thermal state and competing processes are not controlled in the smoke tests; the cold-NVMe sweeps were run alone on the laptop. Table 10 gives bootstrap confidence intervals for every comparison used in the text (`scripts/audit_stats.py`, 10 000 resamples, seed 0, raw logs only): the all-resident parity with mlx-lm holds within -5.9 [-7.1, -4.8] tok/s on the laptop and -0.1 [-0.3, $+0.3$] on M3U96a; every prefetch configuration's interval excludes a gain except the depth-2 transition predictor at 16 slots, whose interval straddles zero (-1.4 [-7.3, $+6.0$]); the fused-kernel differences are within ± 1 tok/s; and the two-node loss is -66 [-72, -55] tok/s.

12. Limitations

- **Scope of evidence.** Version 0.1 reports single-node results on a small model whose experts fit in memory, so the regimes where H1–H3 can matter (expert set larger than the budget of a node) are

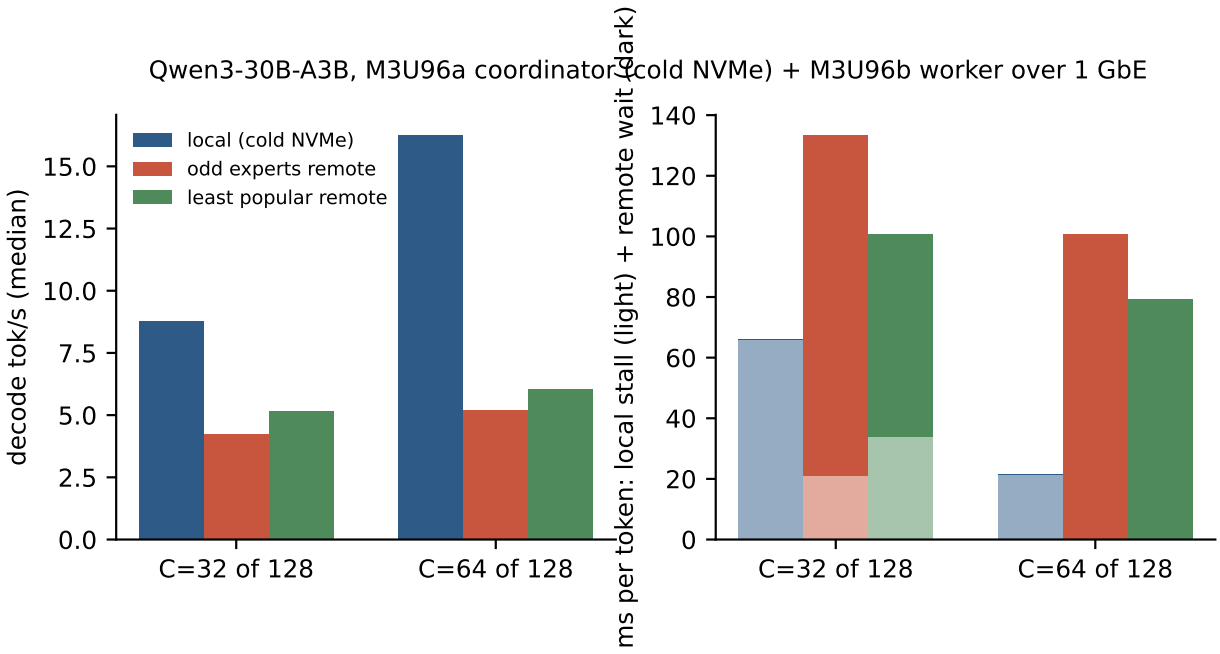


Figure 6. Qwen3-30B-A3B on two Mac Studios over 1 GbE: decode throughput (left) and per-token waiting (right: local stall, light; remote wait, dark) for three placements at two local budgets.

Table 8. H4: fused up+gate gather_qmm vs two kernels (laptop; 64 slots resident, 128 tokens, 12 runs; 16 slots cold, 64 tokens, 4 runs).

