Training in the Gaps: Fine-Tuning a Customer's Adapter inside a Live Inference Engine on a Consumer Mac

Wisdom Okechukwu

October 2026

Abstract

We ask whether a customer's LoRA adapter can be fine-tuned in the idle gaps of a live inference engine on a consumer Mac, on the 4-bit base model the engine already holds in memory, without hurting serving and with exact resumption when the process is killed. We build uli, a training scheduler inside the EngineV2 continuous-batching engine on MLX, plus a small hub that accepts jobs, routes them and serves finished adapters. On one M1 Pro with 16 GiB of memory and Qwen3.5-9B at 4 bits, an adapter trained through the engine's idle calls on an otherwise idle Mac answers 91.2% of 500 held-out Banking77 queries from the query alone, against 73.6% for the same base given the full label list; the configuration was chosen over three rounds graded on the same rows, with one seed and one split. Under a 0.2 requests per second load, training adds p99 0.134 s to a request's time to first token, the longest training piece has p99 0.145 s, and median time to first token stays at 1.41 s against 1.44 s with no training. Training adds 0.296 GB of peak memory and never copies the base. In an end-to-end run through the hub, the node was killed mid-job twice, once by the test script and once by the operator after a stall, and the final adapter matched an uninterrupted run in all 304 arrays. One failure stays open: a memory-pressure gate held training for about 55 minutes and needed manual help. Sharing a base between serving and fine-tuning is not new; our contribution is a measured, narrow data point for consumer Apple Silicon.

1 Introduction

Fine-tuning a small open model on a team's own labelled examples is a common way to fit it to a narrow task, such as intent classification or field extraction [4, 10]. On the task in this paper it moves a 9B model from 73.6% to 91.2% accuracy. The value of such a model comes from the customer's labels, not from the volume of tokens it serves. That makes it a plausible job for machines that cannot compete on token volume, such as consumer Macs.

This paper asks one question. Can a customer's adapter be fine-tuned in the idle gaps of a live inference engine, on a consumer Mac with a 4-bit base held in memory, without hurting serving, and resume exactly after the process is killed?

The question matters now for two reasons. First, networks of provider-owned Macs already serve open models [5]. When checked on 2026-10-27, every model in one such network's catalog required at least 24 GB of memory, so a 16 GB Mac served none of them. Second, OpenAI's self-serve fine-tuning stops accepting new jobs from active customers on January 6, 2027 [15]. The same page states that inference on fine-tuned models continues until the base model shuts down; for gpt-4o-mini and gpt-4.1-mini that date is January 14, 2027. Teams that depend on small custom models will need somewhere else to train and serve them.

What we build. uli is a scheduler that runs training as short, resumable pieces inside the inference engine's own idle loop, on the quantized weights the engine serves. A request is blocked directly by at most one piece. Finished adapters are served per request, side by side with other adapters and the base, in one batch. A hub accepts a dataset, assigns the job to a node, stores checkpoints so a killed node can resume, deletes training data when the job ends, and proxies chat to the adapter.

What we measure. All results come from one M1 Pro (Section 4) running Qwen3.5-9B [17] at 4 bits [6]:

1. The adapter learns the task: 91.2% of 500 held-out Banking77 rows from the query alone, against 73.6% for the base with every label in its prompt, with one seed and one split (Section 5.1).
2. Serving cost is small: at 0.2 requests per second training adds p99 0.134 s to time to first token, the longest training piece has p99 0.145 s, and training adds 0.296 GB of peak memory (Sections 5.2 and 5.3).
3. Resume is exact: after two SIGKILLs mid-job, one scripted and one by the operator after a stall, the final adapter is bit-identical in all 304 arrays to one trained without interruption (Section 5.5).
4. The loop works end to end through the hub, including routing (unit-tested; one node in the live run), adapter serving and data deletion (Section 5.5).

What we do not claim. Sharing one base model between fine-tuning and serving, and suspending fine-tuning at fine granularity to protect inference latency, are published [2, 9, 12, 14]. Our claim is narrower: the mechanism works on Apple Silicon with MLX, in unified memory, on a resident 4-bit base, with bit-exact resume after a kill, and we report what it costs. The design targets memory the owner's other applications also use, but the passing serving and equivalence runs had those applications closed; under their pressure the piece bound failed or the gate stopped training (Section 6). We make no claim about provider earnings, other Macs or other base models.

2 Background and related work

LoRA on a quantized base. LoRA trains a low-rank update beside frozen weights [10]. QLoRA showed that the frozen base can stay quantized while the adapter trains in higher precision [4]. uli trains a rank-8 adapter in float32 on the 4-bit weights the engine already serves, so no second copy of the base exists.

