InvisibleLoRA: Fast Multi-LoRA Serving Without Per-Token LoRA Decoding

Anonymous authors
Paper under double-blind review

Abstract

Low-rank adaptation (LoRA) specializes a single base model into many fine-tuned variants, and production systems now serve hundreds of adapters on a shared base model. Serving many adapters concurrently, however, introduces substantial inference overhead. Because requests are partitioned across active adapters, each low-rank computation covers only a small fraction of the batch and runs as a small, GPU-inefficient operation. Consequently, per-step adapter overhead grows with the number of active adapters even though each adapter holds only a small fraction of the base-model parameters. Existing multi-LoRA systems reduce the cost of each adapter execution through specialized kernels and scheduling, but every active adapter still executes at every token. We propose **InvisibleLoRA**, a speculative decoding method that uses the shared base model as the common drafter for every served adapter. A LoRA-off forward pass proposes the next token for every request in the batch, and a single LoRA-on forward pass verifies all proposals. InvisibleLoRA therefore applies adapter computation once per decoding cycle rather than once per token, while speculative verification preserves the output distribution of standard multi-LoRA decoding. InvisibleLoRA further introduces a hybrid KV cache: the LoRA-off base model writes speculative KV entries into the KV cache of the LoRA-on target model, and the target model overwrites the speculative entries during verification, so the drafter needs no separate KV cache and reads LoRA-conditioned KV states for the committed prefix. InvisibleLoRA also adopts a run-time gate that skips drafting on steps where the adapter cost cannot repay a draft pass. InvisibleLoRA reaches median draft acceptance rates of 92–97% across 456 publicly available LoRA adapters on five base models. Implemented in vLLM, InvisibleLoRA delivers lossless acceleration over an optimized multi-LoRA baseline, improving throughput by up to 1.84× and reducing per-token latency by up to 48%.

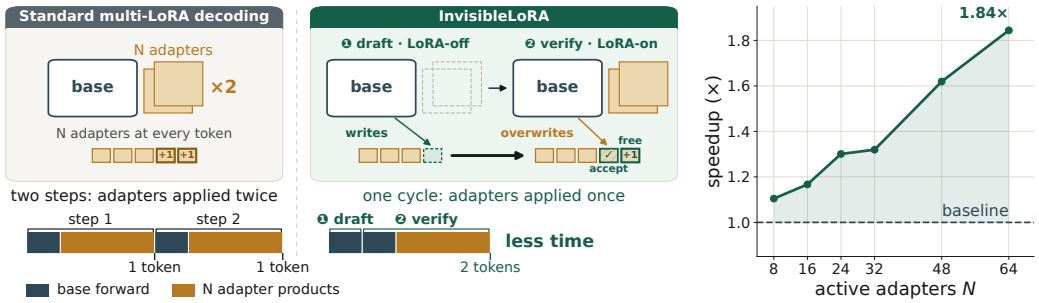
1 Introduction

Low-rank adaptation (LoRA) (Hu et al., 2022) specializes a base model by training a pair of low-rank matrices per layer, at a small fraction of the cost of full fine-tuning (Houlsby et al., 2019; Dettmers et al., 2023). Serving many such adapters on a shared base model has therefore become a common way to offer personalized and task-specific large language models (Chen et al., 2024; Sheng et al., 2024), and production deployments hold hundreds of adapters per base model with mixed ranks and skewed traffic (Jaiswal et al., 2025).

Serving many adapters concurrently is not free. For a homogeneous batch using a single adapter, the adapter can be merged into the base weights ahead of time, eliminating adapter-specific inference overhead (Hu et al., 2022). In heterogeneous multi-LoRA serving, however, different requests invoke different adapters, so no single adapter can be merged into the shared base model and adapter computation is partitioned across active adapters. Consequently, each adapter-specific low-rank computation covers only a small fraction of the batch, resulting in many small, GPU-inefficient operations at every decode step. The aggregate adapter overhead therefore scales with the number of active adapters, whereas the base-model computation remains shared across the entire batch.

Prior work reduces the cost of each adapter execution. Punica batches the adapter-specific computations using specialized GPU kernels (Chen et al., 2024), S-LoRA combines unified paging with

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.



(a) The overview of InvisibleLoRA. (b) Throughput speedup of InvisibleLoRA.

Figure 1: (a) **The overview of InvisibleLoRA.** Standard multi-LoRA decoding applies every active adapter at every token, whereas InvisibleLoRA amortizes adapter execution through LoRA-off drafting and LoRA-on verification on a shared KV cache. (b) **Throughput speedup of InvisibleLoRA.** InvisibleLoRA reaches 1.84× the throughput of the baseline multi-LoRA serving as the number of active adapters grows on Llama-3.1-8B.

heterogeneous batching (Sheng et al., 2024), and subsequent systems jointly schedule adapters and requests to improve GPU utilization (Wu et al., 2024; Predibase, 2024a). These optimizations improve the efficiency of each adapter execution without changing the underlying execution pattern: every active adapter must still be applied at every decode step, and each application serves only the requests assigned to that adapter. Existing multi-LoRA systems therefore make each adapter execution cheaper, but every active adapter still executes at every token.

In this paper, we propose InvisibleLoRA, a speculative decoding method for multi-LoRA serving that addresses the fundamental inefficiency of repeated adapter execution at every decode step. Our key observation is that every adapter adds only a small low-rank correction to one shared base model, so the base model with adapters switched off is a ready-made drafter for all of them. In each cycle, one LoRA-off forward pass proposes the next token for every request, and one LoRA-on forward pass verifies the proposals and commits the tokens, so the output follows the distribution of standard multi-LoRA decoding while the adapters run once per cycle instead of once per token. InvisibleLoRA further introduces a hybrid KV cache: the drafter runs directly on the LoRA-conditioned KV cache of the target model, and verification overwrites the speculative entries, as shown in Figure 1(a). The drafter therefore adds no KV memory, and drafting on LoRA-conditioned KV states rather than base-model approximations pulls the proposals toward the adapter and raises draft acceptance. InvisibleLoRA also employs a run-time gate that turns drafting off on steps where the adapter cost cannot repay a draft pass, since the gain depends on how many active adapters a batch holds. Across 456 publicly available LoRA adapters on five base models, the proposals of the drafter are accepted at a median rate of 92–97%. Our vLLM (Kwon et al., 2023) implementation of InvisibleLoRA delivers lossless acceleration over an optimized multi-LoRA serving baseline, improving throughput by up to 1.84× and reducing per-token latency by up to 48%, as illustrated in Figure 1(b).

In summary, our contributions are as follows.

- **Multi-LoRA decoding without per-token LoRA execution.** To the best of our knowledge, InvisibleLoRA is the first method that reduces the amount of adapter computation in multi-LoRA serving rather than the cost of each adapter execution. By drafting with the LoRA-off base model and verifying with the LoRA-on model, every active adapter runs once per cycle rather than once per token.
- **Hybrid KV cache.** We introduce a hybrid KV cache that removes the separate KV cache that speculative drafting requires and raises draft acceptance simultaneously. By running the LoRA-off drafter directly on the LoRA-conditioned KV cache of the LoRA-on model, the drafter adds no KV memory and reads LoRA-conditioned keys and values, reaching median acceptance rates of 92–97% across 456 public LoRA adapters on five base models.
- **Lossless acceleration in practice.** In our vLLM implementation, InvisibleLoRA improves throughput by up to 1.84×, reduces per-token latency by up to 48%, and raises peak

throughput under Poisson arrivals by up to 1.72× over an optimized multi-LoRA baseline, while preserving the output distribution of standard multi-LoRA decoding.

2 Background and Motivation

2.1 LLM serving with LoRA adapters

Low-rank adaptation (LoRA) reduces fine-tuning cost by freezing each weight matrix $\mathbf{W} \in \mathbb{R}^{d_{\text{out}} \times d_{\text{in}}}$ and learning low-rank matrices $\mathbf{A} \in \mathbb{R}^{r \times d_{\text{in}}}$ and $\mathbf{B} \in \mathbb{R}^{d_{\text{out}} \times r}$ (Hu et al., 2022). For input rows $\mathbf{X}$, an adapted layer computes

$$\mathbf{Y} = \mathbf{X}\mathbf{W}^\top + \underbrace{(\mathbf{X}\mathbf{A}^\top)}_{\text{shrink}} \underbrace{\mathbf{B}^\top}_{\text{expand}}, \tag{1}$$

with the LoRA scaling factor folded into $\mathbf{B}$. When every request uses the same adapter, the update can be merged into the base weight as $\mathbf{W} + \mathbf{B}\mathbf{A}$, eliminating separate per-token adapter computation.

A heterogeneous batch prevents such merging because different requests require different adapters. Multi-LoRA serving therefore keeps the base weight $\mathbf{W}$ shared across the entire batch while executing each adapter-specific low-rank update only for the requests assigned to that adapter (Chen et al., 2024; Sheng et al., 2024). For a batch of B requests with N active adapters, the input rows are partitioned into adapter-specific subsets $\mathbf{X}_i$ of size b_i, where $\sum_i b_i = B$. The shared product $\mathbf{X}\mathbf{W}^\top$ is thus computed once over all B requests, whereas the update $(\mathbf{X}_i\mathbf{A}_i^\top)\mathbf{B}_i^\top$ is computed separately for each active adapter.

LLM serving frameworks implement this execution structure by storing the base model once and keeping multiple adapters resident in GPU memory (Kwon et al., 2023; Zheng et al., 2024). To batch adapters with heterogeneous ranks, the frameworks pad the resident adapters to a common *slot width* w, defined by the largest rank in the resident adapter pool. For a fixed batch size, the average number of requests processed by each active adapter is B/N.

2.2 Adapter overhead in multi-LoRA decoding

Partitioning a fixed batch across more active adapters makes adapter execution increasingly inefficient. The shared base-model product $\mathbf{X}\mathbf{W}^\top$ continues to process all B rows in one dense operation, whereas each adapter update processes only b_i rows. As N grows, the average $b_i = B/N$ decreases, producing increasingly small matrix multiplications with poor GPU utilization (Chen et al., 2024; Sheng et al., 2024; Li et al., 2024b). In Figure 2, the base-model execution time remains nearly constant, while the aggregate adapter execution time grows rapidly with N, reaching 2.9× the base-model time at 64 active adapters. The adapter path can therefore incur substantial serving overhead even though each LoRA adapter contains only a small fraction of the base-model parameters.

Existing multi-LoRA systems reduce the cost of individual adapter executions through specialized kernels, memory management, and scheduling (Chen et al., 2024; Sheng et al., 2024; Wu et al., 2024; Li et al., 2024b; Jaiswal et al., 2025). However, every active adapter is still applied at every decode step. The repeated per-token execution of adapter computation therefore remains unchanged.

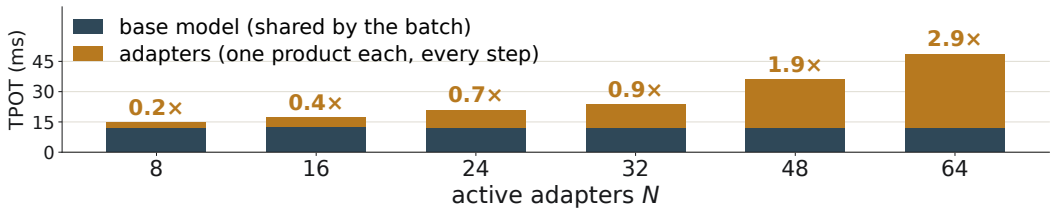


Figure 2: **Adapter overhead in multi-LoRA decoding.** TPOT of the shared base-model computation and aggregate adapter computation as the number of active adapters N grows for Llama-3.1-8B at batch size 64 and slot width 64 on an H100 GPU. Labels report adapter execution time normalized by base-model execution time; the ratio grows with N and reaches 2.9× at $N = 64$.

Algorithm 1 One InvisibleLoRA decoding cycle with drafting length K.

Require: Committed prefix p_t and adapter $a(t)$ for each request $t \in \mathcal{B}$

$\hat{x}_{t,1:K} \leftarrow \text{Draft}(p_t, K; \text{LoRA-off})$ for all $t \in \mathcal{B}$ ▷ no adapter computation

$y_{t,1:K+1} \leftarrow \text{Verify}(p_t, \hat{x}_{t,1:K}; a(t))$ for all $t \in \mathcal{B}$ ▷ one batched LoRA-on pass

for each request $t \in \mathcal{B}$ **do**

  $m_t \leftarrow \max\{m \le K : \hat{x}_{t,j} = y_{t,j},\ \forall j \le m\}$

  commit $\langle y_{t,1}, \ldots, y_{t,m_t+1} \rangle$

end for

2.3 Speculative decoding

Speculative decoding reduces repeated target-model execution by using a cheap *drafter* to propose the next K tokens and verifying all proposals in parallel with the *target* model in one forward pass (Stern et al., 2018; Leviathan et al., 2023; Chen et al., 2023). A single target execution can therefore produce multiple committed tokens rather than only one. Importantly, standard speculative verification is *lossless*: under greedy decoding, verification produces the same token sequence as autoregressive target decoding, while under sampling, rejection sampling preserves the output distribution of the target model.

The same principle provides a natural way to address the repeated adapter execution identified above. By amortizing adapter computation across multiple committed tokens, speculative decoding can reduce the frequency of adapter execution without changing the output distribution of the LoRA-adapted target model. Section 3 develops this approach for multi-LoRA serving.

3 Design of InvisibleLoRA

InvisibleLoRA reduces repeated adapter execution by using the shared base model as a common LoRA-off drafter, as described in Section 3.1. To make this speculative execution practical, InvisibleLoRA introduces a hybrid KV cache in Section 3.2, which reuses the LoRA-conditioned target cache for drafting without maintaining a separate drafter cache. Since the additional draft pass is not always beneficial, InvisibleLoRA further introduces a run-time gate in Section 3.3, which models the resulting cost trade-off and enables drafting only when the saved adapter computation outweighs the drafting overhead.

