FlexLLM: Token-Level Co-Serving of LLM Inference and Finetuning with SLO Guarantees

Gabriele Oliaro* Xupeng Miao*† Xinhao Cheng Vineeth Kada† Mengdi Wu Ruohan Gao Yingyi Huang Remi Delacourt† April Yang Yingcheng Wang† Colin Unger Zhihao Jia

Carnegie Mellon University *Purdue University* *Anthropic PBC*
Stanford University *Mistral AI* *Amazon Web Services*

Abstract

Finetuning large language models (LLMs) is essential for task adaptation, yet today's serving stacks isolate inference and finetuning on separate GPU clusters—wasting resources and under-utilizing hardware. We introduce FlexLLM, the first system to *co-serve* LLM inference and PEFT-based finetuning on shared GPUs by fusing computation at the token level. FlexLLM 's static compilation optimizations—*dependent parallelization* and *graph pruning* significantly shrink activation memory, leading to end-to-end GPU memory savings by up to 80%. At runtime, a novel *token-level finetuning* mechanism paired with a hybrid token scheduler dynamically interleaves inference and training tokens within each co-serving iteration, meeting strict latency SLOs while maximizing utilization. In end-to-end benchmarks on LLaMA-3.1-8B, Qwen-2.5-14B, and Qwen-2.5-32B, FlexLLM maintains inference SLO compliance at up to 20 req/s, and improves finetuning throughput by $1.9-4.8\times$ under heavy inference workloads and $2.5-6.8\times$ under light loads, preserving over 76% of peak finetuning progress even at peak demand. FlexLLM is publicly available at https://flexllm.github.io.

1 Introduction

Recent advancements in large language models (LLMs) such as GPT-5 [67, 70], DeepSeek-R1 [32], LLaMA-4 [29, 58, 84] and QWen 3 [94, 95], have shown strong capabilities of generating natural language texts across various application domains [25, 52, 79, 102]. Given the increasingly high cost of LLM pre-training, there is a trend of standardizing foundational pre-trained LLMs and sharing sub-models across multiple downstream tasks [35, 100] through finetuning (i.e., continuous training on small task-specific datasets). For example, a series of *parameter-efficient finetuning* (PEFT) techniques [33, 35, 36, 45, 47, 71] have been proposed to update a subset of trainable parameters from an LLM or introduce a small number of new parameters into the LLM while keeping the vast majority of the original LLM parameters frozen. The base LLM sharing paradigm provides opportunities to multiplex resources and improve cluster utilization (e.g., Google [11], Microsoft [36], and Tencent [62]). Many recent efforts [14, 37, 76, 91, 105, 112] inspired by these opportunities to build multi-task serving systems, handling inference requests for multiple fine-tuned LLMs simultaneously in a single GPU cluster.

In addition to inference, almost all LLM serving companies (including OpenAI GPT-4o [68], Google Gemini [28], and Databricks Mosaic AI [20]), now offer commercial finetuning APIs to facilitate the creation of custom models. This trend is not only due to the widespread adoption of LLMs but also driven by the growing demand for task-specific model customization [55], data access control [81], and the need for continuous and frequent model updating [26].

Currently, the most popular way to offer both inference and finetuning services is to run each task on separate, *dedicated* clusters [15, 27, 89], which exclusively use hardware resources through specialized systems optimized primarily for latency and throughput, respectively. While straightforward, this practice leads to significant economic inefficiencies. In particular, resources must be overprovisioned for inference tasks, as real-world LLM inference workloads exhibit highly unpredictable, bursty request arrival patterns [88]. For example, public reports from Microsoft and Alibaba cite average GPU utilization rates of just 52% [18] and 10% [18], respectively, with production services showing similar underutilization patterns. While inference tasks must be provisioned for peak burst capacity to maintain SLOs, the resulting idle resources during normal operation cannot be utilized by finetuning tasks, which could tolerate the latency variations (minutes to hours vs. milliseconds). This is economically unsustainable given the high cost and power demands of modern GPUs, which can now exceed 1,000W per chip [18].

In this paper, we explore a key research question: *Can we design a system that* simultaneously *serves both inference and finetuning tasks within a shared GPU cluster, dynamically adapting to unpredictable burst patterns while maintaining strict inference SLOs?* Previous attempts (Section 3) at mul-

*Equal contribution.
†Work done at CMU.

tiplexing inference and finetuning tasks, while successful in increasing the GPU utilization, have seen limited adoption due to their impracticality.

To solve this fundamental challenge, we introduce *co-serving*, a novel multiplexing technique that effectively handles bursty workloads while satisfying strict SLOs. We built FlexLLM, the first system for LLM inference and finetuning that implements this technique. FlexLLM introduces a *PEFT-as-a-service (PaaS)* interface, which unifies inference and finetuning tasks, enabling their joint execution on shared GPU resources. The key insight behind co-serving is that inference and finetuning tasks, when using the same base LLMs, can be merged at fine granularity—at the level of individual tokens rather than entire requests or kernels. This token-level scheduling enables millisecond-scale resource reallocation in response to inference bursts.

This co-serving approach offers critical advantages for handling bursty workloads. When inference requests suddenly spike, FlexLLM can instantly throttle finetuning tokens within the same GPU kernel execution, reallocating resources to maintain inference SLOs without the overhead of context switching or reconfiguration. During normal load, finetuning tasks opportunistically consume the idle capacity reserved for bursts, improving overall utilization. Since the base LLM parameters are shared and frozen during finetuning, memory overhead is minimized, enabling efficient multiplexing.

Overall, this paper makes the following key contributions:

- We introduce FlexLLM, the first system to co-serve LLM inference and PEFT finetuning on shared GPUs through token-level computation fusion.
- We develop static optimizations that reduce GPU memory requirements by up to 80% through dependent parallelization and graph pruning.
- We propose token-level finetuning with hybrid scheduling that maintains strict latency SLOs while maximizing GPU utilization.
- We achieve $1.9-4.8\times$ finetuning throughput improvements under heavy loads and $2.5-6.8\times$ under light loads.

2 Background and Challenges

2.1 Parameter-Efficient Finetuning

Finetuning enables LLMs to adapt to specific domains and downstream tasks [12, 22, 72]. However, full finetuning introduces significant computational and memory overheads. To address this, *parameter-efficient finetuning* (PEFT) methods [24] have been proposed. These include prompt embeddings [45, 47, 53], adapter modules, low-rank decomposition (LoRA) [36], and $(IA)^3$ [51] scaling of the pre-trained weights. Recent unified approaches also combine various PEFT methods using heuristics [33] or neural architecture search [57, 111, 114]. While PEFT approaches minimize trainable parameters, reducing parameter count is sometimes not enough to reduce the memory footprint.

2.2 PEFT-based LLM Inference

LLM inference is increasingly essential in research and industrial applications. Recent systems serve multiple PEFT-based LLMs simultaneously. For example,PetS [112] decouples linear layers into shared and task-specific operations, optimizing scheduling for improved throughput. More recent works focus on optimizations that are specific to generative decoding models. Punica [14] introduces specialized CUDA kernels for multi-tenant LoRA serving. S-LoRA [76] proposes unified paging and heterogeneous batching for thousands of LoRA components. dLoRA [91] improves efficiency through dynamic batching adjustments.

2.3 Challenges in Co-serving Inference and Finetuning

Co-serving inference and finetuning tasks within a single system, as envisioned by FlexLLM, presents significant challenges due to their differing workload characteristics. FlexLLM must address three key challenges.

First, *how can inference and finetuning tasks be co-located while minimizing the memory footprint*? Traditional multi-tenancy ML scheduling systems share resources across tasks and models [92], but suffer from the high memory demands associated with LLMs. Furthermore, existing finetuning systems usually retain all intermediate activations similar to training systems to enable gradient computation during backpropagation, which substantially increases their resource consumption compared to pure inference tasks. This discrepancy makes it challenging to manage system resources between finetuning and inference tasks.

FlexLLM addresses this by co-locating finetuning and inference tasks to share backbone LLM parameters and activation buffers, using *dependent parallelization* to optimize the placement of trainable *bypass networks* (e.g., LoRA adapters [36]) in a distributed fashion (see § 5.1) and *graph pruning* to eliminate unnecessary data dependencies for frozen parameters (see § 5.2). These optimizations reduce finetuning memory overhead by up to $8\times$.

Next, *how can we run inference and finetuning tasks simultaneously while minimizing the interference with inference latency and SLO attainment*? Previous work on traditional ML workloads [73] has shown that co-locating inference with training can significantly degrade inference performance. Unlike conventional training workloads that require coarse-grained temporal or spatial resource sharing [16], LLM finetuning provides a unique opportunity to batch base LLM computation and parameter access across requests from different finetuned variants [91]. However, due to differences in execution logic—auto-regressive decoding in inference versus backpropagation in finetuning—merging the two types of tasks remains difficult. To overcome this challenge, FlexLLM introduces *token-level finetuning*, decomposing finetuning se-

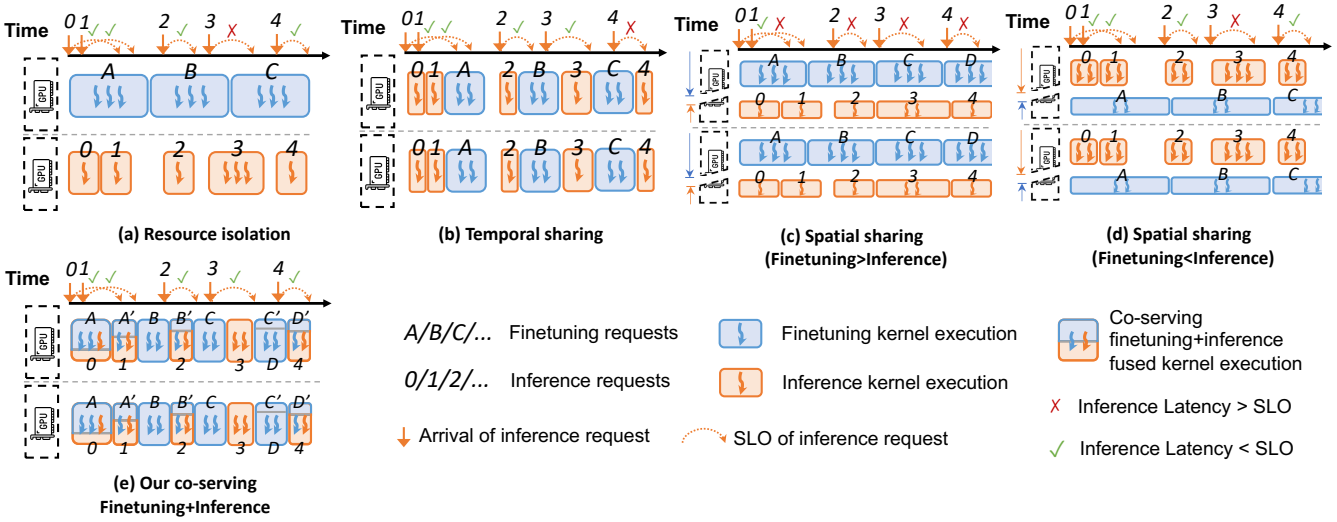


Figure 1: Comparing different resource sharing approaches for serving finetuning and inference. For spatial sharing and co-serving, the height of rounded rectangles illustrates the splitting ratio of GPU resources (e.g., streaming multi-processors).

quences into smaller units (§6.1) that, during the forward pass, follow the same execution order as inference tokens. In this way, both inference and finetuning tokens can be processed together using fused GPU kernels. For the backward pass, FlexLLM launches separate GPU streams for finetuning tokens and adopts a layer-wise execution strategy to minimize memory usage.

