# PREFILLSHARE: A SHARED PREFILL MODULE FOR KV REUSE IN MULTI-LLM DISAGGREGATED SERVING

**Anonymous authors**
Paper under double-blind review

## ABSTRACT

Multi-agent systems increasingly orchestrate multiple specialized language models to solve complex real-world problems, often invoking them over a shared context. This execution pattern repeatedly processes the same prompt prefix across models. Consequently, each model redundantly executes the prefill stage and maintains its own key–value (KV) cache, increasing aggregate prefill load and worsening tail latency by intensifying prefill–decode interference in existing LLM serving stacks. Disaggregated serving reduces such interference by placing prefill and decode on separate GPUs, but disaggregation does not fundamentally eliminate inter-model redundancy in computation and KV storage for the same prompt. To address this issue, we propose PrefillShare, a novel algorithm that enables sharing the prefill stage across multiple fine-tuned models in a disaggregated setting. PrefillShare factorizes the model into prefill and decode modules, freezes the prefill module, and fine-tunes only the decode module. The shared-prefill design allows multiple task-specific models to share a prefill module and the KV cache generated for the same prompt when prefix overlap is high, while shared prefill workers pool prefill capacity across models and mitigate load imbalance even when prefix overlap is limited. We further implement a routing and cache-handoff mechanism in a vLLM-based disaggregated system to support shared-prefill execution across heterogeneous specialized decoders. PrefillShare not only matches full fine-tuning accuracy on a broad range of tasks and models, but also delivers 4.5× lower p95 latency and 2.0× higher throughput in multi-model agent workloads.

## 1 INTRODUCTION

Large language models (LLMs) have demonstrated strong performance across reasoning, code generation, and knowledge-intensive question answering (Chowdhery et al., 2023; Team et al., 2023), motivating agentic workflows in which models use tools, memory, and external resources (Schick et al., 2023; Shinn et al., 2023; Yao et al., 2022). Recent systems extend this paradigm to multi-model workflows, where specialized models collaborate by leveraging distinct fine-tuned capabilities (Hong et al., 2024; Su et al., 2025; Wu et al., 2024). While these workflows improve modularity and task performance, they introduce substantial inference inefficiencies when multiple models operate on a shared context. Modern serving systems rely on KV caches to avoid recomputing attention over previous tokens in decoder-only Transformers (Kwon et al., 2023; Zheng et al., 2024), but KV caches are tied to model parameters and cannot be reused across differently fine-tuned models. As a result, each model must independently prefill the same context and maintain its own KV cache, increasing time-to-first-token (TTFT) and tail latency due to prefill/decode contention while also inflating memory usage. Disaggregated serving mitigates this contention by separating the compute-bound prefill phase from the memory-bound decoding phase (Patel et al., 2024; Zhong et al., 2024); however, it does not eliminate cross-model redundancy, leaving identical prompts to incur duplicated prefill computation and KV-cache across specialized models.

Recent works have begun to address these inefficiencies through cross-model KV reuse. KVCOMM (Ye et al., 2025) reuses KV cache across overlapping contexts by aligning cache offsets with an online anchor pool, while DroidSpeak (Liu et al., 2024) enables partial KV sharing by reusing states from non-sensitive components under specific architectural assumptions. Although these methods show the promise of cross-model reuse, they typically require additional coordination and may not generalize robustly without explicit training for KV-cache sharing. ICaRus (Woo et al., 2026)



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.

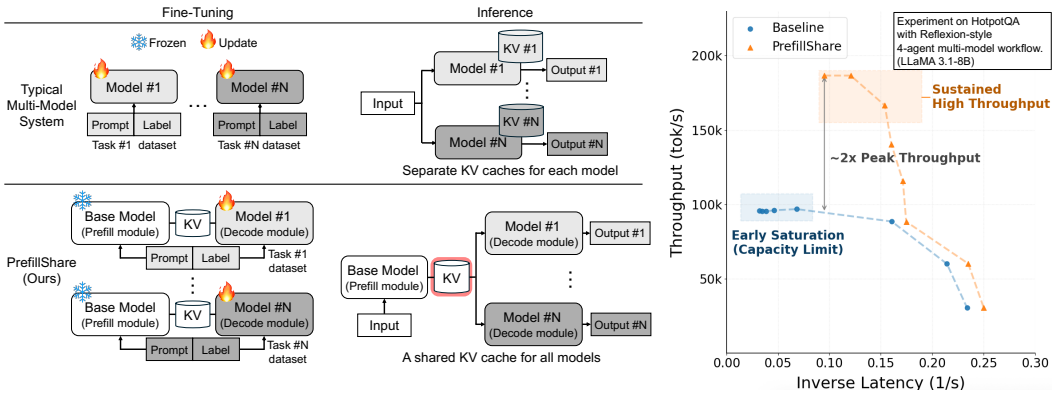



Figure 1: **Comparison between conventional multi-model serving and PrefillShare. (Left)** Typical multi-model systems fine-tune and serve each task-specific model independently, causing redundant prefill computation and KV caches for identical prompts. PrefillShare partitions a decoder-only Transformer into a shared prefill module and task-specific decode modules, allowing multiple specialized decoders to reuse the same prompt KV cache. **(Right)** In disaggregated serving, redundant prefill and per-model KV-cache growth increase computation and trigger cache evictions under high load. By sharing prompt KV caches across models, PrefillShare improves prefix-cache reuse, delays cache saturation, and achieves up to $2\times$ higher throughput.

takes a different approach by decomposing a decoder-only Transformer into a frozen logical encoder and a trainable logical decoder, enabling identical KV reuse across fine-tuned models for the same prompt. However, since its decoder consumes only encoder-generated KV caches, the encoder must still run during decoding, adding compute overhead and limiting scalability under high concurrency.

In this work, we propose PrefillShare, a shared-prefill disaggregated serving algorithm for multi-model agent workloads. As illustrated in Fig. 1, PrefillShare partitions a decoder-only Transformer into a shared prefill module and task-specific decode modules: the prefill module processes the shared context once to produce a reusable KV cache, while specialized decoders generate outputs conditioned on this cache. To enable robust shared-prefill execution, PrefillShare employs cache-conditioned fine-tuning, which freezes the prefill module and trains each task-specific decoder to consume the base model's KV cache. We further implement efficient routing and cache handoff in a vLLM-based disaggregated system to preserve shared-cache locality across heterogeneous model switches. PrefillShare not only eliminates redundant prefill computation and duplicated KV storage when prefix overlap is high, but also pools prefill capacity across models to mitigate load imbalance when prefix overlap is limited. As a result, PrefillShare preserves the accuracy of full fine-tuning across diverse tasks and models while substantially reducing tail latency and improving throughput in multi-model serving workloads.

In summary, we make the following contributions:

- **Cross-model prefill and KV-cache sharing.** We propose PrefillShare, a novel shared-prefill serving algorithm that allows multiple task-specific models to reuse the same prefill computation and KV cache for identical prompts in disaggregated serving, eliminating cross-model redundancy in both computation and memory.
- **Cache-conditioned fine-tuning for robust sharing.** We introduce a disaggregation-aware fine-tuning procedure that freezes the shared prefill module and trains only task-specific decode modules conditioned on the shared KV cache, aligning training with shared-prefill inference while matching full fine-tuning accuracy.
- **Efficient routing and cache handoff.** We implement a routing and cache-handoff mechanism in a vLLM-based heterogeneous disaggregated serving system for different fine-tuned models, making shared-prefill inference practical and achieving up to **4.5**$\times$ lower p95 latency and **2.0**$\times$ higher throughput on multi-model agent workloads.

# 2 BACKGROUND

## 2.1 AUTOREGRESSIVE INFERENCE AND KV CACHING

Let $X = \{x_1, \ldots, x_n\}$ denote an input prompt of length $n$, and let $Y = \{y_1, \ldots, y_T\}$ denote the generated sequence of length $T$. An LLM parameterized by $\theta$ models the conditional likelihood as

$$P(Y|X) = \prod_{t=1}^{T} P(y_t|X, y_{<t}; \theta), \tag{1}$$

where $y_{<t}$ denotes previously generated tokens.

Decoder-only Transformers avoid redundant attention computation by maintaining a KV cache. We write the Transformer forward pass as

$$(y, \Delta\mathcal{C}) = \mathcal{F}_\theta(x, \mathcal{C}_{\text{past}}), \tag{2}$$

where $x$ is the current input, $\mathcal{C}_{\text{past}}$ is the existing cache, $y$ is the next token, and $\Delta\mathcal{C}$ is the newly generated KV state. In the prefill phase, the model processes the entire prompt in parallel:

$$(y_1, \mathcal{C}_n) = \mathcal{F}_\theta(X, \emptyset). \tag{3}$$

It then enters the decode phase, generating tokens autoregressively as

$$(y_t, \Delta\mathcal{C}_t) = \mathcal{F}_\theta(y_{t-1}, \mathcal{C}_{n+t-2}), \quad \mathcal{C}_{n+t-1} = [\mathcal{C}_{n+t-2}; \Delta\mathcal{C}_t]. \tag{4}$$

Since the cache grows linearly with sequence length and must be repeatedly accessed during decoding, the decode phase is typically memory-bound.

## 2.2 MULTI-MODEL INFERENCE AND PREFIX CACHING

Agentic workflows increasingly deploy multiple specialized models $\{\mathcal{M}_1, \ldots, \mathcal{M}_N\}$ to solve complex tasks. In such workflows, different models often process the same shared context $X$, such as a long conversation history, a common system prompt, or a shared document, while producing task-specific outputs. This creates repeated prefill computation over identical or highly overlapping contexts across models.

A standard way to avoid repeated prefill within a single model is prefix caching (Kwon et al., 2023; Zheng et al., 2024). For an incoming request with a prefix $P$, the serving system checks whether the corresponding prefill cache $\mathcal{C}_{|P|}$, produced by Eq. (3), already exists in the cache pool. If a match is found, the cached KV blocks are reused and the corresponding prefill computation is skipped, turning repeated prefill into a lightweight cache lookup.