Co-serving fine-tuning and inference. FlexLLM fuses inference and parameter-efficient fine-tuning at the token level on shared GPUs, interleaving training tokens into every co-serving iteration under latency targets [14]. LLMStation turns a tuning task into a suspendable pipeline that shares one base with serving, pausing backward work per layer or operator [9]. DeltaServe converts idle inference capacity into LoRA fine-tuning throughput through a hook interface over existing engines, admitting fine-tuning only when there is headroom under the inference SLO [2]. LOCAL runs inference and adaptation for an on-device agent on one GPU and manages adapter versions against cached state [12]. LLMStation and DeltaServe already establish training in serving gaps with

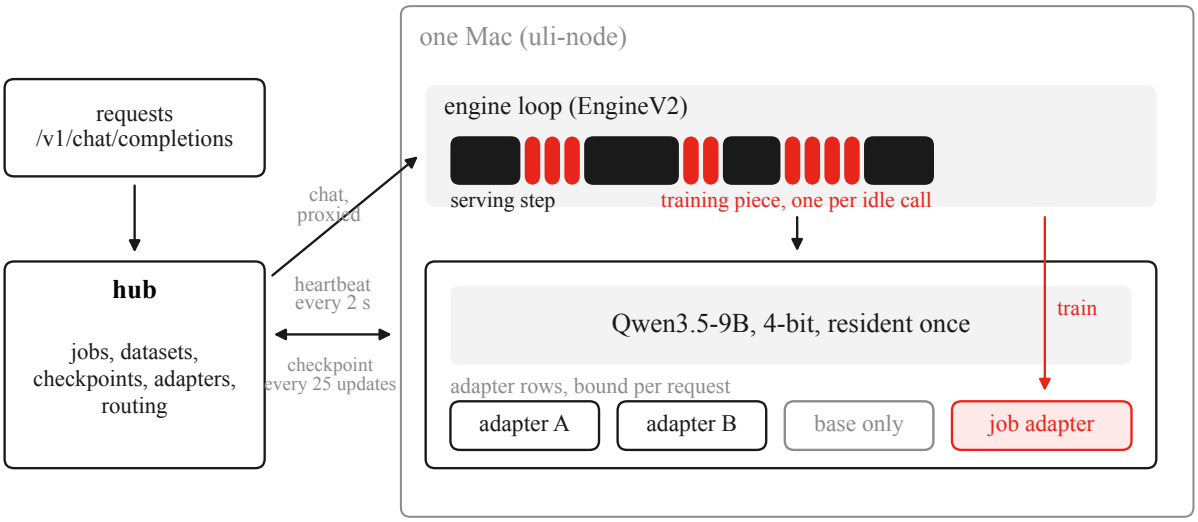


Figure 1: One engine, one resident base. Serving rows and the training job share the same 4-bit weights; each request binds its own adapter row, and training runs only in the loop's idle calls.

fine-grained suspension, and FlexLLM shows that inference and fine-tuning can share every iteration instead. Three of the four were evaluated on consumer GPUs: LLMStation on RTX 3090s as well as H100s, DeltaServe on a single RTX 5090 with 32 GB, and LOCAL on a 7B model on one 24 GB consumer GPU. None of the four reports Apple Silicon, training on a 4-bit base held for serving, or a 16 GiB unified-memory machine. Related systems also exist: FineInfer multiplexes one base between fine-tuning and inference with iteration-level scheduling [8], Loquetier runs LoRA fine-tuning and serving in one runtime [19], and ConServe protects online requests from offline inference by preempting the offline work layer by layer [16].

Serving many adapters. S-LoRA and Punica serve many LoRA adapters over one base by batching requests for different adapters together [3, 18]. uli serves adapters the same way at small scale, binding an adapter, or none, to each row of a batch. We claim nothing new here beyond its correctness on this engine.

3 Design

3.1 Setting and constraints

A node is one Mac holding one base model in memory and serving requests from it. The owner keeps using the Mac for other work, so free memory changes without warning. The engine is EngineV2, a continuous-batching engine in an MIT-licensed fork of mlx-swift-lm on MLX [7, 11]. Serving always has priority: training may only use time when no request is waiting or running. Figure 1 shows the parts.

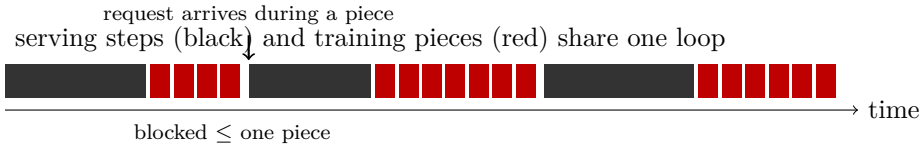


Figure 2: Serving and training share one engine loop. Training runs only when the loop is idle, one piece per idle call, so a new request is blocked directly by at most the piece already running.