Finally, *how can we handle fluctuating inference workloads while preserving their latency SLO*? Given the dynamic nature of inference request arrival patterns, provisioning for peak demand often results in significant resource under-utilization. FlexLLM's fine-grained, token-level execution strategy enables quick adaptation to load fluctuations. To achieve this, FlexLLM introduces a *hybrid token scheduler* (Section 6.2) that prioritizes inference tokens for SLO compliance while opportunistically inserting finetuning tokens in a best-effort manner to maximize GPU utilization.

3 Co-serving LLM Inference and PEFT

Previous studies have proposed different methods for scheduling heterogeneous workloads (in our case, inference and PEFT) on shared GPU resources. These methods can be classified into three categories: *resource isolation*, *temporal sharing*, and *spatial sharing*. This section discusses the limitations inherent to these conventional approaches when applied to co-serving LLM inference and PEFT. It also highlights the fundamental distinctions between FlexLLM's co-serving solution and previous work. Figure 1 depicts the GPU execution timelines associated with different scheduling approaches for managing inference and finetuning tasks.

Resource isolation. A common approach to support both inference and finetuning tasks is *resource isolation*, which partitions the available GPUs into two distinct groups, each dedicated to handling one type of workload (see Figure 1(a)). This approach ensures that throughput-intensive finetuning tasks do not interfere with latency-critical inference tasks. When finetuning an LLM, a dataset of requests is provided to train the PEFT layers, with all requests submitted to FlexLLM simultaneously, making it easier to build larger batches that can be processed sequentially. In contrast, inference requests are submitted independently by users who expect timely processing within predetermined SLOs, with the arrival times of these requests unpredictable.

Resource isolation can introduce significant GPU under-utilization, particularly in scenarios with variable request arrival rates. This is because finetuning and inference requests cannot be batched together if the number of pending requests of either type is insufficient to fully utilize a batch. With fewer GPUs available to parallelize inference computations, it may not be possible to serve requests fast enough to meet their SLO (e.g., request r_3 in Figure 1(a)).

Temporal sharing. Compared to resource isolation, *temporal sharing* [92,93] improves GPU utilization by allowing different tasks to share GPU resources through time slices of different lengths depending on various parameters. This method allows both inference and finetuning requests to access all GPUs within the cluster in turns, as shown in Figure 1(b). By using all GPUs to parallelize computation (e.g., with data parallelism or tensor model parallelism), the latency for each request can be significantly reduced, thus aiding in meeting inference SLOs. For example, while inference request r_3 fails to complete in the scenario illustrated in Figure 1(a) due to resource limitations, it successfully executes in Figure 1(b) by leveraging more resources. However, the arrangement of time slices can also be harmful because of the unpredictable arrival times of inference requests, which may have a strin-

gent latency requirement that cannot be met (e.g., r_4 has been delayed in Figure 1(b) by pre-scheduled finetuning requests).

Spatial sharing. *Spatial sharing* offers an alternative by enabling simultaneous execution of different tasks at the kernel level, where each task utilizes a specific portion of the GPU resources (i.e., streaming multi-processors) This can be implemented on NVIDIA GPUs through multi-stream programming or by utilizing the Multi-Process Service (MPS) [4]. Furthermore, on the latest Hopper and Ampere architectures, spatial sharing benefits from the Multi-Instance GPU (MIG) [3], which allows for dynamic adjustments of GPU partition configurations. However, these adjustments often lack the flexibility and efficiency required to handle dynamic inference workloads effectively [2, 23, 96]. Figures 1(c) and (d) show two scenarios where a substantial portion of GPU resources is allocated to either finetuning or inference tasks. While over-provisioning for inference tasks may appear to provide better SLO guarantees, it still falls short during peak demand periods such as the unexpected urgent request r_3. Additionally, over-provisioning wastes resources and significantly slows down the finetuning process. Therefore, relying solely on spatial sharing to manage finetuning and inference tasks often results in suboptimal performance [16, 106].

FlexLLM's co-serving approach. As shown in Figure 1(e), FlexLLM employs a co-serving approach that uses a fine-grained scheduling mechanism to adaptively manage both inference and finetuning requests.

Unlike temporal or spatial sharing approaches, which schedule GPU kernels of existing systems independently, FlexLLM integrates inference and finetuning kernels. This integration reduces the overhead associated with kernel launches and minimizes accesses to GPU device memory to retrive model weights. In each iteration, FlexLLM dynamically adjusts the allocation of inference and finetuning tokens, strategically balancing the need to meet the SLOs of inference requests with the goal of maximizing GPU utilization. This adaptive approach ensures that FlexLLM adheres to the requirements of inference requests while optimizing the use of available GPU resources.

4 System Overview

Figure 2 shows an overview of FlexLLM, which co-serves both PEFT and inference requests for LLMs. Users can choose to use either finetuning or inference services on a model from the PEFT *model hub*, which stores the backbone LLM and all finetuned variants. The programming interface for both inference and finetuning requests is unified through a *PEFT-as-a-Service (PaaS) interface*. FlexLLM comprises three main components. First, for finetuning requests that execute in parallel across multiple GPUs, the static compilation module (§5) generates a parallel computation graph. This graph specifies the execution of the PEFT model over the distributed environment and optimizes the graph by pruning

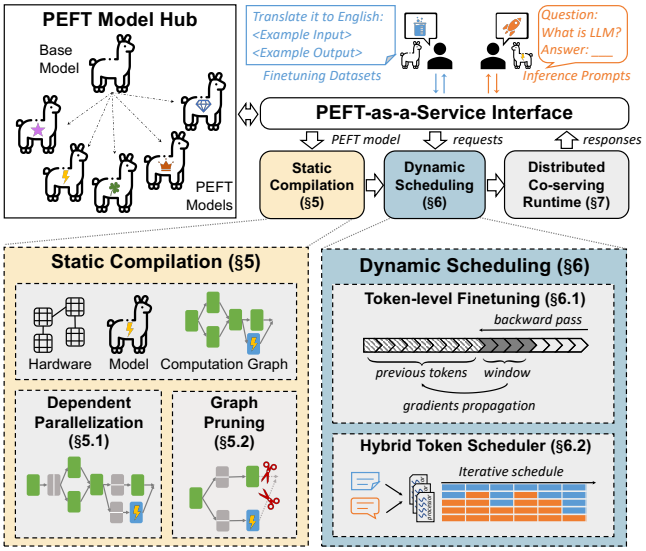


Figure 2: An overview of FlexLLM.

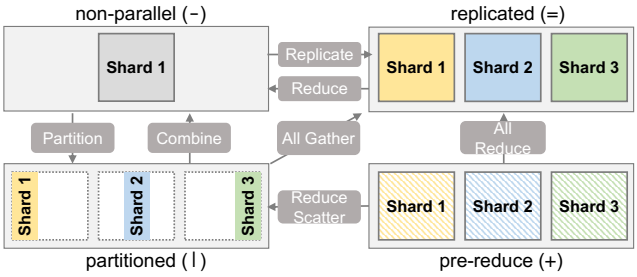
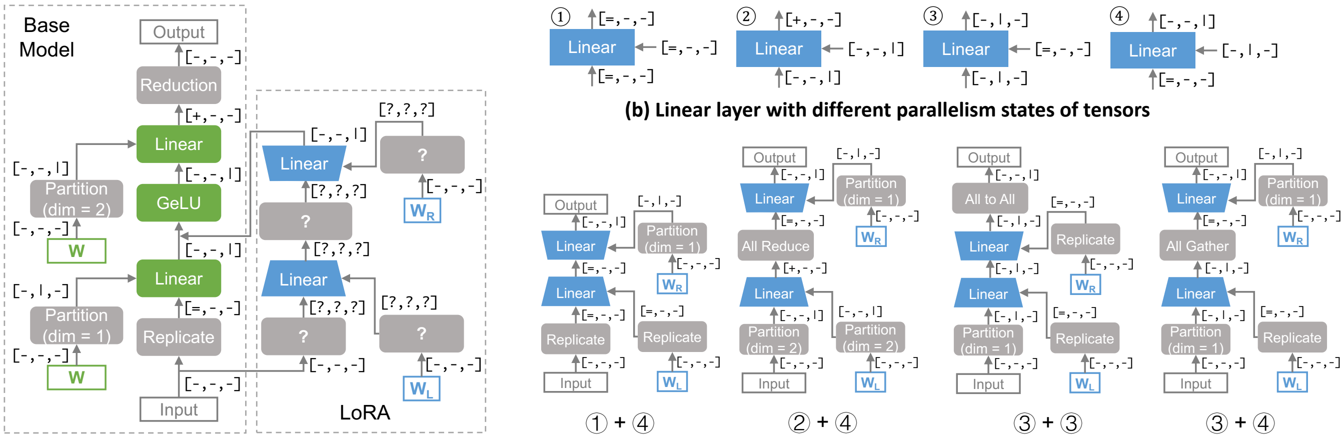


Figure 3: Four possible parallel states for a tensor dimension and their transitions. For each parallel state, the symbol in parenthesis shows the notation FlexLLM used to represent it.

unnecessary tensors for memory saving. Second, the dynamic scheduling module (§6) adapts a token-level finetuning mechanism. It mixes inference and finetuning tokens with a hybrid scheduling policy. Finally, the computation graph and schedule plan are executed by FlexLLM's distributed co-serving runtime (§7).