However, this single-model reuse mechanism does not directly extend to multi-model settings because the cached state produced in Eq. (3) is parameter-dependent and is consumed by the corresponding model during decoding in Eq. (4). Even for the same prompt $X$, two models $\mathcal{M}_i$ and $\mathcal{M}_j$ with $\theta_i \neq \theta_j$ produce different prefill caches, $\mathcal{C}_n^{(i)}$ and $\mathcal{C}_n^{(j)}$, making their KV states incompatible. As a result, typical multi-model systems must recompute prefill and maintain separate KV caches for each specialized model, causing redundant computation and duplicated memory usage.

## 2.3 DISAGGREGATED SERVING

Modern LLM serving systems use continuous batching (Kwon et al., 2023; Yu et al., 2022; Zheng et al., 2024) to improve throughput by interleaving prefill and decode requests in the same scheduling loop. While effective for GPU utilization, this couples two phases with different resource characteristics: prefill is compute-bound, whereas decoding is memory-bound and repeatedly accesses the KV cache. When a new prefill request is inserted into an active batch, it can delay ongoing decoding and cause spikes in inter-token latency (ITL), degrading interactive responsiveness.

Disaggregated serving (Patel et al., 2024; Zhong et al., 2024) mitigates this interference by placing prefill and decode on separate instances. By isolating compute-heavy prefill from the decode loop, it stabilizes ITL and allows prefill capacity to scale independently. However, disaggregation alone does not make KV caches reusable across models. Thus, in multi-model workloads, identical prompts still require per-model prefill computation and separate KV storage, leaving cross-model redundancy unresolved.

# 3 METHOD

## 3.1 DEFINITIONS AND FORMULATION

Building on the autoregressive formulation in Section 2, we first define two operational roles for inference: a prefill module that produces a shared KV cache from the input prompt and a decode module that generates task-specific outputs by consuming the shared cache.

**Base Prefill Module.** Let the base prefill module be a pretrained model $\mathcal{M}_{\text{base}}$ with parameters $\theta_{\text{base}}$. The parameters $\theta_{\text{base}}$ are responsible only for the prefill phase. It processes the input $X$ once to produce a shared KV cache $\mathcal{C}_{\text{base}}$

$$(\cdot, \mathcal{C}_{\text{base}}) = \mathcal{F}_{\theta_{\text{base}}}(X, \emptyset) \tag{5}$$

Here $\mathcal{C}_{\text{base}}$ denotes the shared KV cache. Note that $\mathcal{M}_{\text{base}}$ computes the KV cache but does not participate in the subsequent token generation loop.

**Specialized Decode Module.** For each task, we construct a decode module $\mathcal{M}_{\text{dec}}$ with parameters $\theta_{\text{dec}}$. At inference time, the decode module initializes its cache with $\mathcal{C} \leftarrow \mathcal{C}_{\text{base}}$ and generates the output sequence:

$$(y_t, \Delta\mathcal{C}_t) = \mathcal{F}_{\theta_{\text{dec}}}(y_{t-1}, \mathcal{C}) \tag{6}$$

This separation enables multiple decode modules to reuse the same shared prefill cache while preserving task-specific generation behavior.

## 3.2 CACHE-CONDITIONED FINE-TUNING

A key challenge in PrefillShare is that a task-specific decoder must generate tokens conditioned on KV caches produced by a different parameterization. In standard fine-tuning, prefill and decode are performed by the same fine-tuned model, so the decoder is trained only on its own KV-cache distribution. This setup tightly couples the decoder parameters to their own cache representations observed during training. As shown in Fig. 2, naively reusing an increasing fraction of KV caches from the pretrained base model introduces a distribution mismatch, causing accuracy to degrade and eventually collapse.

To address this mismatch and enable robust shared-prefill inference, we propose cache-conditioned fine-tuning, a disaggregation-aware training procedure that explicitly trains the decode module to consume $\mathcal{C}_{\text{base}}$. For each training example $(X, Y)$, we first compute the shared cache $\mathcal{C}_{\text{base}}$ using the base prefill module $\mathcal{M}_{\text{base}}$ from input prompt $X$, and treat the cache as a constant input so that

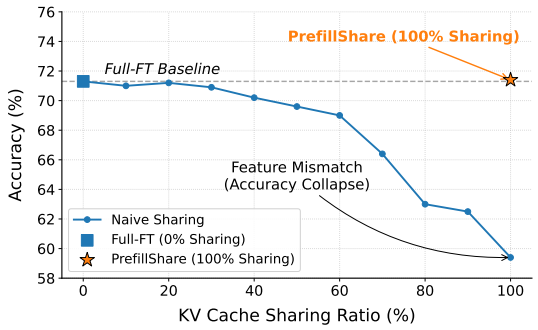



Figure 2: **Naive KV-cache sharing versus PrefillShare.** GSM8K accuracy as a function of the KV-cache sharing ratio between the base and fine-tuned models. Naive sharing collapses at high sharing ratios, whereas PrefillShare preserves near full fine-tuning (Full-FT) accuracy even at 100% sharing.

gradients do not propagate into $\theta_{\text{base}}$. We then fine-tune only the task-specific decode module $\mathcal{M}_{\text{dec}}$ with parameters $\theta_{\text{dec}}$ to predict the target sequence while conditioning on the base cache:

$$\mathcal{L}(\theta_{\text{dec}}) = -\sum_{t} \log P(y_t \mid y_{<t}, \mathcal{C}_{\text{base}}; \theta_{\text{dec}}) \tag{7}$$

We train $\mathcal{M}_{\text{dec}}$ with teacher forcing, feeding the ground-truth prefix $y_{<t}$ while conditioning on the fixed cache $\mathcal{C}_{\text{base}}$. Notably, cache-conditioned fine-tuning preserves the standard next-token prediction objective, but replaces the self-generated cache with the shared base cache as the conditioning signal.

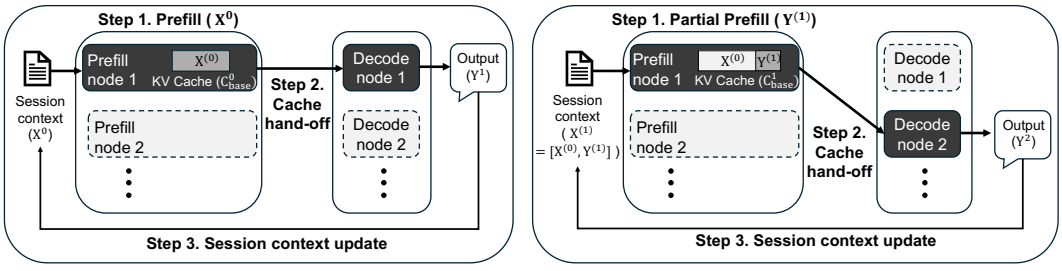
By optimizing Eq. (7), $\mathcal{M}_{\text{dec}}$ learns to generate tokens conditioned on the KV-cache distribution produced by $\mathcal{M}_{\text{base}}$. This adaptation allows $\mathcal{M}_{\text{dec}}$ to effectively interpret the shared base cache $\mathcal{C}_{\text{base}}$, mitigating the representation mismatch that causes naive KV-cache sharing to degrade accuracy, as shown in Fig. 2. Since the training setup matches the shared-prefill inference path, PrefillShare can reliably reuse the same base cache across heterogeneous task-specific decoders without synchronizing their parameters with the base model.

### 3.3 Disaggregated Inference Workflow

Although cache-conditioned fine-tuning enables task-specific decoders compatible with the shared base KV cache, existing disaggregated serving systems are not designed to preserve this cache across model switches. In prior disaggregated systems, prefill requests are scheduled independently for each model or request, so two invocations that share the same session prefix may be routed to different prefill workers. This breaks prefix-cache locality: the worker handling the next model invocation may not hold the previously computed base cache, forcing the system to recompute the shared prefix instead of extending it incrementally.

To preserve prefix-cache locality and realize shared-prefill reuse in practice, we implement a prefix-aware routing and cache-handoff mechanism in a vLLM-based disaggregated serving system. The router pins invocations with the same session prefix to the same prefill worker whenever possible, allowing the worker to keep the base KV cache resident and extend it via partial prefill as new tokens are appended. The resulting cache is then handed off to the selected task-specific decoder, avoiding redundant prefill and duplicated KV-cache storage across model switches as shown in Fig. 3.

Concretely, for the initial invocation, the prefill node performs full prefill over the session context $X^{(0)}$ and constructs $\mathcal{C}^{(0)}_{\text{base}}$, which is handed off to the first decode node to generate $Y^{(1)}$. The generated output is appended to the session context, forming $X^{(1)} = [X^{(0)}; Y^{(1)}]$. For subsequent model switches, the same prefill node performs partial prefill only over the newly appended tokens, extends the cache to $\mathcal{C}^{(1)}_{\text{base}}$, and hands the updated cache to the next selected decode node. Repeating this process enables heterogeneous decoders to reuse a shared prefix cache across model switches, avoiding per-model prefix recomputation and duplicated KV-cache storage. Appendix D.2 compares the memory required for shared-prefix KV caches and model weights under the same GPU budget.




(a) Initial prefill and decode step (b) Subsequent prefill and decode step

Figure 3: **PrefillShare disaggregated inference workflow.** Prefix-aware routing pins same-session requests to the same shared prefill node, allowing the base KV cache to be reused and incrementally extended across model switches. **(a)** The initial step performs full prefill and cache handoff. **(b)** Subsequent steps perform partial prefill and hand off the updated cache to another decode node.

Furthermore, even when prefix overlap is limited, any decoder can be served by any prefill worker, allowing PrefillShare to pool prefill capacity across models, balance load across prefill GPUs, and retain high serving-efficiency.

# 4 EXPERIMENTS

## 4.1 EXPERIMENTAL SETUP

**Training Setup.** We evaluate PrefillShare across multiple model families, sizes, and adaptation domains. We compare cache-conditioned fine-tuning against full fine-tuning (Full-FT) and ICaRus (Woo et al., 2026) using LLaMA-3.1-8B/70B (Grattafiori et al., 2024) and Qwen3-1.7B/8B/14B-Base (Yang et al., 2025). We fine-tune domain-specific models on MetaMathQA-40K (Yu et al., 2023) and DeepMath-1K (He et al., 2025) for math reasoning, EvolInstruct-Code-80K (Luo et al., 2023) for coding, and xLAM-function-calling-60K (Zhang et al., 2025) for tool calling. We evaluate them on GSM8K/GSM+ (Cobbe et al., 2021; Li et al., 2024), MATH-500 (Hendrycks et al., 2021), HumanEval/HumanEval+ (Chen et al., 2021; Liu et al., 2023), and the Simple Python and Multiple subsets of BFCL (Patil et al., 2025), respectively. Additional training details are provided in Appendix A. Appendix D.1 further quantifies the one-time cost of cache-conditioned fine-tuning and its amortization against recurring serving-side savings.

