

Efficient Multi-Tenant LoRA Serving for Encoder Models via Late-Fusion Batched Matrix Multiplication

Ashutosh Dwivedi

Independent Researcher
aashu.dwivedi@gmail.com

Keshav Shrikant

Independent Researcher
keshavshrikant@gmail.com

Jyotsna Rupaliya

Independent Researcher
jagjyotsna@gmail.com

Abstract

Low-Rank Adaptation (LoRA) has become a popular method to adapt pre-trained models to specific domains. Applied to encoders in a multi-tenant service, it lets each tenant specialize a shared model for classification, retrieval, or ranking. Existing multi-tenant LoRA serving systems target autoregressive decoders, and for encoders the available tooling supports only per-request adapter swapping, whose cost grows with the number of registered adapters. We observe that, unlike decoder generation, the encoder forward pass is compute-bound and dominated by the shared base projections, so per-tenant adaptation can be expressed as plain batched matrix multiplication with no custom kernels. We present LateFuse, a pure-PyTorch system that serves many different LoRA-adapted encoders in a single shared forward pass on one GPU. LateFuse fuses adapters late, in activation space: the base projections run once over the mixed-tenant batch, while per-sample low-rank deltas and per-tenant classification heads are applied as batched matrix multiplications. Tenant weights are never merged into the shared base, ruling out cross-tenant contamination by construction. Our evaluation shows that LateFuse’s serving latency stays flat as the adapter pool grows to tens of thousands on a single GPU, and that it achieves $5.6\text{--}21.2\times$ higher throughput than HuggingFace PEFT’s native mixed-batch API and up to $521\times$ over per-adapter swapping, while registering new adapters over $299\times$ faster.

1 Introduction

Enterprise NLP features, such as intent detection (Arora et al., 2024), ticket routing, and text classification, require low latency and predictable cost at high request volume. In multi-tenant environments, these workloads are typically served by a shared bidirectional encoder (Devlin et al., 2019) with tenant-specific task heads. Fine-tuning the encoder

parameters alongside the task head yields consistent accuracy gains over a frozen shared encoder (Tunstall et al., 2022; Jain et al., 2026). Serving an isolated, fully fine-tuned model per tenant, however, is infeasible at scale. At ≈ 1.14 GB per copy in float16 precision (bge-m3, 568M parameters), serving 20,000 tenants would require 22.8 TB of model weights (requiring 285 A100-80GB GPUs) and eliminate any opportunity for cross-tenant request batching. Low-Rank Adaptation (LoRA) (Hu et al., 2022) removes the storage barrier by expressing tenant-specific adaptations as low-rank updates to the shared base encoder, shrinking the per-tenant weight footprint to just 1.573 MB (rank 8, 24 layers), a $722\times$ reduction, at minimal accuracy cost (§2.1). Per-tenant adapters also keep customer data isolated: each is trained only on its own tenant’s data, avoiding the cross-tenant pooling that a single shared fine-tuned encoder would require, a separation that multi-tenant SaaS customers increasingly demand. The remaining challenge is **batched serving** across these heterogeneous adapters on a shared GPU.

Existing multi-tenant LoRA serving systems like Punica (Chen et al., 2024b), S-LoRA (Sheng et al., 2024), vLLM (Kwon et al., 2023), and LoRAX (Predibase, 2023) are tailored for autoregressive decoder workloads, where custom CUDA kernels and paged-memory pools optimize memory-bound, token-by-token generation over a KV cache. Meanwhile, on the encoder side, runtimes do not support mixed-tenant LoRA serving: vLLM lacks a runtime LoRA path for encoder models, TEI (Hugging Face, 2023) serves encoders without adapters, and PEFT’s (Mangrulkar et al., 2022) mixed-batch API is restricted to sequential adapter sub-batching and cannot route per-tenant heads (§3).

Encoder workloads exhibit fundamentally different dynamics from decoder generation: the bidirectional forward pass at sequence length $S \gg 1$ is compute-bound and dominated by the base model’s

projection weights. Consequently, the LoRA compute path accounts for only 9.0% of the total forward GPU time, meaning custom CUDA kernels yield an optimistic upper bound of single-digit performance gains (§5.2, Finding 6). This structural difference makes standard batched matrix multiplication (BMM) the appropriate primitive for encoder serving.

We present LateFuse,¹ a serving system built on this primitive. Rather than merging each $B_i A_i$ into W_0 , which forces per-sample weights to be applied serially, LateFuse keeps the base output $W_0 x$ and the delta $B_i A_i x$ as separate terms and sums them (*activation-space* fusion, not parameter-space merging; §4.1). It runs the shared base path once over a mixed-tenant batch of size B and dispatches the per-sample LoRA paths in parallel via `torch.bmm`, so the batch completes in a single forward pass regardless of how many distinct adapters it references. Per-tenant task heads (variable-size classifiers, regressors, retrieval re-rankers) are batched analogously via zero-padded BMM.

Contributions.

- We identify and characterize the *encoder multi-tenant LoRA serving* problem, and show that existing systems leave it unsolved (§3).
- We present late-fusion BMM serving, an architecture that batches arbitrary mixed-tenant encoder requests without weight merging and without custom CUDA kernels, in standard PyTorch, removing the custom-kernel build step, a common source of deployment friction on managed GPU platforms (§4, Appendix F).
- We validate the served workload: rank-8 LoRA on BAAI/bge-m3 retains 98.1% of full fine-tuning’s mean accuracy and improves over a frozen encoder, justifying per-tenant adaptation over a single shared model (§2.1).
- We empirically demonstrate that adapter count does not drive inference latency: scaling from 100 to 47,000 adapters at batch size 32 on a single A100 leaves p50 latency flat ($\leq 0.6\%$ drift, within seed noise) and p99 bounded (§5).

- We benchmark LateFuse against HuggingFace PEFT on identical hardware: **5.6–21.2** $\times$ higher throughput than PEFT’s native mixed-batch API and up to **521** $\times$ over per-adapter swapping, **>299** $\times$ faster cold-start deployment, isolating PEFT’s two pool-size costs: an $\mathcal{O}(N^2)$ one-time `add_adapter` registration and an $\mathcal{O}(N)$ per-request `set_adapter` swap (§5.3, Finding 7).