4.1 PEFT-as-a-Service Interface

To facilitate the sharing of the backbone LLM across different tasks, FlexLLM represents a PEFT model as a sequence of *bypass networks* attached to the backbone LLM. Each bypass network takes a *single* tensor from the backbone LLM as input and produces a *single* output tensor, which is added to one tensor of the backbone LLM. Let X and Y denote the input and output tensor of the bypass network, and let $f_B(X)$ and $f_A(X)$ denote the neural architecture of the backbone and bypass network for calculating Y. The bypass network can be formulated as $Y = f_B(X) + f_A(X)$. All existing PEFT methods can be represented in this format. For instance, the $(IA)^3$ architecture modifies the topology of the backbone LLM



(a) Base model PCG with LoRA

(b) Linear layer with different parallelism states of tensors

(c) Candidate PCGs for LoRA under different parallelism states and transitions

Figure 4: Illustration of FlexLLM's different dependent parallelization strategies with the LoRA example. Each green (or gray) box indicates a compute (or parallelization) operator, and each edge between operators represents a parallel tensor, and the parallelization states of the tensor's dimensions are shown next to the edge.

by inserting an elementwise multiplication operator. It can be transformed into FlexLLM's bypass network format since $Y = X \odot W = X + X \odot (W - O)$, where $\odot$ is elementwise multiplication and O is a matrix of the same shape as W whose elements are all one. A significant benefit of this interface is that all PEFT models preserve the neural architecture of the backbone LLM, which enables FlexLLM to fuse the computation graphs of different PEFT models. FlexLLM provides a unified user interface to launch inference and finetuning tasks by specifying a PEFT model and committing different types of input data, such as finetuning datasets or inference prompts.

5 Static Compilation

Once a PEFT model is registered, FlexLLM compiles it into a parallel computation graph (PCG) [86]. To optimize finetuning performance, FlexLLM applies two key static compilation optimizations to discover an optimized PCG: *dependent parallelization* (§5.1) and *graph pruning* (§5.2).

5.1 Dependent Parallelization

To minimize memory overhead, FlexLLM allows finetuning and inference tasks to utilize the same backbone LLM and share memory allocated for both model weights and intermediate activations. This approach avoids redundant memory consumption while requiring the parallelization strategies of the PEFT models to be compatible with that of the backbone LLM, a task termed *dependent parallelization*.

Existing auto-parallelization approaches like Alpa [107] and Unity [86] can identify optimal parallelization strategies for backbone LLMs. FlexLLM extends the optimization target to also include the bypass networks in PEFT models. To achieve this goal, FlexLLM generalizes the *parallel computation graph* (PCG) abstraction introduced in Unity [86] by associating a *state* field with tensor dimension.

Figure 3 shows the four possible tensor states when parallelizing a tensor, including *non-parallel* (−), *partitioned* (|), *replicated* (=), and *pre-reduce* (+), as well as the state transitions. Given the fixed parallelization of the backbone LLM, FlexLLM searches for an optimal PCG for each bypass network by enumerating possible parallelization operators—`partition`, `combine`, `replicate`, and `reduce`—inferring output tensor states, and validating their compatibility. Figure 4 shows four candidate PCGs discovered by FlexLLM for parallelizing a LoRA network. To select the best strategy, FlexLLM reuses Unity's profiling-based cost model [86] and chooses the candidate PCG with the lowest estimated execution cost.

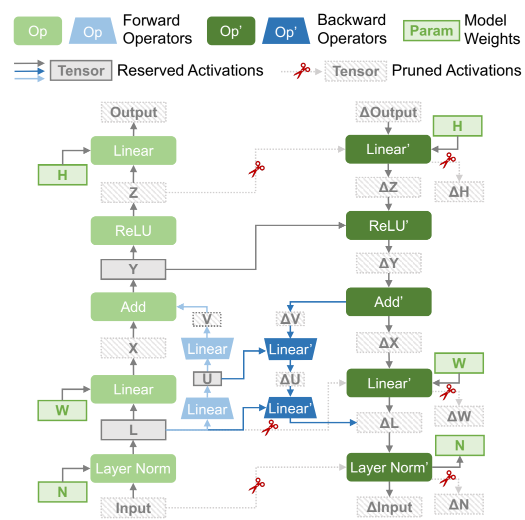


Figure 5: Static graph pruning for an MLP model with LoRA.

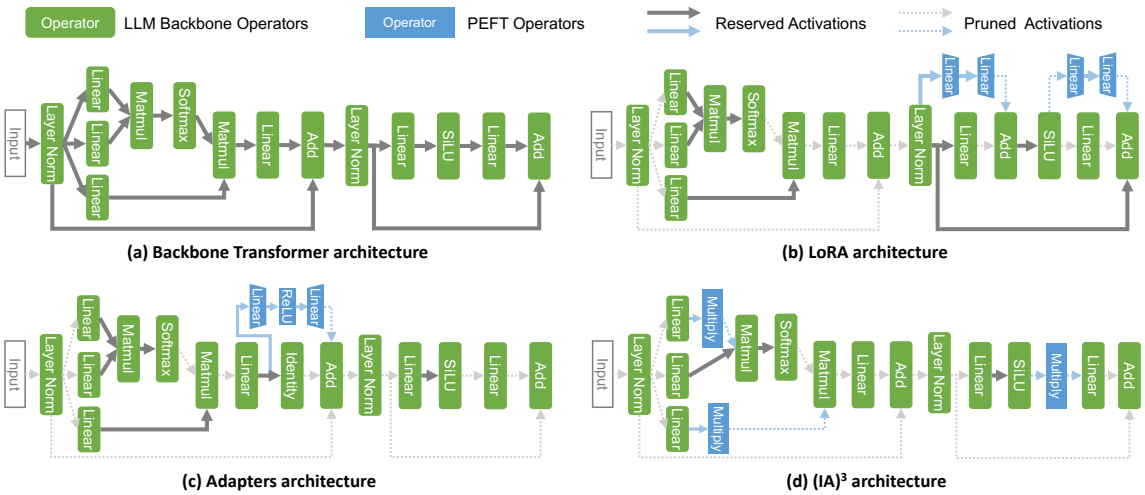


Figure 6: An overview of existing parameter-efficient finetuning (PEFT) methods. Green boxes show the operators of the backbone LLM, while blue boxes are the operators introduced by different PEFT methods. Arrows represent intermediate activations to be reserved and the dashed arrows demonstrate they are pruned by FlexLLM.

5.2 Graph Pruning

FlexLLM's *graph pruning* algorithm takes as input a PEFT model and its backbone LLM, and outputs a minimal set of intermediate activations that must be reserved to perform finetuning of the bypass networks. This pruning process requires reasoning over data dependencies between operators, specifically their input and output tensors. Importantly, this pruning process preserves model quality since gradients with respect to frozen parameters are mathematically unnecessary for PEFT optimization and do not affect the final trained model. To facilitate the analysis of data dependencies, FlexLLM introduces additional notation in the PCG representation. Let $\mathcal{G} = (\mathcal{N}, \mathcal{E})$ denote the PCG of a PEFT model, where each node $n \in \mathcal{N}$ represents a tensor algebra (or parallelization) operator, and each edge $e = (n_1, n_2) \in \mathcal{E}$ is a tensor shared between operators. For a node n, let $I(n)$ and $O(n)$ denote its set of input and output tensors. By definition, $(n_1, n_2) \in \mathcal{E}$ if and only if $O(n_1) \cap I(n_2) \neq \emptyset$, i.e., when at least one output tensor of n_1 serves as an input to n_2.

The design of FlexLLM's graph pruning algorithm is based on two key observations. First, conventional ML training procedures maintain all intermediate activations during the forward pass to compute gradients in the backward pass. However, due to the linear algebra nature of most operators, these intermediate activations are primarily used to compute gradients with respect to the base LLM's trainable parameters. In the case of PEFT, the backbone LLM parameters are frozen, and only the parameters in the bypass networks are updated. This insight allows FlexLLM to eliminate most intermediate activations, retaining only those necessary for computing gradients of the bypass networks. Second, different PEFT methods connect bypass networks at varying locations within the backbone LLMs, and therefore require different subsets of intermediate activations for backpropagation. Figure 6 illustrates examples of how various PEFT methods attach their bypass networks to the base LLM.

Building on these observations, FlexLLM performs *static* graph pruning during the construction of the PCG for each registered PEFT model, and *dynamical* scheduling to manage finetuning and inference tasks, along with corresponding memory allocations (see Section 6). Algorithm 1 shows FlexLLM's graph pruning algorithm, with Figure 5 illustrating the process for an MLP model with LoRA. Given a PEFT model's PCG $\mathcal{G}$, the algorithm generates an execution plan specifying which intermediate activations to cache for efficient backpropagation.

The algorithm constructs the backward graph $\overline{\mathcal{G}}$ using reverse-mode automatic differentiation [34], then prunes gradients for frozen base LLM weights. It iteratively processes operators to discover pruning opportunities, ultimately identifying the minimal set of activations $\mathcal{A}$ required for the backward pass. FlexLLM combines this with *rematerialization* and *activation compression*.

Rematerialization and compression. FlexLLM applies tensor rematerialization [39], selectively discarding tensors in the forward pass and recomputing them during backpropagation. For each tensor $t \in \mathcal{A}$ after graph pruning, FlexLLM rematerializes t if all input tensors are stored and recomputation has low overhead. Additionally, FlexLLM opportunistically applies lossless compression when operators like `ReLU` don't require access to original input tensors. $y = \texttt{ReLU}(x) = \max(x, 0)$ whose derivative is $\partial y / \partial x = 1$ for $x > 0$ and 0 for $x \leq 0$. Therefore, instead of storing the original input tensor x, FlexLLM keeps the bitmask of x.

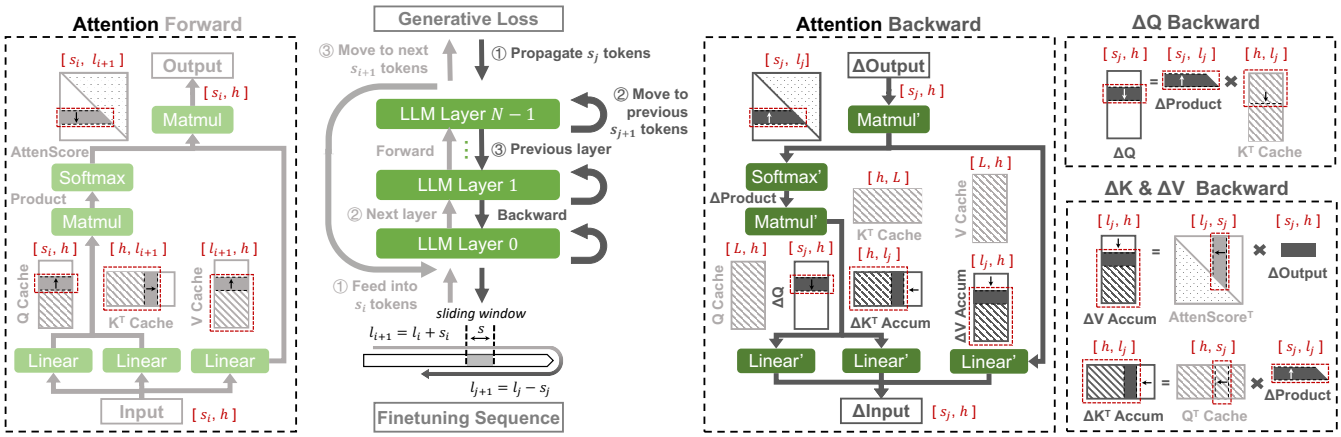


Figure 7: Illustration of attention module's forward and backward execution with FlexLLM's token-level finetuning mechanism.