3.2 Training as pieces

A micro-step is one training example's forward pass, backward pass and, every fourth micro-step, an optimizer update. On this model one micro-step takes several seconds of compute, far too long to make a request wait behind it. uli therefore splits each micro-step into pieces: an embedding piece, two forward pieces per decoder layer (the token mixer and the MLP), a head piece, two backward pieces per layer, and an update piece. A micro-step runs in 131 pieces. Between pieces the trainer is suspended and holds its own state. Each backward piece recomputes its sublayer from one stored hidden state, so activations for only one sublayer are live at a time. The update piece is the only point where a step becomes visible to serving: it replaces the adapter arrays at once.

3.3 The idle hook

uli adds one observer to EngineV2's loop (patch `0003`). When the loop finds no waiting or running request, it calls the observer instead of sleeping. The observer runs at most one training piece and returns, and the loop checks for requests again. A request that arrives during a piece is therefore blocked directly by at most that one piece. Figure 2 shows the pattern.

3.4 The training gate

Before each piece a gate checks the machine. It refuses when MLX's active memory is within a headroom margin of its limit, when macOS reports memory pressure other than normal, or when swap-outs rose in the last 10 seconds. Serving is never gated. MLX's buffer cache is capped at 1 GiB. Section 6 reports how the pressure rule failed in practice.

3.5 Adapters per row

Target linear layers are wrapped once with adapter layers that share the base weights. For each forward pass the engine reports which request owns each row, and uli binds that row's adapter, or none. Each adapter row computes its low-rank term on its own slice, so a row's output does not depend on the rest of its batch. A training job is one more adapter in the same bank.

3.6 Checkpoint and resume

A checkpoint is one safetensors file holding every adapter array, the AdamW first and second moments and step for each array [13], and the micro-step and update counters. It is taken only at an update boundary, when no gradient is pending. Data order, epoch and learning rate are pure functions of the step count, the seed and the configuration, and training draws no randomness after the adapter is created. Restoring writes the saved moments into the optimizer's own state arrays, so

every later update runs the same library code as an uninterrupted run. Resume is therefore exact by construction, and Section 5.5 tests that on the real model.

3.7 Hub and job lifecycle

A hub written in Go holds jobs, datasets, checkpoints and adapters, and routes each request to a live node holding the base, preferring one with the adapter cached. Nodes are clients: each sends a heartbeat every 2 seconds carrying progress, losses and outcomes, and receives assignments and cancellations in reply. A node downloads the dataset into a private job directory, verifies its hash, trains, uploads a checkpoint every 25 updates, uploads the adapter when done, and deletes the job directory on success, failure or cancel. A command-line tool submits, watches, cancels and evaluates jobs and chats with a finished adapter.

4 Methodology

Machine and model. One Apple M1 Pro MacBook Pro with 14 GPU cores and 16 GiB of unified memory, running macOS 26.2. The base is Qwen3.5-9B [17] quantized to 4 bits for MLX, 6.11 GB per the network's catalog [6]. Qwen3.5 interleaves gated-delta layers with attention layers; patch `0001` routes the gated-delta layers to their differentiable path in training mode.

Task and grading. Banking77 is a public intent-classification set with 77 labels [1]. We draw 2,000 training rows and 500 held-out rows, stratified by label with seed 0. Decoding is greedy, and a completion is correct only if it matches the label exactly. The base gets the full label list in its prompt; the adapter gets only the query. The pass threshold, adapter accuracy at least base accuracy plus 15 points, was fixed before the result that passed it existed. The training configuration was not: three rounds were graded on the same 500 held-out rows, with 1 epoch (79.0%, a run its own report marked invalid for two sampler gaps), 2 epochs (86.2%, below the threshold), and 3 epochs with the rate decayed across the third (91.2%). Every result uses one seed and one split, and each reported configuration ran once, with three exceptions: the EngineV2 serving test passed on its second full run (Section 5.2), the quiet-machine serving test on the earlier loop ran twice (Table 3), and the equivalence test was rerun after its first run was invalid. Banking77 stands in for a customer's labelled data; no real customer is involved.

Training configuration. Rank 8, 3 epochs, 6,000 micro-steps and 1,500 optimizer updates with gradient accumulation 4. The learning rate is 10^{-4} for epochs 1 and 2 and decays linearly to zero across epoch 3.