2 Background

2.1 LoRA for Encoder Models

LoRA (Hu et al., 2022) decomposes a weight update $\Delta W \in \mathbb{R}^{d \times d}$ into two low-rank matrices, $\Delta W = BA$ with $A \in \mathbb{R}^{r \times d}$, $B \in \mathbb{R}^{d \times r}$, and $r \ll d$. The single-sample forward pass becomes $h = W_0 x + \frac{\alpha}{r} B A x$, where W_0 is the frozen pre-trained weight and α is a scaling hyperparameter typically set equal to r . For an encoder of hidden size $d = 1024$ with $L = 24$ layers, applying LoRA at rank $r = 8$ to the query and value projections yields $2L \cdot (dr + rd) = 786,432$ trainable parameters per adapter, representing a $64\times$ reduction over full fine-tuning of those layers.

Our evaluation shows that on BAAI/bge-m3 (Chen et al., 2024a), the encoder used by every serving experiment in this paper, rank-8 LoRA retains 98.1% of full fine-tuning’s mean accuracy and beats a frozen encoder baseline on all seven public text-classification benchmarks (Table 1; mean 0.657 vs. 0.638, a 1.9 pp gain over frozen), at $1/722$ the trainable-parameter cost,² confirming that tenant adaptation is worth serving; the remaining barrier is *serving*: combining thousands of these adapters in a single forward pass.

2.2 The Multi-Tenant Serving Problem

Consider a classification service for N customers, each with a custom BAAI/bge-m3 adapter. The naive options all fail at scale: a full model copy per tenant forecloses cross-tenant batching, in-place weight merging ($W' = W_0 + B_i A_i$) serializes requests, and sequential per-tenant batching drives GPU utilization toward $1/N$. An effective solution must instead batch *heterogeneous* adapters (distinct

¹Code and benchmarks: <https://github.com/ashutoshrdwivedi/LateFuse>; pip install latefuse

²We replicate the protocol on paraphrase-mpnet-base-v2 the standard SetFit-MPNet backbone (Tunstall et al., 2022) built using the Sentence Transformer framework (Reimers and Gurevych, 2019), which reproduces SetFit-comparable accuracy and shows the same low-rank-vs-full-fine-tuning pattern (95.5% retained; Table 7, Appendix E).

Table 1: Per-task accuracy on BAAI/bge-m3, the serving model ($N=8$ examples per class, rank=8, mean over 10 seeds; protocol in Appendix E). *Frozen* fits only the classification head on the pre-trained encoder’s embeddings; Δ is LoRA minus Full FT. Scores are accuracy except Amazon-CF (MCC).

Dataset	Frozen	Full FT	LoRA-r8	Δ
SST-2	0.852	0.868	0.862	−0.006
SST-5	0.391	0.422	0.430	0.008
CR	0.878	0.895	0.886	−0.009
Amazon-CF	0.241	0.304	0.248	−0.056
Emotion	0.420	0.469	0.447	−0.022
EnronSpam	0.901	0.909	0.902	−0.007
AG News	0.783	0.824	0.827	0.003
Mean	0.638	0.670	0.657	−0.013
Params	0	568M	786K	1/722

weights, identical shapes) in a single forward pass, applying sample-specific linear operations concurrently without serialization.

3 Related Work

Multi-tenant LoRA serving for decoder LLMs. Punica (Chen et al., 2024b) and S-LoRA (Sheng et al., 2024) build custom CUDA/Triton kernels for the autoregressive prefill→decode pipeline, as do LoRAX (Predibase, 2023), CaraServe (Li et al., 2024), and dLoRA (Wu et al., 2024); none expose the bidirectional-encoder operations (attention, pooling, and task heads) that classification serving requires. vLLM (Kwon et al., 2023) adopts these kernels but its encoder classes do not: v0.23.0 rejects `enable_lora` for BAAI/bge-m3 at initialization, and multi-adapter classification serving remains an unmerged draft.³

Encoder and multi-adapter serving. HuggingFace TEI (Hugging Face, 2023) serves encoder/embedding models at high throughput but has no adapter support (an open feature request as of June 2026).⁴ Vishwanathan et al. (Vishwanathan et al., 2022) serve a shared sentence bi-encoder with per-tenant final-layer weights swapped per request, forcing each request to be single-tenant. PetS (Zhou et al., 2022), the closest prior system, batches multi-task encoder inference for Adapter (Houlsby et al., 2019), BitFit (Ben Zaken et al., 2022), and MaskBERT (Zhao et al., 2020) and related parameter-efficient methods on BERT/DistilBERT (Sanh et al., 2020). It does

³vLLM PR #24596, unmerged as of June 2026.

⁴TEI issue #788, open as of June 2026.

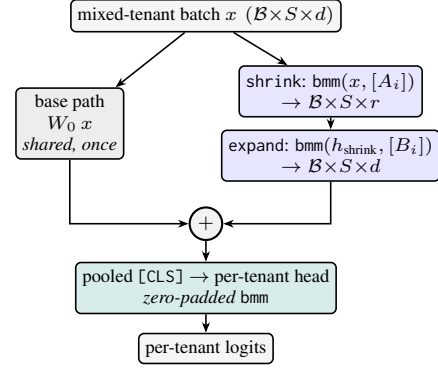


Figure 1: Late-fusion BMM forward pass in LateFuse. The shared base path runs once over the mixed-tenant batch while per-sample LoRA deltas are applied as batched matrix multiplications, so the whole batch completes in a single forward pass regardless of adapter count.

not implement LoRA’s low-rank updates or the per-sample low-rank BMM. LateFuse closes this gap: LoRA-native batching of arbitrary external adapters and their classification heads in a single encoder forward pass.

4 LateFuse: System Design

4.1 Late-Fusion Decomposition

The core insight (Figure 1) is that the LoRA forward pass $h = W_0x + B_iA_ix$ can be split into two *independent* computations:

$$h_{\text{base}} = W_0x \quad (1)$$

$$h_{\text{delta}} = B_i(A_ix) \quad (2)$$

$$h = h_{\text{base}} + h_{\text{delta}} \quad (3)$$

Equation 1 uses the same frozen W_0 for every sample in the batch and is computed by standard PyTorch linear layers. Equation 2 applies a *different* (A_i, B_i) pair to each sample i , which requires batched matrix multiplication rather than a broadcast multiply. Equation 3 is a zero-allocation element-wise addition.

For a batch of B samples with hidden size d and sequence length S :

$$h_{\text{shrink}} = \text{bmm}(x, [A_0, \dots, A_{B-1}]) \in \mathbb{R}^{B \times S \times r} \quad (4)$$

$$h_{\text{delta}} = \text{bmm}(h_{\text{shrink}}, [B_0, \dots, B_{B-1}]) \in \mathbb{R}^{B \times S \times d} \quad (5)$$

where $[A_0, \dots, A_{B-1}] \in \mathbb{R}^{B \times d \times r}$ is the stack of per-sample shrink matrices, assembled from the adapter store at request time. Each bmm call is a

single GPU kernel launch, processing all $\mathcal{B}$ samples simultaneously regardless of how many distinct adapters they reference.