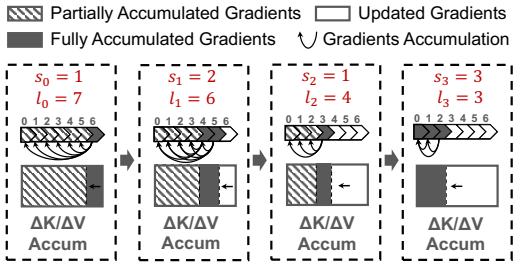


Figure 8: Illustration of KV gradients accumulation in the backward pass of FlexLLM's token-level finetuning mechanism, where s_j and l_j are the window length and the starting position of the j-th slice of the sequence.

6 Dynamic Scheduling

Once static compilation produces an executable generalized PCG for a PEFT model, FlexLLM must determine an execution schedule capable of concurrently handling both finetuning and inference requests. This is particularly challenging due to the distinct characteristics of these two workloads. First, inference requests arrive in an online and highly dynamic manner, requiring the system to rapidly adapt GPU resource allocation to maintain low-latency responses. In constrast, finetuning involves both a forward pass—similar to inference—and a backward pass to update model parameters. Additionally, auto-regressive inference for generative LLMs proceeds in a token-by-token fashion, while finetuning is typically performed at the sequence level to maximize throughput. To address these challenges, FlexLLM introduces a novel *token-level finetuning* mechanism along with a *hybrid token scheduler*.

6.1 Token-Level Finetuning

Processing a finetuning request typically involves performing both forward and backward passes across all model layers for the entire input sequence. When co-served with inference requests, this sequence-level execution can significantly interfere with the token-wise inference kernels and degrade system goodput—this is, the effective inference throughput that meets latency SLOs.

While sequence-level parallelism [38,41] could help reduce the added finetuning latency per GPU, this approach would suffer from coarse-grained scheduling, forcing the system to wait for entire finetuning sequences to complete before reallocating resources to handle inference bursts.

To address this problem and maximize GPU utilization, FlexLLM introduces a *token-level finetuning* mechanism, detailed in Algorithm 2. The key idea is to decompose the finetuning computation into smaller steps using a dynamic sliding window over tokens. This window spans both the forward and backward passes, with its size dynamically determined by the hybrid token scheduler (line 4 and 15) to ensure that inference requests continue to meet their SLO constraints (§6.2).

During the forward pass (lines 3-11), the finetuning sequence is partitioned into windows of s_i tokens (line 5), which are incrementally fed into the model. The model processes each token window layer by layer (lines 7-8), computes the generative loss for the window (line 10), and then advances to the next set of tokens (line 11). To preserve the semantic of full sequence-level finetuning, FlexLLM caches key and value tensors—similar to incremental decoding in inference [97]—as well as query tensors, which are reused during backward attention computations. This caching follows the *causal mask* constraint inherent in LLMs.

During the backward pass (lines 12-21), FlexLLM executes the model layers in reverse order (line 13). Within each layer, the finetuning request is again divided into token windows (lines 14-16). However, unlike the relatively straightforward forward pass, token-level backward execution of the attention module is more complex due to auto-regressive token dependencies.

Figure 7 illustrates the forward and backward PCGs of the attention module. When computing gradients for s_j tokens,

Algorithm 1 Static graph pruning. For an operator n, UPDATEINPUT$(n, O(n))$ returns a set of input tensors needed in order only to compute $O(n)$ of the operator.

Inputs: PCG of a PEFT model $\mathcal{G}$
Outputs: $\mathcal{A}$ is a set of tensors to be memorized, $\mathcal{R}$ is a set of tensors to be rematerialized
▷ Step 1: computation graph pruning
$\overline{\mathcal{G}} = \text{REVERSEAUTODIFF}(\mathcal{G})$
$\mathcal{Q} = \emptyset$ ▷ $\mathcal{Q}$ is a queue of updated operators
for operator $n \in \overline{\mathcal{G}}$ **do**
  for output tensor $t \in O(n)$ **do**
    if t is the weight gradient of the base LLM **then**
      $O(n) = O(n) \setminus \{t\}$
      $I(n) = \text{UPDATEINPUT}(n, O(n))$
      $\mathcal{Q}$.push_back(n)
while $\mathcal{Q}$ is not empty **do**
  $n = \mathcal{Q}$.pop_front()
  for output tensor $t \in O(n)$ **do**
    if $\nexists u.t \in I(u)$ **then**
      $O(n) = O(n) \setminus \{t\}$
      $I(n) = \text{UPDATEINPUT}(n, O(n))$
      $\mathcal{Q}$.push_back(n)
$\mathcal{A} = \emptyset$
for operator $n \in \mathcal{G}$ **do**
  for tensor t *in* $O(n)$ **do**
    if $\exists u \in \overline{\mathcal{G}}.t \in I(u)$ **then**
      $\mathcal{A} = \mathcal{A} \cup \{t\}$
▷ Step 2: opportunistically rematerializing tensors
for tensor $t \in \mathcal{A}$ **do**
  Let n be the operator that outputs t (i.e., $t \in O(n)$)
  if $I(n) \subseteq \mathcal{A}$ and $\text{COST}(n) < threshold$ **then**
    $\mathcal{A} = \mathcal{A} \setminus \{t\}$, $\mathcal{R} = \mathcal{R} \cup \{t\}$
return $\mathcal{A}, \mathcal{R}$

the resulting gradients ΔQ, ΔK, ΔV have shapes $[s_j, h]$ for queries but $[l_j, h]$ for keys and values due to autoregressive attention dependencies. FlexLLM accumulates KV gradients across steps, applying them only after complete backward traversal. This accumulation strategy minimally increases memory consumption due to layer-wise execution enabling workspace memory reuse.

6.2 Hybrid Token Scheduler

FlexLLM's *hybrid token scheduler* coordinates token-level GPU execution in two stages. First, FlexLLM determines the scheduling of available inference requests based on specific scheduling policies. By default, FlexLLM adopts Orca's *iteration-level scheduling* [97], which maintains a fixed maximum batch size and dynamically replaces each completed request with a new one whenever available. To further mitigate blocking caused by long input sequences, FlexLLM incorporates the chunked-prefill optimization [6].

Algorithm 2 Token-level finetuning mechanism.

Inputs: PEFT model M depth of N; Finetuning request r with sequence length of L
Outputs: PEFT model gradients ΔM
for $i \leftarrow 0, l_0 \leftarrow 0$; $l_i < L$; i++ **do** ▷ Forward pass
  $s_i \leftarrow \text{HYBRIDTOKENSCHEDULER}(l_i, L)$
  $r_i \leftarrow \text{SLICE}(r, l_i, s_i)$
  $X_{0,i} \leftarrow \text{TOKENIZE}(r_i)$
  for model layer index $n \in \text{RANGE}(0, N)$ **do**
    $X_{n+1,i}, Q_{n,i}, K_{n,i}, V_{n,i} \leftarrow \text{FORWARD}(M_n, X_{n,i})$
    $QKVCache_n \leftarrow \text{APPEND}(Q_{n,i}, K_{n,i}, V_{n,i})$
  $Loss_i \leftarrow \text{GENERATIVELOSS}(X_{N,i}, r_i)$
  $l_{i+1} \leftarrow l_i + s_i$
$Y_N \leftarrow Loss$
for model layer index $n \in \text{RANGE}(N-1, -1, -1)$ **do**
  for $j \leftarrow 0, l_0 \leftarrow L$; $l_j > 0$; j++ **do** ▷ Backward pass
    $s_j \leftarrow \text{HYBRIDTOKENSCHEDULER}(l_j, 0)$
    $Y_{n+1,j} \leftarrow \text{SLICE}(Y_{n+1}, l_j, s_j)$
    $Y_{n,j}, \Delta K_{n,j}, \Delta V_{n,j} \leftarrow \text{BACKWARD}(M_n, Y_{n+1,j}, QKVCache_n, \Delta KVAccum_n)$
    $\Delta KVAccum_n \leftarrow \text{ADD}(\Delta K_{n,j}, \Delta V_{n,j})$
    $G_{n,j} \leftarrow \text{SLICE}(\Delta KVAccum_n, l_j, s_j)$
    $\Delta M_n \leftarrow \Delta M_n + \text{CALCULATEGRADS}(M_n, G_{n,j})$
  $l_{j+1} \leftarrow l_j - s_j$
return ΔM

Second, after determining the inference schedule for a given iteration, FlexLLM opportunistically appends as many finetuning tokens as possible—determined by the sliding window size s—to maximize GPU utilization. The number of finetuning tokens added is determined automatically using the formula $s = \arg\max f(c, s) \leq \text{SLO}$, where $f(\cdot, \cdot)$ is the latency estimation function and c is the number of inference tokens scheduled in the current iteration. Here $f(\cdot, \cdot)$ is derived via offline profiling of the LLM's execution [61]. Such window-based, best-effort scheduling mechanism ensures that FlexLLM can automatically adapt to inference workload fluctuations, maintaining inference SLOs while opportunistically allocating compute to finetuning tasks (§8.3).