Serving load. Poisson arrivals at 0.2 requests per second, 200-token prompts and 64 output tokens, in four 10-minute arms. Arms come in pairs that replay the same arrival schedule and prompts, one with no training (baseline) and one with an attached job (candidate). Added wait is the candidate's time to first token minus the baseline's for the same request index. At this request size the engine on this Mac saturates at about 0.36 requests per second, so 0.2 is a moderate load, not a light one.

Table 1: Banking77 held-out accuracy. All rows hold the same adapter weights. The base sees all 77 labels in its prompt; the adapter sees only the query.

Grading path	Rows	Base	Adapter	Gain
Single-request loop (M0), trained here	500	73.8% (369)	91.2% (456)	+17.4
EngineV2 node (M1), same file	500	73.6% (368)	91.2% (456)	+17.6
Through the hub (M2), retrained, identical	100	69 of 100	90 of 100	+21

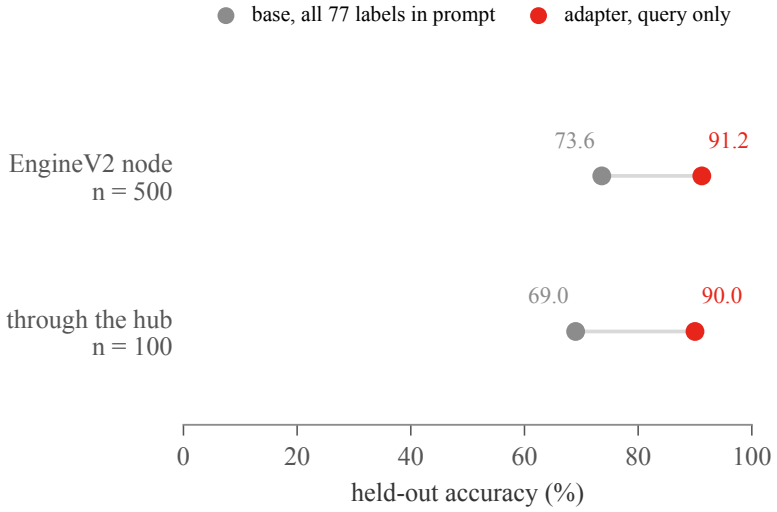


Figure 3: The adapter, given only the query, beats the same base given every label: on 500 held-out rows on the EngineV2 node, and on every fifth of those rows through the hub.

Evidence. Every run writes an evidence bundle with a sha256 manifest, the commit, a clean-tree flag, machine state samples every 500 ms, and per-piece events. A run with a sampler gap or a missing event is marked invalid.

5 Results

5.1 The adapter learns the task

Table 1 and Figure 3 give accuracy. All three rows grade the same weights. The adapter was trained once on the earlier single-request loop; the EngineV2 row grades that same file again on the EngineV2 node, and the end-to-end run of Section 5.5 trained a new adapter through the hub that is bit-identical to it in all 304 arrays. That identity is itself a result: training on two different engine loops, one of them through the hub with two restarts, produced the same weights. On the EngineV2 node the adapter answers 456 of 500 held-out rows from the query alone, against 368 of 500 for the base with every label listed. On the earlier loop it scored the same 456 of 500, against 369 of 500 for the base. The adapter's 44 misses are mostly neighbouring intents, such as two top-up intents or two transfer states, and no label accounts for more than 3 of them. Training on the idle Mac took 25,673 s (7.13 hours) for 420,789 training tokens, 16.4 tokens per second.

Training through the engine's idle calls computes exactly what the trainer computes alone. Over 200 micro-steps, every loss and all 304 adapter arrays were bit-identical between the two, and each

Table 2: Serving at 0.2 requests per second, 200 prompt and 64 output tokens, EngineV2 node, 226 requests per side. Baseline arms run no training; candidate arms run a job in the gaps. Added wait is the candidate's time to first token minus the paired baseline request's; the longest piece is taken per micro-step, over 114 micro-steps.

Measure	Baseline	Candidate
TTFT p50	1.44 s	1.41 s
TTFT p90	4.03 s	4.14 s
TTFT p99	8.95 s	8.87 s
Failed requests	0	0
Added wait p99 (max)	n/a	0.134 s (0.169 s)
Longest piece p99 (max)	n/a	0.145 s (0.147 s)
Optimizer updates per minute, slowest arm	n/a	1.3

of 26,200 idle calls ran one piece. This test passed after its gate was reduced to the MLX headroom check alone; its first run was invalid, because memory pressure from other applications made the full gate refuse every piece.

5.2 Serving cost

Two latency numbers must not be confused. Time to first token (TTFT) is the whole wait a request sees on this slow machine under load, with or without training. Added wait is what training adds on top of that. Table 2 gives both, with the serving-only baseline beside the candidate.