**Inference Setup.** We evaluate PrefillShare in a four-agent, multi-turn serving scenario where agents are invoked sequentially over a largely shared prefix, creating frequent model switches and opportunities for prefill reuse. We use representative agentic prompting patterns, including ReAct and Reflexion, following prior LLM agent and serving studies (Kim et al., 2025; Woo et al., 2026). Additional inference details are provided in Appendix A.

We compare against a disaggregated serving baseline with independent prefill/decode pipelines per model. The baseline uses one prefill GPU and one decode GPU per model, forming four isolated prefill/decode pairs (8 GPUs total). PrefillShare uses the same GPU budget, pooling the four prefill GPUs and sharing prefill caches across decoders. Unless stated otherwise, experiments use LLaMA-3.1-8B on a single A100 node; both systems extend vLLM's disaggregated serving pipeline.

For each arrival rate, we sweep the maximum number of concurrent sessions, an admission-control limit that bounds the active KV-cache footprint, and report the best-performing setting. Appendices C.4 and C.5 analyze the effects of the concurrency limit and input length on serving performance, respectively.

## 4.2 TRAINING AND ACCURACY EVALUATION

**Robust accuracy across tasks.** Table 1 demonstrates that PrefillShare achieves accuracy comparable to Full-FT across math, coding, and tool-calling tasks. Across all benchmarks, PrefillShare remains within 1% of Full-FT and even outperforms it on GSM8K, HumanEval, and the tool-calling benchmark. We observe a similar trend for LLaMA-3.1-70B, where PrefillShare remains competitive

Table 1: **Accuracy on math, coding, and tool-calling tasks.** LLaMA-3.1-8B and LLaMA-3.1-70B after Full-FT and PrefillShare fine-tuning, evaluated on GSM8K/GSM+, HumanEval/HumanEval+, and BFCL, respectively. PrefillShare matches full fine-tuning (Full-FT) accuracy while enabling KV-cache sharing.

| Configuration | KV Sharing | Math | | Coding | | Tool calling | |
|---|---|---|---|---|---|---|---|
| | | GSM8K | GSM+ | HumanEval | HumanEval+ | Simple Python | Multiple |
| LLaMA-3.1-8B | Inherent | 25.9 | 18.0 | 36.6 | 29.9 | 70.5 | 45.5 |
| Full-FT | Not Supported | 71.3 | **49.8** | 48.2 | **45.7** | 90.0 | 88.0 |
| PrefillShare (Ours) | Supported | **71.4** | 49.3 | **48.8** | 45.1 | **90.7** | **88.5** |
| LLaMA-3.1-70B | Inherent | 31.9 | 24.6 | 56.7 | 50.6 | 75.0 | 57.5 |
| Full-FT | Not Supported | 79.2 | 57.9 | **73.2** | **64.6** | **90.0** | 89.5 |
| PrefillShare (Ours) | Supported | **80.1** | **58.9** | 72.3 | 63.8 | **90.0** | **90.5** |

Table 2: **Accuracy across Qwen3 model sizes.** All models are fine-tuned on DeepMath-1K and evaluated on MATH-500. Q3-1.7B/8B/14B-Base denotes Qwen3-1.7B/8B/14B, respectively. PrefillShare achieves accuracy comparable to fine-tuning across all evaluated model sizes.

| Configuration | Model | | |
|---|---|---|---|
| | Q3-1.7B-Base | Q3-8B-Base | Q3-14B-Base |
| Baseline | 60.6 | 69.8 | 81.6 |
| Fine-tuning | **62.8** | 72.4 | 82.8 |
| PrefillShare (Ours) | 62.6 | **72.6** | **83.0** |

as model scale increases. We attribute this robustness to cache-conditioned fine-tuning, which trains the decode module to interpret KV representations produced by the frozen shared prefill module. The same accuracy parity extends to additional math and coding benchmarks and to an out-of-domain evaluation, where PrefillShare remains within 0.55 points of Full-FT across all evaluated settings. Appendix D.3 provides the full results.

**Scaling across model sizes.** We next evaluate whether PrefillShare preserves the reasoning benefits of fine-tuning across model scales using Qwen3-1.7B/8B/14B-Base under a math reasoning fine-tuning setup using LoRA (Hu et al., 2022). As shown in Table 2, fine-tuning on DeepMath-1K consistently improves MATH-500 accuracy over the base models. PrefillShare exhibits the same trend, matching fine-tuning within 0.2% across all model sizes and slightly outperforming it at 8B and 14B, so cache-conditioned fine-tuning preserves the reasoning gains of fine-tuning across model sizes.

### 4.3 Serving Performance under Multi-Model Agent Workloads

Fig. 4 compares the serving performance of PrefillShare against the disaggregated baseline under increasing session arrival rates for both ReAct and Reflexion workloads. At low load, both systems achieve similar latency and throughput. As the arrival rate increases, however, the baseline experiences rapidly growing tail latency and diminished throughput due to redundant prefill work, duplicated KV cache management across model switches, and per-model prefill workers that cannot lend capacity to one another. In contrast, PrefillShare amortizes shared-prefix processing across decoders, sustaining higher throughput while keeping tail latency low. Overall, PrefillShare achieves up to 4.1× lower p95

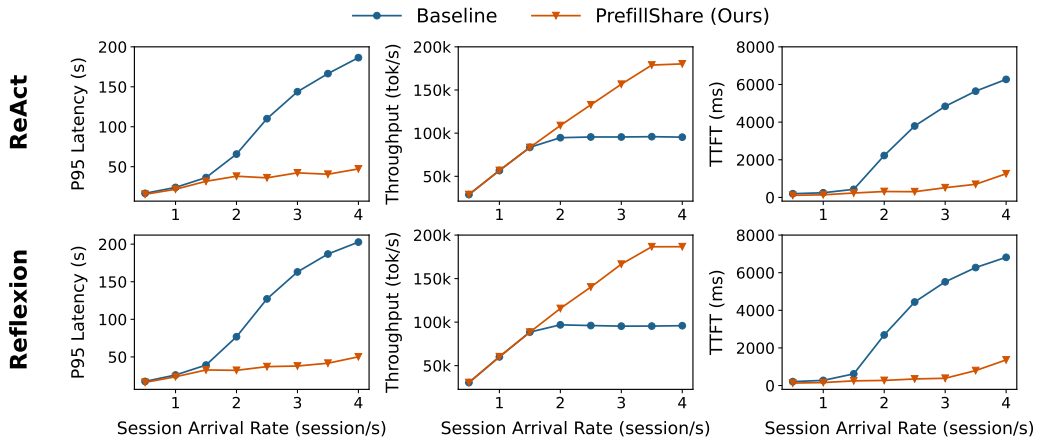



Figure 4: **Serving performance on multi-model agent workloads.** Comparison between PrefillShare and a disaggregated baseline under increasing session arrival rates for two representative agentic patterns: ReAct (top) and Reflexion (bottom). PrefillShare consistently reduces p95 latency and time to first token (TTFT) while improving throughput, with larger gains at higher arrival rates.

Table 3: **Serving performance of LLaMA-3.1-70B on four H100 nodes.** One TP=8 instance per node. Baseline: a prefill and a decode node per model; PrefillShare: two shared prefill nodes.

| Session Arrival Rate (session/s) | Throughput (tok/s) | | TTFT (ms) | |
|---|---|---|---|---|
| | Baseline | PrefillShare (Ours) | Baseline | PrefillShare (Ours) |
| 4 | 37.4k | **39.0k (1.04×)** | 470 | **391 (1.20×)** |
| 8 | 36.6k | **56.8k (1.55×)** | 1,559 | **717 (2.17×)** |
| 48 | 36.5k | **59.8k (1.64×)** | 2,300 | **1,781 (1.29×)** |

Table 4: **Serving performance under low prefix overlap.** An orchestrator and three specialist agents share only a 16-token (0.2%) prefix of an 8,145-token prompt. Qwen3-14B, one 8×H100 node.

| Session Arrival Rate (session/s) | | 0.010 | 0.018 | 0.026 | 0.032 |
|---|---|---|---|---|---|
| p95 latency (s) | Baseline | **5.54** | 9.33 | 31.42 | 36.46 |
| | PrefillShare (Ours) | 5.59 (0.99×) | **8.13 (1.15×)** | **12.07 (2.60×)** | **15.46 (2.36×)** |
| Throughput (tok/s) | Baseline | **17.5k** | 26.5k | 18.9k | 17.0k |
| | PrefillShare (Ours) | 17.4k (0.99×) | **27.6k (1.04×)** | **33.9k (1.79×)** | **36.7k (2.16×)** |

latency and 1.9× higher throughput on ReAct, and up to 4.5× lower p95 latency and 2.0× higher throughput on Reflexion. Appendices B.1–B.2 further show that these gains persist across backbone models and agent invocation patterns. Appendix C.1 separately analyzes the contributions of prefill pooling and cross-model KV reuse. Appendix C.2 further compares PrefillShare with a predictive prefill baseline that assumes perfect next-role prediction.

**Scaling to LLaMA-3.1-70B on multiple nodes.** We further serve LLaMA-3.1-70B on four H100 nodes, each hosting one tensor-parallel instance, with two specialized models. Appendix A.2 provides the detailed multi-node configuration. As Table 3 shows, the baseline saturates at about 36.5k tok/s from 4 sessions/s onward, whereas PrefillShare reaches 59.8k tok/s (1.64×) and reduces TTFT by up to 2.17×. The two metrics peak at different loads: the TTFT speedup is largest near the baseline's capacity limit and shrinks to 1.29× at 48 sessions/s, where decode pressure propagates back to the prefill nodes and both systems queue, whereas the throughput gain keeps growing because pooled prefill capacity keeps more work in flight. PrefillShare reduces only prefill work, so its savings surface only when decode capacity suffices.