Figure 9 illustrates the execution timeline of FlexLLM and shows an example a token scheduling plan for a finetuning mini-batch A, under a representative inference requests arrival pattern (i.e., r_0 to r_4). To preserve the semantics of finetuning, FlexLLM enforces the execution dependencies between the forward and backward passes of each mini-batch. During the forward pass, FlexLLM leverages fused GPU kernels to jointly process both inference and finetuning tokens. This approach avoids additional kernel launch overhead and is enabled by the shared token-wise computation logic between inference and finetuning. In particular, finetuning tokens follow the same execution pattern and causal masking rules as inference tokens during the prefill phase, guaranteeing com-

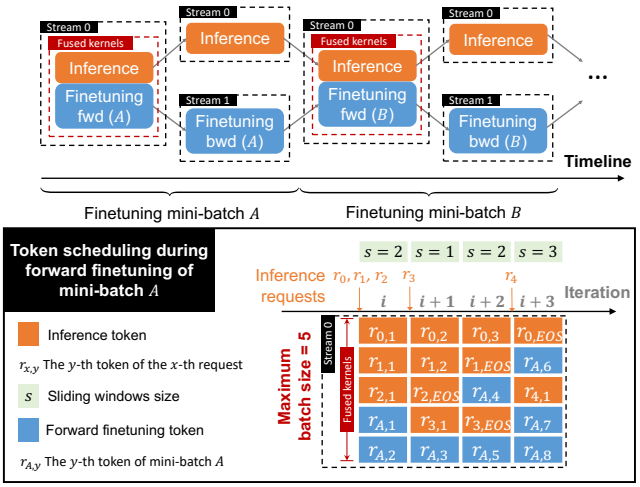


Figure 9: Execution timeline and token scheduling when co-serving inference requests and a finetuning mini-batch.

patibility at the kernel level. For the backward pass, FlexLLM adapts a space-sharing approach, using two separate GPU streams: one for inference tokens and another for finetuning tokens. This design allows concurrent execution without interference.

7 Implementation

FlexLLM is built on FlexFlow Serve [59] with 9K lines of C++ and CUDA code. It provides C++ and Python bindings, with full HuggingFace compatibility for `PEFT` [56] and `transformers` [90] libraries. FlexLLM supports PEFT models with LLAMA [84], GPT [12], OPT [104], Falcon [9], or MPT [82] backbones, among others.

Memory management. FlexLLM manages the required GPU memory using a combination of static and dynamic allocation. Static allocation reserves GPU space for the backbone weights and the KV cache required for incremental decoding during inference. Dynamic allocation is used for gradients, activations, and optimizer states. When a finetuning request is received, FlexLLM allocates memory during the first forward pass and reuses it during subsequent backward passes, freeing up space when it's no longer needed. For inference memory management, FlexLLM employs paged attention [43] with chunked prefill [7] to dynamically allocate KV cache pages and minimize evictions. New inference requests are only admitted if the entire prompt can fit within available KV cache pages, preventing memory fragmentation and ensuring stable inference performance during co-serving.

Key-value gradient accumulator. To support token-level finetuning, it is necessary to accumulate key-value gradients for every preceding token in a given sequence during backpropagation. FlexLLM utilizes static allocation to reserve space for key-value gradient accumulation. During backpropagation, the gradients for keys and values of a specific segment are obtained, and the position of this segment in the overall sequence is determined. FlexLLM then accumulates the new key-value gradients of all subsequently scheduled segments. When a segment is scheduled during backpropagation, it includes accumulated gradients from future tokens.

8 Evaluation

LLMs and PEFT models. We evaluated FlexLLM across three publicly available LLMs: LLaMA-3.1-8B [85], Qwen-2.5-14B [94], and Qwen-2.5-32B [94] from HuggingFace [1]. Since existing LLM serving systems only support base models and LoRA-based PEFT, we focus on these configurations for fair comparison. We applied LoRA with rank 16 to MLP down projection layers, yielding 9.4M, 14.5M, and 25.16M trainable parameters respectively.

Platform. Experiments were performed on Perlmutter [5] nodes with AMD EPYC 7763 processors, 256 GB DRAM, and four NVIDIA A100-SXM4-80GB GPUs connected via HPE Slingshot (200 Gb/s). We used tensor-model parallelism for LLM serving and finetuning.

Workload. We synthesized inference workloads using ShareGPT [83] prompt/generation lengths and Azure ChatGPT production traces [87] for realistic arrival patterns, following recent work [7, 43, 80, 99, 110, 113]. The original trace spans multiple days and captures real production arrival patterns from ChatGPT serving on Azure OpenAI, providing realistic inference workload characteristics including bursty request arrivals and varying concurrency patterns. We sampled 20-minute intervals, adjusting the arrival times to simulate different average arrival rates. We set TPOT SLOs to 50ms (8B model) and 75ms (14B/32B models) following the methodology from prior work [7], with 5s maximum TTFT to prevent excessive queueing.

We sampled each finetuning request from the `Sky-T1_data_17k` dataset [64], used to finetune the `Sky-T1-32B-Preview` reasoning model [63], a finetuned Qwen2.5 variant with performance comparable to OpenAI's o1-preview [69] on math and coding benchmarks. We truncated sequences to 8192 tokens and used the Adam optimizer [40].

8.1 End-to-end Comparison

We compare FlexLLM's co-serving approach with existing approaches that employ separate clusters for inference and finetuning tasks. In the separate cluster approach, we use vLLM [43] and LlamaFactory [109] as the inference and finetuning systems, respectively, as they represent state-of-the-art solutions for these workloads. We selected vLLM over alternatives like DeepSpeed-MII [21] and TensorRT-LLM [65] because recent vLLM optimizations have closed previous performance gaps, while PipeSwitch [10] lacks support for autoregressive LLM generation. We enable all available optimizations, including `torch.compile`, and chunked

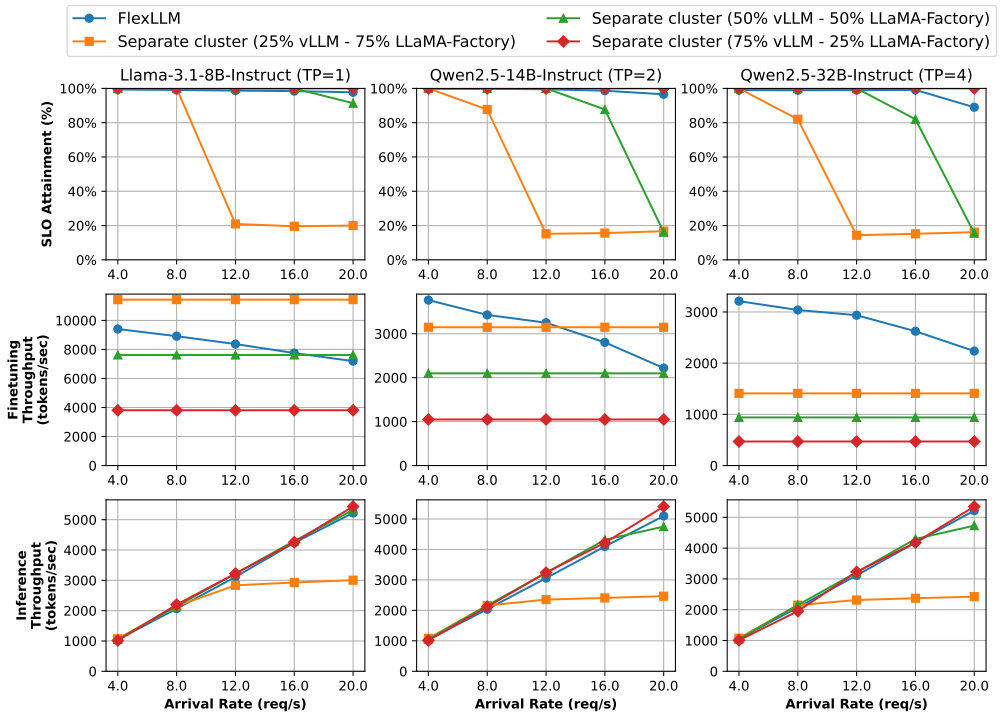


Figure 10: End-to-end comparison between co-serving and using separate resources on three models.

prefills for vLLM v1; DeepSpeed ZeRO Stage 3, Unsloth and FlashAttention for LlamaFactory. We allocated 4, 8, and 16 A100 GPUs for the three models respectively.

Figure 10 shows the results. The separate approach uses tensor parallelism (degree 1, 2, 4) for the three models respectively, creating four pipelines in total. We explored various strategies to distribute resources between inference and finetuning tasks. Configurations like 25% vLLM - 75% LlamaFactory allocate 25% of resources (i.e., 1 pipeline) to inference, 75% (i.e., 3 pipelines) to finetuning.

Configurations with fewer inference pipelines (25%-50% vLLM) handle only lightweight workloads, achieving limited inference throughput and SLO attainment under more intensive workloads. Dedicating more pipelines to inference (75% vLLM) maintains high SLO attainment but reduces finetuning throughput.

FlexLLM maintains near-optimal SLO attainment without compromising finetuning performance by prioritizing latency-sensitive inference and opportunistically maximizing finetuning tokens.

Across all three models, FlexLLM matches the 75% vLLM - 25% LlamaFactory configuration in inference SLO attainment (at or above 90% even at 20 req/s) and inference throughput, while dramatically improving finetuning throughput. Specifically, under heavy inference loads (20 req/s), FlexLLM sustains finetuning throughputs of 7.2K, 2.2K and 2.2K tokens/s for LLaMA-3.1-8B, Qwen-2.5-14B, and Qwen-2.5-32B respectively, compared to only 3.8K, 1.0K, and 0.5K tokens/s in the 75% vLLM - 25% LlamaFactory setup—i.e. a **1.9×-4.8×** improvement. Under light inference loads (4.0 req/s), FlexLLM sustains finetuning throughputs of 9.4K, 3.7K, and 3.2K tokens/s for the same models, translating to a **2.5×-6.8×** improvement over the 75% vLLM - 25% LlamaFactory setup.

8.2 GPU Scheduling

This section compares FlexLLM's co-serving mechanism with two commonly used GPU scheduling strategies: *temporal sharing* and *spatial sharing*. Temporal sharing interleaves the execution of inference and finetuning tasks over time, as illustrated in Figure 1. Spatial sharing simultaneously launches inference and finetuning computations using separate CUDA resources. Both strategies are implemented on top of FlexLLM by replacing the co-serving mechanism with their respective scheduling strategies. To ensure a fair comparison, all memory optimizations are enabled in each baseline.

Figure 11 shows the results. For temporal sharing, interleaving one inference iteration with one finetuning iteration would violate the SLOs for nearly all inference requests. This is because each inference iteration is expected to complete within tens of milliseconds, whereas a finetuning iteration requires several seconds. To mitigate this issue, in the temporal sharing experiments, we interleave each finetuning iteration with n inference iterations, where n is the inference frequency shown in the figure.