3.1 LoRA-off drafting and LoRA-on verification

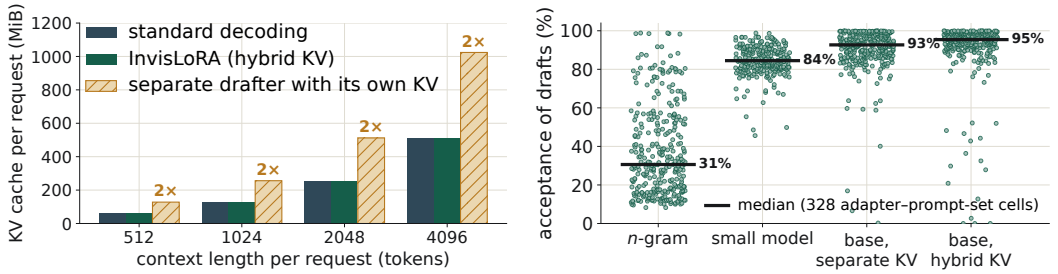
As analyzed in Section 2.2, a substantial fraction of multi-LoRA serving overhead comes from GPU-inefficient adapter computation, particularly when many adapters are active. Rather than reducing the cost of individual adapter executions, we propose InvisibleLoRA, which losslessly reduces adapter execution frequency by amortizing a single LoRA-on verification over multiple output tokens.

InvisibleLoRA adopts speculative decoding for multi-LoRA serving, as summarized in Algorithm 1. Although requests use different LoRA adapters, all targets share the same base model, allowing the LoRA-off base model to serve as a common drafter without an additional draft model or adapter-specific training. For drafting length K, the drafter proposes K tokens per request without adapter computation, and a single batched LoRA-on pass verifies all proposals. InvisibleLoRA then commits the longest accepted draft prefix together with one target token, so each cycle executes the adapters once while advancing a request by up to $K + 1$ tokens.

Let α_K denote the average fraction of the K drafted tokens contained in the accepted prefix. Since verification contributes one additional target token beyond the accepted draft prefix, a single LoRA-on execution advances each request by

$$\mathbb{E}[C_K] = 1 + K\alpha_K \tag{2}$$

tokens on average. Thus, both drafting length and draft acceptance determine how many output tokens amortize one adapter execution. For the default $K = 1$, we write $\alpha = \alpha_1$, giving $\mathbb{E}[C_1] = 1 + \alpha$; at 95% acceptance, one LoRA-on execution advances each request by 1.95 tokens on average. Larger K can amortize one LoRA-on verification over more tokens, but lower acceptance at later draft positions increases wasted speculative computation as analyzed in Appendix C.



(a) KV cache per request.

(b) Acceptance by drafter.

Figure 3: **Effect of the hybrid KV cache.** (a) An independent base-model drafter requires a second KV lineage, doubling the KV footprint, whereas InvisibleLoRA retains the KV footprint of standard decoding. (b) Reusing LoRA-conditioned KV increases draft acceptance relative to a separate base-model cache.

3.2 Hybrid KV cache

As described in Section 3.1, InvisibleLoRA reduces adapter execution without additional model parameters by using the shared base model as a drafter. Making InvisibleLoRA practical, however, requires addressing two issues. First, an independent LoRA-off drafter requires a separate base-model KV cache in addition to the LoRA-conditioned target cache, duplicating KV memory. Second, the amortization in Equation 2 depends directly on draft acceptance, while a separate base-model cache conditions the drafter only on base-computed representations of the committed prefix.

InvisibleLoRA addresses both issues with a hybrid KV cache that directly reuses the LoRA-conditioned target cache for drafting. For clarity, we omit layer and head indices. Let T denote the length of the committed prefix and (K^a, V^a) the KV states produced with adapter a. Drafting and verification then operate as

$$\begin{aligned} \text{draft:} \quad o_i^{\text{base}} &= \text{Attn}\Big(q_i^{\text{base}}, [K^a_{\le T}; K^{\text{base}}_{T+1:i}], [V^a_{\le T}; V^{\text{base}}_{T+1:i}]\Big), \\ \text{verify:} \quad o_i^{a} &= \text{Attn}\big(q_i^{a}, [K^a_{\le T}; K^{a}_{T+1:i}], [V^a_{\le T}; V^{a}_{T+1:i}]\big). \end{aligned} \tag{3}$$

The LoRA-off drafter therefore reads the LoRA-conditioned KV of the committed prefix and writes base-model KV only for newly drafted positions. The LoRA-on verification pass recomputes the speculative positions with adapter a and overwrites the temporary base-model KV before the positions become committed. Consequently, every committed position remains LoRA-conditioned across decoding cycles, preserving the target state required for lossless speculative verification.

The hybrid KV cache provides two complementary benefits. First, by reusing the target cache, InvisibleLoRA eliminates the second KV lineage required by an independent base-model drafter and retains the same KV footprint as standard decoding as illustrated in Figure 3(a). Second, the LoRA-off drafter attends to LoRA-conditioned prefix representations, allowing adapter-specific information to propagate into the draft states even without adapter computation at the drafted positions. The resulting reduction in mismatch with the LoRA-on target increases median draft acceptance from 93% to 95% in Figure 3(b), thereby increasing the number of output tokens amortized over each LoRA-on execution.

3.3 Cost model and run-time gating

InvisibleLoRA is not beneficial for every multi-LoRA batch. Accepted drafts avoid adapter executions that standard multi-LoRA decoding would otherwise perform, whereas rejected drafts spend base-model computation without advancing the committed sequence. Batched verification can also incur additional cost relative to a standard decode step. Let $\Gamma(B)$ denote the cost of a LoRA-off base-model step over batch size B, $A(N, w)$ the adapter cost for N active adapters with slot width w, and $\Gamma_{\text{ver}}(B, K)$ the base-model cost of verification with drafting length K. Using the acceptance rate

α_K defined in Section 3.1, InvisibleLoRA is beneficial when

$$\underbrace{K\alpha_K A(N,w)}_{\text{skipped adapter computation}} > \underbrace{K(1-\alpha_K)\Gamma(B)}_{\text{rejected draft computation}} + \underbrace{[\Gamma_{\text{ver}}(B,K)-\Gamma(B)]}_{\text{verification overhead}}. \tag{4}$$

Equation 4 captures the central trade-off of InvisibleLoRA. Higher draft acceptance both skips more adapter executions and reduces wasted speculation, while larger adapter overhead increases the amount of computation that can be amortized. Accordingly, InvisibleLoRA is most beneficial when adapter computation is substantial and draft acceptance remains high. Appendix B derives Equation 4 by comparing the cost of producing the same number of committed tokens and validates the model against measured execution times.

The exact execution costs in Equation 4 are not available before a decoding step, so the scheduler requires an observable run-time proxy. Section 2.2 shows that adapter overhead increases with both the number of active adapters N and the slot width w, while a larger batch B distributes the adapter burden across more requests. This motivates the following gate.

Proposition 1 (Run-time gate)**.** *Define the adapter burden per request as*

$$g = \frac{Nw}{B}. \tag{5}$$

InvisibleLoRA enables drafting when $g \geq \tau_{\text{gate}}$ and otherwise falls back to standard multi-LoRA decoding.

A larger g indicates a regime with more adapter computation available to amortize per request. The threshold τ_{gate} is calibrated once per deployment. Appendix B provides the detailed derivation of Proposition 1. Empirically, a single threshold $\tau_{\text{gate}} = 10$ correctly separates beneficial and non-beneficial drafting regimes across nearly 100 serving configurations spanning different slot widths, batch caps, arrival patterns, pool sizes, and base models, as detailed in Appendix I.

4 Results

4.1 Experimental Setup

We evaluate InvisibleLoRA on five base models, Llama-3.1-8B, Mistral-7B-v0.3, Qwen3-8B, Qwen3-14B, and Qwen3-32B (Grattafiori et al., 2024; Jiang et al., 2023; Yang et al., 2025), each with its own pool of publicly available LoRA adapters collected with a single rule: 146, 72, 88, 86, and 64 adapters, respectively (Appendix F). We measure acceptance over every adapter of each pool on GSM8K, MATH-500, HumanEval, MBPP, Alpaca, and MT-Bench (Cobbe et al., 2021; Lightman et al., 2024; Chen et al., 2021; Austin et al., 2021; Taori et al., 2023; Zheng et al., 2023), with 32 prompts each, 256 generated tokens, and greedy decoding.

All experiments run on H100 80GB GPUs, with TP=2 for Qwen3-32B. The baseline is the multi-LoRA path of vLLM (Kwon et al., 2023), which implements the kernels of Punica (Chen et al., 2024), and InvisibleLoRA is implemented as a patch to the same engine (Appendix E). For a fair comparison, both systems use the same LoRA kernel configuration, tuned for the baseline (Appendix D). Fixed-batch experiments use batches of 64 requests with 128 input and 256 output tokens, whose N adapters are drawn at random from the pool with two seeds, and each configuration is measured at least twice (Appendix E); results for all five base models are in Appendix H.

Because verification is lossless, we report serving performance rather than task accuracy. In batched bf16 inference, which is not bitwise reproducible even for the baseline, scored MATH-500 answers of InvisibleLoRA and the baseline agree on 94% of problems, while two runs of the baseline agree on 92% (Appendix A).

4.2 Draft acceptance

The speedup of InvisibleLoRA is bounded by $1 + \alpha$ (Equation 2), so its benefit depends first on how often the adapters accept the drafts of the LoRA-off base model. In this section, we show that the LoRA-off base model is an accurate drafter across tasks, that the hybrid KV cache raises its acceptance further, and that both results hold across model families and sizes.

Table 1: **Draft acceptance of four drafters.** Acceptance (%) on five base models and six prompt sets, the median over every adapter of each pool. "Separate KV" is the base model drafting from its own KV cache, and "small model" is an off-the-shelf small model of the same family (†none for Mistral). InvisibleLoRA achieves the highest acceptance on every base model and prompt set.

		Math		Code		Chat		
Model	Drafter	GSM8K	MATH-500	HumanEval	MBPP	Alpaca	MT-Bench	Avg
Llama-3.1-8B	*n*-gram	36.0	31.4	36.9	22.7	13.8	20.3	26.9
	small model†	89.3	92.2	89.3	87.8	84.4	80.7	87.3
	base model (separate KV)	95.6	96.9	96.6	96.2	94.4	92.9	95.4
	InvisibleLoRA (hybrid KV)	**97.2**	**97.7**	**97.9**	**97.4**	**95.7**	**95.2**	**96.9**
Mistral-7B-v0.3	*n*-gram	36.3	34.5	44.3	26.4	18.1	26.0	30.9
	base model (separate KV)	89.1	89.5	90.9	88.9	86.8	85.3	88.4
	InvisibleLoRA (hybrid KV)	**93.0**	**94.3**	**94.3**	**93.1**	**89.9**	**89.7**	**92.4**
Qwen3-8B	*n*-gram	28.0	27.0	75.7	79.2	18.3	24.9	42.2
	small model†	91.7	92.4	94.9	96.1	81.0	78.8	89.2
	base model (separate KV)	96.9	97.2	98.0	98.3	93.5	91.7	95.9
	InvisibleLoRA (hybrid KV)	**97.8**	**97.7**	**98.9**	**99.1**	**95.1**	**94.1**	**97.1**
Qwen3-14B	*n*-gram	26.5	27.3	38.1	38.8	12.4	16.6	26.6
	small model†	91.0	91.8	89.1	85.4	78.8	76.4	85.4
	base model (separate KV)	97.0	97.3	96.3	96.5	94.1	92.4	95.6
	InvisibleLoRA (hybrid KV)	**98.0**	**98.0**	**98.5**	**98.3**	**96.1**	**95.7**	**97.4**
Qwen3-32B	*n*-gram	25.3	26.1	44.1	27.2	10.7	14.9	24.7
	small model†	89.5	91.2	88.8	81.3	76.4	73.4	83.5
	base model (separate KV)	95.6	96.9	96.7	95.3	92.2	90.0	94.5
	InvisibleLoRA (hybrid KV)	**96.5**	**97.0**	**98.0**	**96.7**	**93.7**	**92.3**	**95.7**

Accurate drafts across tasks. Table 1 demonstrates that InvisibleLoRA achieves the highest acceptance among all drafters on every base model and every prompt set. On Llama-3.1-8B, 96.9% of the drafted tokens are accepted on average, and acceptance remains consistently high from math and code to open-ended chat prompts. Its acceptance is up to 18.0% higher than that of a small drafter model of the same family.

Effect of the hybrid KV cache. The hybrid KV cache further raises acceptance on every base model. Compared with the same base model drafting from its own separate KV cache, InvisibleLoRA improves average acceptance by 1.2 to 4.0 points, and the gain is largest where the base model is least accurate on its own, such as on chat prompts. This result shows that the LoRA-conditioned KV cache carries the behavior of the adapter.

Consistency across model families and sizes. The same trend holds across model families and sizes. On Mistral-7B-v0.3, InvisibleLoRA likewise outperforms the strongest alternative drafter, the base model with a separate KV cache, by 4.0 points on average. On the Qwen3 family, InvisibleLoRA stays above 95% from 8B to 32B (97.1%, 97.4%, and 95.7%, respectively). In contrast, the acceptance of the small drafter model decreases as the target grows, so the margin of InvisibleLoRA widens from 8.9% on Qwen3-8B to 14.6% on Qwen3-32B. This is because a fixed small model falls further behind a larger target, whereas the LoRA-off drafter is the base model of the target itself.

4.3 Throughput and latency as active adapters grow

In this section, we show that the speedup of InvisibleLoRA grows with the number of active adapters N, and that InvisibleLoRA nearly halves the per-token latency when many adapters are active.