TTFT under this load is long because the Mac is slow, not because of training: the candidate's median and p99 are at or below the baseline's, and its p90 is 0.11 s above it. The added wait a request sees, measured against the same request in the paired baseline, has p99 0.134 s and a maximum of 0.169 s. The longest piece in any of the 114 micro-steps has p99 0.145 s and a maximum of 0.147 s, below the 0.25 s bound we set. The one-piece bound covers direct blocking only; added wait also carries the engine's other effects, and on earlier code it exceeded the longest piece (Table 3). The first run of this test failed that bound on two pieces in the process's first 30 seconds of training, so the node now runs a short training warm-up first; the warm-up took 4 micro-steps and 16.8 s. Figure 4 shows a real minute of the candidate arm, and Figure 5 the full distributions.

Piece granularity sets the direct-blocking bound. Table 3 traces it across the design's history on the same Mac. A whole micro-step took 2.46 to 7.80 s. Layer pieces brought the longest piece p99 to 1.04 s. Sublayer pieces first measured 0.83 s under heavy memory pressure, then 0.136 s and 0.130 s in two runs on a quiet machine. The wait a request actually gains follows piece length only loosely: on the single-request loop its p99 was 4.17 s under memory pressure and 0.35 s and 0.68 s in the two quiet runs, against 0.134 s on EngineV2. On the earlier loop, a single run's added-wait p99 varied about twofold between two runs; the 0.134 s is one run's value, not a bound.

5.3 Memory

With a job attached, MLX's peak memory rose by 0.296 GB over serving alone, against a 1.0 GB bound. Active memory rose by 258.6 MB, under the bound of adapter plus optimizer state plus 64 MiB, 260.6 MB. The base is not copied. The first measurement, before training was split into

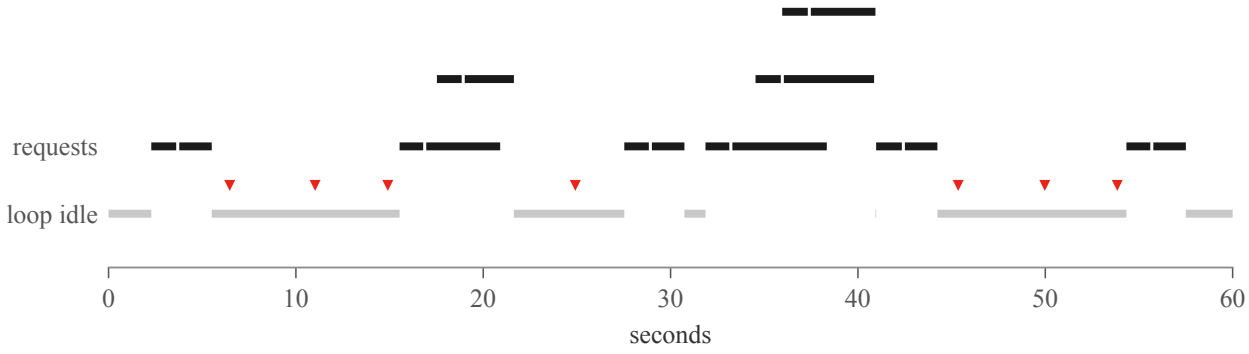
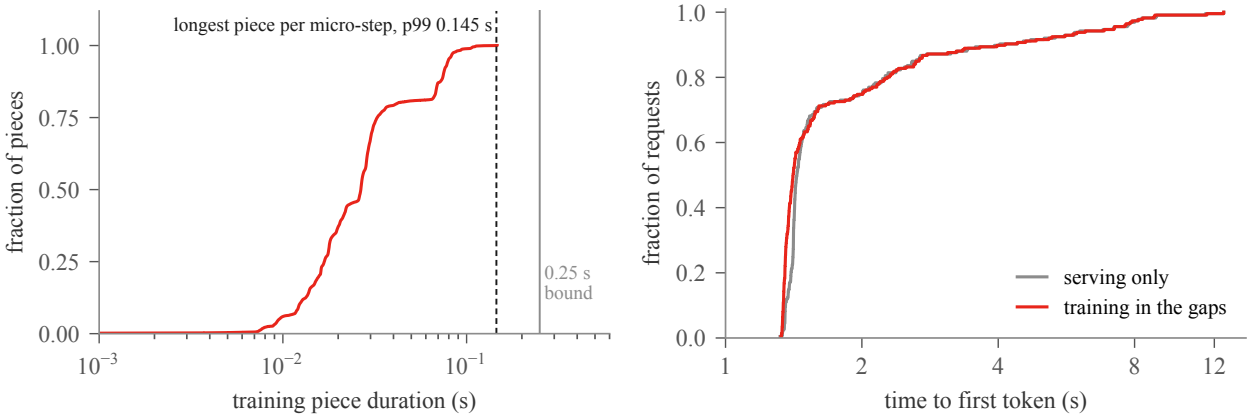


Figure 4: A real 60 s window of a candidate arm at 0.2 requests per second. Requests in flight are black, with a white tick at the first token; the loop's idle time between them is grey; red triangles mark completed training micro-steps. Every micro-step in the window completes inside an idle span.