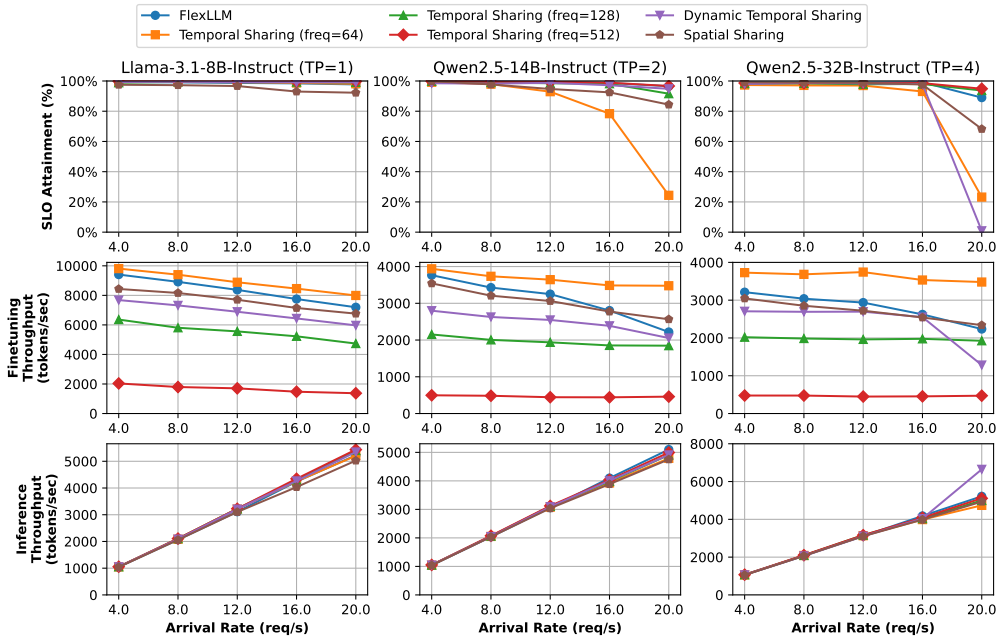


Figure 11: End-to-end comparison of co-serving with temporal and spatial sharing strategies.

While adopting a low inference frequency in temporal sharing (i.e., frequency = 64) maximizes finetuning throughput, it adversely impacts the SLO attainment and throughput for inference requests. Employing a frequency of 128 enables temporal sharing to match the inference throughput and SLO attainment rate of co-serving. However, it also reduces finetuning throughput by 0.57x-0.86× compared to co-serving.

Finally, we also implemented a dynamic temporal sharing baseline (see Appendix A), where the interleaving frequency is determined dynamically based on queue lengths, batch sizes, arrival rates, and completion rates. Dynamic temporal sharing outperforms fixed-frequency temporal sharing by adapting to workload conditions, maintaining SLO attainment above 90% in most scenarios, and achieving impressive inference throughputs of 5.4K, 4.9K, and 6.6K tokens/s for the three models under heavy loads (20 req/s). However, it still lags behind co-serving, whose finetuning throughput is **1.0–1.7**× higher, and shows instability under the heaviest loads as SLO attainment drops significantly for the 32B model.

Spatial sharing allocates separate GPU resources for inference and finetuning, achieving comparable finetuning throughput to co-serving. However, it remains suboptimal in SLO attainment under heavy inference workloads due to the interference between inference and finetuning tasks.

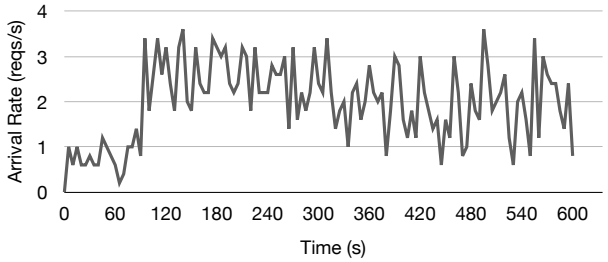
8.3 Case Study

This case study demonstrates FlexLLM's ability to dynamically adapt to fluctuating inference workloads in real-time. We evaluate FlexLLM using a 10-min interval of the BurstGPT trace with Qwen-2.5-14B model. To ensure compatibility with our experiment environments, we replayed this trace segment and re-scaled its arrival intensity, as previous works have done [8, 30, 60, 103]. In the experiment, each inference request and inference request are sampled from the ShareGPT dataset and the Sky-T1 dataset respectively.

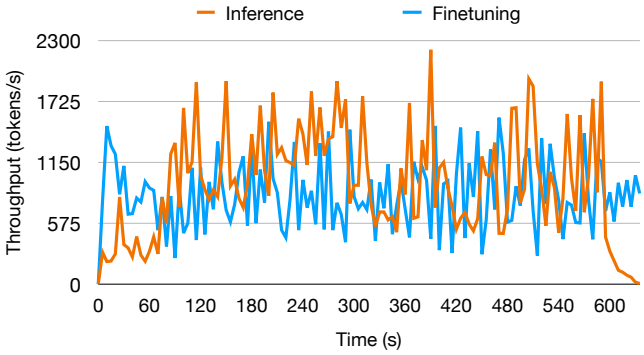
In our case study, as shown in Figure 12, we observed that the arrival rate of the inference requests initially increased to a peak level after around 90 seconds, and then gradually decreased with some peaks. FlexLLM could automatically detect the fluctuations in the workload and improve the ratio of inference tokens (vs finetuning tokens) in each iteration's batch. This significantly increased inference throughput from a few hundreds to 2.25K.

8.4 Memory Optimization

FlexLLM introduces a series of memory optimizations to reduce the memory overhead of PEFT finetuning and allow co-serving finetuning with inference workloads. To understand the effectiveness of FlexLLM memory optimizations, we perform an ablation study that incrementally turns off these optimizations (graph pruning, rematerialization, and token-level finetuning) and measures the memory requirements. Figure 13 shows the activation memory requirements of FlexLLM for different finetuning methods on a 70B LLM and a sequence length of 1024. FlexLLM saves 85%-87% memory requirements for activations compared to existing approaches. The major improvement comes from graph pruning. With graph pruning only, FlexLLM achieves 71%-74% activation memory overhead reduction. Rematerialization and



(a) Arrival Inference Requests



(b) System throughput

Figure 12: Case study of FlexLLM's system throughput for fluctuating inference workload.

token-level finetuning further reduce memory overhead by 0-8% and 4%-10%, respectively. These memory optimizations are highly effective in practice, enabling FlexLLM to retain sufficient memory for the inference requests' KV cache, ensuring eviction rates of 0% in most cases and peaking at only 1.2% for the largest model (Qwen2.5-32B) under the heaviest loads (see Appendix B).

9 Related Work

ML serving and inference systems. Numerous ML serving systems [19, 30, 31, 44, 73] address challenges like latency prediction, scalability, swapping [10], preemption [17], workload estimation [103], and cost-effectiveness [101]. Systems like Nexus [74] and Gpulet [16] focus on batching and GPU virtualization for small models. Recent LLM serving systems include vLLM [43], Sarathi [7], SGLang [108], AlpaServe [49], Punica [14], and S-LoRA [76], which improve throughput through batching and specialized PEFT kernels. Aqua [42] introduces fine-grained fair scheduling among multiple inference workloads. Speculative decoding [13, 59, 66] can reduce serving latency and improve SLO attainment [50] and goodput [54]; combining it with co-serving is future work.

GPU resource sharing for ML. GPU resource sharing focuses on efficient allocation among multiple ML users. Lyra [46] allocates dedicated resources with dynamic adjustment. Gandiva [92] and Antman [93] use time-slicing for better cluster utilization. GSLICE [23] develops adaptive allocation schemes for different SLO constraints. Although these approaches can not directly solve the resource utilization problem in PEFT service, we believe GPU resource sharing will be increasingly important as GPUs become more powerful.

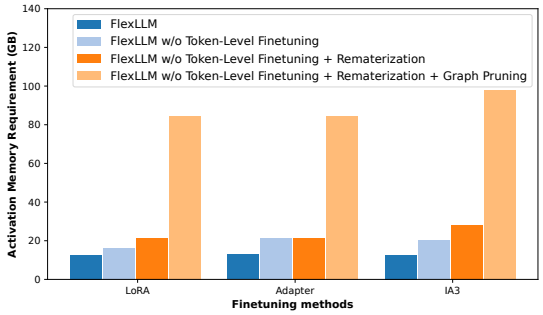


Figure 13: Ablation study of memory optimizations.

10 Limitations and Future Work

While FlexLLM demonstrates significant improvements in GPU utilization and finetuning throughput, we identify several promising directions for future work. First, extending our implementation beyond offline finetuning scenarios where sequences are processed individually (batch size 1), to reinforcement-learning-based approaches like RLHF [75, 78] and Group Relative Policy Optimization (GRPO) [32, 48, 98] presents an exciting opportunity, as our token-level co-serving architecture naturally fits these methods where auto-regressive generation and gradient updates are tightly coupled. Second, developing adaptive strategies for workloads with consistently high inference demand could further expand co-serving opportunities. Finally, advancing beyond our current simple scheduling heuristics to more sophisticated algorithms that optimize for task priorities, deadlines, and QoS requirements could unlock additional system efficiency gains.

11 Conclusion

This paper proposes FlexLLM, a system for co-serving parameter-efficient finetuning and inference of large language models. We observe the distinct GPU resource utilization features between finetuning and inference workloads and propose a holistic PEFT serving system that supports to co-serve finetuning requests without affecting inference latency.

Acknowledgement

We would like to thank the anonymous reviewers and our shepherd, Hitesh Ballani, for their valuable comments and suggestions. This research is partially supported by NSF awards CNS-2211882 and CNS-2239351, a Sloan research fellowship, and research awards from Amazon, Cisco, Google, Meta, NVIDIA, Oracle, Qualcomm, and Samsung.