Backend	cfg	*n*	decode tok/s (median)	p95 ITL ms	hit	stall s	TTFT ms
titan	C=16, fused=False, cold	4	35.0	36.9	0.492	1.17	348
titan	C=16, fused=True, cold	4	35.9	35.0	0.492	1.10	342
titan	C=64, fused=False	12	476.7	2.2	1.000	0.00	17
titan	C=64, fused=True	12	475.6	2.2	1.000	0.00	18

Sources: `benchmarks/raw/gateD/fused_upgate_laptop.jsonl`.

only approached by artificially bounding the slot tables. The medium model on the 36–64 GB nodes is the first realistic regime.

- **Topology.** The cluster is connected by gigabit Ethernet with no Thunderbolt link; RDMA/JACCL results would require cabling the Studios (a physical change outside the agent's authority). A LAN fetch of one expert costs about 28 ms, which rules out moving weights during decode and confines the remote tier to remote *execution*.
- **Synchronisation floor.** Any demand-loading design must read the router's choice on the host once per layer; on this model that costs 2.4× in throughput. Hiding it requires either complete tables (no benefit from caching) or speculative execution on predicted experts with verification, which changes nothing semantically only if the verification is exact – a Gate D question.

Table 9. Energy per generated token on the laptop (M5 Max), OLMoE 4-bit, 256-token greedy decode, held-out general/code prompts; package power from `powermetrics` at 1 s; idle baseline 13.2 W. Source: `benchmarks/raw/gateG/energy_laptop/`.

Backend	residency	runs	tok/s	mean W	J/token (gross)	J/token (net of idle)
mlx-lm	all resident	12	463.0	47.2	0.101	0.073
titan	all resident	12	461.9	46.4	0.101	0.072
titan	16 slots, cold NVMe, probe	12	42.0	18.4	0.441	0.119
titan	16 slots, cold NVMe	12	31.9	15.9	0.498	0.086

- **Predictors.** Only simple predictors are tested; the probe is fitted on 48 prompts and evaluated on 17, all synthetic.
- **Install path.** The identified bottleneck of H1 (nine lazy slice writes per installed expert) is an implementation property of this version, not of the approach; a single-copy install may reverse the verdict and is the first task of v0.2.
- **Measurements.** Stall times in simulation use point estimates of tier latencies; online runs use the page-cache tier unless stated; energy is not yet measured.
- **Baselines.** llama.cpp and colibri comparisons are pending and will carry precision flags because their quantization formats differ from MLX's.

13. Conclusion

We have built an MLX-native MoE runtime whose bounded expert residency is semantically invisible (bit-identical logits to mlx-lm on two architectures, on one node and across two) and whose all-resident path costs nothing (489 vs 494 tok/s; 121.6 vs 121.7 on the medium model), characterised the cluster as it actually is (gigabit Ethernet, no Thunderbolt), and tested the five hypotheses. Three findings carry over to any similar system: the storage layout of experts decides the cold tier (8 to 36 tok/s); the number of host-side operations per installed expert, not prediction accuracy, decides whether prefetching pays (a probe within 0.16 of the oracle hit rate only wins on long generations); and on gigabit Ethernet a remote expert costs more than a local cold miss, so placement across nodes buys capacity and, when cost-aware, a better loss than static sharding, but not speed. Kernel fusion was immaterial because decoding at batch size one is launch-bound. The next version follows from the attribution: a single-copy install, the probe on the medium model, nodes whose budget the model exceeds, energy on the cluster, and, if cabled, the Thunderbolt tier. Every number here has a log; the losses are reported with the same care as the wins.

Table 10. Statistical audit: medians with bootstrap 95 % CIs (10 000 resamples) of decode throughput (tok/s) and of the difference A − B.