**Serving under low prefix overlap.** We further evaluate PrefillShare under a supervisor pattern commonly used in real-world agent workflows (Hong et al., 2024; Wu et al., 2024), where a main agent delegates tasks to subagents and consequently shares less prefix across model switches. As detailed in Appendix A.2, only 0.2% of the prefix is reusable across roles in this workload. As shown in Table 4, even under this near-zero-overlap setting, PrefillShare reduces p95 latency by 62% and improves throughput by 79% at 0.026 sessions/s. The gain primarily comes from better utilization of prefill capacity. In our measurements in Appendix C.3, the max–min worker-load ratio decreases from 42.2× under role-affine placement to 2.07× with shared prefill workers. Concentrating requests on a dedicated worker also induces repeated prefill computation, with the orchestrator worker performing 7.55× its session-local no-eviction prefill work; pooling reduces the aggregate recomputation amplification to 1.06×.

### 4.4 Comparisons to Prior Work

We compare PrefillShare with ICaRus, a prior approach to robust KV-cache reuse across fine-tuned models. As shown in Fig. 5, PrefillShare achieves higher accuracy than ICaRus across all evaluated task categories while also improving serving efficiency. In particular, PrefillShare improves math accuracy by 1.3% over ICaRus, matches Full-FT on coding, and achieves the highest accuracy on tool calling. On the systems side, PrefillShare reduces TPOT from 17.1 ms to 12.2 ms at batch size 1, corresponding to a 28.7% reduction. The benefit becomes larger at higher load: at batch size 8, TPOT decreases from 26.9 ms to 16.5 ms, yielding a 38.7% reduction.

| Configuration | Math | Coding | Tool Calling |
|---|---|---|---|
| Full-FT | 60.6 | **47.0** | 89.0 |
| ICaRus | 59.1 | 46.1 | 88.3 |
| PrefillShare (Ours) | **60.4** | **47.0** | **89.6** |

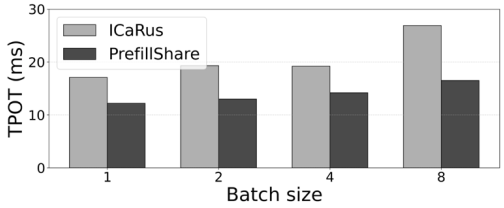



Figure 5: **Comparison between PrefillShare and ICaRus on LLaMA-3.1-8B. (Left)** Accuracy across math (GSM8K/GSM+), coding (HumanEval/HumanEval+), and tool calling (BFCL Simple Python/Multiple) tasks. **(Right)** Time per output token (TPOT) measured on a single A100 GPU with input length 1K. PrefillShare outperforms ICaRus in both task accuracy and serving efficiency.

The difference comes from where KV-cache sharing is enforced. ICaRus keeps the frozen base model on the decoding path so that KV caches remain compatible across fine-tuned variants. This enables reuse of KV states generated during decoding, but the base model and its task-specific adaptation must run together at every decode step; adapter-based tuning such as LoRA reduces this cost, yet limits adaptation capacity relative to full fine-tuning and adds adapter overhead to decoding. PrefillShare instead uses the base model only for shared prefill: the base model produces the shared KV cache once, and fully fine-tuned task-specific decoders consume this cache for generation. This separation preserves full fine-tuning capacity while removing adapter overhead from decoding, which explains PrefillShare's gains in both accuracy and TPOT. Appendix E.1 compares PrefillShare and ICaRus under the same LoRA configuration to control for adaptation capacity. Appendix E.2 provides a qualitative comparison with DroidSpeak and KVCOMM.

## 5 Related Work

**Efficient LLM Serving and KV Management.** vLLM's PagedAttention (Kwon et al., 2023) introduces block-based KV-cache management that reduces fragmentation and improves batching efficiency. Prefix caching (Gim et al., 2024) reuses KV states for repeated prompt prefixes, and SGLang's RadixAttention (Zheng et al., 2024) organizes them in a radix tree for cross-request reuse.

**Disaggregated Serving.** Disaggregated serving addresses the prefill–decode interference of continuous batching (Yu et al., 2022) by separating the two phases. DistServe (Zhong et al., 2024) assigns each phase to dedicated resources to improve latency stability, while Splitwise (Patel et al., 2024) formalizes phase separation as a system-level optimization for goodput. Mooncake (Qin et al., 2026) further decouples KV-cache management from execution in large-scale serving.

**Multi-Model and Agentic Inference.** Agentic systems combine reasoning, planning, and tool use (Schick et al., 2023; Shinn et al., 2023; Wang et al., 2023; Wei et al., 2022; Yao et al., 2022). Multi-agent frameworks such as LangChain (Team, 2023) and AutoGen (Wu et al., 2024) assign specialized roles to agents, while heterogeneous orchestration systems such as ToolOrchestra (Su et al., 2025) combine models and tools. DroidSpeak and KVCOMM address related inefficiencies via layer-wise and anchor-based KV reuse, respectively (Liu et al., 2024; Ye et al., 2025), whereas PrefillShare targets shared-prefill reuse across task-specific decoders.

## 6 Conclusion

We proposed PrefillShare, a disaggregated serving algorithm that eliminates redundant prefill computation and duplicated KV caches in multi-model LLM workloads by separating inference into a shared prefill module and task-specific decode modules. Enabled by cache-conditioned fine-tuning, which aligns specialized decoders with a frozen shared prefill module, our approach preserves task accuracy while substantially improving tail latency and throughput across diverse tasks, model scales, and agent workloads, even when agents share almost no prefix. Shared-prefill execution is thus a promising foundation for scalable multi-model and agentic LLM serving.

### AI USE STATEMENT

We did not use generative AI tools for any task in the required-disclosure category of the ICLR 2027 AI Policy for Authors; in particular, the formulation of PrefillShare, the design of cache-conditioned fine-tuning and the disaggregated routing mechanism, the experimental design, and the interpretation of results were carried out entirely by the authors. Generative AI tools were used for tasks in the recommended-disclosure category: assisting with code review and debugging of the author-written implementation, editing the manuscript for readability and grammar, and converting the LaTeX source between conference formats. All AI-assisted code changes were verified and tested by the authors, and all AI-assisted edits were reviewed by the authors, who take full responsibility for the final content of this work.

### REPRODUCIBILITY STATEMENT

The experimental setup, including models, datasets, hardware, and the disaggregated serving configuration, is described in Section 4.1, with full training and inference details provided in Appendix A. The method is formally specified in Section 3.

## REFERENCES

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al. Palm: Scaling language modeling with pathways. *Journal of Machine Learning Research*, 24(240):1–113, 2023.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, et al. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*, 2021.

Sourav Das, Tanmay Joshi, and Kripabandhu Ghosh. Can we optimize the performance-carbon emission break-even point?: The quest for greener llms. *arXiv preprint arXiv:2608.08744*, 2026. URL https://arxiv.org/abs/2608.08744.

Shizhe Diao, Rui Pan, Hanze Dong, Ka Shun Shum, Jipeng Zhang, Wei Xiong, and Tong Zhang. Lmflow: An extensible toolkit for finetuning and inference of large foundation models, 2024.

Jiangfei Duan, Runyu Lu, Haojie Duanmu, Xiuhong Li, Xingcheng Zhang, Dahua Lin, Ion Stoica, and Hao Zhang. MuxServe: Flexible spatial-temporal multiplexing for multiple llm serving. In *Forty-first International Conference on Machine Learning*, 2024.

Yao Fu, Leyang Xue, Yeqi Huang, Andrei-Octavian Brabete, Dmitrii Ustiugov, Yuvraj Patel, and Luo Mai. ServerlessLLM: Low-latency serverless inference for large language models. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pp. 135–153, 2024.

In Gim, Guojun Chen, Seung-seob Lee, Nikhil Sarda, Anurag Khandelwal, and Lin Zhong. Prompt cache: Modular attention reuse for low-latency inference. *Proceedings of Machine Learning and Systems*, 6:325–338, 2024.

Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, et al. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.

Zhiwei He, Tian Liang, Jiahao Xu, Qiuzhi Liu, Xingyu Chen, Yue Wang, Linfeng Song, Dian Yu, Zhenwen Liang, Wenxuan Wang, Zhuosheng Zhang, Rui Wang, Zhaopeng Tu, Haitao Mi, and Dong Yu. Deepmath-103k: A large-scale, challenging, decontaminated, and verifiable mathematical dataset for advancing reasoning. *CoRR*, abs/2504.11456, 2025. doi: 10.48550/ARXIV.2504.11456.

Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt. Measuring mathematical problem solving with the math dataset. In *Advances in Neural Information Processing Systems*, 2021.

Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Steven Yau, Zijuan Lin, Liyang Zhou, et al. Metagpt: Meta programming for a multi-agent collaborative framework. In *International Conference on Learning Representations*, volume 2024, pp. 23247–23275, 2024.

Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. Lora: Low-rank adaptation of large language models. In *The Tenth International Conference on Learning Representations, ICLR 2022, Virtual Event, April 25-29, 2022*. OpenReview.net, 2022.

Jiin Kim, Byeongjun Shin, Jinha Chung, and Minsoo Rhu. The cost of dynamic reasoning: Demystifying ai agents and test-time scaling from an ai infrastructure perspective. *arXiv preprint arXiv:2506.04301*, 2025.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th symposium on operating systems principles*, pp. 611–626, 2023.

Qintong Li, Leyang Cui, Xueliang Zhao, Lingpeng Kong, and Wei Bi. Gsm-plus: A comprehensive benchmark for evaluating the robustness of llms as mathematical problem solvers. *arXiv preprint arXiv:2402.19255*, 2024.

Zhuohan Li, Lianmin Zheng, Yinmin Zhong, Vincent Liu, Ying Sheng, Xin Jin, Yanping Huang, Zhifeng Chen, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. AlpaServe: Statistical multiplexing with model parallelism for deep learning serving. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, 2023.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *Advances in Neural Information Processing Systems*, 36:21558–21572, 2023.