References

[1] Huggingface Models. https://huggingface.co/models, 2023. 9

[2] NVIDIA MIG Partitioning Limitations and Resetting. https://docs.nvidia.com/datacenter/tesla/mig-user-guide/index.html#partitioning, 2023. 4

[3] NVIDIA Multi-Instance GPU (MIG). https://www.nvidia.com/en-us/technologies/multi-instance-gpu/, 2023. 4

[4] NVIDIA Multi-Process Service (MPS). https://docs.nvidia.com/deploy/mps/index.html, 2023. 4

[5] Perlmutter supercomputer. https://docs.nersc.gov/systems/perlmutter/architecture/, 2023. 9

[6] Amey Agrawal, Nitin Kedia, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav Gulavani, Alexey Tumanov, and Ramachandran Ramjee. Taming {Throughput-Latency} tradeoff in {LLM} inference with {Sarathi-Serve}. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 117–134, 2024. 8

[7] Amey Agrawal, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav S Gulavani, and Ramachandran Ramjee. Sarathi: Efficient llm inference by piggybacking decodes with chunked prefills. *arXiv preprint arXiv:2308.16369*, 2023. 9, 12

[8] Ahsan Ali, Riccardo Pinciroli, Feng Yan, and Evgenia Smirni. Optimizing inference serving on serverless platforms. *Proceedings of the VLDB Endowment*, 15(10), 2022. 11

[9] Ebtesam Almazrouei, Hamza Alobeidli, Abdulaziz Alshamsi, Alessandro Cappelli, Ruxandra Cojocaru, Merouane Debbah, Etienne Goffinet, Daniel Heslow, Julien Launay, Quentin Malartic, Badreddine Noune, Baptiste Pannier, and Guilherme Penedo. Falcon-40B: an open large language model with state-of-the-art performance. 2023. 9

[10] Zhihao Bai, Zhen Zhang, Yibo Zhu, and Xin Jin. {PipeSwitch}: Fast pipelined context switching for deep learning applications. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 499–514, 2020. 9, 12

[11] Paul Barham, Aakanksha Chowdhery, Jeff Dean, Sanjay Ghemawat, Steven Hand, Daniel Hurt, Michael Isard, Hyeontaek Lim, Ruoming Pang, Sudip Roy, et al. Pathways: Asynchronous distributed dataflow for ml. *Proceedings of Machine Learning and Systems*, 4:430–449, 2022. 1

[12] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020. 2, 9

[13] Charlie Chen, Sebastian Borgeaud, Geoffrey Irving, Jean-Baptiste Lespiau, Laurent Sifre, and John Jumper. Accelerating large language model decoding with speculative sampling, 2023. 12

[14] Lequn Chen, Zihao Ye, Yongji Wu, Danyang Zhuo, Luis Ceze, and Arvind Krishnamurthy. Punica: Multi-tenant lora serving. *Proceedings of Machine Learning and Systems*, 6:1–13, 2024. 1, 2, 12

[15] Zhenqian Chen, Xinkui Zhao, Chen Zhi, and Jianwei Yin. Deepboot: Dynamic scheduling system for training and inference deep learning tasks in gpu cluster. *IEEE Transactions on Parallel and Distributed Systems*, 2023. 1

[16] Seungbeom Choi, Sunho Lee, Yeonjae Kim, Jongse Park, Youngjin Kwon, and Jaehyuk Huh. Serving heterogeneous machine learning models on {Multi-GPU} servers with {Spatio-Temporal} sharing. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 199–216, 2022. 2, 4, 12

[17] Yujeong Choi and Minsoo Rhu. Prema: A predictive multi-task scheduling algorithm for preemptible neural processing units. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 220–233. IEEE, 2020. 12

[18] Patrick H. Coppock, Brian Zhang, Eliot H. Solomon, Vasilis Kypriotis, Leon Yang, Bikash Sharma, Dan Schatzberg, Todd C. Mowry, and Dimitrios Skarlatos. Lithos: An operating system for efficient machine learning on gpus, 2025. 1

[19] Daniel Crankshaw, Xin Wang, Guilio Zhou, Michael J Franklin, Joseph E Gonzalez, and Ion Stoica. Clipper: A {Low-Latency} online prediction serving system. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 613–627, 2017. 12

[20] Databricks. Finetuning api in databricks mosaic ai training. https://docs.mosaicml.com/projects/mcli/en/latest/finetuning/finetuning.html, 2024. 1

[21] DeepSpeed Team. DeepSpeed-MII, 2022. GitHub repository. 9

[22] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018. 2

[23] Aditya Dhakal, Sameer G Kulkarni, and KK Ramakrishnan. Gslice: controlled spatial sharing of gpus for a scalable inference platform. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, pages 492–506, 2020. 4, 12

[24] Ning Ding, Yujia Qin, Guang Yang, Fuchao Wei, Zonghan Yang, Yusheng Su, Shengding Hu, Yulin Chen, Chi-Min Chan, Weize Chen, et al. Parameter-efficient fine-tuning of large-scale pre-trained language models. *Nature Machine Intelligence*, 5(3):220–235, 2023. 2

[25] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, et al. An image is worth 16x16 words: Transformers for image recognition at scale. *arXiv preprint arXiv:2010.11929*, 2020. 1

[26] Qiankun Gao, Chen Zhao, Yifan Sun, Teng Xi, Gang Zhang, Bernard Ghanem, and Jian Zhang. A unified continual learning framework with general parameter-efficient tuning. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 11483–11493, 2023. 1

[27] Wei Gao, Qinghao Hu, Zhisheng Ye, Peng Sun, Xiaolin Wang, Yingwei Luo, Tianwei Zhang, and Yonggang Wen. Deep learning workload scheduling in gpu datacenters: Taxonomy, challenges and vision. *arXiv preprint arXiv:2205.11913*, 2022. 1

[28] Google. Fine-tuning with the gemini api. https://ai.google.dev/gemini-api/docs/model-tuning, 2024. 1

[29] Aaron Grattafiori et al. The llama 3 herd of models, 2024. 1

[30] Arpan Gujarati, Reza Karimi, Safya Alzayat, Wei Hao, Antoine Kaufmann, Ymir Vigfusson, and Jonathan Mace. Serving DNNs like clockwork: Performance predictability from the bottom up. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 443–462. USENIX Association, November 2020. 11, 12

[31] Jashwant Raj Gunasekaran, Cyan Subhra Mishra, Prashanth Thinakaran, Bikash Sharma, Mahmut Taylan Kandemir, and Chita R Das. Cocktail: A multidimensional optimization for model serving in cloud. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 1041–1057, 2022. 12

[32] Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025. 1, 12

[33] Junxian He, Chunting Zhou, Xuezhe Ma, Taylor Berg-Kirkpatrick, and Graham Neubig. Towards a unified view of parameter-efficient transfer learning. *arXiv preprint arXiv:2110.04366*, 2021. 1, 2

[34] Robin J Hogan. Fast reverse-mode automatic differentiation using expression templates in c++. *ACM Transactions on Mathematical Software (TOMS)*, 40(4):1–16, 2014. 6

[35] Neil Houlsby, Andrei Giurgiu, Stanislaw Jastrzebski, Bruna Morrone, Quentin De Laroussilhe, Andrea Gesmundo, Mona Attariyan, and Sylvain Gelly. Parameter-efficient transfer learning for nlp. In *International Conference on Machine Learning*, pages 2790–2799. PMLR, 2019. 1

[36] 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. *arXiv preprint arXiv:2106.09685*, 2021. 1, 2

[37] Nikoleta Iliakopoulou, Jovan Stojkovic, Chloe Alverti, Tianyin Xu, Hubertus Franke, and Josep Torrellas. Chameleon: Adaptive caching and scheduling for many-adapter llm inference environments. *arXiv preprint arXiv:2411.17741*, 2024. 1

[38] Sam Ade Jacobs, Masahiro Tanaka, Chengming Zhang, Minjia Zhang, Shuaiwen Leon Song, Samyam Rajbhandari, and Yuxiong He. Deepspeed ulysses: System optimizations for enabling training of extreme long sequence transformer models, 2023. 7

[39] Paras Jain, Ajay Jain, Aniruddha Nrusimha, Amir Gholami, Pieter Abbeel, Joseph Gonzalez, Kurt Keutzer, and Ion Stoica. Checkmate: Breaking the memory wall with optimal tensor rematerialization. In *Proceedings of Machine Learning and Systems 2020*, pages 497–511. 2020. 6

[40] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014. 9

[41] Vijay Korthikanti, Jared Casper, Sangkug Lym, Lawrence McAfee, Michael Andersch, Mohammad Shoeybi, and Bryan Catanzaro. Reducing activation recomputation in large transformer models, 2022. 7

[42] Abhishek Vijaya Kumar, Gianni Antichi, and Rachee Singh. Aqua: Network-accelerated memory offloading for llms in scale-up gpu domains, 2025. 12

[43] 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*, SOSP '23, page 611–626, New York, NY, USA, 2023. Association for Computing Machinery. 9, 12

[44] Yunseong Lee, Alberto Scolari, Byung-Gon Chun, Marco Domenico Santambrogio, Markus Weimer, and Matteo Interlandi. {PRETZEL}: Opening the black box of machine learning prediction serving systems. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 611–626, 2018. 12

[45] Brian Lester, Rami Al-Rfou, and Noah Constant. The power of scale for parameter-efficient prompt tuning. *arXiv preprint arXiv:2104.08691*, 2021. 1, 2

[46] Jiamin Li, Hong Xu, Yibo Zhu, Zherui Liu, Chuanxiong Guo, and Cong Wang. Lyra: Elastic scheduling for deep learning clusters. In *Proceedings of the Eighteenth European Conference on Computer Systems*, pages 835–850, 2023. 12

[47] Xiang Lisa Li and Percy Liang. Prefix-tuning: Optimizing continuous prompts for generation. *arXiv preprint arXiv:2101.00190*, 2021. 1, 2

[48] Xuying Li, Zhuo Li, Yuji Kosuga, and Victor Bian. Optimizing safe and aligned language generation: A multi-objective grpo approach, 2025. 12

[49] Zhuohan Li, Lianmin Zheng, Yinmin Zhong, Vincent Liu, Ying Sheng, Xin Jin, Yanping Huang, Zhifeng Chen, Hao Zhang, Joseph E Gonzalez, et al. {AlpaServe}: Statistical multiplexing with model parallelism for deep learning serving. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 663–679, 2023. 12

[50] Zikun Li, Zhuofu Chen, Remi Delacourt, Gabriele Oliaro, Zeyu Wang, Qinghan Chen, Shuhuai Lin, April Yang, Zhihao Zhang, Zhuoming Chen, Sean Lai, Xinhao Cheng, Xupeng Miao, and Zhihao Jia. Adaserve: Accelerating multi-slo llm serving with slo-customized speculative decoding, 2025. 12

[51] Haokun Liu, Derek Tam, Mohammed Muqeeth, Jay Mohta, Tenghao Huang, Mohit Bansal, and Colin A Raffel. Few-shot parameter-efficient fine-tuning is better and cheaper than in-context learning. *Advances in Neural Information Processing Systems*, 35:1950–1965, 2022. 2

[52] Jiachang Liu, Dinghan Shen, Yizhe Zhang, Bill Dolan, Lawrence Carin, and Weizhu Chen. What makes good in-context examples for gpt-3? *arXiv preprint arXiv:2101.06804*, 2021. 1

[53] Xiao Liu, Yanan Zheng, Zhengxiao Du, Ming Ding, Yujie Qian, Zhilin Yang, and Jie Tang. Gpt understands, too. *AI Open*, 2023. 2