Comparison (A vs B)	n_A/n_B	A [CI]	B [CI]	A−B [CI]
laptop: TITAN all-resident (64 slots, preloaded) vs mlx-lm	12/12	489.1 [488.2, 489.5]	495.0 [494.1, 495.9]	-5.9 [-7.1, -4.8]
laptop: TITAN 64 slots not preloaded (sync path) vs mlx-lm	12/12	201.4 [201.0, 202.7]	495.0 [494.1, 495.9]	-293.6 [-294.4, -292.0]
M3U96a Qwen3-30B: TITAN all-resident (128 slots) vs mlx-lm	18/18	121.6 [121.4, 121.8]	121.7 [121.4, 121.8]	-0.1 [-0.3, +0.3]
laptop cold NVMe, 16 slots: transition m=8 vs no prefetch	12/12	27.9 [26.5, 30.5]	36.4 [34.1, 42.0]	-8.5 [-14.5, -5.1]
laptop cold NVMe, 16 slots: transition2 m=8 vs no prefetch	12/12	35.0 [33.7, 41.5]	36.4 [34.1, 42.0]	-1.4 [-7.3, +6.0]
laptop cold NVMe, 16 slots: transition3 m=8 vs no prefetch	12/12	29.1 [27.7, 34.7]	36.4 [34.1, 42.0]	-7.3 [-13.1, -1.1]
laptop cold NVMe, 16 slots: transition2 m=16 vs no prefetch	12/12	25.9 [25.0, 29.6]	36.4 [34.1, 42.0]	-10.4 [-16.1, -5.9]
laptop cold NVMe, 16 slots: prev_rank m=16 vs no prefetch	12/12	19.1 [18.7, 20.0]	36.4 [34.1, 42.0]	-17.2 [-22.9, -14.8]
laptop cold NVMe, 32 slots: transition m=8 vs no prefetch	12/12	41.3 [35.5, 46.9]	50.6 [46.6, 56.9]	-9.2 [-17.5, -2.6]
laptop cold NVMe, 32 slots: transition2 m=8 vs no prefetch	12/12	42.0 [38.5, 46.8]	50.6 [46.6, 56.9]	-8.5 [-15.2, -2.7]
laptop cold NVMe, 32 slots: transition3 m=8 vs no prefetch	12/12	41.0 [37.6, 46.6]	50.6 [46.6, 56.9]	-9.5 [-16.2, -3.0]
laptop cold NVMe, 32 slots: transition2 m=16 vs no prefetch	12/12	41.0 [35.4, 46.0]	50.6 [46.6, 56.9]	-9.6 [-17.7, -3.6]
laptop cold NVMe, 32 slots: prev_rank m=16 vs no prefetch	12/12	28.4 [27.0, 32.5]	50.6 [46.6, 56.9]	-22.2 [-28.6, -16.9]
laptop cold NVMe, 16 slots (general): probe m=8 vs no prefetch	4/4	31.8 [31.6, 32.8]	33.0 [32.6, 33.3]	-1.2 [-1.6, -0.2]
laptop cold NVMe, 16 slots (general): probe m=16 vs no prefetch	4/4	25.3 [24.8, 26.2]	33.0 [32.6, 33.3]	-7.7 [-8.4, -6.8]
laptop cold NVMe, 16 slots (general): probe2 m=8 vs no prefetch	4/4	31.3 [31.0, 32.0]	33.0 [32.6, 33.3]	-1.7 [-2.1, -0.9]
laptop cold NVMe, 16 slots (general): probe3 m=8 vs no prefetch	4/4	30.7 [29.3, 32.6]	33.0 [32.6, 33.3]	-2.3 [-3.7, -0.4]
laptop cold NVMe, 16 slots (general): transition2 m=8 vs no prefetch	4/4	27.1 [26.0, 28.2]	33.0 [32.6, 33.3]	-5.9 [-7.1, -4.5]
laptop cold NVMe, 16 slots (general): transition2t50 m=8 vs no prefetch	4/4	27.9 [26.8, 29.4]	33.0 [32.6, 33.3]	-5.1 [-6.2, -3.6]
laptop cold NVMe, 16 slots (general): transition2t70 m=8 vs no prefetch	4/4	27.8 [26.7, 29.4]	33.0 [32.6, 33.3]	-5.2 [-6.3, -3.6]
laptop H4: fused up+gate vs two kernels, 64 slots resident	12/12	475.6 [473.5, 476.2]	476.7 [475.7, 478.2]	-1.1 [-3.6, +0.1]
laptop H4: fused up+gate vs two kernels, 16 slots cold	4/4	35.9 [35.7, 36.7]	35.0 [34.1, 35.5]	+1.0 [+0.4, +2.0]
laptop cold NVMe, 16 slots, 256-token generations (energy run): probe m=8 vs no prefetch	12/12	42.0 [39.1, 45.1]	31.9 [26.8, 35.2]	+10.1 [+5.2, +16.9]
laptop energy run: TITAN all-resident vs mlx-lm (256 tokens)	12/12	461.9 [460.0, 463.3]	463.0 [457.2, 467.4]	-1.1 [-6.3, +4.9]
M3U96a+M3U96b OLMoE 32 slots: odd experts remote vs local	16/16	15.1 [14.8, 15.5]	81.3 [70.5, 86.9]	-66.2 [-71.8, -55.2]
M3U96a+M3U96b Qwen3-30B 32 slots cold: odd placement vs local cold NVMe	16/16	4.2 [4.2, 4.3]	8.8 [8.3, 9.3]	-4.5 [-5.1, -4.0]
M3U96a+M3U96b Qwen3-30B 32 slots cold: popular placement vs local cold NVMe	16/16	5.1 [4.6, 5.6]	8.8 [8.3, 9.3]	-3.6 [-4.4, -2.9]
M3U96a+M3U96b Qwen3-30B 64 slots cold: odd placement vs local cold NVMe	16/16	5.2 [5.2, 5.3]	16.3 [14.9, 19.7]	-11.1 [-14.5, -9.7]
M3U96a+M3U96b Qwen3-30B 64 slots cold: popular placement vs local cold NVMe	16/16	6.0 [5.8, 6.7]	16.3 [14.9, 19.7]	-10.2 [-13.6, -8.8]