Speedup with more active adapters. Figure 4 demonstrates that the speedup of InvisibleLoRA grows monotonically with the number of active adapters on all three base models, from about 1.1× at N=8 to up to 1.84× at N=64. This is because the two systems slow down at different rates as N grows. Every additional active adapter adds an adapter update to every decode step of the baseline, so its throughput drops by 2.9–3.2× from N=8 to N=64, whereas InvisibleLoRA executes the adapters once per cycle, so its throughput drops by only about 1.9×. The more adapters a batch holds, the

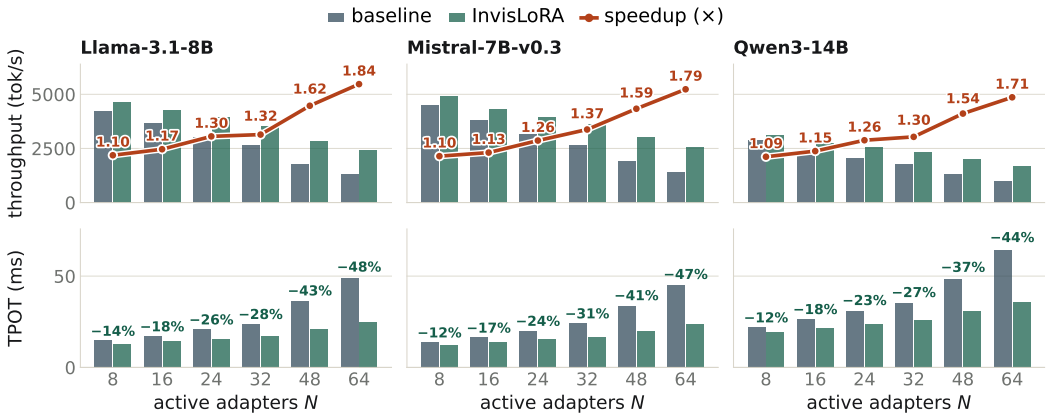


Figure 4: **Throughput and latency with increasing active adapters.** Throughput (top) and time per output token (bottom) of the baseline and InvisibleLoRA on three base models as the number of active adapters N grows; the line gives the speedup of InvisibleLoRA. The speedup grows with N on every base model, reaching 1.84× at N=64, while the time per output token drops by up to 48%.

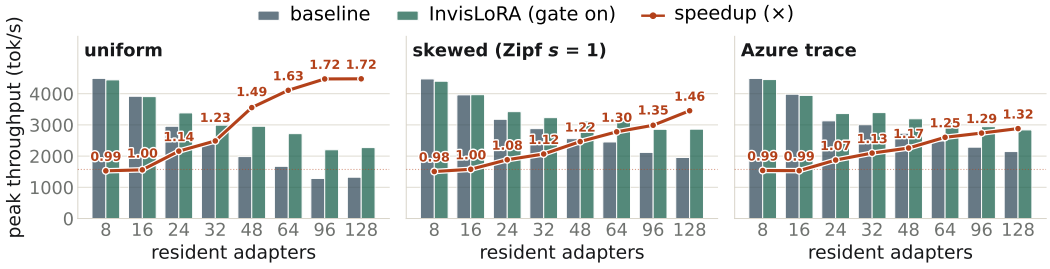


Figure 5: **Serving under Poisson arrivals.** Peak throughput of the baseline and InvisibleLoRA on Llama-3.1-8B as the number of resident adapters grows, under uniform, skewed, and trace-driven arrivals; both systems are measured at the offered load that maximizes the throughput of InvisibleLoRA, and the line gives the speedup. The speedup grows with the number of resident adapters under every arrival pattern, and the run-time gate keeps the smallest pools at parity with the baseline.

more adapter execution InvisibleLoRA saves. Qwen3-8B and Qwen3-32B follow the same trend, reaching 1.68× at N=64 and 1.50× at N=48, and the gain persists with longer inputs: with 512 and 2048 input tokens, the speedup at N=64 is 1.74× and 1.73×, because both systems pay the same prefill while only the baseline pays the adapter cost at every decode step (Appendix H).

Per-token latency. InvisibleLoRA also cuts the time per output token (TPOT), and the cut is largest when many adapters are active. On Llama-3.1-8B with 64 active adapters, TPOT drops from 49 ms to 25 ms, a 48% reduction (Figure 4, bottom). Across the three base models, the reduction grows from 12–14% at N=8 to 44–48% at N=64, so the latency gain scales with N in the same way as the throughput gain. Each cycle commits nearly two tokens per adapter execution, and the adapter execution is the part of the step that grows with N.

4.4 Serving under load

In online serving, requests arrive over time, so the number of distinct adapters in a batch depends on the arrival pattern rather than being fixed. We drive Llama-3.1-8B with open-loop Poisson arrivals over 8 to 128 resident adapters and a batch cap of 64, with uniform, skewed ($\propto \text{rank}^{-1}$), and Azure Functions trace-driven (Shahrad et al., 2020) adapter selection.

Peak throughput under arrivals. Figure 5 shows that InvisibleLoRA raises peak throughput under every arrival pattern, by up to 1.72× with 128 resident adapters under uniform arrivals. The speedup follows how many distinct adapters a batch holds. With 64 resident adapters under uniform arrivals, a full batch holds about 51 distinct adapters rather than 64, so the speedup of 1.63× sits slightly below the 1.84× of Section 4.3. Uniform arrivals spread requests over many adapters and gain the most, whereas skewed and trace-driven arrivals concentrate requests on fewer adapters, about 23 to 26 distinct adapters per batch, and gain less (1.46× and 1.32×), yet all three keep rising as the pool grows. The gain does not come at the cost of latency: with 64 resident adapters, the arrival rate at which 90% of requests meet a service-level objective of 6 s to the first token and 50 ms per output token (Sheng et al., 2024; Zhong et al., 2024) rises by 1.16 to 1.49× (Appendix I). The median TPOT stays under 40 ms in every configuration, and the bound that limits the capacity is the time to first token, which queueing determines rather than the extra drafting pass.

Effect of the run-time gate. The run-time gate keeps InvisibleLoRA at least as fast as the baseline when few adapters are active. For pools of 8 and 16 resident adapters, whose batches hold few distinct adapters, the gate falls back to standard decoding (Appendix I). Larger pools keep their full speedup, and the same threshold serves all three arrival patterns (Figure 5). The threshold $\tau_{\text{gate}}=10$ is calibrated once on the uniform runs and reused unchanged for the skewed and trace-driven arrivals, so the gate requires no per-workload tuning; the same value also holds on Qwen3-8B, which the calibration did not use (Appendix I).

5 Related Work

Multi-LoRA serving systems (Chen et al., 2024; Sheng et al., 2024; Wu et al., 2024; Li et al., 2024b; Jaiswal et al., 2025; Li et al., 2026; Shi et al., 2026) make each adapter cheaper to apply through gathered kernels, paging, caching and scheduling; InvisibleLoRA leaves them in place and instead reduces how often adapter computation runs (others change what is trained, Gowda et al., 2025; Kong et al., 2024). Adapters have been paired with speculation before: LoRAX supports per-request Medusa adapters and jointly trained Medusa–LoRA adapters (Predibase, 2024b), and Kodavanti et al. (2026) combine runtime-switched LoRAs with a self-speculative decoder on a phone; the first adds Medusa adapters, the second a separate speculative mechanism, and InvisibleLoRA adds neither. Speculative decoding (Leviathan et al., 2023; Chen et al., 2023; Stern et al., 2018) amortizes a target model's computation over verified proposals from a draft model, learned heads or a reduced target (Miao et al., 2024; Cai et al., 2024; Li et al., 2024a; Zhou et al., 2024; Liu et al., 2024); self-speculative decoding (Zhang et al., 2024; Elhoushi et al., 2024; Chen et al., 2025) is the closest family, but it drafts by skipping or early-exiting parts of the target; ours is the unchanged base model all adapters share. PEFT-BD (Javat and Kazakov, 2026) drafts with the adapter and verifies with the backbone, the mirror image of ours, and does not pay; Activated LoRA (Li et al., 2025) reuses the base model's KV cache for an adapted model, the reverse of InvisibleLoRA's hybrid cache. Shared-module serving (Woo et al., 2026a;b) shares a frozen module across fine-tuned models, InvisibleLoRA the adapter-free forward pass.

6 Conclusion

In this work, we propose InvisibleLoRA, which uses the shared base model, with its adapters switched off, as the drafter for every adapter it serves, so that adapter computation runs once per cycle rather than once per token, without training or additional parameters. A hybrid KV cache removes the separate draft cache, and the committed tokens are provably those of standard decoding. Across five base models, and under three arrival patterns on Llama-3.1-8B, InvisibleLoRA raises throughput and lowers per-token latency, and an analytical model bounds the gain, predicts it without fitting to speedup measurements, and names the regimes in which the method loses. In practice, InvisibleLoRA pays off wherever a single base model serves many adapters at once, and most where the baseline degrades fastest.

AI Use Statement

In this work, we used generative AI tools to implement the method and the benchmark scripts, to run the benchmarks, to provide feedback on the experimental design, to help develop the cost model and the proof of losslessness, to filter the collected adapter pools, to assist with translation, and to help interpret the results. We have not used generative AI tools to propose the research idea or hypotheses, and generating synthetic data and qualitative data analysis are not applicable to this work. Additionally, we used generative AI tools to create figures and tables, to write and debug code, to draft and edit the paper, and to search the literature and verify references. We have reviewed all AI-assisted work: every reported number is recomputed from the measured result files, and every reference was checked against its original record. We take responsibility for the final content of this work, including text, claims, and artifacts produced with the aid of generative AI.

References

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. Program synthesis with large language models. arXiv:2108.07732, 2021.

Tianle Cai, Yuhong Li, Zhengyang Geng, Hongwu Peng, Jason D. Lee, Deming Chen, and Tri Dao. Medusa: Simple LLM inference acceleration framework with multiple decoding heads. *ICML*, 2024. arXiv:2401.10774.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, et al. Evaluating large language models trained on code. arXiv:2107.03374, 2021.

Charlie Chen, Sebastian Borgeaud, Geoffrey Irving, Jean-Baptiste Lespiau, Laurent Sifre, and John Jumper. Accelerating large language model decoding with speculative sampling. arXiv:2302.01318, 2023.

Lequn Chen, Zihao Ye, Yongji Wu, Danyang Zhuo, Luis Ceze, and Arvind Krishnamurthy. Punica: Multi-tenant LoRA serving. *MLSys*, 2024. arXiv:2310.18547.

Longze Chen, Renke Shan, Huiming Wang, Lu Wang, Ziqiang Liu, Run Luo, Jiawei Wang, Hamid Alinejad-Rokny, and Min Yang. CLaSp: In-context layer skip for self-speculative decoding. *ACL*, 2025. arXiv:2505.24196.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. Training verifiers to solve math word problems. arXiv:2110.14168, 2021.

Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. QLoRA: Efficient finetuning of quantized LLMs. *NeurIPS*, 2023. arXiv:2305.14314.

Mostafa Elhoushi, Akshat Shrivastava, Diana Liskovich, Basil Hosmer, Bram Wasti, Liangzhen Lai, Anas Mahmoud, Bilge Acun, Saurabh Agarwal, Ahmed Roman, Ahmed A. Aly, Beidi Chen, and Carole-Jean Wu. LayerSkip: Enabling early exit inference and self-speculative decoding. *ACL*, 2024. arXiv:2404.16710.

Dhananjaya Gowda, Seoha Song, Harshith Goka, and Junhyun Lee. zFLoRA: Zero-latency fused low-rank adapters. *EMNLP*, 2025. arXiv:2510.25784.

Aaron Grattafiori et al. The Llama 3 herd of models. arXiv:2407.21783, 2024.

Neil Houlsby, Andrei Giurgiu, Stanislaw Jastrzebski, Bruna Morrone, Quentin de Laroussilhe, Andrea Gesmundo, Mona Attariyan, and Sylvain Gelly. Parameter-efficient transfer learning for NLP. *ICML*, 2019. arXiv:1902.00751.

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. *ICLR*, 2022. arXiv:2106.09685.

Shashwat Jaiswal, Shrikara Arun, Anjaly Parayil, Ankur Mallick, Spyros Mastorakis, Alind Khare, Chloi Alverti, Renee St Amant, Chetan Bansal, Victor Rühle, and Josep Torrellas. Serving heterogeneous LoRA adapters in distributed LLM inference systems. arXiv:2511.22880, 2025.

Abdurrahman Javat and Allan Kazakov. Accepted prefixes are not all you need: A negative result on PEFT-based block-diffusion drafting. arXiv:2607.12422, 2026.

Albert Q. Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, Lélio Renard Lavaud, Marie-Anne Lachaux, Pierre Stock, Teven Le Scao, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. Mistral 7B. arXiv:2310.06825, 2023.

Sravanth Kodavanti, Sowmya Vajrala, Srinivas Soumitri Miriyala, Utsav Tiwari, Uttam Kumar, Utkarsh Kumar Mahawar, Achal Pratap Singh, Arya D, Narendra Mutyala, Vikram Nelvoy Rajendiran, Sharan Kumar Allur, Euntaik Lee, Dohyoung Kim, HyeonSu Lee, Gyusung Cho, and JungBae Kim. Unlocking the edge deployment and on-device acceleration of multi-LoRA enabled one-for-all foundational LLM. *Findings of ACL*, 2026. arXiv:2604.18655.

Rui Kong, Qiyang Li, Xinyu Fang, Qingtian Feng, Qingfeng He, Yazhu Dong, Weijun Wang, Yuanchun Li, Linghe Kong, and Yunxin Liu. LoRA-Switch: Boosting the efficiency of dynamic LLM adapters via system-algorithm co-design. arXiv:2405.17741, 2024.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with PagedAttention. *SOSP*, 2023. arXiv:2309.06180.

Yaniv Leviathan, Matan Kalman, and Yossi Matias. Fast inference from transformers via speculative decoding. *ICML*, 2023. arXiv:2211.17192.