Yuhan Liu, Yuyang Huang, Jiayi Yao, Shaoting Feng, Zhuohan Gu, Kuntai Du, Hanchen Li, Yihua Cheng, Junchen Jiang, Shan Lu, et al. Droidspeak: Kv cache sharing for cross-llm communication and multi-llm serving. *arXiv preprint arXiv:2411.02820*, 2024.

Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. Wizardcoder: Empowering code large language models with evol-instruct. *arXiv preprint arXiv:2306.08568*, 2023.

Rima Mittal, Ankit Gubrani, and Satyanarayana Kakollu. Lifecycle-optimal tokenization: Vocabulary size as a deployment-regime-dependent infrastructure parameter. *arXiv preprint arXiv:2608.11361*, 2026. URL `https://arxiv.org/abs/2608.11361`.

Pratyush Patel, Esha Choukse, Chaojie Zhang, Aashaka Shah, Íñigo Goiri, Saeed Maleki, and Ricardo Bianchini. Splitwise: Efficient generative llm inference using phase splitting. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*, pp. 118–132. IEEE, 2024.

Shishir G Patil, Huanzhi Mao, Fanjia Yan, Charlie Cheng-Jie Ji, Vishnu Suresh, Ion Stoica, and Joseph E Gonzalez. The berkeley function calling leaderboard (bfcl): From tool use to agentic evaluation of large language models. In *Forty-second International Conference on Machine Learning*, 2025.

Ruoyu Qin, Zheming Li, Weiran He, Jialei Cui, Heyi Tang, Feng Ren, Teng Ma, Shangming Cai, Yineng Zhang, Mingxing Zhang, et al. Mooncake: A kvcache-centric disaggregated architecture for llm serving. *ACM Transactions on Storage*, 22(4):1–38, 2026.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools. *Advances in Neural Information Processing Systems*, 36:68539–68551, 2023.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. *Advances in Neural Information Processing Systems*, 36:8634–8652, 2023.

Jyotika Singh, Fang Tu, Aziza Mirsaidova, Amit Agarwal, Hitesh Laxmichand Patel, Sandip Ghoshal, Miguel Ballesteros, Karan Dua, Yassine Benajiba, Weiyi Sun, Tao Sheng, Graham Horwood, Sujith Ravi, and Dan Roth. Gsm-sem: Benchmark and framework for generating semantically variant augmentations. 2026. URL https://arxiv.org/abs/2605.07053.

Hongjin Su, Shizhe Diao, Ximing Lu, Mingjie Liu, Jiacheng Xu, Xin Dong, Yonggan Fu, Peter Belcak, Hanrong Ye, Hongxu Yin, et al. Toolorchestra: Elevating intelligence via efficient model and tool orchestration. *arXiv preprint arXiv:2511.21689*, 2025.

Biao Sun, Ziming Huang, Hanyu Zhao, Wencong Xiao, Xinyi Zhang, Yong Li, and Wei Lin. Llumnix: Dynamic scheduling for large language model serving. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, 2024.

Gemini Team, Rohan Anil, Sebastian Borgeaud, Jean-Baptiste Alayrac, Jiahui Yu, Radu Soricut, Johan Schalkwyk, Andrew M Dai, Anja Hauth, Katie Millican, et al. Gemini: a family of highly capable multimodal models. *arXiv preprint arXiv:2312.11805*, 2023.

LangChain Team. Langchain: Building applications with large language models. https://github.com/langchain-ai/langchain, 2023. Accessed: 2025-01.

M-A-P Team, Xinrun Du, Yifan Yao, Kaijing Ma, Bingli Wang, Tianyu Zheng, Kang Zhu, Minghao Liu, Yiming Liang, Xiaolong Jin, Zhenlin Wei, Chujie Zheng, Kaixing Deng, Shuyue Guo, Shian Jia, Sichao Jiang, Yiyan Liao, Rui Li, Qinrui Li, Sirun Li, Yizhi Li, Yunwen Li, Dehua Ma, Yuansheng Ni, Haoran Que, Qiyao Wang, Zhoufutu Wen, Siwei Wu, Tianshun Xing, Ming Xu, Zhenzhu Yang, Zekun Moore Wang, Junting Zhou, Yuelin Bai, Xingyuan Bu, Chenglin Cai, Liang Chen, Yifan Chen, Chengtuo Cheng, Tianhao Cheng, Keyi Ding, Siming Huang, Yun Huang, Yaoru Li, Yizhe Li, Zhaoqun Li, Tianhao Liang, Chengdong Lin, Hongquan Lin, Yinghao Ma, Zhongyuan Peng, Zifan Peng, Qige Qi, Shi Qiu, Xingwei Qu, Yizhou Tan, Zili Wang, Chenqing Wang, Hao Wang, Yiya Wang, Yubo Wang, Jiajun Xu, Kexin Yang, Ruibin Yuan, Yuanhao Yue, Tianyang Zhan, Chun Zhang, Jingyang Zhang, Xiyue Zhang, Xingjian Zhang, Yue Zhang, Yongchi Zhao, Xiangyu Zheng, Chenghua Zhong, Yang Gao, Zhoujun Li, Dayiheng Liu, Qian Liu, Tianyu Liu, Shiwen Ni, Junran Peng, Yujia Qin, Wenbo Su, Guoyin Wang, Shi Wang, Jian Yang, Min Yang, Meng Cao, Xiang Yue, Zhaoxiang Zhang, Wangchunshu Zhou, Jiaheng Liu, Qunshu Lin, Wenhao Huang, and Ge Zhang. Supergpqa: Scaling llm evaluation across 285 graduate disciplines, 2025. URL https://arxiv.org/abs/2502.14739.

Lei Wang, Wanyu Xu, Yihuai Lan, Zhiqiang Hu, Yunshi Lan, Roy Ka-Wei Lee, and Ee-Peng Lim. Plan-and-solve prompting: Improving zero-shot chain-of-thought reasoning by large language models. *arXiv preprint arXiv:2305.04091*, 2023.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 35, pp. 24824–24837, 2022.

Sunghyeon Woo, Jaeun Kil, Hoseung Kim, Minsub Kim, Joonghoon Kim, Ahreum Seo, Sungjae Lee, Minjung Jo, Jiwon Ryu, Baeseong Park, Se Jung Kwon, and Dongsoo Lee. Icarus: Identical cache reuse for efficient multi-model inference. In *International Conference on Learning Representations (ICLR)*, 2026.

Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, et al. Autogen: Enabling next-gen llm applications via multi-agent conversations. In *First Conference on Language Modeling*, 2024.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629*, 2022.

Hancheng Ye, Zhengqi Gao, Mingyuan Ma, Qinsi Wang, Yuzhe Fu, Ming-Yu Chung, Yueqian Lin, Zhijian Liu, Jianyi Zhang, Danyang Zhuo, et al. Kvcomm: Online cross-context kv-cache communication for efficient llm-based multi-agent systems. *arXiv preprint arXiv:2510.12872*, 2025.

Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. Orca: A distributed serving system for {Transformer-Based} generative models. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pp. 521–538, 2022.

Longhui Yu, Weisen Jiang, Han Shi, Jincheng Yu, Zhengying Liu, Yu Zhang, James T Kwok, Zhenguo Li, Adrian Weller, and Weiyang Liu. Metamath: Bootstrap your own mathematical questions for large language models. *arXiv preprint arXiv:2309.12284*, 2023.

Zhaojian Yu, Yilun Zhao, Arman Cohan, and Xiao-Ping Zhang. HumanEval pro and MBPP pro: Evaluating large language models on self-invoking code generation task. In Wanxiang Che, Joyce Nabende, Ekaterina Shutova, and Mohammad Taher Pilehvar (eds.), *Findings of the Association for Computational Linguistics: ACL 2025*, pp. 13253–13279, Vienna, Austria, July 2025. Association for Computational Linguistics. ISBN 979-8-89176-256-5. doi: 10.18653/v1/2025.findings-acl.686. URL https://aclanthology.org/2025.findings-acl.686/.

Jianguo Zhang, Tian Lan, Ming Zhu, Zuxin Liu, Thai Quoc Hoang, Shirley Kokane, Weiran Yao, Juntao Tan, Akshara Prabhakar, Haolin Chen, et al. xlam: A family of large action models to empower ai agent systems. In *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pp. 11583–11597, 2025.

Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Livia Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E Gonzalez, et al. Sglang: Efficient execution of structured language model programs. *Advances in neural information processing systems*, 37: 62557–62583, 2024.

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. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pp. 193–210, 2024.

Kan Zhu, Mathew Jacob, Chenxi Ma, Yi Pan, Stephanie Wang, Arvind Krishnamurthy, and Baris Kasikci. Tracelab: Characterizing coding agent workloads for llm serving. *arXiv preprint arXiv:2606.30560*, 2026.

# APPENDICES

## A EXPERIMENTAL DETAILS

### A.1 TRAINING SETUP

All training experiments reported in Table 1 were carried out on a single node with 8 A100-SXM4-80G GPUs. For fine-tuning, we used the LMFlow (Diao et al., 2024) framework with three datasets: MetaMathQA-40K (Yu et al., 2023), EvolInstruct-Code-80K (Luo et al., 2023), and xLAM-function-calling-60K (Zhang et al., 2025).

Fine-tuning tasks across all datasets were consistently performed with a batch size of 1, gradient accumulation over 16 steps and a maximum sequence length of 1024 for one epoch. Learning rates were selected via a grid search over $\{1 \times 10^{-4}, 1 \times 10^{-5}, 2 \times 10^{-5}, 5 \times 10^{-5}, 2 \times 10^{-6}, 5 \times 10^{-6}\}$ for each dataset. AdamW with $\beta_1 = 0.9$ and $\beta_2 = 0.999$ was used as the optimizer for fine-tuning, together with a weight decay of 0.1 and a warmup ratio of 0.03. There were no explicit regularization setups, including gradient clipping or dropout.

### A.2 INFERENCE SETUP

**Workloads and measurement.** All inference experiments follow the serving configuration described in Section 4.1. For a fair comparison, both systems are evaluated under the same workload and generation settings. We replay the same session-level traces for the disaggregated baseline and PrefillShare, so each system receives the same session arrivals and agent invocation sequence.