References

Alfarizy, Alfarizy, Hung Truong Thanh Nguyen, René Richard, Roozbeh Razavi-Far, and Hung Cao, "Does Mixture-of-Experts Actually Help Inference on Consumer and Edge Hardware? An Empirical Study," *arXiv preprint arXiv:2606.21428*, 2026.

Apple Machine Learning Research, "Local and Distributed Large LLM Inference with MLX and RDMA on Mac Studio," https://machinelearning.apple.com/research/mlx-distributed-inference 2026. Published 2026-10-15. Accessed 2026-11-01.

Benazir, Afsara and Felix Xiaozhu Lin, "Efficient Mixture-of-Experts LLM Inference with Apple Silicon NPUs," *arXiv preprint arXiv:2604.18788*, 2026.

Borzunov, Alexander, Dmitry Baranchuk, Tim Dettmers, Maksim Riabinin, Younes Belkada, Artem Chumachenko, Pavel Samygin, and Colin Raffel, "Petals: Collaborative Inference and Fine-tuning of Large Models," in "Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 3: System Demonstrations)" 2023, pp. 558–568.

Cai, Ruisi, Yeonju Ro, Geon-Woo Kim, Peihao Wang, Babak Ehteshami Bejnordi, Aditya Akella, and Zhangyang Wang, "Read-ME: Refactorizing LLMs as Router-Decoupled Mixture of Experts with System Co-Design," in "Advances in Neural Information Processing Systems (NeurIPS)" 2024. arXiv:2410.19123.

Cao, Shiyi, Shu Liu, Tyler Griggs, Peter Schafhalter, Xiaoxuan Liu, Ying Sheng, Joseph E. Gonzalez, Matei Zaharia, and Ion Stoica, "MoE-Lightning: High-Throughput MoE Inference on Memory-constrained GPUs," in "Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)" 2025. arXiv:2411.11217; ACM DOI not retrieved.

Eliseev, Artyom and Denis Mazur, "Fast Inference of Mixture-of-Experts Language Models with Offloading," *arXiv preprint arXiv:2312.17238*, 2023.

exo labs, "exo: Run frontier AI locally," https://github.com/exo-explore/exo 2025. Accessed 2026-11-01.