Yuhui Li, Fangyun Wei, Chao Zhang, and Hongyang Zhang. EAGLE: Speculative sampling requires rethinking feature uncertainty. *ICML*, 2024a. arXiv:2401.15077.

Suyi Li, Hanfeng Lu, Tianyuan Wu, Minchen Yu, Qizhen Weng, Xusheng Chen, Yizhou Shan, Binhang Yuan, and Wei Wang. CaraServe: CPU-assisted and rank-aware LoRA serving for generative LLM inference. arXiv:2401.11240, 2024b.

Allison Li, Kristjan Greenewald, Thomas Parnell, and Navid Azizan. Efficient multi-adapter LLM serving via cross-model KV-cache reuse with Activated LoRA. arXiv:2512.17910, 2025.

Shaolong Li, Xiang Yang, Qi Qi, Haifeng Sun, Zirui Zhuang, Bo He, Wanyi Ning, and Jingyu Wang. M-LoRA: Efficient serving for concurrent LoRA adapters with memory-aware speculative scheduler on single GPU. *IJCAI*, 2026. doi:10.24963/ijcai.2026/502.

Hunter Lightman, Vineet Kosaraju, Yura Burda, Harri Edwards, Bowen Baker, Teddy Lee, Jan Leike, John Schulman, Ilya Sutskever, and Karl Cobbe. Let's verify step by step. *ICLR*, 2024. arXiv:2305.20050.

Xiaoxuan Liu, Lanxiang Hu, Peter Bailis, Alvin Cheung, Zhijie Deng, Ion Stoica, and Hao Zhang. Online speculative decoding. *ICML*, 2024. arXiv:2310.07177.

Xupeng Miao, Gabriele Oliaro, Zhihao Zhang, Xinhao Cheng, Zeyu Wang, Zhengxin Zhang, Rae Ying Yee Wong, Alan Zhu, Lijie Yang, Xiaoxiang Shi, Chunan Shi, Zhuoming Chen, Daiyaan Arfeen, Reyna Abhyankar, and Zhihao Jia. SpecInfer: Accelerating generative LLM serving with tree-based speculative inference and verification. *ASPLOS*, 2024. arXiv:2305.09781.

Reiner Pope, Sholto Douglas, Aakanksha Chowdhery, Jacob Devlin, James Bradbury, Anselm Levskaya, Jonathan Heek, Kefan Xiao, Shivani Agrawal, and Jeff Dean. Efficiently scaling transformer inference. *MLSys*, 2023. arXiv:2211.05102.

Predibase. LoRAX: Multi-LoRA inference server that scales to 1000s of fine-tuned LLMs. `https://github.com/predibase/lorax`, 2024a.

Predibase. LoRAX v0.10.0: Medusa speculative decoding adapters, per-request Medusa adapters, and jointly trained Medusa + LoRA adapters. https://github.com/predibase/lorax/releases/tag/v0.10.0, May 2024b.

Mohammad Shahrad, Rodrigo Fonseca, Íñigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In *USENIX Annual Technical Conference*, 2020.

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. *MLSys*, 2024. arXiv:2311.03285.

Jiuchen Shi, Hang Zhang, Yixiao Wang, Quan Chen, Yizhou Shan, Kaihua Fu, Wei Wang, and Minyi Guo. ELORA: Efficient LoRA and KV cache management for multi-LoRA LLM serving. *HPCA*, 2026.

Mitchell Stern, Noam Shazeer, and Jakob Uszkoreit. Blockwise parallel decoding for deep autoregressive models. *NeurIPS*, 2018. arXiv:1811.03115.

Rohan Taori, Ishaan Gulrajani, Tianyi Zhang, Yann Dubois, Xuechen Li, Carlos Guestrin, Percy Liang, and Tatsunori B. Hashimoto. Stanford Alpaca: An instruction-following LLaMA model. https://github.com/tatsu-lab/stanford_alpaca, 2023.

Sunghyeon Woo, Ahreum Seo, Jaegwang Lee, Jaeeun Kil, Hanbae Seo, Joonghoon Kim, Baeseong Park, Se Jung Kwon, and Dongsoo Lee. SUN: Shared use of next-token prediction for efficient multi-LLM disaggregated serving. arXiv:2603.02599, 2026a.

Sunghyeon Woo, Hoseung Kim, Sunghwan Shim, Minjung Jo, Hyunjoon Jeong, Jeongtae Lee, Joonghoon Kim, Sungjae Lee, Baeseong Park, Se Jung Kwon, and Dongsoo Lee. PrefillShare: A shared prefill module for KV reuse in multi-LLM disaggregated serving. arXiv:2602.12029, 2026b.

Bingyang Wu, Ruidong Zhu, Zili Zhang, Peng Sun, Xuanzhe Liu, and Xin Jin. dLoRA: Dynamically orchestrating requests and adapters for LoRA LLM serving. *OSDI*, 2024.

An Yang et al. Qwen3 technical report. arXiv:2505.09388, 2025.

Jun Zhang, Jue Wang, Huan Li, Lidan Shou, Ke Chen, Gang Chen, and Sharad Mehrotra. Draft & Verify: Lossless large language model acceleration via self-speculative decoding. *ACL*, 2024. arXiv:2309.08168.

Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric P. Xing, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. Judging LLM-as-a-judge with MT-Bench and Chatbot Arena. *NeurIPS Datasets and Benchmarks*, 2023. arXiv:2306.05685.

Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark Barrett, and Ying Sheng. SGLang: Efficient execution of structured language model programs. *NeurIPS*, 2024. arXiv:2312.07104.

Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. DistServe: Disaggregating prefill and decoding for goodput-optimized large language model serving. *OSDI*, 2024. arXiv:2401.09670.

Yongchao Zhou, Kaifeng Lyu, Ankit Singh Rawat, Aditya Krishna Menon, Afshin Rostamizadeh, Sanjiv Kumar, Jean-François Kagy, and Rishabh Agarwal. DistillSpec: Improving speculative decoding via knowledge distillation. *ICLR*, 2024. arXiv:2310.08461.

A Correctness, losslessness measurements and derivations

Throughout the appendix, the two *arms* of a comparison are the baseline and InvisibleLoRA run on the same engine, with the same kernel configuration, on the same cell. We write F^{off} for the LoRA-off forward pass, F_i^{on} for the LoRA-on forward pass with adapter i, and p_i for the next-token distribution of the latter.

Algorithm 2 One InvisibleLoRA cycle over a batch, drafting length K=1

Require: batch $\mathcal{B}$, shared KV cache $\mathcal{K}$, adapter map $a(\cdot)$

1: $\hat{x} \leftarrow F^{\text{off}}(\mathcal{B}, \mathcal{K})$, then **write** its keys and values into $\mathcal{K}$ at the speculated positions ▷ draft: LoRA-off, no adapter is applied

2: $y \leftarrow F^{\text{on}}_{a(\cdot)}(\mathcal{B} \oplus \hat{x}, \mathcal{K})$, then **overwrite** those positions with y's ▷ verify: LoRA-on, $2|\mathcal{B}|$ rows, each active adapter applied once

3: **for all** $t \in \mathcal{B}$ **do**

4: commit $\langle y_{t,1}, y_{t,2} \rangle$ if $\hat{x}_t = \arg\max y_{t,1}$, else $\langle y_{t,1} \rangle$ ▷ two tokens, or one and the draft is discarded

5: **end for**

Proof of Theorem 1. The proof is by induction on committed prefixes, carrying a second invariant about the cache: at the start of every cycle, each committed position of a request t holds keys and values computed with $a(t)$ attached. The invariant is maintained because the draft writes LoRA-off entries only at the position it speculates, which is not yet committed, and the verify pass recomputes and overwrites exactly those positions with the adapter attached (lines 1 and 2 of Algorithm 2) before any later cycle reads them. This is the step the hybrid cache of §3.2 turns on, and it is what a shared cache has to get right.

Consider a request t whose committed prefix is h. By the invariant the cache holds the LoRA-on keys and values for h, so row 1 of the verify forward computes $F^{\text{on}}_{a(t)}$ on h, exactly as standard decoding would, so $y_{t,1}$ is $\arg\max p_{a(t)}(\cdot \mid h)$ and the first committed token is the one standard decoding emits. A second token is committed only when the draft $\hat{x}_t$ equals $\arg\max y_{t,1}$. In that case row 2 of the verify forward computes $F^{\text{on}}_{a(t)}$ on $h \| y_{t,1}$, so $y_{t,2}$ is $\arg\max p_{a(t)}(\cdot \mid h \| y_{t,1})$, which is what standard decoding would emit next. If the draft differs, only $y_{t,1}$ is committed and the draft is discarded. In both cases the committed prefix after the cycle is a prefix of the standard-decoding output, and the induction continues. The draft therefore changes only how many tokens a cycle commits, never which. □

Sampling. Under temperature sampling, row 1 of the verify forward yields for each request the LoRA-on next-token distribution $p = p_{a(t)}$ — by the cache invariant, computed with $a(t)$ attached over LoRA-on keys and values — and the engine's rejection sampler accepts the draft $\hat{x}_t \sim q$ with probability $\min(1, p(\hat{x}_t)/q(\hat{x}_t))$ and otherwise samples from the normalized positive residual $(p - q)_+$, which returns a sample from p (Leviathan et al., 2023). When the draft is accepted the second token is drawn from row 2, again LoRA-on. The invariant is maintained exactly as in the greedy case, so in exact arithmetic the committed tokens follow the distribution of standard multi-LoRA sampling. The paragraphs below measure this in bf16.

Losslessness in bf16. Batched bf16 decoding is not bit-reproducible, so the baseline's agreement with itself is the reference level against which any method is read. Over the 16 grid cells, or 1,024 requests, the baseline reproduces its own output exactly on 473 requests when repeated (46%), InvisibleLoRA on 461, and the two arms agree on 462; scored MATH-500 (Lightman et al., 2024) answers agree at 94% against a 92% self-repeat level. Tables 2 and 3 measure the sampling case.

Temperature. Table 2 repeats two cells of the offline grid under temperature sampling. Above T=0 the engine's rejection sampler decides acceptance against the LoRA-on distribution, so that distribution is preserved by construction and what moves is the acceptance rate: it falls roughly linearly in T, the speedup at about half that rate, as $\mathbb{E}[\tau_1] = 1 + \alpha$ requires, and the baseline is unchanged by temperature (950–967 tok/s across the four settings at slot 128). Run-to-run spread is within 1.3% on the baseline arm and 2.8% on ours, with one cell at 7.9%.

Table 2: Acceptance and speedup under temperature sampling, K=1, batch 64, 256 new tokens, both arms pinned to the same kernel configuration, two repetitions averaged. α is $\mathbb{E}[\tau_1] - 1$ from the engine's acceptance statistics.

Pool		T=0	T=0.3	T=0.7	T=1.0
slot 128, N=64:	α	0.977	0.950	0.900	0.855
	speedup	**1.71**	1.68	1.63	**1.57**
slot 64, N=32:	α	0.960	0.927	0.877	0.811
	speedup	**1.30**	1.22	1.17	**1.14**

The output distribution itself. Table 3 measures what the previous paragraph asserts. For each of three adapters spanning the acceptance range and eight MATH-500 prompts we draw 256 completions at T=1 from the baseline and from InvisibleLoRA, each request with its own seed, and compare the empirical distributions by total variation (TV): per position over the first eight tokens, and jointly over the first four. The floor is the TV between two independent baseline replicates, which is large where the adapter's distribution is broad (position-wise 0.34 for the α=0.22 adapter, whose 256 samples are nearly all distinct) and near zero where it is peaked. InvisibleLoRA sits at the floor everywhere: over 24 adapter–position pairs the median ratio of test to floor is 0.90.

Table 3: Distribution preservation under sampling, T=1, Llama-3.1-8B, 256 samples per prompt, medians over 8 prompts. Each cell is TV(baseline, independent baseline replicate) / TV(baseline, InvisibleLoRA).

adapter α	position 1	positions 2–8 (median)	first 4 tokens jointly
0.996	0.002 / 0.000	0.014 / 0.000	0.002 / 0.000
0.851	0.010 / 0.002	0.031 / 0.023	0.035 / 0.043
0.215	0.418 / 0.014	0.340 / 0.318	0.908 / 0.887

Symbols. The derivations behind §3.3 use the following constants. W_b = 16.06 GB and P_b = 8.03B for Llama-3.1-8B in bf16; $W_\mathrm{a}(w) = 2w \sum_{\text{layers}} \sum_{\text{modules}} (d_\text{in} + d_\text{out})$, which for the seven projections of 32 layers gives 5.24 MB per unit of rank and 336 MB at w=64, matching the measured slot size; BW = 3.35 TB/s and F = 990 TFLOP/s for H100 SXM. $\Gamma(B)$ is the measured LoRA-off step (N=1, one rank-16 slot):

B	16	32	64	128	196	256
$\Gamma(B)$, ms	11.19	11.49	12.34	14.40	17.06	19.03
compute term $2P_\mathrm{b}B/F$, ms	0.26	0.52	1.04	2.08	3.18	4.15
$\Gamma(B)/(W_\mathrm{b}/\mathrm{BW})$	2.3	2.4	2.6	3.0	3.6	4.0

Roofline. A decode step reads the weights once and performs $2P_\mathrm{b}$ FLOP per token, so the roofline bound (Pope et al., 2023) on its time is

$$t_\text{step}(B) \;\gtrsim\; \max\Big(\frac{W_\mathrm{b}}{\mathrm{BW}},\; \frac{2P_\mathrm{b}B}{F}\Big), \tag{6}$$

memory-bound until the two terms meet at $B^* = F/\mathrm{BW}$; the table above puts every batch of this paper below it. The adapter term (11) that §3.3 fits on top of it is what the grid of Appendix B measures.