Unlike Punica and S-LoRA, whose custom kernels group inputs by adapter id for the prefill→decode pipeline, `torch.bmm` applies a distinct (A_i, B_i) to every item of the dense (S, d) encoder forward pass (§3).

4.2 Batched Classification Heads

In the multi-tenant classification setting, each tenant trains a logistic-regression (LR) head on top of the encoder’s pooled [CLS] representation. Tenants also differ in their class count c_i , so co-batched heads have mismatched shapes and cannot be stacked directly.

LateFuse serves all heads in a single BMM call. Given a batch of pooled representations $p \in \mathbb{R}^{\mathcal{B} \times 1 \times d}$ and per-tenant weight matrices $W_i^{\text{cls}} \in \mathbb{R}^{c_i \times d}$ and bias vectors $b_i^{\text{cls}} \in \mathbb{R}^{c_i}$, we zero-pad all weight matrices to the maximum label count $c_{\max} = \max_i c_i$ in the batch and form a stacked tensor $W^{\text{cls}} \in \mathbb{R}^{\mathcal{B} \times c_{\max} \times d}$. The logit computation is:

$$\text{logits} = \text{bmm}(p, W^{\text{cls}^\top}) + b^{\text{cls}} \in \mathbb{R}^{\mathcal{B} \times 1 \times c_{\max}} \quad (6)$$

Callers slice $[:, :, : c_i]$ per sample before softmax, so the padded entries never enter the normalization and leave each tenant’s distribution unchanged.

4.3 Implementation

Pre-allocated buffers. A naive implementation allocates a fresh output tensor for each of the $2|\mathcal{M}|L$ bmm calls per forward pass, adding several hundred microseconds of allocator overhead. LateFuse instead pre-allocates two reusable buffers (`out_A`, `out_B`), passed via each bmm’s `out=` argument; the two suffice for all layers because each expand consumes `out_A` before the next shrink overwrites it.

Adapter store and batch assembly. At service startup, LateFuse pre-loads all tenant LoRA adapters into GPU memory as an `AdapterStore`: a dictionary mapping adapter IDs to `LoraWeight` objects holding $W_A \in \mathbb{R}^{L \times d \times r}$ and $W_B \in \mathbb{R}^{L \times r \times d}$, stored in inference-transposed layout so bmm requires no runtime transpose. For a batch of adapter IDs, the `BatchAssembler` gathers each adapter’s per-layer W_A/W_B slices into per-layer Python lists. This CPU-side gather (a loop over samples, layers, and target modules) is the dominant per-request

overhead before the GPU forward pass (§5.2). The lists are concatenated lazily inside the forward, one `torch.cat` per layer before its bmm, so this cost is attributed to forward rather than assembly time.

4.4 Complexity Analysis

Memory. Total GPU memory is $\mathcal{O}(d^2L + NrdL)$: a tenant-independent base plus $\mathcal{O}(rdL)$ per adapter. Finding 4 (§5.2) reports the per-adapter footprint and per-GPU ceilings.

Inference time. A single forward pass is independent of N , depending only on batch size $\mathcal{B}$, sequence length S , and depth L ; adapter count affects only CPU-side batch assembly, $\mathcal{O}(\mathcal{B} \cdot L \cdot |\mathcal{M}|)$ for $|\mathcal{M}|$ target modules. We validate both empirically in §5.2 (Findings 1 and 5).

5 Experiments

5.1 Setup

Base model. Primary experiments use BAAI/bge-m3, a 24-layer XLM-RoBERTa-derived encoder with hidden size $d = 1024$, 16 attention heads, and 568M parameters.

LoRA configuration. We apply LoRA to the query and value projection matrices of each attention layer (`target_modules = [query, value]`). We sweep rank $r \in \{4, 8, 16, 32\}$ with $\alpha = r$ (no scaling), matching the configuration used by our PEFT-equivalence numerical tests.

Hardware and software. Primary experiments run on a single NVIDIA A100-SXM4-80GB GPU on a cloud-managed node in float16 precision, with PyTorch 2.11.0+cu128 (CUDA 12.8), transformers 4.57.6, and peft 0.19.1.

Benchmark protocol. We measure end-to-end latency (batch assembly + encoder forward pass + LR head) at sequence length $S=128$ using 50/200 warm-up/measured iterations for LateFuse; the PEFT baselines use 10/50, since PEFT-grouped reaches ~ 80 s/iter at the largest configuration. GPU-side work is timed with reusable `torch.cuda.Event` pairs and synchronized once per iteration; CPU-side assembly is timed with `time.perf_counter` between identical synchronization points. All configurations are evaluated across 5 independent random seeds, sweeping adapter count $N \in \{100, 1,000, 5,000, 10,000, 20,000, 40,000, 47,000\}$, batch size $\mathcal{B} \in \{8, 16, 32, 64, 128\}$, and rank $r \in$

Table 2: Serving latency (mean $\pm$ s.d. over 5 seeded runs), throughput, and peak GPU memory (setup in §5.1). The top block sweeps adapter count, the middle batch size, and the bottom LoRA rank.

N	B	r	p50 (ms)	p99 (ms)	QPS (s/s)	GPU (GB)
100	32	8	40.1 $\pm$ 0.4	46.5 $\pm$ 8.0	772	1.43
1,000	32	8	40.1 $\pm$ 0.3	44.7 $\pm$ 3.9	781	2.87
5,000	32	8	39.8 $\pm$ 0.1	41.0 $\pm$ 0.3	786	9.24
10,000	32	8	39.8 $\pm$ 0.1	41.2 $\pm$ 1.4	782	17.21
20,000	32	8	39.8 $\pm$ 0.1	41.3 $\pm$ 0.8	792	33.15
40,000	32	8	39.8 $\pm$ 0.1	41.1 $\pm$ 0.5	789	65.03
47,000	32	8	39.9 $\pm$ 0.1	40.8 $\pm$ 1.2	788	76.18
1,000	8	8	22.1 $\pm$ 0.1	22.7 $\pm$ 0.1	356	2.77
1,000	16	8	26.1 $\pm$ 0.2	28.4 $\pm$ 1.0	591	2.80
1,000	64	8	75.4 $\pm$ 0.1	220.7 $\pm$ 19.8	801	3.00
1,000	128	8	147.4 $\pm$ 0.4	306.9 $\pm$ 29.3	792	3.25
1,000	32	4	40.4 $\pm$ 0.4	44.8 $\pm$ 0.9	768	2.08
1,000	32	16	40.1 $\pm$ 0.3	42.9 $\pm$ 2.0	772	4.44
1,000	32	32	39.9 $\pm$ 0.1	42.7 $\pm$ 3.7	784	7.74

{4, 8, 16, 32}. Each configuration samples mixed-tenant batches uniformly at random from the adapter pool.