Fang, Zhiyuan, Yuegui Huang, Zicong Hong, Yufeng Lyu, Wuhui Chen, Yue Yu, Fan Yu, and Zibin Zheng, "Klotski: Efficient Mixture-of-Expert Inference via Expert-Aware Multi-Batch Pipeline," *arXiv preprint arXiv:2502.06888*, 2025.

_ , Zicong Hong, Yuegui Huang, Yufeng Lyu, Wuhui Chen, Yue Yu, Fan Yu, and Zibin Zheng, "Fate: Fast Edge Inference of Mixture-of-Experts Models via Cross-Layer Gate," *arXiv preprint arXiv:2502.12224*, 2025.

Fornaro, Vincenzo, "colibri: Tiny engine, immense model," https://github.com/JustVugg/colibri 2026. Version 2.1.0, Apache-2.0. Accessed 2026-11-01. No paper or citation provided by the project.

Gerganov, Georgi and ggml-org contributors, "llama.cpp: LLM inference in C/C++," https://github.com/ggml-org/llama.cpp 2026. Accessed 2026-11-01; RPC backend documented at tools/rpc/README.md.

Hannun, Awni, Jagrit Digani, Angelos Katharopoulos, and Ronan Collobert, "MLX: Efficient and flexible machine learning on Apple silicon," https://github.com/ml-explore/mlx

2023. Accessed 2026-11-01; distributed docs: https://ml-explore.github.io/mlx/build/html/usage/distributed.html.

He, Xin, Shunkang Zhang, Kaijie Tang, Shaohuai Shi, Yuxin Wang, Zihao Zeng, Zhenheng Tang, Xiaowen Chu, Haiyan Yin, Ivor W. Tsang, and Yew Soon Ong, "ExpertFlow: Efficient Mixture-of-Experts Inference via Predictive Expert Caching and Token Scheduling," in "Proceedings of the Design Automation Conference (DAC)" 2026. arXiv:2410.17954.

Hwang, Ranggi, Jianyu Wei, Shijie Cao, Changho Hwang, Xiaohu Tang, Ting Cao, and Mao Yang, "Pre-gated MoE: An Algorithm-System Co-Design for Fast and Scalable Mixture-of-Expert Inference," in "Proceedings of the 51st ACM/IEEE Annual International Symposium on Computer Architecture (ISCA)" 2024. arXiv:2308.12066.

Javat, Abdurrahman and Allan Kazakov, "Silicon Showdown: Performance, Efficiency, and Ecosystem Barriers in Consumer-Grade LLM Inference," *arXiv preprint arXiv:2605.00519*, 2026.

Kamahori, Keisuke, Tian Tang, Yile Gu, Kan Zhu, and Baris Kasikci, "Fiddler: CPU-GPU Orchestration for Fast Inference of Mixture-of-Experts Models," in "International Conference on Learning Representations (ICLR)" 2025. arXiv:2402.07033.

Kannan, Rajeswari, Raj Firke, Shreya Bengle, and Srushti Deshmukh, "GreenBench: Benchmarking Energy Efficiency and Carbon Footprint of Open-Source LLM Inference on Apple Silicon," in "IEEE International Conference on Computing, Communication, Control and Automation (ICCUBEA)" 2026. arXiv:2608.28667.

Kim, Byeongju, Jungwan Lee, Donghyeon Han, Hoi-Jun Yoo, and Sangyeob Kim, "Flash-MoE: Reducing SSD I/O Bottlenecks via ML-Based Cache Replacement for Mixture-of-Experts Inference on Edge Devices," *arXiv preprint arXiv:2601.17063*, 2026.

Kong, Jinwei, Runqi Meng, Fanyi Wang, Wentao Qiu, Haotian Hu, Yongjian Zhou, and Zhenhua Ge, "SpecPrefetch: Parameter-Efficient Expert Prefetching for Sparse MoE Foundation Models," *arXiv preprint arXiv:2607.24787*, 2026.

Li, Jiamin, Yimin Jiang, Yibo Zhu, Cong Wang, and Hong Xu, "Accelerating Distributed MoE Training and Inference with Lina," in "Proceedings of the USENIX Annual Technical Conference (USENIX ATC)" 2023, pp. 945–959. arXiv:2210.17223.