General drafting length. With per-position acceptance α and drafting length K, a cycle commits j+1 tokens with probability $\alpha^j(1-\alpha)$ for $j < K$ and K+1 tokens with probability α^K, so

$$\mathbb{E}[\tau_K] \;=\; \sum_{j=0}^{K-1} (j{+}1)\,\alpha^j(1-\alpha) + (K{+}1)\,\alpha^K \;=\; \sum_{i=0}^{K} \alpha^i \;=\; \frac{1-\alpha^{K+1}}{1-\alpha}, \tag{7}$$

which is $1+\alpha$ at K=1. A cycle then runs K LoRA-off drafts and one LoRA-on verification over $(K{+}1)B$ positions, so with $\rho = A/\Gamma(B)$ the speedup over standard multi-LoRA decoding is

$$S_K \;=\; \mathbb{E}[\tau_K]\,\frac{\Gamma(B)+A}{K\,\Gamma(B)+\Gamma\big((K{+}1)B\big)+A} \;=\; \mathbb{E}[\tau_K]\,\frac{1+\rho}{K+\Gamma\big((K{+}1)B\big)/\Gamma(B)+\rho}, \tag{8}$$

whose case K=1 is

$$S \;=\; (1+\alpha)\,\frac{1+\rho}{1+\Gamma(2B)/\Gamma(B)+\rho}, \tag{9}$$

with break-even $\rho_K^* = \big(K+\Gamma((K{+}1)B)/\Gamma(B)-\mathbb{E}[\tau_K]\big)/(\mathbb{E}[\tau_K]-1)$.

What a mixed batch adds. With request t bound to adapter $a(t)$, the LoRA correction is $\Delta y_t = (x_t U_{a(t)}^\top)V_{a(t)}^\top$, with $U_{a(t)}$ and $V_{a(t)}$ the two low-rank factors of that adapter. In the accounting of (6), a batch with N active adapters of slot width w adds

$$\text{bytes:}\;\; +\,N\,W_{\mathrm{a}}(w) \qquad\qquad \text{FLOP:}\;\; +\,2P_{\mathrm{a}}B, \tag{10}$$

where $W_{\mathrm{a}}(w)$ is the bytes of one padded slot (336 MB at w=64 for Llama-3.1-8B) and P_{a} the adapter parameters one request touches ($P_{\mathrm{a}}/P_{\mathrm{b}} \approx 2\%$). At N=64 the byte term is 21.5 GB per step, 1.3× the base model.

What a perfect kernel would leave. InvisibleLoRA does not reduce the price of applying an adapter, only how often it is paid, so kernel work and InvisibleLoRA multiply. Even an adapter kernel at the bandwidth floor, reading each slot's bytes at peak bandwidth and nothing more, would leave about 1.10× for InvisibleLoRA to amortize at slot 64 and N=64, from (9) with A set to the byte floor of 0.10 ms per slot (A = 6.4 ms against $\Gamma(64)$ = 12.4 ms, so ρ = 0.52 and $1.945\times 1.52/2.69 \approx 1.10$). A perfect kernel leaves the argument and removes most of the multiplier.

Memory and compute forms. A K-draft cycle reads KW_{b} for the drafts and $W_{\mathrm{b}}+NW_{\mathrm{a}}$ for the verify; a baseline step reads $W_{\mathrm{b}}+NW_{\mathrm{a}}$. Dividing time per token, $S_{\mathrm{mem}} = \mathbb{E}[\tau_K](W_{\mathrm{b}}+NW_{\mathrm{a}})/((K{+}1)W_{\mathrm{b}}+NW_{\mathrm{a}})$, which tends to $\mathbb{E}[\tau_K]$ as $NW_{\mathrm{a}}/W_{\mathrm{b}} \to \infty$ and to $\mathbb{E}[\tau_K]/(K{+}1) < 1$ as $NW_{\mathrm{a}}/W_{\mathrm{b}} \to 0$: with no adapters the draft is pure additional reading. The break-even in bytes is $N^*_{\mathrm{mem}} = W_{\mathrm{b}}\,(K{+}1-\mathbb{E}[\tau_K])/(W_{\mathrm{a}}\,(\mathbb{E}[\tau_K]-1))$, which is 2.8 rank-64 slots at K=1 and 5.7 at K=3 for α = 0.945. In FLOP a cycle does $K\cdot 2P_{\mathrm{b}}B$ for the drafts and $(K{+}1)\cdot 2(P_{\mathrm{b}}+P_{\mathrm{a}})B$ for the verify, against $2(P_{\mathrm{b}}+P_{\mathrm{a}})B$ per baseline step, so $S_{\mathrm{comp}} = \mathbb{E}[\tau_K](P_{\mathrm{b}}+P_{\mathrm{a}})/(KP_{\mathrm{b}}+(K{+}1)(P_{\mathrm{b}}+P_{\mathrm{a}})) = 0.65, 0.57, 0.53$ at $K = 1, 2, 3$: compute-bound, every drafting length loses. The measured Γ ratios give $\rho_K^* = 0.23, 0.30, 0.32$ for $K = 1, 2, 3$ at B=64.

B Cost model: fits and the check against the grid

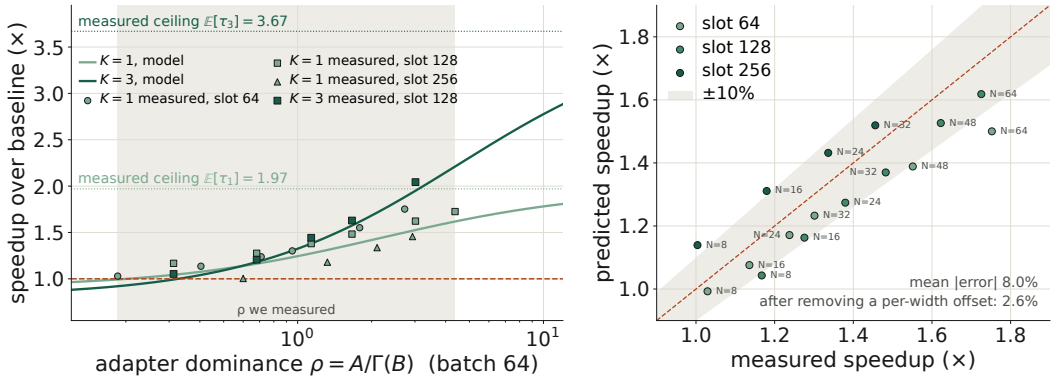
The adapter term. Over the offline grid the baseline step time is fit as $\Gamma(B)+A(N,w)$ with

$$A(N,w) \;=\; N\,(\kappa_0+\kappa_1 w), \tag{11}$$

one rank-independent and one rank-proportional cost per slot (Table 4).

Numbers behind §3.3. The adapters cost 2.9× (N=8) to 5.2× (N=64) the time to read their bytes at peak bandwidth (Figure 7). At the pooled α = 0.945, (7) gives 1.945 tokens per cycle at K=1 against 1.97 read from the engine's acceptance counter (Appendix C). The break-even ρ^* = 0.23 at B=64 is an adapter cost of 2.9 ms per step, about three rank-128 slots (0.86 ms each) or five rank-64 slots (0.61 ms each).

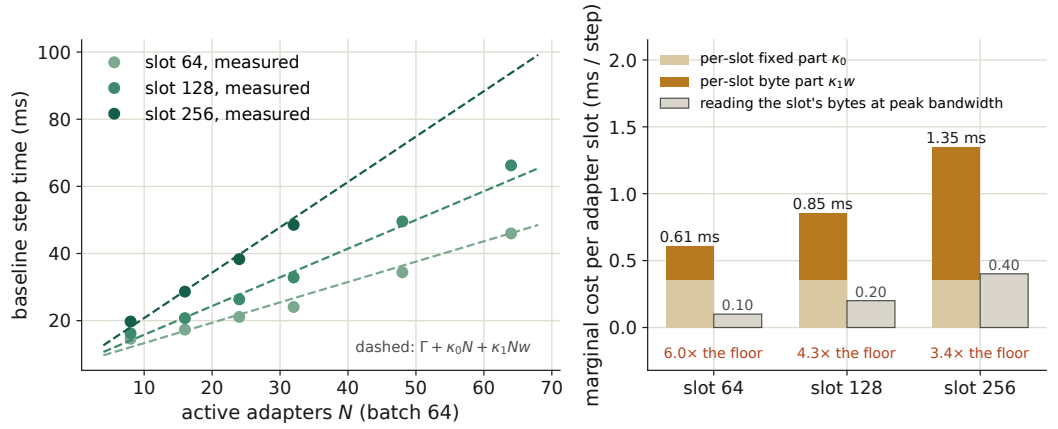
Table 4 lists the fits and Figure 6 the check of (9) against every cell. The prediction uses no parameter fitted to the speedup: from each cell's measured Γ and the pooled α it tracks the shape of the measurements and misses their level by one constant per slot width, a mean absolute error of 8.0% that removing one constant per width brings to 2.6%. The slot-256 offset is largely an acceptance effect. Each column of the grid is built from adapters of one rank, whose measured rates are 0.954 (rank 64), 0.978 (rank 128) and 0.886 (rank 256) against the pooled 0.945; re-solving each column with its own rate closes most of the slot-256 over-prediction, from +4–14% to within 9%. The remaining under-prediction at slots 64 and 128 comes from costing the verify pass as $\Gamma(2B)$: measured with a single adapter, every forward of the cycle costs about one B-row forward whether it carries one row per request or two, so $\Gamma(2B)$ overstates the cycle by about 8%.



(a) Speedup against dominance.

(b) Predicted vs. measured.

Figure 6: The cost model against measurements. (a) Speedup predicted by (8) as a function of the adapter-to-base time ratio ρ of Equation 9 for $K \in \{1, 3\}$, with the measured ceilings $\mathbb{E}[\tau_K]$ dotted and the measured cells of the offline grid as points; the shaded band is the range of ρ those cells cover, and outside it the curves are extrapolation. (b) Predicted against measured speedup for the 16 cells of the grid, using each cell's measured Γ and the measured $\mathbb{E}[\tau_1]$. The mean absolute error is 8.0%, but it is a per-width offset rather than scatter: removing one constant per slot width leaves 2.6%.



(a) Step time and the fit.

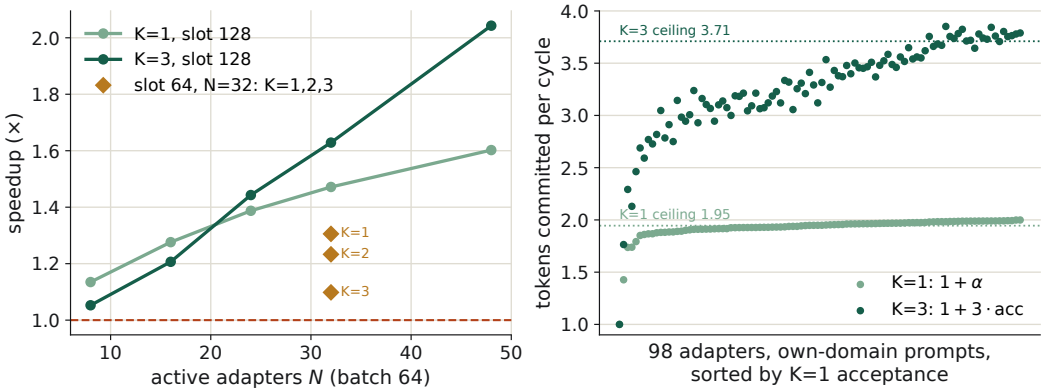
(b) Cost of one slot.

Figure 7: The adapter cost. (a) Baseline decode step time against N for three slot widths (H100, batch 64, both arms' kernel configuration pinned); dashed lines are the two-term fit (11). (b) The marginal cost of one adapter slot decomposed into the part that does not depend on rank and the part that does, against the time to read the slot's bytes at peak bandwidth.

Table 4: Fits of the baseline step time (ms) over the 16-cell offline grid (Llama-3.1-8B, batch 64, both arms pinned; three slot widths, six adapter counts).

Model	coefficients	R^2	max residual
one term: $\Gamma + \kappa\, Nw$	$11.60 + 0.00556\, Nw$	0.861	25.2%
two terms: $\Gamma + \kappa_0 N + \kappa_1 Nw$	$7.25 + 0.358\, N + 0.00388\, Nw$	0.983	17.3%
per-slot marginal cost $\kappa_0 + \kappa_1 w$: 0.61 (slot 64), 0.86 (128), 1.35 ms (256)			
share of the slot cost that does not depend on rank: 59%, 42%, 26%			
time to read the slot's bytes at 3.35 TB/s: 0.10, 0.20, 0.40 ms			

C Drafting length



(a) Speedup against K.

(b) Tokens per cycle.

Figure 8: (a) Speedup at slot 128 for K=1 and K=3 as N grows, both arms pinned; the pool is the single rank-128 adapter replicated to N slots, so acceptance is that adapter's. Diamonds: slot 64, N=32, $K \in \{1, 2, 3\}$. (b) Tokens committed per cycle implied by each adapter's measured acceptance at K=1 $(1 + \alpha)$ and at K=3 (1 + 3·acc, where acc is accepted over drafted tokens), own-domain prompts, the grid pool's 98 adapters of rank ≤ 64 (Appendix F).

Table 5: Drafting length, both arms pinned. Left: speedup against K on the grid pool (Appendix F); each cell replicates the adapters of that slot width. Right: own-domain acceptance — accepted over drafted tokens, each adapter on the prompts of its own domain — on both pools. The serving pool shows 145 of its 146 adapters: the K=3 run set the engine's maximum LoRA rank to each adapter's own rank, a setting the engine accepts only from $\{8, 16, 32, 64, \ldots\}$, so its one rank-25 adapter was skipped there; the same adapter loaded for the K=1 measurements of Table 1.