Adapters and correctness. Benchmarks use synthetic LoRA weights to isolate system overhead. Unit tests enforce numerical equivalence to reference HuggingFace and PEFT models within $\text{atol} = \text{rtol} = 10^{-4}$ in fp32, across five routing and padding scenarios and both BERT and XLM-RoBERTa position-id conventions.

Baselines. We compare against HuggingFace PEFT (Mangrulkar et al., 2022) in four configurations described in §5.3, including its native mixed-batch API (adapter_names), which does not route per-request classification heads,⁵ a requirement of our workload. We omit vLLM and LoRAX from quantitative comparison: neither serves encoder LoRA (§3). The timing asymmetry (PEFT omits the per-tenant head) makes the comparison conservative in PEFT’s favor.

5.2 Latency and Throughput Scaling

Table 2 reports p50/p99 latency and throughput across a one-factor-at-a-time sweep of this space, varying N , B , and r in turn around the $N=1,000$, $B=32$, $r=8$ operating point. We report all our findings below.

Finding 1: Adapter count does not drive latency. Scaling from 100 to 47,000 adapters at $B=32$, $r=8$ shifts p50 latency by only 0.6%, within

⁵<https://github.com/huggingface/peft/issues/1960>

one between-seed s.d. (Table 2), a $470\times$ increase in served customers at negligible latency cost, sustained to the per-GPU memory ceiling (Finding 4). The GPU-side forward pass time is constant at 26.0 ms across the entire sweep, and the only N -dependent component is the CPU-side batch assembler, whose share stays at or below 36.9% of total latency, empirically validating the $\mathcal{O}(1)$ -in- N scaling of §4.4 (Figure 2, Appendix A).

Finding 2: Batch size drives throughput up to saturation at $B=64$. Throughput scales from 356 samples/sec at $B=8$ to 801 samples/sec at $B=64$, then plateaus: $B=128$ delivers no further gain (792 samples/sec). The shared-base encoder forward becomes GPU-bound at $B=64$; beyond $B=32$, larger batches mainly widen tail latency (Finding 5), making $B=32$ the operating point under strict p99 SLAs and $B=64$ the throughput-optimal choice when tails are tolerable.

Finding 3: LoRA rank is a free latency knob in the small-rank regime. At $N=1,000$, $B=32$, p50 latency is flat across $r \in \{4, 8, 16, 32\}$, and tail latency shows no resolvable rank trend (the 5% p99 spread is comparable to the between-seed s.d.; Table 2). While $r \ll d$, the low-rank path adds only $2r/d$ of a base projection’s arithmetic ($\approx 1\%$ of the full encoder forward), so the per-batch GPU forward stays at ≈ 26.0 ms across all four ranks; the knob ceases to be free once $2r$ approaches d . Within this regime, rank is a pure quality–memory tradeoff: doubling it leaves forward latency unchanged but doubles the per-adapter footprint, halving the per-GPU tenant ceiling (Finding 4).

Finding 4: Predictable memory footprint and per-GPU ceiling. Per-adapter cache memory measures 1.573 MB at $r=8$ in fp16, matching the analytic formula $2 \cdot M \cdot L \cdot d \cdot r \cdot \text{dtype}$ exactly. On an 80 GB A100, after reserving 2.26 GB for base-model parameters and activation buffers, this yields a clean empirical ceiling of **47,000** adapters at $r=8$ (the largest pool we verified, occupying 76.18 GB), **23,500** at $r=16$, and **11,750** at $r=32$. Our $N=50,000$ runs OOM’d at all batch sizes.

Finding 5: Batch assembly is the residual bottleneck and drives tail latency. The CPU-side batch-assembly path (gather per-sample LoRA slices into per-layer lists, pad the LR head, host-to-device copy) consumes 17.4% of iteration time at $B=8$ and **43.0%** at $B=128$, scaling linearly with B while the encoder forward saturates at $B=64$.

This same path drives tail latency: the p99/p50 ratio is $1.11\times$ at $\mathcal{B}=32$ but reaches $2.93\times$ at $\mathcal{B}=64$, with profiling localizing the spike to occasional Python/GC stalls during assembly; the GPU forward exhibits no such variance.

The bottleneck is CPU-side, so as an alternative we evaluate moving assembly onto the GPU in a separate benchmark ($N=2,000$ adapters, 5 seeds): the adapter cache held in one contiguous GPU tensor and gathered with a single `torch.index_select` per module, still without any custom kernel.

Since assembly’s share grows as the encoder gets cheaper, we construct an adversarial case: a high-QPS regime on a small encoder, all-MiniLM-L6-v2 (Reimers and Gurevych, 2019), whose forward is short enough that CPU-side assembly dominates each iteration. There assembly consumes 58.0–65.3% of latency at $\mathcal{B} \geq 64$ and caps single-stream throughput at 5,192 samples/s; the GPU-resident assembler lifts that to 15,315 samples/s, makes assembly negligible, and stabilizes the tail (Appendix C). On bge-m3 the same change gives up to $1.80\times$ speedup. Both assemblers ship in our library; however, the results in §5 use the CPU path.

Finding 6: Custom LoRA kernels offer only single-digit upside on encoders. Could custom kernels (Punica’s SGMV, S-LoRA’s MBGMV) replacing our `torch.bmm` help? Three measurements at $\mathcal{B}=32$, $N=1,000$, $r=8$ bound the upside to single digits. First, the *analytic FLOP ratio* per layer is $3d/r=384$, so even an instantaneous LoRA kernel cuts at most 0.26% of FLOPs. Second, a *profiler trace* (self CUDA time; the LoRA shrink/expand isolated by an apply-LoRA on/off toggle) charges only **4.94%** of forward GPU time to the LoRA `bmm`, versus **50.6%** for the base projections. Third, a *no-LoRA ablation* skipping the entire LoRA path (gather, shrink/expand, merge) saves **2.4 ms of 26.4 ms (9.0%)**; here the gather and merge, not the `bmm`, dominate the gap above the 4.94% kernel cost. Custom fused kernels touch only this path, so 9.0% is the hard ceiling.

5.3 Comparison with Baselines

We compare against HuggingFace PEFT in four configurations, all attaching N adapters via `add_adapter`. *Grouped* buckets a mixed batch by adapter id, paying one `set_adapter` and forward per distinct adapter present. *Mixed* is PEFT’s

native mixed-batch API (`adapter_names`): one shared base forward with a per-layer loop over the adapters present in the batch (§5.1). *Homogeneous*, PEFT’s friendliest case, uses single-tenant batches: one `set_adapter` and one batched forward each. *Base* is the bare model with no adapters (N -independent).