For the ReAct (Yao et al., 2022) and Reflexion (Shinn et al., 2023) agents, each session contains four agents invoked in a round-robin order for four rounds, resulting in 16 model invocations per session. The input and output token lengths of each invocation are fixed according to the token-length statistics reported in prior agent-serving work (Kim et al., 2025; Woo et al., 2026). Each session proceeds synchronously: after receiving a model response, it immediately issues the next model invocation until all turns are completed, without injecting additional delays between model calls. Thus, the session arrival rate specifies how frequently new sessions enter the system, rather than the rate of individual model invocations.

Across agent transitions, both systems append generated responses to the session context using the same prompt format. Latency and throughput are measured at the client-facing proxy to capture the end-to-end serving behavior observed by requests.

**Multi-node configuration.** The four H100 nodes are connected by InfiniBand, and each hosts one TP=8 instance so that decode work is spread over all eight GPUs of the node. The baseline dedicates one prefill and one decode node to each of the two specialized models, whereas PrefillShare uses two shared prefill nodes and two task-specific decode nodes. Batching, chunked prefill, prefix caching, decode capacity, and session arrivals are identical for both systems, and the ReAct workflow follows the single-node setup above. Input contexts are longer than in a single node workload, so absolute throughput is not comparable across the two settings.

**Low-overlap orchestrator workload.** We construct a supervisor-style workload in which a main agent delegates role-specific tasks to three subagents in the sequence `main` $\rightarrow$ `A` $\rightarrow$ `main` $\rightarrow$ `B` $\rightarrow$ `main` $\rightarrow$ `C` $\rightarrow$ `main`. Each role maintains an independent history, and the main agent provides each subagent only with its role-specific instruction, substantially reducing the prefix shared across model switches. Per-step input and output lengths follow TraceLab (Zhu et al., 2026), a trace of approximately 4,300 real Claude Code and Codex sessions.

The resulting prompts share only 30 tokens across roles out of 8,145 tokens (0.4%). With the 16-token KV-cache block granularity used by our serving system, only one block remains reusable, corresponding to 0.20% of the prompt. Both systems serve Qwen3-14B-Base on one 8$\times$H100 node with four prefill and four decode workers. We use identical batching, chunked prefill, prefix caching, decode capacity, and session arrivals for the baseline and PrefillShare.

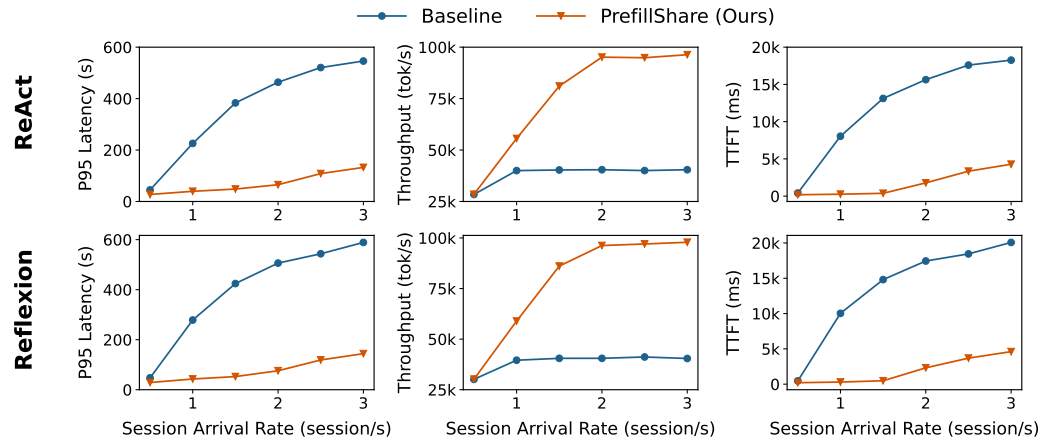



Figure 6: **Serving performance with Qwen3-14B.** We replicate the setup of Fig. 4, replacing the LLaMA-3.1-8B backbone with Qwen3-14B while keeping all other settings identical. Results are shown for ReAct (top) and Reflexion (bottom), reporting p95 end-to-end latency, throughput, and time-to-first-token (TTFT).

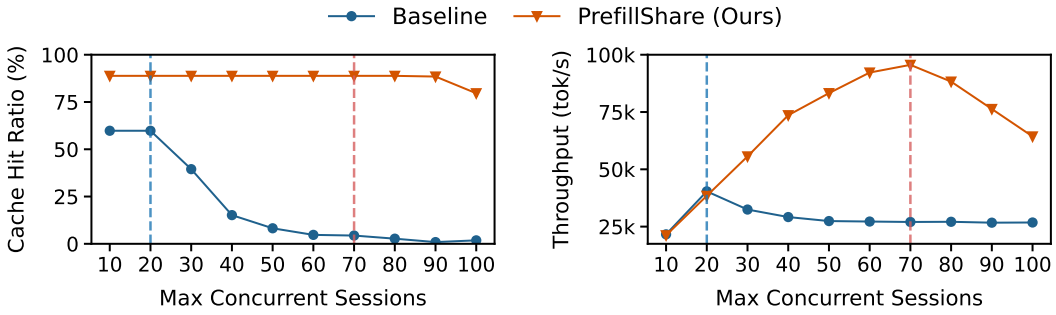



Figure 7: **Concurrency ablation with Qwen3-14B.** This figure mirrors the setup of Fig. 8, with the backbone model replaced by Qwen3-14B. Left: prefix cache hit ratio (%). Right: throughput (tok/s).

**KV transfer at high concurrency.** Our prototype is implemented on top of vLLM's disaggregated serving pipeline. In the common case, KV caches produced by prefill workers are transferred to decode workers and kept resident in GPU memory, where they are directly consumed during autoregressive decoding.

At high concurrency, when many sessions are concurrently active, the aggregate GPU-resident KV footprint on the decode side can temporarily exceed the available memory capacity. In such cases, vLLM stages a subset of KV caches in CPU memory and reloads them to the GPU when the corresponding sessions are scheduled for decoding. This swap-like staging and reload behavior increases CPU–GPU data movement, which can increase latency and reduce throughput at high concurrency.

We emphasize that this behavior is an implementation-level mechanism in vLLM for handling transient KV memory pressure, rather than an inherent limitation of shared-prefill execution. In principle, alternative designs could mitigate overflow-induced staging via stricter admission control, decode-to-prefill backpressure, or per-session reservation of GPU-resident KV buffers. Exploring such alternatives is orthogonal to PrefillShare and left for future system optimizations.

# B Robustness Across Backbones and Agent Invocation Patterns

## B.1 Qwen3-14B Backbone

To examine whether the serving behavior observed in Section 4.3 depends on a specific backbone model, we repeat the same experiments using Qwen3-14B in place of LLaMA-3.1-8B. All workload configurations, scheduling policies, and evaluation protocols remain identical to those used in the main experiments. As shown in Fig. 6 and Fig. 7, PrefillShare continues to achieve substantially higher throughput and lower tail latency than the baseline, while maintaining a high prefix cache hit ratio over a wide concurrency range.

## B.2 Randomized Swarm Pattern

Table 5: **Serving performance under a randomized four-agent swarm pattern.** TTFT and throughput comparison between the disaggregated baseline and PrefillShare on LLaMA-3.1-8B with four sequential agents on a single A100 node. Each agent randomly hands off the request to another agent after completing its task segment. PrefillShare consistently reduces time-to-first-token (TTFT) and improves throughput under higher load.

| Session Arrival Rate (session/s) | TTFT (ms) | | Throughput (tok/s) | |
|---|---|---|---|---|
| | Baseline | PrefillShare (Ours) | Baseline | PrefillShare (Ours) |
| 0.5 | 152 | **76 (2.0×)** | 7,314 | **7,319 (1.0×)** |
| 1.5 | 280 | **81 (3.5×)** | 21,094 | **21,280 (1.0×)** |
| 2.5 | 733 | **94 (7.8×)** | 28,206 | **34,272 (1.2×)** |
| 3.5 | 741 | **141 (5.3×)** | 28,325 | **45,919 (1.6×)** |

We further evaluate PrefillShare under a randomized four-agent swarm pattern on LLaMA-3.1-8B with a single A100 node, using the same setup as Appendix A.2 except for the agent invocation pattern. In other words, unlike the fixed round-robin patterns in Section 4.3, the swarm workload lets each agent randomly hand off the request to another agent after completing its task segment, while preserving a shared and growing session context across agents.

Table 5 reports TTFT and throughput, showing that PrefillShare consistently reduces TTFT across all session arrival rates, achieving up to 7.8× lower TTFT. These results show that the benefit of PrefillShare is not specific to a fixed agent invocation pattern. Regardless of whether model transitions follow a fixed sequence or a randomized swarm, the baseline must repeatedly recompute prefill and maintain duplicated KV caches across specialized models. PrefillShare removes this structural redundancy by sharing the base prefix cache across model handoffs, making its efficiency gains largely independent of the particular agent invocation order.

# C Analysis of Serving Mechanisms and Resource Utilization

## C.1 Attribution of Prefill Pooling and Cross-Model KV Reuse

PrefillShare combines two mechanisms that can independently improve serving efficiency: sharing prefill capacity across models and reusing KV caches across model switches. To separate their contributions, we evaluate an intermediate configuration that enables the shared prefill pool, prefix caching, and prefix-aware routing while disabling cross-model KV reuse. In this configuration, any model can use any prefill worker, but each model still recomputes its own prefix.

Pooling alone substantially improves serving efficiency: compared with the per-model baseline, the shared-pool configuration improves throughput by 84% and reduces p95 latency by 59%. Exact cross-model KV reuse provides an additional gain beyond pooling. Moving from the shared-pool configuration to PrefillShare further improves throughput by 20% and reduces p95 latency by 34%, while increasing the APC hit rate from 58.8% to 94.4% and reducing local prefill from 4.92M to 0.68M tokens.