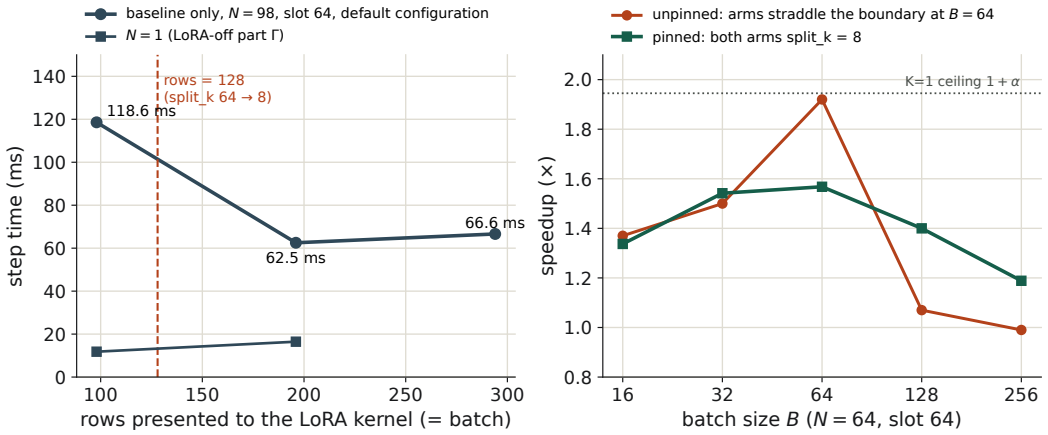
Cell	K=1	K=2	K=3
slot 128, N=32	1.49	1.65	1.67
slot 128, N=64	1.73	2.29	**2.48**
slot 64, N=32	**1.31**	1.23	1.10
slot 128, N=8 / 16 / 24 / 48	1.13 / 1.28 / 1.39 / 1.60	—	1.05 / 1.21 / 1.44 / 2.04

Own-domain acceptance	K=1	K=3
grid pool, its 98 adapters of rank ≤ 64		
median	94.9%	77.6%
10th percentile	88.0%	59.3%
adapters below 80%	5	54
serving pool, 145 of its 146 adapters		
median	95.8%	79.1%
10th percentile	88.7%	61.3%
adapters below 80%	6	77

Figure 8 and Table 5 give the measurements. K=3 raises the ceiling from 1.95 to 3.71 tokens per cycle and, where the adapter cost is large, the measured gain follows it: 2.48× at slot 128 and N=64, where the replicated rank-128 adapter commits 3.66 tokens per cycle. Where the adapter cost is small the two extra base forwards cost more than they save, and K=1 wins (slot 64, N=32; slot 128, $N \leq 16$). The pool bears K=3 unevenly: because every cycle drafts three tokens whether or not the first is accepted, the median adapter's acceptance falls from 94.9% to 77.6% and the tenth percentile's from 88.0% to 59.3%, so 54 of the grid pool's 98 adapters fall below 80% at K=3 against 5 at K=1, and the serving pool reads the same (95.8% to 79.1%). The break-even is therefore per

adapter, and we keep K=1 as the method. Losslessness holds at K=3: over the 64 requests of the slot-128 cell, two K=3 runs agree exactly on 26 and a K=3 run agrees with the baseline on 29, against 32 for K=1, all within the bf16 self-repeat level of Appendix A.

D Baseline fairness: kernel configuration and engine version



(a) The kernel boundary.

(b) Stock vs. pinned.

Figure 9: (a) Baseline step time alone, N=98 rank-64 adapters, default configuration: doubling the rows from 98 to 196 halves the step, because the row count crosses 128 and the kernel switches from `split_k`=64 to 8. (b) The batch axis at N=64 under the stock configuration and pinned: under the stock configuration B=64 is the only batch at which the verify pass ($2B$ rows) has crossed the boundary while the baseline has not, and it carries a peak that no smooth cost model can produce.

Figure 9 shows the boundary in the baseline alone and the artifact it produces in a comparison under the stock configuration.

Launch geometry. The shrink kernel's grid is (`split_k`$\cdot\lceil M/\texttt{BM}\rceil\cdot\lceil N/\texttt{BN}\rceil$, `slices`, `max_loras`), so each adapter slot receives its own set of thread blocks and blocks belonging to empty slots exit early, a waste the upstream source notes in a `TODO`. The configuration heuristic does not see `max_loras`, so at high adapter counts the split multiplies an already over-subscribed grid. At N=98 and rank 64 the baseline shape launches 301,056 blocks against the verify shape's 65,856. This per-slot block issue is the physical origin of the fixed term κ_0 in (11).

Cost decomposition. At B=98 with N=98 the adapter term of the baseline step is 106.8 ms under the default configuration and 55.5 ms under the pinned one, for identical bytes whose bandwidth floor is 9.8 ms.

Selecting the pinned value. Sweeping the baseline over $\{\text{default}, 64, 8, 1\}$ gives 828, 818, 1,456 and 1,243 tok/s at N=98, and 937, 929, 1,387, 1,131 at N=64. Pinning 64 reproduces the default within 1.2%, as it must at these batches, which checks that the override itself changes nothing else. The optimum is 8 rather than the smallest value, and 8 is also the baseline's optimum at slot 256 (1,322 against 680 default), so the pinned grid does not disadvantage the baseline at the widest slot.

What the stock configuration shows. With the default heuristic in both arms, the slot-64 batch axis at N=64 reads 1.92× at B=64 and 1.07× at B=128 against the pinned 1.57 and 1.40: the two arms run different kernels, which inflates the method at one batch and deflates it at the next. The main text claims only the pinned numbers.

Engine version. We build on the vLLM release that was current when this work began, 0.11.2, and compare it against the latest release at submission, 0.28.0. Two pieces of evidence say the

baseline we beat is not simply an old one. The first requires no measurement: the LoRA launch heuristic of Appendix D is byte-identical in 0.11.2, 0.19.0, 0.27.1 and 0.28.0. Upstream did improve the surrounding grid (from 0.20 the third grid axis counts *active* adapters rather than all resident slots), but the configuration boundary is unchanged. The second is measured (Table 6): multi-LoRA baseline throughput agrees within 5% on all three cells, and to 0.02% on the first.

Table 6: Baseline throughput (tok/s) on both releases, the grid pool's 98 adapters registered with 32 resident slots on each. Only the baseline arm is measured: the question is whether the baseline itself has moved.

Cell	0.11.2	0.28.0	ratio
B=98, default configuration	596.7	596.6	1.00
B=98, `split_k` pinned to 8	725.5	704.0	0.97
B=196, default configuration	1264.7	1204.4	0.95

Release 0.28.0 runs under NVIDIA's CUDA forward-compatibility package on our driver and could not hold more than 32 resident rank-64 slots on an 80 GB device, so both releases are compared at 32 slots.

E Experimental setup, measurement discipline and reproduction

Adapter pools. The five pools of the body, one per base model, gathered from the public hub by one rule: one adapter per repository, at most two per owner, rank ≤ 64 (the grid pool of Appendix F, used only for the cost-model grid and the drafting-length axis, reaches rank 256), no output-layer tensors (re-checked after download), weight files hash-pinned, and every adapter loadable by the engine (Appendix F gives the screen). Domain labels for design B come from the model card's dataset and tag metadata first and from the repository name last. The admission audit removes 3, 1, 0, 9 and 3 adapters at 0.70 (Llama-3.1-8B, Qwen3-8B, Qwen3-14B, Mistral-7B-v0.3, Qwen3-32B) and leaves 134, 84, 82, 41 and 53 at 0.90. Where an audited pool is smaller than N — Mistral-7B-v0.3 at N=48 and 64 — the remaining slots repeat adapters drawn from the pool; each slot holds its own copy of the weights under its own LoRA id, so the kernel and the gate see N adapters and only the acceptance statistics repeat. These cells therefore stress the adapter-slot cost axis while repeating the acceptance composition; Llama-3.1-8B, Qwen3-8B and Qwen3-14B fill N=64 with distinct adapters.

Drafters of Table 1. The small-model drafter is an off-the-shelf sibling sharing the tokenizer, used without training: Llama-3.2-1B for Llama-3.1-8B and Qwen3-1.7B for the three Qwen3 models; Mistral-7B-v0.3 has no released small sibling on its tokenizer, so that row is empty. The n-gram drafter proposes the token that followed the longest matching suffix, 2 to 4 tokens long, of the prompt and the output so far; a position with no match counts as a miss.

Prompt sets. GSM8K and MATH-500 (mathematics), HumanEval and MBPP (code), Alpaca and MT-Bench (chat); 32 prompts each at their natural length (median 98, 40 and 21 tokens by domain), 256 new tokens, K=1, greedy. The three non-InvisibleLoRA drafter rows of Table 1 are teacher-forced over the committed tokens of the InvisibleLoRA run, so all four drafters are scored on the same committed prefixes (in exact arithmetic these are the target's greedy prefixes, §2.3), and one run per (adapter, prompt set) scores all four drafters.

Throughput cells. A cell is one (model, N); both arms run in the same engine over identical prompts at batch 64 with 256 new tokens. The N slots are populated from the audited pool by the replication rule above; where the pool holds at least N distinct adapters they are a random sample drawn once per seed and nested across N, so the set at N=16 contains the set at N=8; we report the mean of three seeds and show their range. Because vLLM picks the LoRA kernel configuration from the number of rows in a forward pass, we pin `split_k` to 8 in both arms, the baseline's own optimum (Appendix D). Each arm of each cell is measured twice; when the two repeats differ by more than 5% we measure a third and report the median. Qwen3-32B does not fit on one H100 with a pool (65.6 GB of weights against 1.07 GB per rank-64 slot), so it alone runs under TP=2.

Two arms, two floors. The reported quantity is a ratio, so a noise check that watches only the baseline admits cells whose other arm is moving. Over the 41 cells of the grid, the batch axis and the pools, the baseline's repeat spread has a median of 0.3% and a maximum of 1.1%; the InvisibleLoRA arm's has a median of 1.2% and a maximum of 13%, and 7 cells exceed 5%. Six repeats of one cell on one GPU spread by 1.0% and six on six different GPUs by 0.4%, so GPU-to-GPU variation does not appear to explain the larger InvisibleLoRA spread; it is consistent with bf16 token trajectories diverging slightly between runs and changing which drafts are accepted, and the cells that move are those near break-even. We report both spreads and do not judge a cell whose worse arm exceeds 5%.

Warm-up. At small batches the measurement window is short enough that adapter slots still loading during the window register as baseline cost; we warm up with the full batch composition for 32 tokens before every measured window.

Implementation details that decide whether it works. The LoRA-off forward pass must skip the LoRA kernels rather than multiply by zero, or the draft costs what it was meant to save; and the proposer must take the target's attention-layer list so that drafts land in the target's cache. A third mistake fails silently: if the sampled token is treated as the drafter's first *proposal* rather than its first *input*, the scheme degrades to plain decoding plus wasted drafts while still producing correct output. Check the accepted-token count per cycle, not the output.

Hook sites. The three hook sites are method registration in the speculative configuration, proposer construction and invocation in the GPU model runner, and a short-circuit on Punica's GPU path. The patch, the pool manifest, the launch-configuration override used to pin both arms, and every per-cell result file are released with the paper. Hook locations move between releases, and the upstream `custom_class` proposer hook passes only host-side token ids, so it cannot host a drafter that runs a forward pass.

F The adapter pool

How a pool is gathered. One adapter per repository and at most two per owner, so that no single fine-tuning recipe dominates; no adapter carrying output-layer tensors, re-checked after download rather than taken from the model card; repositories whose names mark them as translation or long chain-of-thought models excluded (domain labels are assigned afterwards from metadata, which is why a few translation adapters remain); and every file pinned by hash. The same rule builds every pool in the paper, applied to each base model's own public adapters, and the audit below runs on the result.

Admission audit. The acceptance an adapter will see is measured before it is admitted, not read from the engine's mean: one LoRA-on run of 16 requests and 128 new tokens with drafting enabled, about 40 seconds on one H100, reporting the accepted-token fraction. An adapter below the floor a deployment sets (0.90 in this paper) is served without drafting.

Two pool generations. Two pools for Llama-3.1-8B-Instruct stand behind this paper, both built by that rule and both released. The *grid pool* holds 103 adapters with ranks {8:14, 16:38, 32:27, 64:19, 128:1, 256:4}. Its five members of rank > 64 are the only adapters wider than 64 in either pool, and they are what makes the slot-128 and slot-256 columns of the offline grid possible, so the cost model's grid (Appendix B) and the drafting-length axis (Appendix C) use it. By domain its 98 adapters of rank ≤ 64 are 58 chat, 13 classification, 8 structured output, 6 medical, 5 summarization, 3 translation, 3 code and 2 mathematics. The *serving pool* is the later and larger one, 146 adapters with ranks {8:24, 16:54, 25:1, 32:35, 64:32}, capped at rank 64 because that is the natural slot width (Appendix H); the acceptance survey of §4.2, the adapter axis of §4.3 and the serving runs of §4.4 draw from it. The two are not interchangeable: replaying a result against the wrong manifest changes the slot width and with it the adapter cost, so every result file names its pool.

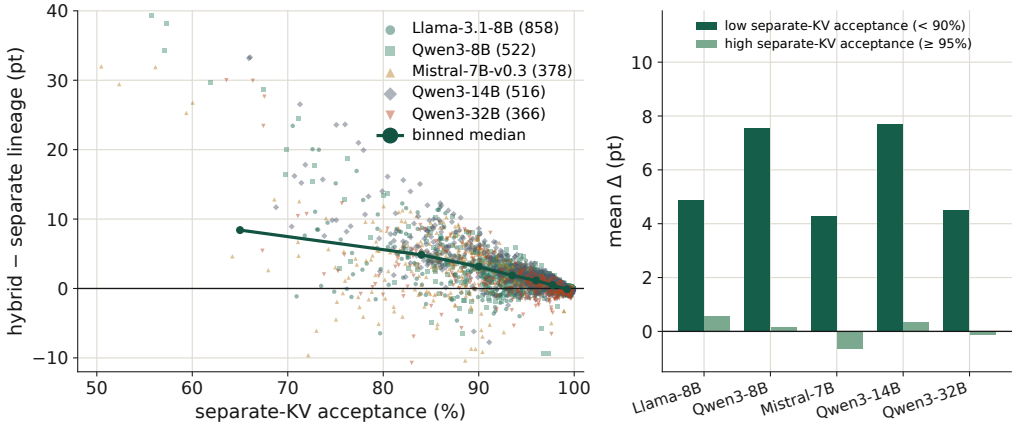
Every adapter in both is pinned by repository and by the SHA-256 of the weight file as measured. The builder verifies each hash on download and halts on drift rather than substituting a replacement, so a pool built from the manifest is byte-identical to the one behind every number reported here.