Table 3: Latency/throughput vs. PEFT in the four configurations of §5.3 ($r=8$; PEFT measured at $\mathcal{B} \in \{8, 32, 128\}$). Speedup rows are QPS ratios (QPS column) of LateFuse over the like-for-like *mixed* baseline.

System	$N = 100$		$N = 1,000$	
	p50 (ms)	QPS	p50 (ms)	QPS
Base ($\mathcal{B}=8$)	11	717.8	11	717.8
Base ($\mathcal{B}=32$)	17	1837.0	17	1837.0
Base ($\mathcal{B}=128$)	61	2077.5	61	2077.5
PEFT grouped ($\mathcal{B}=8$)	605	13.6	5,287	1.5
PEFT grouped ($\mathcal{B}=32$)	2107	15.2	21,225	1.5
PEFT grouped ($\mathcal{B}=128$)	5550	23.1	79,979	1.6
PEFT mixed ($\mathcal{B}=8$)	140	56.3	476	16.8
PEFT mixed ($\mathcal{B}=32$)	371	86.6	762	41.9
PEFT mixed ($\mathcal{B}=128$)	936	137.2	1,829	69.8
PEFT homog. ($\mathcal{B}=8$)	77	103.0	656	12.1
PEFT homog. ($\mathcal{B}=32$)	82	386.8	658	48.5
PEFT homog. ($\mathcal{B}=128$)	127	999.5	708	180.6
LateFuse ($\mathcal{B}=8$)	22.1	359	22.1	356
LateFuse ($\mathcal{B}=32$)	40.1	772	40.1	781
LateFuse ($\mathcal{B}=128$)	152.3	774	147.4	792
Speedup_{mixed} ($\mathcal{B}=8$)	6.4$\times$		21.2$\times$	
Speedup_{mixed} ($\mathcal{B}=32$)	8.9$\times$		18.6$\times$	
Speedup_{mixed} ($\mathcal{B}=128$)	5.6$\times$		11.3$\times$	

Finding 7: PEFT does not scale to large tenant pools. *Homogeneous* batching, PEFT’s best case, is competitive at $N=100$, even edging LateFuse at $\mathcal{B}=128$ (999.5 vs. 774 samples/sec; Table 3). This edge is a small-pool artifact: at the same $\mathcal{B}=128$, growing the pool to 1,000 collapses it from 999.5 to 180.6 samples/sec (up to $8.5\times$ at smaller batches), leaving LateFuse 4.4–29.4 $\times$ faster at $N=1,000$. This isolates the collapse to `set_adapter`, which rescans the per-layer `ModuleDict` on every call, an $\mathcal{O}(N)$ cost by construction. *Grouped* fares worse: mixed batches rarely share an adapter, so LateFuse leads by 51–521 $\times$ at $\mathcal{B}=32$. PEFT’s native *mixed* API sits between the two: it shares the base forward, but loops over the batch’s adapters at every LoRA layer, and its per-call validation scales with N : at $\mathcal{B}=8$, where distinct adapters per batch stay essentially fixed, p50 still grows from 140.2 to 475.9 ms as N grows $100 \rightarrow 1,000$, leaving LateFuse 5.6–21.2 $\times$ faster across the measured grid. Registration scales worse still: `add_adapter` rescans that same

Table 4: LateFuse across encoders and GPU classes (protocol: §5.1). $\Delta p50$: p50 at each configuration’s tenant ceiling versus $N=1,000$. *Spread*: max–min p50 over the sweep. *vs. PEFT*: throughput ratio over PEFT’s mixed-batch API, on cells where it runs. [†]the configuration of §5.2–5.3.

Encoder	GPU	Params	Ceiling N ($r=8$)	$\Delta p50$ at ceil.	Spread (sweep)	vs. PEFT mixed
bge-m3 [†]	A100-80GB	568M	47,000	+0.65%	3.31%	5.6–21.2×
ELECTRA-large	A100-80GB	334M	51,000	−0.16%	1.36%	6.2–20.9×
DeBERTa-v2-xlarge	A100-80GB	885M	64,000	+0.43%	1.16%	2.4–7.1×
XLM-RoBERTa-XL	A100-80GB	3.5B	12,000	−0.38%	0.70%	2.3–19.5×
bge-m3	L40S-48GB	568M	28,000	+1.29%	1.60%	3.1–32.7×

ModuleDict as the pool grows, an $\mathcal{O}(N^2)$ cumulative cost, so loading the pool takes 14.4 s at $N=100$ but 1180.7 s at $N=1,000$, versus 0.05 s and 0.29 s for LateFuse’s linear-time AdapterStore preload, a 299–4071× cold-start reduction. We cap the sweep at $N=1,000$, but both costs grow with the pool by construction, so the throughput and load-time gaps above are conservative lower bounds that only widen, rendering PEFT impractical for large tenant pools.

5.4 Generalization Across Models and Hardware

To empirically verify that $\mathcal{O}(1)$ -in- N latency scaling is a property of the decomposition (§4.1) rather than an artifact of a specific encoder or GPU, we replicate the sweep protocol of §5.1 (5 seeds, float16, sequence length 128) on 3 additional encoders and a second GPU class: ELECTRA-large (Clark et al., 2020) (replaced-token detection); DeBERTa-v2-xlarge (He et al., 2021) (disentangled attention)⁶; XLM-RoBERTa-XL (Goyal et al., 2021) (36 layers, $d=2560$); and bge-m3 on an L40S-48GB.

To serve these models we use a generic attachment wrapper that binds LateFuse to any HuggingFace encoder exposing standard projection modules, rather than the BERT-family encoder reimplementation used in §5.2–5.3; we also enable PyTorch’s `expandable_segments` to avoid allocator fragmentation. Neither change applies to bge-m3/A100, including its row in Table 4, and neither alters the decomposition of §4.1.

Core scaling invariants generalize across models and GPUs. As summarized in Table 4, all headline properties replicate across all 5 configurations.

⁶Adapters target value projections only, as DeBERTa-v2 shares the query projection with batch-shared relative-position embeddings. Since fewer modules are wrapped, PEFT incurs less mixed-batch overhead, making our reported speedup conservative.