Table 6: **Attribution of prefill pooling and cross-model KV reuse.** All configurations use the same Qwen3-14B 4P+4D deployment with 64 sessions at 0.5 sessions/s.

| Configuration | Mean TTFT (s) | p95 (s) | Throughput (tok/s) | Local prefill (M tok) | APC hit (%) |
|---|---|---|---|---|---|
| Baseline (per-model prefill) | 8.28 | 322.60 | 28.68k | 9.56 | 20.0 |
| Shared pool, no cross-model reuse | 3.05 | 132.09 | 52.71k | 4.92 | 58.8 |
| PrefillShare | 1.63 | 87.02 | 63.13k | 0.68 | 94.4 |

Cross-model cache reuse cannot be obtained through scheduling alone because KV states produced by independently fine-tuned models are not interchangeable. Cache-conditioned fine-tuning instead trains each task-specific decoder to consume the shared base-model KV cache, enabling exact reuse without recomputing the shared prefix. Fig. 2 shows the corresponding effect on accuracy: naive KV sharing fails, whereas PrefillShare preserves Full-FT accuracy. These results show that prefill pooling and exact cross-model KV reuse make complementary contributions to PrefillShare.

### C.2 Predictive Prefill Overlap

We evaluate an optimistic predictive-overlap baseline that assumes the next role is known before the current one finishes decoding. While agent $i$ decodes, the baseline prefills the available prefix for the next agent on the worker that will serve it, capped at two warmups per rank. Agent $i$'s own output and the following tool result cannot be prefetched because neither exists yet. All three configurations use the same Qwen3-14B 4P+4D deployment, 64 sessions at 0.5 sessions/s.

Table 7: **Predictive prefill overlap.** Comparison with a predictive-overlap baseline that assumes perfect next-role prediction.

| Configuration | Mean TTFT (s) | p95 (s) | Throughput (tok/s) | Local prefill (M tok) | APC hit (%) |
|---|---|---|---|---|---|
| Baseline (per-model prefill) | 8.28 | 322.60 | 28.68k | 9.56 | 20.0 |
| Predictive prefill overlap | 7.81 | 331.88 | 28.09k | 9.72 | 40.7 |
| PrefillShare | **1.63** | **87.02** | **63.13k** | **0.68** | **94.4** |

Predictive prefill overlap reduces mean TTFT by 6% and increases the prefix-cache hit rate from 20.0% to 40.7%, but does not improve throughput or tail latency at this load. The overlap moves prefill work earlier without removing it, and local prefill increases from 9.56M to 9.72M tokens. The overlap is useful when prefill capacity is not the bottleneck: at 0.05 sessions/s, it reduces TTFT by 31%. This advantage disappears as the prefill workers saturate because the prefetched work still consumes the same serving capacity. PrefillShare removes the redundant work instead, reducing local prefill to 0.68M tokens and continuing to improve throughput under load.

### C.3 Prefill-Worker Load under Low Prefix Overlap

We analyze why pooling provides a large serving benefit in the low-overlap workload. Each workflow invokes the main agent four times but each specialist only once. Under the baseline, each role is bound to one prefill worker, so the asymmetric invocation frequency concentrates requests on the main-agent worker while leaving capacity on the specialist workers underutilized. PrefillShare removes this role affinity and allows requests from any role to use any of the four prefill workers.

We measure the resulting worker-level prefill load at 0.02625 sessions/s, corresponding to the 0.026 sessions/s setting in Table 4. Prefill work is measured as prefix-cache query tokens minus prefix-cache hit tokens, i.e., the number of tokens actually computed by each prefill worker.

The baseline exhibits severe load imbalance: the busiest worker processes 42.2× as many prefill tokens as the least-loaded worker. PrefillShare reduces this max–min ratio to 2.07× by distributing role requests across the shared worker pool. The concentrated baseline load also causes repeated prefill computation. The main-agent worker performs 7.55× the prefill work required if each

Table 8: **Prefill-worker load under low prefix overlap.** Prefill tokens computed by each worker at 0.02625 sessions/s. Baseline workers are bound to individual roles, whereas PrefillShare workers are shared across roles.

| Worker | Baseline assignment | Baseline prefill tokens | PrefillShare prefill tokens |
|---|---|---|---|
| Worker 1 | Main agent | 4,668,967 | 391,369 |
| Worker 2 | Specialist A | 254,739 | 550,692 |
| Worker 3 | Specialist B | 168,097 | 265,681 |
| Worker 4 | Specialist C | 110,625 | 390,771 |
| Total | | 5,202,428 | 1,598,513 |

session-local prefix were computed once without eviction, whereas PrefillShare reduces the aggregate recomputation amplification to 1.06×.

This behavior explains why pooling remains effective even when little prefix can be reused across models. In the baseline, requests queue behind the overloaded role-affine worker while prefill capacity on other workers remains available. PrefillShare makes this otherwise idle capacity available to the queued requests, reducing both queueing and repeated prefill computation. Static GPU reallocation does not provide the same flexibility when all roles must remain resident: removing a role from a GPU incurs model-loading latency when the role is next invoked (Fu et al., 2024), while the most heavily loaded role can change across tasks and over time (Sun et al., 2024; Duan et al., 2024; Li et al., 2023).

### C.4 Effect of the Concurrency Limit

To isolate the effect of cache pressure, we introduce and ablate the max concurrent session limit, an admission-control knob that caps the number of sessions kept active in the system at a time; sessions beyond the limit remain queued until capacity becomes available. This limit controls the active KV-cache footprint and prevents unbounded session admission from causing excessive cache pressure. As concurrency increases, the disaggregated baseline must retain separate prefix KV caches for each specialized model, increasing memory pressure, cache evictions, and prefill recomputation.

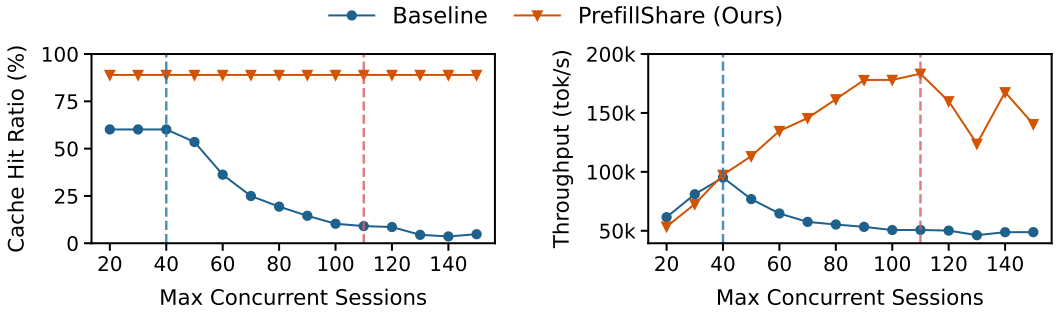



Figure 8: **Effect of the concurrency limit.** Prefix cache hit ratio (left) and throughput (right) under varying maximum concurrent sessions. The baseline degrades beyond ≈40 sessions as reduced prefix reuse lowers both metrics, whereas PrefillShare sustains higher throughput over a wider range, with high-concurrency saturation driven by handoff overheads.

For Fig. 4, we sweep this concurrency limit and report the best-performing configuration for each setting. In Fig. 8, we fix the ReAct arrival rate to 4 sessions/s and vary the max concurrent sessions. The baseline begins to degrade beyond roughly 40 concurrent sessions: its prefix-cache hit ratio peaks at about 60% and then drops sharply, which reduces throughput. In contrast, PrefillShare maintains a hit ratio near 89% across concurrency levels by storing the shared prefix cache only once, allowing throughput to scale over a wider range.

At extreme concurrency, around 110 concurrent sessions, PrefillShare throughput eventually saturates and declines even though the hit ratio remains stable. This indicates that the bottleneck shifts from prefix-cache reuse to decode-side KV pressure and cache-handoff overhead in our vLLM prototype as detailed in Appendix A. The same qualitative trend holds for Qwen3-14B under identical settings, with additional results in Fig. 7.

### C.5 Effect of Input Length

We further study how input length affects the serving benefit of PrefillShare. We evaluate LLaMA-3.1-8B with four sequential agents on a single A100 node under two input lengths: a short context of 64 tokens and a long context of 4096 tokens. As shown in Table 9, PrefillShare consistently reduces TTFT across both input lengths. For short inputs, PrefillShare reduces TTFT from 0.52s to 0.27s, corresponding to a 1.9× reduction, while throughput remains unchanged at 1.0× because the prefill cost is relatively small. For long inputs, the benefit becomes much more pronounced: PrefillShare reduces TTFT from 20.18s to 6.11s, achieving a 3.3× reduction, and improves throughput from 40,218 tok/s to 88,323 tok/s, a 2.2× increase. These results confirm that PrefillShare is particularly effective for long-context workloads, where shared-prefix reuse eliminates substantial redundant prefill computation and KV cache memory. Together with the low-overlap analysis, this ablation separates the two regimes: the benefit from exact cache reuse grows with the reusable prefix, while prefill pooling remains useful even when the reusable prefix is small.

Table 9: **Effect of input length on serving performance.** Ablation study on time-to-first-token (TTFT) and throughput for LLaMA-3.1-8B with four sequential agents on a single A100 node. PrefillShare provides larger benefits for longer inputs, where shared-prefix reuse avoids more redundant prefill computation and KV-cache storage.

| Input Length | TTFT (s) | | Throughput (tok/s) | |
|---|---|---|---|---|
| | Baseline | PrefillShare (Ours) | Baseline | PrefillShare (Ours) |
| 64 | 0.52 | **0.27 (1.9×)** | 2,515 | **2,515 (1.0×)** |
| 4096 | 20.18 | **6.11 (3.3×)** | 40,218 | **88,323 (2.2×)** |

## D Training Cost, Memory Accounting, and Other Benchmarks

### D.1 Cache-Conditioned Fine-Tuning Cost and Amortization

A method that shifts cost from serving to training should account for both sides. We therefore measure the one-time cost of cache-conditioned fine-tuning and compare it with the recurring serving-side savings. One epoch requires 2.0, 3.0, and 7.1 H100 GPU-hours for math, tool calling, and coding, respectively, for a total of 12.1 GPU-hours across the three task-specific decoders.