The grid pool's five adapters of rank > 64 carry no domain label, so their own-domain acceptance is measured on instruction prompts.

How the grid's pools are built. A column of the offline grid (Appendix B) at slot width w uses only adapters of rank exactly w, so that the slot width is set by construction rather than by whichever adapter happens to be widest. The pool holds 19 adapters of rank 64, one of rank 128 and four of rank 256; a cell with N slots takes the largest divisor d of N with d adapters available and replicates each N/d times into distinct slots. The slot-64 column therefore uses 8 to 16 distinct adapters, the slot-128 column one, and the slot-256 column four. Each replica is a separate LoRA slot to the engine and is read separately, so the cost side of the measurement is that of N adapters; the acceptance side is that of the replicated adapters, which for the rank-128 adapter is $\alpha = 0.97$ on mathematics prompts, near the pool median.

G Acceptance: the other design and the paired differences

Table 1 in the body reports one of two designs; this appendix gives the other, and the per-adapter paired differences that decide which drafter is ahead. *Design A*, the one in the body, places every adapter of a pool on every prompt set, so a cell is the pool median over 64 to 146 adapters. *Design B* (Table 7) places only the adapters of the matching domain, so a cell is the median over the adapters trained for that kind of prompt, with the count shown. It is the deployment case, where a request is routed to an adapter of its own domain, but its cells are thin: mathematics adapters for the 32B model are two, because only two on the public hub pass the pool-building rule. Neither design applies an acceptance floor; the 0.90 floor of §4.1 enters only in the throughput study.



(a) Gain against separate-KV acceptance.

(b) Mean gain, low vs. high separate-KV acceptance.

Figure 10: Acceptance gain of the shared cache against separate-KV acceptance, five models, the 2,640 (adapter, prompt set) cells of the 440 adapters passing the 0.70 screen. (a) The x axis is acceptance with base-computed keys and values, the y axis the gain from reading the LoRA-conditioned cache instead; the line is the binned median. (b) Mean gain for cells with low separate-KV acceptance ($x < 90$) and high ($x \geq 95$).

Medians are not subtracted. The two base-model rows of Table 1 are medians over the same adapters, but a median of one row and a median of the other need not come from the same adapter, so their difference is not a paired quantity. Table 8 subtracts adapter by adapter and reports the median of the differences with the number of adapters on which InvisibleLoRA is ahead. Every domain-matched cell is at or above zero. In the pool-wide design one cell, Qwen3-32B on MATH-500, has a paired median of −0.3 (24 of 64 adapters ahead); every other cell is at or above zero.

Table 7: The design not shown in the body: domain-matched cells, with the adapter count in gray. Acceptance of drafts (%), otherwise as in Table 1.

		Math		Code		Chat		
Model	Drafter	GSM8K	MATH-500	HumanEval	MBPP	Alpaca	MT-Bench	Avg
Llama-3.1-8B	*n*-gram	45.4 (12)	43.1 (12)	49.9 (14)	40.0 (14)	13.7 (64)	21.7 (64)	35.6
	small model†	88.7 (12)	91.1 (12)	90.0 (14)	87.3 (14)	84.1 (64)	80.8 (64)	87.0
	base model (separate KV)	92.0 (12)	94.5 (12)	94.7 (14)	93.7 (14)	92.5 (64)	92.1 (64)	93.3
	InvisibleLoRA (hybrid KV)	**96.5** (12)	**97.1** (12)	**96.0** (14)	**94.5** (14)	**94.7** (64)	**94.1** (64)	**95.5**
Mistral-7B-v0.3	*n*-gram	49.5 (2)	57.8 (2)	64.6 (5)	56.9 (5)	18.0 (13)	27.0 (13)	45.6
	base model (separate KV)	86.6 (2)	86.0 (2)	80.5 (5)	89.7 (5)	85.6 (13)	84.0 (13)	85.4
	InvisibleLoRA (hybrid KV)	**87.0** (2)	**88.0** (2)	**83.0** (5)	**96.5** (5)	**89.9** (13)	**89.8** (13)	**89.0**
Qwen3-8B	*n*-gram	32.3 (4)	28.3 (4)	75.6 (15)	79.8 (15)	14.2 (5)	23.0 (5)	42.2
	small model†	90.5 (4)	92.3 (4)	94.9 (15)	96.5 (15)	80.4 (5)	78.6 (5)	88.9
	base model (separate KV)	93.7 (4)	95.9 (4)	98.5 (15)	99.6 (15)	90.1 (5)	89.4 (5)	94.5
	InvisibleLoRA (hybrid KV)	**95.7** (4)	**96.4** (4)	**98.7** (15)	**99.7** (15)	**97.4** (5)	**96.5** (5)	**97.4**
Qwen3-14B	*n*-gram	33.9 (6)	28.7 (6)	34.8 (5)	57.2 (5)	12.7 (2)	17.8 (2)	30.9
	small model†	84.8 (6)	90.1 (6)	89.1 (5)	89.8 (5)	78.5 (2)	76.6 (2)	84.8
	base model (separate KV)	87.9 (6)	92.9 (6)	98.4 (5)	98.6 (5)	93.7 (2)	92.7 (2)	94.0
	InvisibleLoRA (hybrid KV)	**96.3** (6)	**97.5** (6)	**98.6** (5)	**99.4** (5)	**94.9** (2)	**93.0** (2)	**96.6**
Qwen3-32B	*n*-gram	25.1 (2)	29.2 (2)	43.5 (7)	28.1 (7)	19.5 (8)	30.6 (8)	29.3
	small model†	79.3 (2)	84.5 (2)	88.7 (7)	81.0 (7)	67.3 (8)	67.4 (8)	78.0
	base model (separate KV)	90.0 (2)	92.9 (2)	98.7 (7)	97.6 (7)	82.5 (8)	83.1 (8)	90.8
	InvisibleLoRA (hybrid KV)	**93.3** (2)	**94.0** (2)	**98.6** (7)	**94.5** (7)	**85.7** (8)	**86.4** (8)	**92.1**

Table 8: Paired acceptance difference, InvisibleLoRA minus separate KV, per adapter: median difference in points and, in gray, the number of adapters on which InvisibleLoRA is ahead over the number paired.

	Math		Code		Chat		
Model	GSM8K	MATH-500	HumanEval	MBPP	Alpaca	MT-Bench	Avg
Design A: every adapter							
Llama-3.1-8B	+1.4 (123/146)	+0.5 (103/146)	+1.0 (122/146)	+1.1 (118/146)	+0.7 (97/146)	+1.4 (101/146)	+1.0
Mistral-7B-v0.3	+2.8 (55/72)	+3.0 (51/72)	+1.7 (53/72)	+2.4 (57/72)	+2.3 (52/72)	+3.5 (55/72)	+2.6
Qwen3-8B	+0.8 (71/88)	+0.3 (60/88)	+0.4 (71/88)	+0.2 (65/88)	+1.0 (61/88)	+1.2 (62/88)	+0.6
Qwen3-14B	+0.5 (53/86)	+0.4 (55/86)	+1.3 (64/86)	+1.5 (67/86)	+1.3 (51/86)	+1.8 (61/86)	+1.1
Qwen3-32B	+0.3 (39/64)	-0.3 (24/64)	+0.6 (45/64)	+0.7 (38/64)	+0.1 (33/64)	+0.7 (35/64)	+0.3
Design B: domain-matched							
Llama-3.1-8B	+3.5 (11/12)	+1.0 (9/12)	+1.0 (9/14)	+1.4 (11/14)	+1.5 (44/64)	+1.9 (43/64)	+1.7
Mistral-7B-v0.3	+0.5 (1/2)	+2.0 (1/2)	+2.5 (3/5)	+4.2 (5/5)	+3.1 (8/13)	+3.4 (10/13)	+2.6
Qwen3-8B	+2.0 (3/4)	+0.5 (3/4)	+0.4 (15/15)	+0.1 (10/15)	+3.9 (5/5)	+2.9 (5/5)	+1.6
Qwen3-14B	+5.8 (5/6)	+4.0 (4/6)	+0.2 (4/5)	+0.9 (4/5)	+1.2 (2/2)	+0.4 (1/2)	+2.1
Qwen3-32B	+3.3 (2/2)	+1.2 (2/2)	+0.1 (4/7)	-0.0 (3/7)	+3.0 (5/8)	+3.1 (5/8)	+1.8

H The adapter axis: input length, slot width and admission floor

All five models. Figure 11 repeats the throughput axis of Figure 4 on all five base models under the same protocol; the two not shown in the body follow the same curve, Qwen3-8B reaching 1.68× at N=64 and Qwen3-32B 1.50× at N=48 under TP=2 (§4.3).

Input length. Figure 12 pads prompts to 128, 512 and 2048 tokens from real dataset text (output fixed at 256). The drafter pays the prefill as the baseline does, so the ordering of the two arms is unchanged at every length and every N we measured; what moves is the level. For Llama-3.1-8B the range over N reads 1.10–1.84× at 128 tokens, 1.07–1.74× at 512 and 1.03–1.73× at 2048, and at a fixed N the gain falls with length (at N=8, 1.10, 1.07 and 1.03×), because the prefill both arms pay grows while the adapter term that InvisibleLoRA amortizes does not. The absolute throughput of both arms falls with length and the time to first token rises for both, by more for InvisibleLoRA because the draft pass precedes the first verification. The 2048-token column is not monotone in N

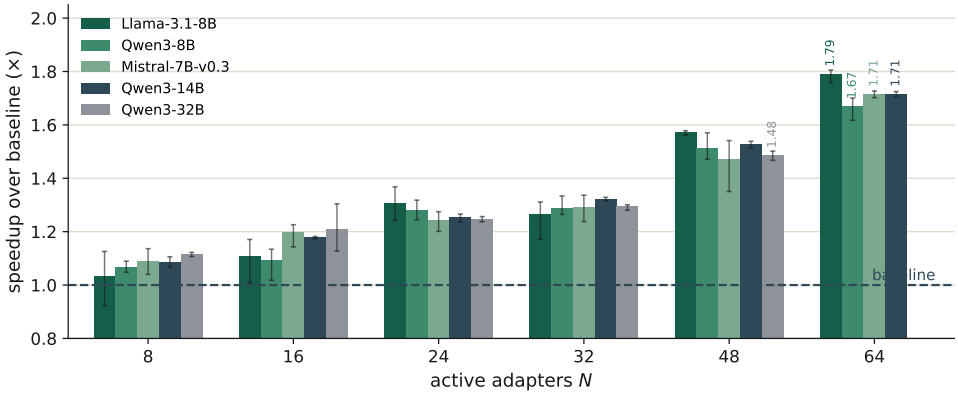


Figure 11: Throughput speedup against the number of active adapters on all five base models, 128-token inputs; audited pools (floor 0.90), slot width 64, batch 64, both arms pinned. Bars are the mean over three random draws of N slots from each audited pool and whiskers their range; Mistral-7B-v0.3 repeats adapters above N=41 (Appendix E). Qwen3-32B runs under TP=2 and stops at N=48.

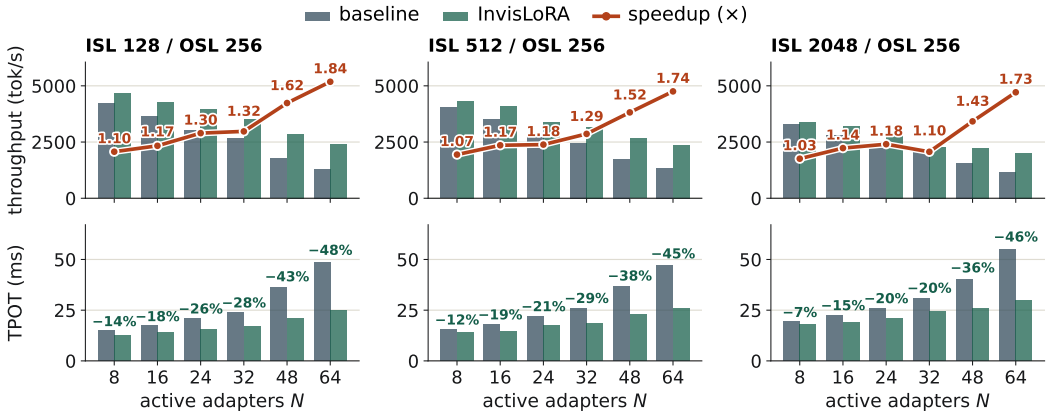


Figure 12: The input-length axis, Llama-3.1-8B: throughput (top) and time per output token (bottom) against N at three input lengths (columns; output 256 tokens). Same protocol as Figure 4.

for three of the four models; the cells that break the order are those where the adapter share of the step is smallest, where (9) is flattest and the noise floor of Appendix E largest relative to the effect. We report them as measured. For Qwen3-32B the InvisibleLoRA arm runs out of memory at 512 and 2048 tokens under TP=2 at batch 64, so those two columns cover four models.