Growing the resident pool from $N=1,000$ to each configuration’s memory ceiling (up to 64× more tenants) moves p50 latency by at most 1.29%, negative on two configurations, consistent with zero effect and seed noise (Finding 1). Sweeping $r \in \{4, 8, 16, 32\}$ leaves p50 flat within 1.19% across all 5 configurations (Table 5, Appendix B), confirming that rank remains a cost-free quality–memory knob (Finding 3). Tenant ceilings range from 12,000 adapters for XLM-RoBERTa-XL to 64,000 for DeBERTa-v2-xlarge and track per-adapter footprint exactly (5.90 MB and 1.18 MB), each validated by an OOM at the subsequent probe point (Finding 4). Finally, throughput remains multi-fold above PEFT’s mixed-batch API in every case (Finding 7). Findings 2, 5, and 6 concern per-batch execution rather than pool size, so tenant scaling does not affect them.

This shows that latency generalizes across models and hardware; capacity is set by available memory. XLM-RoBERTa-XL, the largest model tested, sustains the smallest pool, and bge-m3 fits 28,000 adapters on the L40S against 47,000 on the A100. The Spread column in Table 4 reflects seed noise rather than systematic drift.

6 Conclusion

Multi-tenant NLP encoder features increasingly require per-tenant adaptation: it improves accuracy over a shared frozen encoder while keeping each tenant’s data and weights isolated. LoRA provides this cheaply, for serving this at scale we presented LateFuse, which serves many LoRA adapters in one shared forward pass via late-fusion BMM. Gathering per-tenant deltas and classification heads into batched matrix multiplications, keeps latency flat as the adapter pool grows to tens of thousands on a single GPU. A per-tenant encoder LoRA feature can therefore run as a shared service that amortizes GPU cost across tenants and scales out as stateless, tenant-sharded replicas (Appendix F).

Limitations

Uniform rank within a batch. All adapters in a batch share one rank r and one set of target modules, so a heterogeneous fleet must pad every adapter to the largest rank present. A padded batch matches a uniform one at the same rank to within 0.21% in latency (Appendix D). Padding costs memory rather than latency, and per-rank stores recover it for a small gather-time cost. A masked-BMM variant that skips padded lanes remains future work.

Pre-loaded adapter cache. LateFuse holds all adapters in GPU memory permanently. Workloads whose adapter population exceeds a single GPU’s capacity require an LRU eviction policy with CPU offload, or paged adapter storage as in S-LoRA’s unified paging (Sheng et al., 2024); either is a natural extension.

Batch assembler tradeoff. The default BatchAssembler keeps adapters in a per-adapter store, so a tenant can be admitted or evicted without touching the others, but its CPU-side gather scales with batch size and becomes the bottleneck as the encoder gets cheaper (§5.2). The GPU-resident assembler removes that cost, at the price of a contiguous tensor whose capacity is fixed at construction: continuous admission needs slot reservation and a grow strategy, which we did not measure. We ship both assemblers; §5 reports the CPU path.

Single-node evaluation. All latency and throughput results are measured on one GPU. The architecture targets horizontal scale-out via stateless, tenant-sticky replicas, with a reference Kubernetes deployment provided (Appendix F); cluster-level throughput and behavior under skewed (hot-tenant) traffic remain to be evaluated.

Evaluation scope. Accuracy is averaged over 10 random seeds (Tables 7 and 1); the remaining per-seed variance is non-trivial on the smaller few-shot tasks. The LateFuse serving sweeps report the mean over 5 seeded runs per configuration; the PEFT baselines are single runs, adequate for the large multiplicative gaps ($\geq 5.6\times$ on mixed-tenant traffic) they support (§5.3).

Ethics Statement

This work addresses inference-time efficiency for multi-tenant NLP systems. We identify no specific

ethical concerns beyond those inherent to the underlying models (e.g., bias in pre-trained encoders). The system is intended for legitimate multi-tenant SaaS deployments. Adapter isolation is enforced by design: per-tenant weights are never merged into the shared base model, eliminating cross-tenant contamination risks present in weight-merging approaches.

Acknowledgements

We thank the organizers and reviewers of EMNLP 2026 for the opportunity to present this work.

The authors acknowledge the use of generative AI tools to assist with code development and manuscript editing. Specifically, they were used to assist with development of proof-of-concept code and polish the language of the manuscript. All AI-assisted code and text were reviewed and verified by the authors, who remain responsible for the final content and all reported results.

References

- Gaurav Arora, Shreya Jain, and Srujana Merugu. 2024. [Intent detection in the age of LLMs](#). In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing: Industry Track*, pages 1559–1570, Miami, Florida, US. Association for Computational Linguistics.
- Elad Ben Zaken, Yoav Goldberg, and Shauli Ravfogel. 2022. [BitFit: Simple parameter-efficient fine-tuning for transformer-based masked language-models](#). In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pages 1–9.
- Jianlyu Chen, Shitao Xiao, Peitian Zhang, Kun Luo, Defu Lian, and Zheng Liu. 2024a. [M3-embedding: Multi-linguality, multi-functionality, multi-granularity text embeddings through self-knowledge distillation](#). In *Findings of the Association for Computational Linguistics: ACL 2024*, pages 2318–2335, Bangkok, Thailand. Association for Computational Linguistics.
- Lequn Chen, Zihao Ye, Yongji Wu, Danyang Zhuo, Luis Ceze, and Arvind Krishnamurthy. 2024b. [Punica: Multi-tenant LoRA serving](#). In *Proceedings of Machine Learning and Systems*, volume 6.
- Kevin Clark, Minh-Thang Luong, Quoc V. Le, and Christopher D. Manning. 2020. [ELECTRA: Pre-training text encoders as discriminators rather than generators](#). In *ICLR*.
- Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. [BERT: Pre-training of](#)

- deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186.
- Naman Goyal, Jingfei Du, Myle Ott, Giri Anantharaman, and Alexis Conneau. 2021. **Larger-scale transformers for multilingual masked language modeling**. In *Proceedings of the 6th Workshop on Representation Learning for NLP (RepL4NLP-2021)*, pages 29–33, Online. Association for Computational Linguistics.
- Pengcheng He, Xiaodong Liu, Jianfeng Gao, and Weizhu Chen. 2021. **DeBERTa: Decoding-enhanced BERT with disentangled attention**. In *International Conference on Learning Representations*.
- Neil Houlsby, Andrei Giurgiu, Stanisław Jastrzebski, Bruna Morrone, Quentin de Laroussilhe, Andrea Gesmundo, Mona Attariyan, and Sylvain Gelly. 2019. **Parameter-efficient transfer learning for NLP**. In *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pages 2790–2799.
- Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2022. **LoRA: Low-rank adaptation of large language models**. In *International Conference on Learning Representations*.
- Hugging Face. 2023. Text embeddings inference. <https://github.com/huggingface/text-embeddings-inference>. Accessed: June 17, 2026.
- Prateek Jain, Shabari S Nair, Ritesh Goru, Prakhar Agarwal, Ajay Yadav, Yoga Sri Varshan Varadharajan, and Constantine Caramanis. 2026. **Succeeding at scale: Automated dataset construction and query-side adaptation for multi-tenant search**. *Preprint*, arXiv:2601.04646.
- Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. 2023. **Efficient memory management for large language model serving with PagedAttention**. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pages 611–626.
- Suyi Li, Hanfeng Lu, Tianyuan Wu, Minchen Yu, Qizhen Weng, Xusheng Chen, Yizhou Shan, Binhang Yuan, and Wei Wang. 2024. **CaraServe: CPU-assisted and rank-aware LoRA serving for generative LLM inference**. *Preprint*, arXiv:2401.11240.
- Sourab Mangrulkar, Sylvain Gugger, Lysandre Debut, Younes Belkada, Sayak Paul, and Benjamin Bossan. 2022. **PEFT: State-of-the-art parameter-efficient fine-tuning methods**. <https://github.com/huggingface/peft>. Accessed: June 17, 2026.
- Predibase. 2023. LoRAX: Serve hundreds of fine-tuned models on a single GPU. <https://github.com/predibase/lorax>. Accessed: June 17, 2026.
- Nils Reimers and Iryna Gurevych. 2019. **Sentence-BERT: Sentence embeddings using siamese BERT-networks**. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 3982–3992.
- Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. 2020. **Distilbert, a distilled version of bert: smaller, faster, cheaper and lighter**. *Preprint*, arXiv:1910.01108.
- 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. 2024. **S-LoRA: Scalable serving of thousands of LoRA adapters**. In *Proceedings of Machine Learning and Systems*, volume 6.
- Lewis Tunstall, Nils Reimers, Unso Eun Seo Jo, Luke Bates, Daniel Korat, Moshe Wasserblat, and Oren Pereg. 2022. **Efficient few-shot learning without prompts**. *arXiv preprint arXiv:2209.11055*.
- Asha Vishwanathan, Rajeev Warrier, Gautham Vadakkekara Suresh, and Chandra Shekhar Kandpal. 2022. **Multi-tenant optimization for few-shot task-oriented FAQ retrieval**. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing: Industry Track*, pages 188–197, Abu Dhabi, UAE. Association for Computational Linguistics.
- Bingyang Wu, Ruidong Zhu, Zili Zhang, Peng Sun, Xuanzhe Liu, and Xin Jin. 2024. **dLoRA: Dynamically orchestrating requests and adapters for LoRA LLM serving**. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 911–927, Santa Clara, CA. USENIX Association.
- Mengjie Zhao, Tao Lin, Fei Mi, Martin Jaggi, and Hinrich Schütze. 2020. **Masking as an efficient alternative to finetuning for pretrained language models**. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 2226–2241.
- Zhe Zhou, Xuechao Wei, Jiejing Zhang, and Guangyu Sun. 2022. **PetS: A unified framework for parameter-efficient transformers serving**. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 489–504.

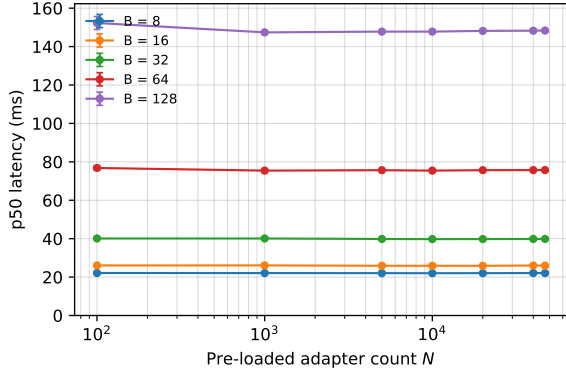


Figure 2: p50 latency as a function of pre-loaded adapter count at five batch sizes (rank = 8, sequence length = 128; setup in §5.1). Curves are flat across $N \in [100, 47,000]$, empirically confirming the $\mathcal{O}(1)$ -in- N scaling property of LateFuse.

Table 5: p50 latency (ms) at each LoRA rank ($B=32$, $N=1,000$, mean over 5 seeds). *Spread* is (max – min)/min across the four ranks.

Configuration	$r=4$	$r=8$	$r=16$	$r=32$	Spread
bge-m3 / A100-80GB	40.41	40.10	40.13	39.93	1.19%
ELECTRA-large / A100-80GB	37.28	37.01	37.01	37.27	0.73%
DeBERTa-v2-xl / A100-80GB	75.92	75.76	75.84	76.08	0.41%
XLNet-RoBERTa-XL / A100-80GB	123.35	122.64	122.95	123.46	0.67%
bge-m3 / L40S-48GB	39.22	39.22	39.22	39.25	0.07%

Appendix A Adapter-Count Latency Scaling

Figure 2 is the visual companion to Finding 1 (§5.2): p50 latency stays flat as the pre-loaded adapter pool grows from 100 to 47,000 adapters across all five batch sizes.

Appendix B Rank Sweep Across Encoders and GPUs

Table 5 details the rank-insensitivity claim of §5.4, with ranks interleaved within each seed pass. No configuration shows a resolvable trend in r : the largest spread across the four ranks is 1.19%, below the 3.41% between-seed s.d. of the noisiest cell we measure.

Appendix C GPU-Resident Batch Assembly

Finding 5 identifies CPU-side batch assembly as the residual bottleneck. The alternative, a GPU-resident assembler, holds the adapter store as one contiguous tensor per target module: shape (N, L, d, r) for the shrink matrices and (N, L, r, d) for the expand matrices. Each batch is then built

with one `torch.index_select` per module, which gathers all L layers for the whole batch on the GPU. We sweep both encoders at $N=2,000$, $r=8$, float16, sequence length 128, 5 seeds, on the same A100. Table 6 reports the result.

Table 6: CPU baseline assembler (*base*) versus the GPU-resident `index_select` assembler (*idx-sel*) at $N=2,000$, $r=8$, float16, 5 seeds. *Asm. share* is assembly as a fraction of end-to-end latency, before $\rightarrow$ after.