Li, Zonghang, Tao Li, Wenjiao Feng, Rongxing Xiao, Jianshu She, Hong Huang, Mohsen Guizani, Hongfang Yu, Qirong Ho, Wei Xiang, and Xue Liu, "Prima.cpp: Fast 30-70B LLM Inference on Heterogeneous and Low-Resource Home Clusters," *arXiv preprint arXiv:2504.08791*, 2025.

Liang, Fangzhou, Yibin Shen, Jianmin Hu, Jiayang Xu, Hanchi Gao, Minxian Xu, and Zili Meng, "SSD-LLaMA: SSD-Native Inference for Trillion-Parameter MoE at 1+ Token/s on a Consumer PC," *arXiv preprint arXiv:2609.18110*, 2026.

Liang, Jingcong, Siyuan Wang, Miren Tian, Yitong Li, Duyu Tang, and Zhongyu Wei, "Not All Models Suit Expert Offloading: On Local Routing Consistency of Mixture-of-Expert Models," in "International Conference on Learning Representations (ICLR)" 2026. arXiv:2505.16056.

Muennighoff, Niklas, Luca Soldaini, Dirk Groeneveld, Kyle Lo, Jacob Morrison, Sewon Min, Weijia Shi, Pete Walsh, Oyvind Tafjord, Nathan Lambert, Yuling Gu, Shane Arora, Akshita Bhagia, Dustin Schwenk, David Wadden, Alexander Wettig, Binyuan Hui, Tim Dettmers, Douwe Kiela, Ali Farhadi, Noah A. Smith, Pang Wei Koh, Amanpreet Singh, and Hannaneh Hajishirzi, "OLMoE: Open Mixture-of-Experts Language Models," *arXiv preprint arXiv:2409.02060*, 2024.

Rajbhandari, Samyam, Conglong Li, Zhewei Yao, Minjia Zhang, Reza Yazdani Aminabadi, Ammar Ahmad Awan, Jeff Rasley, and Yuxiong He, "DeepSpeed-MoE: Advancing Mixture-of-Experts Inference and Training to Power Next-Generation AI Scale," in "Proceedings of the 39th International Conference on Machine Learning (ICML)" 2022. arXiv:2201.05596.

Shen, Zixu, Kexin Chu, Yifan Zhang, Dawei Xiang, Runxin Wu, and Wei Zhang, "ExpertFlow: Adaptive Expert Scheduling and Memory Coordination for Efficient MoE Inference," *arXiv preprint arXiv:2510.26730*, 2025.

Skliar, Andrii, Ties van Rozendaal, Romain Lepert, Todor Boinovski, Mart van Baalen, Markus Nagel, Paul Whatmough, and Babak Ehteshami Bejnordi, "Mixture of Cache-Conditional Experts for Efficient Mobile Device Inference," *Transactions on Machine Learning Research*, 2025. arXiv:2412.00099.

Song, Xiaoniu, Zihang Zhong, Rong Chen, and Haibo Chen, "ProMoE: Fast MoE-based LLM Serving using Proactive Caching," *arXiv preprint arXiv:2410.22134*, 2024.

Wang, Zihan, Yuqi Wang, Lei Gong, Cheng Tang, Wenqi Lou, Teng Wang, Chao Wang, and Xuehai Zhou, "SeqMoE: Toward Full-Load Performance via Predictive and Graph-Compatible MoE Offloading," *arXiv preprint arXiv:2609.12978*, 2026.

Xu, Tairan, Leyang Xue, Zhan Lu, Adrian Jackson, and Luo Mai, "MoE-Gen: High-Throughput MoE Inference on a Single GPU with Module-Based Batching," *arXiv preprint arXiv:2503.09716*, 2025.

Xue, Leyang, Yao Fu, Zhan Lu, Luo Mai, and Mahesh Marina, "MoE-Infinity: Efficient MoE Inference on Personal Machines with Sparsity-Aware Expert Cache," *arXiv preprint arXiv:2401.14361*, 2024.