(a) training piece durations (b) time to first token

Figure 5: Training stays out of serving's way. (a) All 14,934 pieces of the 114 completed micro-steps in the candidate arms; the longest piece per micro-step has p99 0.145 s, under the 0.25 s bound. (b) Time to first token for the 226 requests of the serving-only arms and the 226 of the candidate arms: the two distributions overlap from median to tail.

layer pieces, added 1.19 GB of peak memory; keeping one layer's activations at a time removed most of it.

5.4 Correctness of shared serving

With two adapters and base rows sharing forward passes, 20 of 20 prompts produced the same first 64 tokens as the same adapter on a separate model instance through upstream LoRA. Base requests produced identical output with and without a trainer attached, alone and beside adapter rows. Serving between training pieces left training unchanged: on a small test model, final adapter arrays agreed within 10^{-6}.

Table 3: Training's cost to waiting requests, by piece granularity. The first row is micro-step duration with no serving load, measured on battery power; the rest ran under the load of Table 2. Rows 1 to 5 ran on the earlier single-request loop, row 6 on EngineV2.

Granularity	Machine state	Longest piece p99	Added wait p99
Whole micro-step	battery, no load	2.46 to 7.80 s each	n/a
Layer pieces (66 per micro-step)	not recorded	1.04 s	0.56 s
Sublayer pieces	heavy memory pressure	0.83 s	4.17 s
Sublayer pieces	quiet, 5.8 GiB free	0.136 s	0.35 s
Sublayer pieces, repeat	quiet	0.130 s	0.68 s
Sublayer pieces, EngineV2	normal pressure	0.145 s	0.134 s

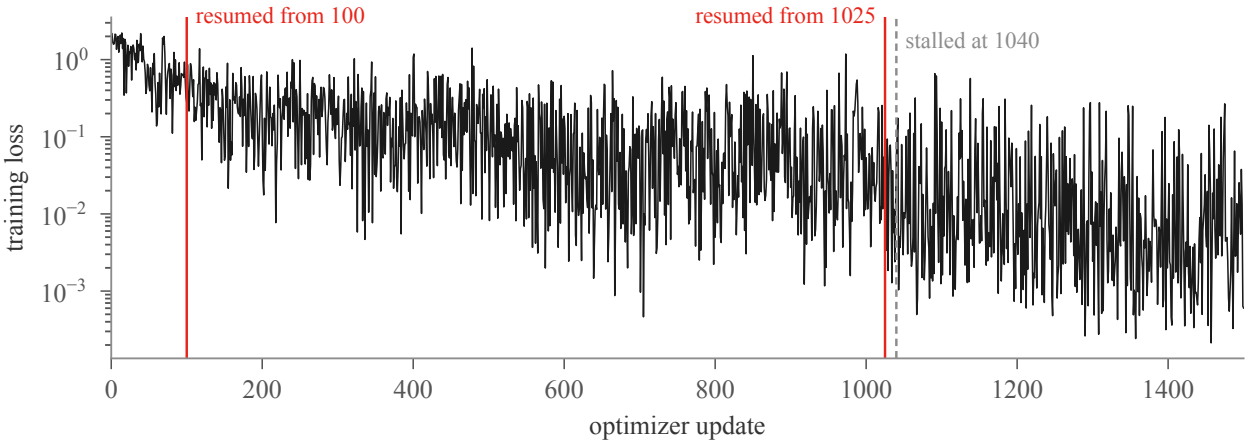


Figure 6: The overnight job's loss at each of its 1,500 optimizer updates, on a log scale. Both resumes restart from the last checkpoint, at 100 and 1025; the stall was found at update 1040. The finished adapter matches an uninterrupted run in all 304 arrays.