Table 10: **Training cost and recurring serving-side savings.** The training cost is measured once for the three task-specific decoders. Serving-side measurements use the Qwen3-14B 4P+4D workload from Appendix C.1.

| Quantity | When paid | Measured |
|---|---|---|
| Cache-conditioned fine-tuning | Once | 2.0 / 3.0 / 7.1 H100 GPU-h |
| Redundant prefill compute | Every workflow | 9.56 → 0.68 M tokens |
| End-to-end throughput | Every workflow | 28.68k → 63.13k tok/s |

The training cost is paid once, whereas the serving savings recur on every workflow. In this workload, PrefillShare reduces local prefill from 9.56M to 0.68M tokens per 64-session run and increases throughput from 28.68k to 63.13k tok/s. Under the same GPU budget, the baseline therefore requires roughly twice the GPU time to deliver the same amount of work, allowing the measured 12.1 GPU-hours of cache-conditioned fine-tuning to be recovered within a few hours of sustained serving. This one-time-versus-recurring trade-off is consistent with recent analyses of LLM deployment, which

model total deployment cost as a one-time training term plus an inference-volume-weighted recurring term, and with recent work noting that repeated inference can substantially exceed a one-time fine-tuning cost (Mittal et al., 2026; Das et al., 2026). Thus, the offline fine-tuning cost is amortized as the system serves repeated multi-model workflows rather than recurring on every request.

### D.2 Practical Memory Accounting

The main paper reports the asymptotic reduction in KV memory. We additionally account for the absolute KV footprint and resident model weights. For four LLaMA-3.1-8B specialists with a 4K shared prefix, LLaMA-3.1-8B stores 128 KiB of KV state per token. The baseline therefore retains four model-specific copies of the shared prefix, whereas PrefillShare retains a single reusable copy.

The KV-memory reduction is not offset by additional model-weight residency. Under the same 4P+4D GPU budget, the baseline keeps specialized models resident on both prefill and decode workers, whereas PrefillShare keeps the shared base model on prefill workers and task-specific models on decode workers. The resident parameter footprint is therefore comparable between the two systems.

The distinction becomes more important as context length and concurrency increase: model weights remain fixed per worker, whereas aggregate KV memory grows with the number of active sessions and their context lengths. Table 9 reflects this trend: PrefillShare provides no throughput improvement at 64 input tokens, where prefill is inexpensive, but improves throughput by 2.2× at 4,096 input tokens. Fig. 8 similarly shows that avoiding duplicated prefix KV allows PrefillShare to sustain cache reuse over a wider concurrency range.

### D.3 Other Benchmarks and Out-of-Domain Accuracy

We additionally evaluate PrefillShare on other math and coding benchmarks and on an out-of-domain task. For math, we use GSM-SEM (Singh et al., 2026); for coding, we use CodeEval-Pro (Yu et al., 2025), a harder reconstruction of HumanEval and MBPP. For the out-of-domain evaluation, the Full-FT and PrefillShare models are fine-tuned only on coding data and then evaluated on the non-CS disciplines of SuperGPQA (Team et al., 2025), with no overlap between the fine-tuning and evaluation domains.

PrefillShare stays within 0.26 and 0.55 points of Full-FT on GSM-SEM and CodeEval-Pro, respectively, extending the accuracy parity observed in the main evaluation to additional math and coding benchmarks. For the out-of-domain evaluation, Base, Full-FT, and PrefillShare remain within 0.41 points on SuperGPQA non-CS. Cache-conditioned fine-tuning therefore preserves the in-domain

Table 11: **Practical memory accounting for a 4K shared prefix.** We compare four LLaMA-3.1-8B specialists under the same GPU budget. The baseline stores one shared-prefix KV copy per specialized model, whereas PrefillShare stores one shared copy across models. Resident model-weight memory remains comparable between the two systems.

| Memory Component | Baseline | PrefillShare |
|---|---|---|
| Shared-prefix KV copies | 4 | 1 |
| Prompt KV per session | 2.0 GiB | 0.5 GiB |
| Resident model weights | ∼16 GiB / worker | ∼16 GiB / worker |

Table 12: **Accuracy on other benchmarks and an out-of-domain task.** GSM-SEM evaluates mathematical reasoning, CodeEval-Pro evaluates coding on harder variants of HumanEval and MBPP, and SuperGPQA non-CS evaluates out-of-domain performance after coding-only fine-tuning.

| Method | GSM-SEM | CodeEval-Pro | SuperGPQA non-CS |
|---|---|---|---|
| Base | 61.76 | 52.21 | 24.54 |
| Full-FT | 76.90 | 59.78 | 24.72 |
| PrefillShare (Ours) | 76.64 | 59.23 | 24.95 |

Table 13: **Matched-adaptation comparison with ICaRus.** ICaRus and PrefillShare use the same LoRA configuration to control for adaptation capacity. TPOT is measured with input and output lengths of 1,024 and 128 tokens, respectively. Math averages GSM8K and GSM+, coding averages HumanEval and HumanEval+, and tool calling averages the Simple Python and Multiple categories of BFCL.

| | TPOT (ms) | | | Accuracy | | |
|---|---|---|---|---|---|---|
| **Method** | **B=1** | **B=4** | **B=16** | **Math** | **Coding** | **Tool** |
| ICaRus (LoRA) | 16.5 | 19.9 | 22.9 | 59.1 | 46.1 | 88.3 |
| PrefillShare (matched LoRA) | 15.8 | 19.6 | 20.7 | 60.0 | 45.7 | 89.0 |
| PrefillShare (Full-FT) | 12.1 | 15.5 | 14.8 | 60.4 | 47.0 | 89.6 |

gains of conventional fine-tuning without introducing additional out-of-domain degradation in this evaluation.

# E Comparison with Prior KV-Cache Sharing Methods

## E.1 Matched Adaptation Comparison with ICaRus

To isolate the architectural contribution from adaptation capacity, we compare ICaRus and PrefillShare under the same LoRA configuration: rank 128, $\alpha = 256$, with LoRA applied to the q, o, gate, up, and down projections. Both methods use the same logical batch size on a single A100 with LLaMA-3.1-8B, TP=1, and input/output lengths of 1024/128 tokens.

At equal adaptation budget, PrefillShare has lower TPOT at every evaluated batch size, including about 10% at B=16, while maintaining comparable task accuracy (+0.9 Math, +0.7 Tool, $-0.4$ Coding). This matched comparison removes adaptation capacity as the explanation for the serving gap. The remaining difference is architectural. ICaRus keeps the frozen base model on the decode path, so every step runs both the base model for compatible KV and the base-plus-adapter path for generation. PrefillShare separates the phases completely: the base model runs only during prefill and the task-specific model runs during decode, so generation carries no additional base-model computation.

The separation also removes the adapter-based specialization constraint. ICaRus requires the base-model path to remain compatible with the task-specific path during generation, whereas PrefillShare can use full fine-tuning for the task-specific decode model. The Full-FT row therefore represents a configuration that PrefillShare can support without retaining an additional base-model path during decode.

## E.2 Comparison with KVCOMM and DroidSpeak

We further compare PrefillShare with other KV-cache sharing algorithms, including KVCOMM (Ye et al., 2025) and DroidSpeak (Liu et al., 2024). A direct end-to-end quantitative comparison is difficult because the methods differ in target setting and implementation availability. KVCOMM focuses on anchor-based KV reuse across different system prompts, while PrefillShare targets shared-prefix reuse across differently fine-tuned task-specific decoders in multi-model agent workloads. Applying KVCOMM to this setting would require adapting its anchor-based reuse mechanism, making the result sensitive to nontrivial design choices. DroidSpeak is more closely related in target setting, but its official implementation is not publicly available, preventing a faithful reproduction. We therefore provide a normalized qualitative and asymptotic comparison in Table 14.

Table 14 compares the methods under a common multi-agent abstraction with $N$ agents and sequence length $L$. KVCOMM and DroidSpeak have the practical advantage of being training-free. However, their reuse is conditional or partial: KVCOMM depends on anchor retrieval, and DroidSpeak recomputes sensitive layers. These inference-time costs are paid repeatedly during serving. PrefillShare makes a different trade-off by introducing cache-conditioned fine-tuning, an offline cost that is amortized after deployment. This training aligns the decoder with the shared base KV cache, enabling

Table 14: **Comparison of KV reuse algorithms in a multi-agent system.** $N$ and $L$ denote the number of agents and sequence length, respectively; $V$ and $r$ denote the number of anchors and anchor retrieval hit rate in KVCOMM; and $\alpha$ denotes the fraction of sensitive layers in DroidSpeak.

| **Aspect** | **KVCOMM** | **DroidSpeak** | **PrefillShare (Ours)** |
|---|---|---|---|
| Method | Anchor-based KV reuse | Selective recomputation | KV-sharing-aware fine-tuning |
| Target | Different system prompts | Different fine-tuned models | Different fine-tuned models |
| Training Overhead | Training-free | Training-free | Fine-tuning |
| Inference Overhead | Anchor lookup | Calibration | No overhead |
| KV sharing | Not full KV sharing (anchor-guided KV reuse) | Not full KV sharing (layer-wise KV reuse) | Full KV sharing |
| Accuracy | Limited by train-test discrepancy | Limited by train-test discrepancy | No train-test discrepancy; higher accuracy |
| KV memory | $O((1+V)L)$ | $O((1+\alpha(N-1))L)$ | $O(L)$ |
| # Prefill Compute | $O((1+(1-r)(N-1))L^2)$ | $O((1+\alpha(N-1))L^2)$ | $O(L^2)$ |

full shared-prefix KV reuse without additional anchor lookup or layer-wise recomputation during inference.

Under this abstraction, KVCOMM requires memory proportional to the anchor pool, yielding $O((1+V)L)$ KV memory, and its prefill compute depends on the anchor hit rate $r$, giving $O((1+(1-r)(N-1))L^2)$. DroidSpeak reduces redundant prefill by reusing non-sensitive layers, but still recomputes an $\alpha$ fraction of sensitive layers for each additional agent, resulting in $O((1+\alpha(N-1))L)$ KV memory and $O((1+\alpha(N-1))L^2)$ prefill compute. PrefillShare stores the shared-prefix KV cache once and performs the shared prefill once, leading to $O(L)$ KV memory and $O(L^2)$ prefill compute for the shared context. Thus, while training-free methods are attractive for deployment simplicity, PrefillShare shifts the cost to offline fine-tuning and reduces repeated inference-time overhead in multi-agent shared-context serving.