Yang, An, Anfeng Li, Baosong Yang et al., "Qwen3 Technical Report," *arXiv preprint arXiv:2505.09388*, 2025.

Yao, Jinghan, Quentin Anthony, Aamir Shafi, Hari Subramoni, and Dhabaleswar K. Panda, "Exploiting Inter-Layer Expert Affinity for Accelerating Mixture-of-Experts Model Inference," *arXiv preprint arXiv:2401.08383*, 2024.

Yu, Enda, Dezun Dong, Zhaoning Zhang, Zhe Bai, Weiling Yang, Haojie Wang, Dongsheng Li, Yongwei Wu, and Xiangke Liao, "LayerScope: Predictive Cross-Layer Scheduling for Efficient Multi-Batch MoE Inference on Legacy Servers," in "Proceedings of the International Conference on Supercomputing (ICS)" 2026. arXiv:2509.23638.

Yu, Hanfei, Xingqi Cui, Hong Zhang, Hao Wang, and Hao Wang, "Taming Latency-Memory Trade-Off in MoE-Based LLM Serving via Fine-Grained Expert Offloading," *arXiv preprint arXiv:2502.05370*, 2025.

Zhang, Yujie, Shivam Aggarwal, and Tulika Mitra, "DAOP: Data-Aware Offloading and Predictive Pre-Calculation for Efficient MoE Inference," in "Design, Automation & Test in Europe Conference (DATE)" 2025. arXiv:2501.10375.

Zhang, Yuning, Grant Pinkert, Nan Yang, Yanli Li, and Dong Yuan, "DuoServe-MoE: Dual-Phase Expert Prefetch and Caching for LLM Inference QoS Assurance," *arXiv preprint arXiv:2509.07379*, 2025.

Zhao, Chenggang et al., "DeepEP: an efficient expert-parallel communication library," https://github.com/deepseek-ai/DeepEP 2025. Accessed 2026-11-01; author list abbreviated from the repository's own BibTeX.

Zhong, Shuzhang, Yanfan Sun, Ling Liang, Runsheng Wang, Ru Huang, and Meng Li, "HybriMoE: Hybrid CPU-GPU Scheduling and Cache Management for Efficient MoE Inference," in "Proceedings of the Design Automation Conference (DAC)" 2025. arXiv:2504.05897.

Appendices

A. Reproducibility

Software. Python 3.12 virtual environment with MLX 0.32.4 and mlx-lm 0.32.2 (`requirements-lock.txt`); reference repositories audited at colibri `0a5504a` (2026-10-31), mlx `bd63d38` (2026-11-01), mlx-lm `736ab0d` (2026-10-30), llama.cpp `a91eab6` (2026-10-31).

Commands. Correctness: `.venv/bin/python -m pytest tests -q`. Capacity sweep: `benchmarks/script -backend {mlx_lm,titan} -capacity C -predictor P -traces T`. Traces: `benchmarks/scripts/coll` Simulation: `benchmarks/scripts/simulate_cache.py`. SSD tier: `benchmarks/scripts/ssd_bandwidth.p` after `sudo purge`. Node preparation: `scripts/cluster_setup.sh`.

Raw data. `benchmarks/raw/gateA/` (smoke baseline, SSD), `gateB/` (design microbenchmarks, capacity sweep, tests), `gateC/` (traces, online prefetch, miss-cost breakdown, Qwen3 baselines); processed simulations in `benchmarks/processed/`. Logs are append-only JSON lines.

Prior-art table. `research/prior_art.csv` (59 rows, evidence URL per row) and `research/RELATED_WORK.m`

AI-assistance disclosure. The software, experiments and this manuscript were produced by Claude Code (Anthropic, model Claude Fable 5.1) acting as an autonomous research engineer under the author's written mandate (`docs/MANDATE.md`); the author set the research question, hypotheses, integrity rules and gates, and is responsible for the content. Every quantitative claim traces to a logged measurement; the agent's decisions are recorded in `research/DECISIONS.md` and its chronology in `research/LOG.md`.