5.5 End to end, with two kills

The final test is one script: start the hub and a node, submit the 2,000-row job through the command-line tool at the configuration above, kill the node mid-job, restart it, and grade the adapter through the hub's chat endpoint. The run did not finish under that script alone; the operator stepped in, as described below. The run script sent SIGKILL once the hub held a checkpoint at update 100 and the job had reached update 110. Training resumed from update 100. Later the memory gate stalled training (Section 6). The operator stopped the run script without its cleanup, killed and restarted the node, and continued the run with a copy of the script that skips its startup steps and runs the same grading steps. Training resumed from update 1025, the last checkpoint, but stayed paused until the operator sent three chat requests through the hub at 00:52 UTC. The script's own resume check failed, because it requires exactly one resume and the run had two; the resume criterion in our specification, that each restart resumes from the last checkpoint and the job completes, held. The job completed all 1,500 of 1,500 updates and 1,500 losses. The final adapter is bit-identical, in all 304 arrays, to the adapter from the uninterrupted training run of Table 1's first row. Through the hub, the adapter answered 90 of 100 held-out rows (every fifth row of the 500) against 69 of 100 for the base. Figure 6 shows the job's loss with both resumes.

The three checkpoint writes the job snapshots record took 0.05 to 0.17 s, and the finished adapter

is 64.5 MB. After the job ended, none of 20 training queries appeared under the node's store or in its logs, and the hub recorded that the node had deleted the data; the same held for a smaller job that succeeded, one that was cancelled and one that failed. The run took 8 hours 29 minutes of wall-clock time, about 55 minutes of it paused.

6 Failures and limitations

The gate stall. The gate refuses training whenever macOS reports memory pressure other than normal. From about 23:58 UTC in the end-to-end run, macOS reported warning while 44% of memory was available, and swap-outs did not rise. Clearing the file cache and briefly allocating 2 GB left the warning in place. Killing the node returned pressure to normal at once; reloading the model raised it again. The first of three chat requests the operator sent through the hub returned pressure to normal, and training ran to the end. A paused node does not leave this state on its own, and the node did not log which check refused, so diagnosis read the gate's inputs from outside. An earlier equivalence run on the same Mac hit the same refusal under pressure from other applications. The rule as written is too strict for unattended use; a fix that pauses only while available memory is actually low, and logs the refusing check, is the first follow-up. The second kill in Section 5.5 was the operator's response to this stall, not a scheduled test.

Memory pressure sets the tail. On the same code, the longest piece p99 was 0.83 s when free memory was under 0.2 GiB and the Mac was swapping gigabytes per arm, and 0.136 s when the run was repeated on a quiet machine. On a shared consumer machine the owner's other applications decide the tail, and the gate's job is to stop training before that happens.

Thin update margin. Training advanced at 1.3 updates per minute in the slowest arm against a bound of 1.0, and one arm skipped 196 idle calls on swap-outs. At heavier load the gaps shrink and training slows; we did not measure where it stops.

Scope of the evidence. Every number comes from one 16 GiB M1 Pro and one base model, Qwen3.5-9B at 4 bits. No other Mac, memory size or base has been measured. Banking77 is a public benchmark standing in for a customer's data. The training configuration was chosen over three rounds graded on the same held-out rows, with one seed and one split, so 91.2% is an optimistic estimate of accuracy on unseen data. No graded adapter was trained under serving traffic: the accuracy runs trained on an otherwise idle Mac, the end-to-end run had no chat traffic beyond the operator's three requests, and under the serving load of Section 5.2 training reached only 13 and 15 optimizer updates per 10-minute arm. The serving tests ran at 0.2 requests per second against a ceiling of about 0.36 on this Mac, so the results say nothing about a busy node.

7 Discussion

The results answer the question for one machine. This Mac can fine-tune a useful adapter through its own inference engine's idle calls, on the weights it serves, on an otherwise idle Mac; in separate serving runs the cost to serving is hard to see in TTFT; and it can resume exactly after being killed. The measurements also show where it is fragile: the gate reads macOS's pressure signal, which can stay at warning long after memory is available.

The timing of this question is not only technical. When OpenAI's self-serve fine-tuning stops accepting new jobs [15], teams that customized small models there must move to other hosts or run open weights themselves. Whether customization stays available to small teams then depends on where open-weight fine-tuning can run. Fine-tuning that fits in the idle time of hardware people already own is one answer, and this paper measures its cost on the smallest machine we expect it to run on.

Two directions follow. The first is the gate: decide on available memory rather than the pressure level, and record every refusal. The second is breadth: more Macs, more memory sizes and other bases, with real chat traffic during training.

8 Conclusion

uli trains a customer's LoRA adapter inside a live inference engine on an M1 Pro laptop, in the engine's idle calls, on the 4-bit base the engine serves. The adapter, trained on an otherwise idle Mac, reached 91.2% of 500 held-out Banking77 rows against 73.6% for the base. Under load, training added p99 0.134 s to time to first token, and the longest training piece had p99 0.145 s; training adds 0.296 GB of peak memory; and a job killed twice resumed to a bit-identical adapter. The idea of co-serving is not new; what this adds is evidence that it works, and what it costs, on consumer Apple Silicon.