Slot width. The engine pads every LoRA slot to the widest rank in the pool. Our pools cap rank at 64, so 64 is the natural width; forcing 128, as one rank-128 adapter would, reads 1.15, 1.34, 1.26 and 1.70× at $N = 8, 16, 32, 64$ against 1.04, 1.04, 1.21 and 1.81× at the natural width in the same measurement campaign: the multiplier rises where the adapter cost is small and falls at N=64, and it moves by lowering the baseline, whose throughput drops 30% at N=64 (1,346 → 940 tok/s) while InvisibleLoRA's drops 34%. A ratio that improves because the denominator worsens is not a gain; we report the natural width in the body and this column as the effect of one wide adapter in the pool.

Admission floor. At a floor of 0.70 the same protocol reads, at N=8, 1.05 (Llama-3.1-8B), 0.88 (Qwen3-8B), 0.92 (Mistral-7B-v0.3), 1.11 (Qwen3-14B) and 1.09× (Qwen3-32B), rising to 1.74, 1.63, 1.59, 1.39 and 1.48× at the largest N. The two cells below parity are the pools with the widest acceptance tail: a fixed-length run ends when its slowest request does, so at small N the minimum acceptance in the batch sets the wall clock. The floor of 0.90 used in the body is where (9) breaks

even at N=8 for these pools; it removes 8% of the Llama-3.1-8B pool and 43% of the Mistral-7B-v0.3 pool (12 of 146 and 31 of 72 adapters).

Tensor parallelism. Sharding the adapters across devices divides the adapter cost and the base step together, so ρ stays put (0.196 against 0.217) while the multiplier falls by about 10% (1.32 → 1.20× at N=64, TP=2).

I Serving: load axis, batch composition and the gate

Figure 5 in the body reads both arms at the offered load that maximizes InvisibleLoRA's throughput. Figure and table axes in this appendix report the resident pool size; N keeps its meaning of §2.1, the distinct adapters in a batch, and is reconstructed below. Reading each arm at its own peak load instead changes two of the 24 cells — the uniform 96-adapter pool from 1.72 to 1.69× and the trace at 64 resident adapters from 1.25 to 1.20× — and leaves the rest identical, because below saturation both arms carry the offered load and above it the baseline's throughput is nearly flat. This appendix gives the load axis behind the figure at 64 resident adapters, and then the batch composition that both figures turn on. The trace supplies one function per resident adapter: for a pool of R adapters the R busiest functions, read from a 64-function extract for pools up to 64 and a 128-function extract above, with function rank shuffled onto adapter identity by a fixed seed and each minute of the trace replayed as one second.

Unlike Figure 4, which holds the batch at 64, each column of Figure 5 is read at its own peak load (16 to 40 req/s across the grid) and a step's batch is whatever the scheduler assembled; the gate reads both through $g = Nw/B$.

The load axis at 64 resident adapters. Figure 13 sweeps the arrival rate for the three patterns, with the service-level objective marked.

Where the objective binds. Of the 449,895 requests in the serving study, 1,084 miss the per-token bound alone against 187,008 that miss the first-token bound alone: queueing decides the objective, and the per-token gain acts by draining the queue. Below the knee the two arms coincide, InvisibleLoRA's first token up to 13 ms later, the cost of one draft pass; past it the baseline's first-token latency rises by two orders of magnitude within two or three grid points, while the median time per output token stays under 40 ms in every cell. At 64 resident adapters the SLO capacity rises from 10.3 to 15.3 req/s under uniform arrivals, from 14.2 to 18.4 under skew and from 16.2 to 18.9 on the trace.

How many adapters a batch holds. The engine does not log the composition of each step, so §4.4 reconstructs it from the request records. At a grid of instants over the middle 80% of a run, the requests in flight are those that have arrived and not finished, and the running batch is the oldest 64 of them, which is the order the scheduler admits in under a cap of 64. Medians over instants:

distinct adapters in the batch	8 req/s	12	16	20	24
uniform	23	46	51	51	51
skewed	12	19	24	26	26
Azure trace	14	18	23	26	25

Uniform arrivals saturate the resident pool by 16 req/s; the two skewed patterns level off at half of it. That count is the N in ρ, which is why their absolute per-token latency is lower and their relative gain smaller.

The gate under load. Table 9 is the gate result of §4.4 in full, and Figure 14 repeats Figure 5 without the gate. Two things about how it is read. The speedup of a gated cell is taken against the baseline arm of the *same* run group, never against the ungated group's baseline: the baseline does not depend on the gate, but it varies between run groups by up to 11%, and comparing across groups turns that variation into an apparent gate effect. And g is reconstructed, not logged — the engine prints the gate variable only for the first step — from the slot width, the running batch and the adapter count of the paragraph above.

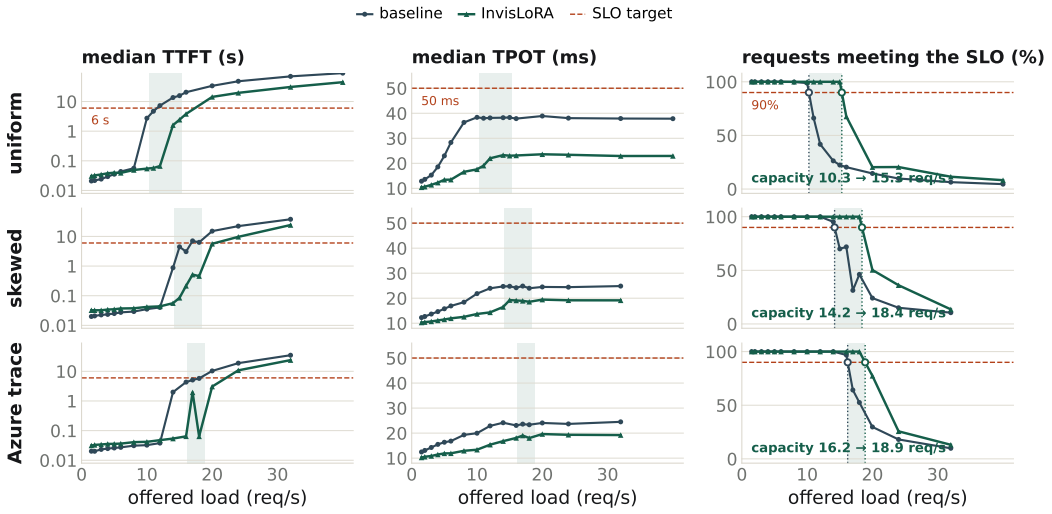


Figure 13: Serving under Poisson arrivals: Llama-3.1-8B, 64 resident adapters, batch cap 64. Rows are arrival patterns — uniform, skewed with popularity $\propto$ rank^{-1}, and a replay of the Azure Functions trace. A request meets the SLO when its first token arrives within 6 s (Sheng et al., 2024) and its mean time per output token stays under 50 ms; the SLO capacity is the largest rate at which 90% comply (Zhong et al., 2024), where the hollow markers sit. Shading spans the loads InvisibleLoRA serves within the objective and the baseline does not. Cells are medians over requests and over two to eight 60-second windows.

Table 9: The gate under an arrival process, Llama-3.1-8B, batch cap 64. Each entry is ungated / τ_{gate}=10, a within-group ratio of median throughput over two to five 60-second windows; a dash is a cell we did not run. $g = Nw/B$ at the saturating batch. Bold marks the cells the ungated method loses; the gate closes for pools of 8 and 16 and stays open from 24 resident adapters. For the 8-adapter pool the gated runs used the batched threshold of 5, which closes the same cells.

				speedup, ungated / τ=10		
pool	w	load	g	uniform	skewed	Azure trace
8	32	16	4.0	0.98 / 1.00	1.01 / —	1.00 / 1.00
8	32	20	4.0	1.00 / 1.00	**0.94** / —	1.04 / 1.00
8	32	24	4.0	**0.80** / 1.01	**0.93** / 1.00	**0.82** / 0.99
8	32	40	4.0	**0.85** / 0.99	**0.79** / 0.98	**0.89** / 0.99
16	32	16	8.0	1.01 / —	1.00 / —	1.00 / —
16	32	20	8.0	0.99 / —	0.99 / —	0.99 / —
16	32	24	8.0	**0.95** / 1.00	**0.93** / 1.00	**0.93** / 1.00
16	32	32	8.0	**0.95** / 1.00	**0.91** / 1.00	**0.93** / 0.99
16	32	40	8.0	**0.85** / 1.00	**0.94** / 1.00	**0.92** / 0.98
24	64	24	24.0	1.15 / 1.16	1.08 / 1.09	1.07 / 1.08
64	64	16	64.0	1.56 / —	1.13 / —	1.09 / —

The threshold, its calibration on the uniform rows of this table, and the range over which it holds are in Appendix I.

The gate. Section 3.3 gates drafting on $g = Nw/B$ and §4.4 reports the threshold; the rest of this appendix gives the calibration, the range over which the value holds, and the form's derivation.

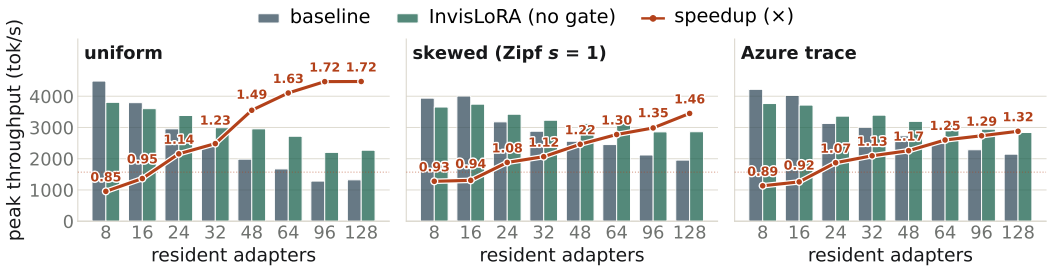


Figure 14: Figure 5 without the gate. Everything else is identical, so the two read together: below 24 resident adapters the drafter costs more than it saves, by 5 to 15% at the load where InvisibleLoRA peaks, and the gate of §3.3 is what holds those pools at parity in the body figure rather than any property of the drafter. The crossing sits between 16 and 24 adapters at the pool's own slot width, and between 8 and 16 when the width is pinned to 64.

Calibration returns an interval, not a point. The calibration is the uniform rows of Table 9: pool sizes 8 to 64 at two to four loads past the knee, both arms, 60-second windows. Order by g the cells whose worse arm repeats within the 5% floor of Appendix E. Every losing cell sits at $g \leq 8.0$ and every winning one at $g \geq 12.1$; no judged cell lies in the gap, so any threshold inside it decides all of them alike. Sweeping the threshold, 9 through 12 misjudge none, 8 misjudges two and 14 one. We take $\tau_{\text{gate}} = 10$; the gate drafts when $g \geq \tau_{\text{gate}}$, so a cell at $g = 8.0$ drafts under a threshold of 8.

The value holds over what a deployment can change. We calibrated on uniform arrivals alone and left the other two patterns out of it. Over 97 judged cells the same threshold misjudges none across

- **slot width** — pinning the width to 64 for pools of 8 and 16, where the pool's own maximum rank is 32, moves g from 4–8 to 8–16 and the measurements follow: g=8 still loses (0.88–0.95×) and g=14 does not (0.99–1.03×), on all three patterns;
- **batch cap** — raising it from 64 to 128 for the 16-adapter pool leaves twelve cells, all decided correctly;
- **arrival pattern** — skewed and trace-driven runs, which the calibration never saw;
- **pool size** — 8 to 128 resident adapters;
- **base model** — repeating the procedure on Qwen3-8B gives losing cells at $g \leq 8.0$ and winning ones at $g \geq 17.0$, an interval that contains 10 as well.

Simpler variables do not separate. Dropping either factor breaks the split once both models are pooled: N alone, N/B, N/B_{cap} and B/N all overlap. Solving for exponents in Nw^a/B^b, the error-free region spans $a \in [0.25, 1.5]$ and $b \in [0.5, 1.5]$; $a = b = 1$ sits inside it with the losing and winning classes 52% apart. Both factors are load-bearing: with $b = 0$ the classes overlap at every a, and with $a = 0$ at every b.

Where the form comes from. Setting $S_K = 1$ in (8) and solving for the adapter cost rather than for ρ gives the break-even tax

$$A^*(B) = \frac{\Gamma((K+1)B) + \big(K - \mathbb{E}[\tau_K]\big)\,\Gamma(B)}{\mathbb{E}[\tau_K] - 1}, \tag{12}$$

which grows almost in proportion to B even though Γ itself barely does, because it is a difference of Γ at two points: A^*/B runs over 0.044 to 0.057 ms per request for B from 16 to 128. A break-even

that is a constant on A/B is a constant on Nw/B once the adapter cost is taken as proportional to Nw, which is (5), and at w=64 on an H100 it puts the threshold at 5.0.

Two things about the form are measured rather than derived. The tax of (11) is $N(\kappa_0 + \kappa_1 w)$, so the variable the derivation hands over is $N(\kappa_0 + \kappa_1 w)/B$ rather than Nw/B; measured, the κ-weighted variable separates the losing and winning classes by 14% against 52%, and its crossing moves with the slot width ($[0.121, 0.183]$ at w=32, $[0.076, 0.138]$ at w=64) where Nw/B holds one crossing at both. Under an arrival process the slot width therefore carries more of the decision than the batched fit's κ_1 assigns to it, and we use the variable the measurements select.

The form transfers; the value is measured. Under an arrival process the threshold is twice that. The drafter, and not the baseline, pays an extra cost there: for the 8-adapter pool the baseline's step is unchanged from the batched grid (14.6 → 14.5 ms) while InvisibleLoRA's rises (14.6 → 16.4 ms). Carrying a per-request term P through (12) reproduces $\tau_{\text{gate}} = 10$ at $P \approx 0.047B$ ms, and the step times measure $P = 3.5$ ms at B=64 against the 3.1 the threshold implies. We can therefore say where the threshold moves and that it moves by a per-request cost — which is what keeps (5)'s form intact, since a cost that did not scale with B would break it — but we do not derive the cost itself. Acceptance is flat across batch sizes (0.95), so the shortfall is not an acceptance effect.