B	Latency (ms)		Speed	Asm. share		p99/p50
	base	idx-sel		-up	base $\rightarrow$ idx	
bge-m3						
8	21.39	18.75	$1.14\times$	$14.8 \rightarrow 2.2\%$	$1.12 \rightarrow 1.08$	
16	26.85	19.10	$1.41\times$	$29.8 \rightarrow 2.2\%$	$1.14 \rightarrow 1.10$	
32	41.26	26.62	$1.55\times$	$36.8 \rightarrow 1.8\%$	$1.18 \rightarrow 1.02$	
64	83.52	48.41	$1.73\times$	$42.8 \rightarrow 1.0\%$	$4.29 \rightarrow 1.02$	
128	166.66	92.85	$1.80\times$	$45.0 \rightarrow 0.5\%$	$2.76 \rightarrow 1.01$	
all-MiniLM-L6-v2						
8	5.47	4.86	$1.13\times$	$15.3 \rightarrow 4.1\%$	$1.20 \rightarrow 1.13$	
16	6.54	5.07	$1.29\times$	$24.8 \rightarrow 3.9\%$	$1.21 \rightarrow 1.17$	
32	7.90	4.77	$1.66\times$	$40.7 \rightarrow 4.6\%$	$1.17 \rightarrow 1.13$	
64	13.00	5.27	$2.47\times$	$58.0 \rightarrow 4.4\%$	$1.24 \rightarrow 1.11$	
128	24.65	8.98	$2.75\times$	$61.6 \rightarrow 3.1\%$	$1.48 \rightarrow 1.06$	
256	52.12	16.72	$3.12\times$	$65.3 \rightarrow 1.8\%$	$5.28 \rightarrow 1.03$	

Appendix D Mixed-Rank Fleets

A batch must be rank-homogeneous, so a fleet with mixed ranks pads each adapter to the batch maximum. Finding 3 predicts this is cheap: p50 is flat across ranks, so padding to the fleet maximum should cost little while the base path dominates the forward. We test it directly on bge-m3/A100 ($B=32$, 5 seeds), serving a heterogeneous fleet two ways: a pre-padded store, and per-rank stores that pad only during batch assembly.

Across an $r \in \{4, 16\}$ fleet swept to $N=20,000$ and a wider $r \in \{4, 8, 16, 32\}$ fleet swept to $N=10,000$, a pre-padded mixed batch matches a uniform batch already running at the fleet maximum to within 0.21% mean end-to-end latency, measured with the GPU-resident assembler of §5.2. Latency also stays flat as the pool grows: the worst p50 spread across N is 0.78%. Since every sampled batch contained the fleet maximum, these are worst-case mixed batches.

The two stores trade memory against assembly. Pre-padding holds every adapter at the fleet maximum, so a batch gathers ready-shaped tensors; per-rank stores hold each at its own rank, but must pad during assembly, a small gather-time cost.

Appendix E Protocol Validation on the SetFit Backbone

To confirm that our few-shot protocol reproduces results comparable to the SetFit literature, we replicate it on paraphrase-mpnet-base-v2 (Reimers and Gurevych, 2019), the standard SetFit backbone (Tunstall et al., 2022), across seven public benchmarks: SST-2, SST-5, CR, Amazon-Counterfactual, Emotion, EnronSpam, and AG News. For each dataset we train two variants at $N=8$ labeled examples per class: a full fine-tune of the encoder (109M trainable parameters) and a LoRA variant at rank 8 (294,912 trainable parameters, a $371\times$ reduction). LoRA trains with body learning rate 5×10^{-4} and full fine-tuning with 2×10^{-5} : adapters need the larger learning rate, and applying either model’s best rate to the other understates that model’s accuracy.

On mpnet, LoRA retains 95.5% of full fine-tuning’s mean accuracy, losing 2.96 percentage points (Table 7); the gap is smallest on EnronSpam (0.7pp) and largest on Amazon-CF (5.1pp, reported as MCC, where the low absolute scores magnify the difference). The protocol reproduces SetFit-comparable accuracy and shows the same LoRA full fine-tuning pattern on a second backbone, confirming that the protocol we apply to the serving model in §2.1 is sound. We report a frozen-encoder baseline only for the serving model (§2.1); the corresponding mpnet comparison is given by Tunstall et al. (2022). In the case where the LoRA full-fine-tuning gap is unacceptable, rank can be increased at zero latency cost (Finding 3); the only price is GPU memory.

Table 7: Per-task accuracy of LoRA adapters versus full encoder fine-tuning on paraphrase-mpnet-base-v2, the standard SetFit backbone ($N=8$ examples per class, rank=8, mean over 10 seeds). Scores are accuracy except Amazon-CF (MCC); Δ is LoRA minus Full FT.

Dataset	Full FT	LoRA-r8	Δ
SST-2	0.830	0.796	−0.034
SST-5	0.431	0.408	−0.023
CR	0.871	0.829	−0.042
Amazon-CF	0.272	0.221	−0.051
Emotion	0.443	0.415	−0.028
EnronSpam	0.907	0.900	−0.007
AG News	0.821	0.799	−0.022
Mean	0.654	0.624	−0.030
Params	109M	295K	1/371

Appendix F Horizontal Deployment

Our measurements isolate a single GPU node; this appendix describes how the same serving primitive composes into a horizontally scaled cluster. We provide a reference deployment (Kubernetes manifests and a FastAPI serving layer) that is intentionally minimal: it demonstrates the architecture rather than optimizing it, and is meant to be adapted to a given workload’s batching, autoscaling, and adapter-placement policy. We do not evaluate it empirically.

Stateless replicas. The base encoder is replicated across pods and adapters are loaded from S3-compatible object storage, so the only per-node state is the in-GPU adapter cache, which is reconstructible from storage. Each pod runs the dynamic batcher and LateFuse engine of §4 unchanged.

Tenant-sticky routing. An ingress layer routes by a consistent hash of the tenant identifier (NGINX upstream-hash-by on an X-Tenant-Id header, mirrored by an application-level hash ring with virtual nodes). Each tenant is thus pinned to a single pod that holds its adapter resident in GPU memory, so the $\mathcal{O}(1)$ -in- N latency we measure holds at each pod, and because the replicas are independent, aggregate request throughput grows with the number of pods ($\approx K \times$ a single node for K pods). Serving more adapters than a single node’s adapter ceiling (Finding 4) additionally requires partitioning the adapters across pods by the same hash, so each pod loads only its shard; the reference deployment instead replicates a fixed adapter set per pod (simpler, and adequate when the population fits one node), leaving shard-aware placement as a configuration change.

Elasticity and hot reload. A horizontal pod autoscaler adds or removes replicas on utilization metrics. Consistent hashing remaps only $\approx 1/K$ of tenants on a scaling event, so disruption is bounded to those tenants, whose adapters are then (re)loaded on their new pod from object storage. Adapter retraining is propagated by a publish/subscribe channel (Redis) that triggers in-place reload without a pod restart.

Routing caveat. Sticky routing pins a tenant to one pod, so a single high-traffic tenant cannot be spread across nodes without sacrificing cache locality; cluster behavior under skewed tenant traffic is left to future work.