References

[1] Iñigo Casanueva, Tadas Temčinas, Daniela Gerz, Matthew Henderson, and Ivan Vulić. Efficient intent detection with dual sentence encoders. In *Proceedings of the 2nd Workshop on Natural Language Processing for Conversational AI*, 2020. arXiv:2003.04807.

[2] Jiaxuan Chen, Jianshu She, Ye Yuan, Rajat Ghosh, Karan Gupta, Qirong Ho, Xue Liu, and Oana Balmau. DeltaServe: Host-agnostic co-serving of inference and fine-tuning for LLMs. *arXiv:2607.28848*, 2026.

[3] Lequn Chen, Zihao Ye, Yongji Wu, Danyang Zhuo, Luis Ceze, and Arvind Krishnamurthy. Punica: Multi-tenant LoRA serving. *arXiv:2310.18547*, 2023.

[4] Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. QLoRA: Efficient finetuning of quantized LLMs. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2023. arXiv:2305.14314.

[5] Eigen Labs. Darkbloom. Model catalog, `https://api.darkbloom.dev/v1/models/catalog`, snapshot 2026-10-27 between 11:00 and 12:00 UTC; site `https://www.darkbloom.ai/`, 2026.

[6] Eigen Labs. Qwen3.5-9b MLX 4-bit artifact. Hugging Face `EigenLabs/Qwen3.5-9B-MLX-4bit-mtp`, revision 556dcba5, 2026.

[7] Awni Hannun, Jagrit Digani, Angelos Katharopoulos, and Ronan Collobert. MLX: Efficient and flexible machine learning on Apple silicon. `https://github.com/ml-explore/mlx`, 2023.

[8] Yongjun He, Yao Lu, and Gustavo Alonso. Deferred continuous batching in resource-efficient large language model serving. In *Proceedings of the 4th Workshop on Machine Learning and Systems (EuroMLSys)*, 2024. FineInfer.

[9] Yongjun He, Haofeng Yang, Yao Lu, Ana Klimovic, and Gustavo Alonso. Resource multiplexing in tuning and serving large language models. In *USENIX Annual Technical Conference (ATC)*, 2025. LLMStation. `https://www.usenix.org/conference/atc25/presentation/he-yongjun`.

[10] Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. LoRA: Low-rank adaptation of large language models. In *International Conference on Learning Representations (ICLR)*, 2022. arXiv:2106.09685.

[11] Layr Labs. mlx-swift-lm, layr-labs fork with the EngineV2 continuous-batching engine. `https://github.com/Layr-Labs/mlx-swift-lm`, MIT licence, commit aee6ebb6, 2026.

[12] Xinxin Liu, Jiaxin Li, Zibo Wang, Yun Ji, Zhangqi Zhu, Qing Hu, Zhibin Wang, Rong Gu, Sheng Zhong, and Chen Tian. LOCAL: Enabling learning on-device contiguously for agent LLMs. *arXiv:2608.15241*, 2026.

[13] Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. In *International Conference on Learning Representations (ICLR)*, 2019.

[14] Gabriele Oliaro, Xupeng Miao, Xinhao Cheng, Vineeth Kada, Mengdi Wu, Ruohan Gao, Yingyi Huang, Remi Delacourt, April Yang, Yingcheng Wang, Colin Unger, and Zhihao Jia. FlexLLM: Token-level co-serving of LLM inference and finetuning with SLO guarantees. *arXiv:2402.18789*, 2024. NSDI 2026.

[15] OpenAI. Deprecations. `https://developers.openai.com/docs/deprecations`, accessed 2026-10-30, 2026.

[16] Yifan Qiao, Shu Anzai, Shan Yu, Haoran Ma, Yang Wang, Miryung Kim, and Harry Xu. ConServe: Harvesting GPUs for low-latency and high-throughput large language model serving. *arXiv:2410.01228*, 2024.

[17] Qwen Team. Qwen3.5-9B. Hugging Face `Qwen/Qwen3.5-9B`, Apache-2.0, 2026.

[18] Ying Sheng, Shiyi Cao, Dacheng Li, Coleman Hooper, Nicholas Lee, Shuo Yang, Christopher Chou, Banghua Zhu, Lianmin Zheng, Kurt Keutzer, Joseph E. Gonzalez, and Ion Stoica. S-LoRA: Serving thousands of concurrent LoRA adapters. *arXiv:2311.03285*, 2023.

[19] Yuchen Zhang, Hanyue Du, Chun Cao, and Jingwei Xu. Loquetier: A virtualized multi-LoRA framework for unified LLM fine-tuning and serving. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2025.