[54] Xiaoxuan Liu, Jongseok Park, Langxiang Hu, Woosuk Kwon, Zhuohan Li, Chen Zhang, Kuntai Du, Xiangxi Mo, Kaichao You, Alvin Cheung, Zhijie Deng, Ion Stoica, and Hao Zhang. Turbospec: Closed-loop speculation control system for optimizing llm serving goodput, 2025. 12

[55] Xinyue Liu, Harshita Diddee, and Daphne Ippolito. Customizing large language model generation style using parameter-efficient finetuning. In *Proceedings of the 17th International Natural Language Generation Conference*, pages 412–426, 2024. 1

[56] Sourab Mangrulkar, Sylvain Gugger, Lysandre Debut, Younes Belkada, Sayak Paul, and Benjamin Bossan. Peft: State-of-the-art parameter-efficient fine-tuning methods. https://github.com/huggingface/peft, 2022. 9

[57] Yuning Mao, Lambert Mathias, Rui Hou, Amjad Almahairi, Hao Ma, Jiawei Han, Wen-tau Yih, and Madian Khabsa. Unipelt: A unified framework for parameter-efficient language model tuning. *arXiv preprint arXiv:2110.07577*, 2021. 2

[58] Meta AI. The llama 4 herd: Natively multimodal ai innovation, April 2025. Accessed: September 2025. 1

[59] Xupeng Miao, Gabriele Oliaro, Zhihao Zhang, Xinhao Cheng, Zeyu Wang, Rae Ying Yee Wong, Alan Zhu, Lijie Yang, Xiaoxiang Shi, Chunan Shi, Zhuoming Chen, Daiyaan Arfeen, Reyna Abhyankar, and Zhihao Jia. Specinfer: Accelerating generative large language model serving with speculative inference and token tree verification, 2023. 9, 12

[60] Xupeng Miao, Chunan Shi, Jiangfei Duan, Xiaoli Xi, Dahua Lin, Bin Cui, and Zhihao Jia. Spotserve: Serving generative large language models on preemptible instances. *ASPLOS*, 2024. 11

[61] Deepak Narayanan, Keshav Santhanam, Peter Henderson, Rishi Bommasani, Tony Lee, and Percy Liang. Cheaply estimating inference efficiency metrics for autoregressive transformer models. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023. 8

[62] Xiaonan Nie, Yi Liu, Fangcheng Fu, Jinbao Xue, Dian Jiao, Xupeng Miao, Yangyu Tao, and Bin Cui. Angel-ptm: A scalable and economical large-scale pre-training system in tencent. *Proceedings of the VLDB Endowment*, 16(12):3781–3794, 2023. 1

[63] NovaSky-AI. Sky-T1-32B-Preview. https://huggingface.co/NovaSky-AI/Sky-T1-32B-Preview, 2025. 9

[64] NovaSky-AI. Sky-T1_data_17k. https://huggingface.co/datasets/NovaSky-AI/Sky-T1_data_17k, 2025. 9

[65] NVIDIA Corporation. TensorRT-LLM, 2023. GitHub repository. 9

[66] Gabriele Oliaro, Zhihao Jia, Daniel Campos, and Aurick Qiao. Suffixdecoding: Extreme speculative decoding for emerging ai applications, 2025. 12

[67] OpenAI. GPT-4 technical report. *CoRR*, abs/2303.08774, 2023. 1

[68] OpenAI. Fine-tuning now available for gpt-4o. https://openai.com/index/gpt-4o-fine-tuning/, 2024. 1

[69] OpenAI. Learning to reason with llms, 2024. Accessed: 2024-09-12. 9

[70] OpenAI. Gpt-5 system card. Technical report, OpenAI, August 2025. Accessed: September 2025. 1

[71] Jonas Pfeiffer, Aishwarya Kamath, Andreas Rücklé, Kyunghyun Cho, and Iryna Gurevych. Adapterfusion: Non-destructive task composition for transfer learning. *arXiv preprint arXiv:2005.00247*, 2020. 1

[72] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019. 2

[73] Francisco Romero, Qian Li, Neeraja J Yadwadkar, and Christos Kozyrakis. {INFaaS}: Automated model-less inference serving. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 397–411, 2021. 2, 12

[74] Haichen Shen, Lequn Chen, Yuchen Jin, Liangyu Zhao, Bingyu Kong, Matthai Philipose, Arvind Krishnamurthy, and Ravi Sundaram. Nexus: A gpu cluster engine for accelerating dnn-based video analysis. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 322–337, 2019. 12

[75] Guangming Sheng, Chi Zhang, Zilingfeng Ye, Xibin Wu, Wang Zhang, Ru Zhang, Yanghua Peng, Haibin Lin, and Chuan Wu. Hybridflow: A flexible and efficient rlhf framework. page 1279–1297, 2025. 12

[76] Ying Sheng, Shiyi Cao, Dacheng Li, Coleman Hooper, Nicholas Lee, Shuo Yang, Christopher Chou, Banghua Zhu, Lianmin Zheng, Kurt Keutzer, et al. Slora: Scalable serving of thousands of lora adapters. *Proceedings of Machine Learning and Systems*, 6:296–311, 2024. 1, 2, 12

[77] Ying Sheng, Shiyi Cao, Dacheng Li, Banghua Zhu, Zhuohan Li, Danyang Zhuo, Joseph E. Gonzalez, and Ion Stoica. Fairness in serving large language models, 2024. 19, 20

[78] Nisan Stiennon, Long Ouyang, Jeff Wu, Daniel M. Ziegler, Ryan Lowe, Chelsea Voss, Alec Radford, Dario Amodei, and Paul Christiano. Learning to summarize from human feedback, 2022. 12

[79] Nisan Stiennon, Long Ouyang, Jeffrey Wu, Daniel Ziegler, Ryan Lowe, Chelsea Voss, Alec Radford, Dario Amodei, and Paul F Christiano. Learning to summarize with human feedback. *Advances in Neural Information Processing Systems*, 33:3008–3021, 2020. 1

[80] Yifan Tan, Cheng Tan, Zeyu Mi, and Haibo Chen. Pipellm: Fast and confidential large language model services with speculative pipelined encryption. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, ASPLOS '25, page 843–857, New York, NY, USA, 2025. Association for Computing Machinery. 9

[81] Zhaoxuan Tan, Qingkai Zeng, Yijun Tian, Zheyuan Liu, Bing Yin, and Meng Jiang. Democratizing large language models via personalized parameter-efficient fine-tuning. *arXiv preprint arXiv:2402.04401*, 2024. 1

[82] MosaicML NLP Team. Introducing mpt-7b: A new standard for open-source, commercially usable llms, 2023. Accessed: 2023-05-05. 9

[83] ShareGPT Team, 2023. 9

[84] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023. 1, 9

[85] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. Llama 2: Open foundation and fine-tuned chat models, 2023. 9

[86] Colin Unger, Zhihao Jia, Wei Wu, Sina Lin, Mandeep Baines, Carlos Efrain Quintero Narvaez, Vinay Ramakrishnaiah, Nirmal Prajapati, Patrick S. McCormick, Jamaludin Mohd-Yusof, Xi Luo, Dheevatsa Mudigere, Jongsoo Park, Misha Smelyanskiy, and Alex Aiken. Unity: Accelerating DNN training through joint optimization of algebraic transformations and parallelization. In *16th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2022, Carlsbad, CA, USA, July 11-13, 2022*, pages 267–284. USENIX Association, 2022. 5

[87] Yuxin Wang, Yuhan Chen, Zeyu Li, Xueze Kang, Zhenheng Tang, Xin He, Rui Guo, Xin Wang, Qiang Wang, Amelie Chi Zhou, and Xiaowen Chu. Burstgpt: A real-world workload dataset to optimize llm serving systems, 2024. 9

[88] Yuxin Wang, Yuhan Chen, Zeyu Li, Zhenheng Tang, Rui Guo, Xin Wang, Qiang Wang, Amelie Chi Zhou, and Xiaowen Chu. Towards efficient and reliable llm serving: A real-world workload study. *arXiv preprint arXiv:2401.17644*, 2024. 1

[89] Qizhen Weng, Wencong Xiao, Yinghao Yu, Wei Wang, Cheng Wang, Jian He, Yong Li, Liping Zhang, Wei Lin, and Yu Ding. {MLaaS} in the wild: Workload analysis and scheduling in {Large-Scale} heterogeneous {GPU} clusters. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 945–960, 2022. 1

[90] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. Transformers: State-of-the-art machine learning for pytorch, tensorflow, and jax. https://github.com/huggingface/transformers, 2022. 9

[91] Bingyang Wu, Ruidong Zhu, Zili Zhang, Peng Sun, Xuanzhe Liu, and Xin Jin. {dLoRA}: Dynamically orchestrating requests and adapters for {LoRA}{LLM} serving. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 911–927, 2024. 1, 2

[92] Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, Zhenhua Han, Pratyush Patel, Xuan Peng, Hanyu Zhao, Quanlu Zhang, et al. Gandiva: Introspective cluster scheduling for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 595–610, 2018. 2, 3, 12

[93] Wencong Xiao, Shiru Ren, Yong Li, Yang Zhang, Pengyang Hou, Zhi Li, Yihui Feng, Wei Lin, and Yangqing Jia. {AntMan}: Dynamic scaling on {GPU} clusters for deep learning. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 533–548, 2020. 3, 12

[94] An Yang et al. Qwen2.5 technical report, 2025. 1, 9

[95] An Yang et al. Qwen3 technical report, 2025. 1

[96] Fuxun Yu, Di Wang, Longfei Shangguan, Minjia Zhang, Chenchen Liu, and Xiang Chen. A survey of multi-tenant deep learning inference on gpu. *arXiv preprint arXiv:2203.09040*, 2022. 4

[97] 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)*, pages 521–538, 2022. 7, 8

[98] Qiying Yu, Zheng Zhang, Ruofei Zhu, Yufeng Yuan, Xiaochen Zuo, Yu Yue, Tiantian Fan, Gaohong Liu, Lingjun Liu, Xin Liu, Haibin Lin, Zhiqi Lin, Bole Ma, Guangming Sheng, Yuxuan Tong, Chi Zhang, Mofan Zhang, Wang Zhang, Hang Zhu, Jinhua Zhu, Jiaze Chen, Jiangjie Chen, Chengyi Wang, Hongli Yu, Weinan Dai, Yuxuan Song, Xiangpeng Wei, Hao Zhou, Jingjing Liu, Wei-Ying Ma, Ya-Qin Zhang, Lin Yan, Mu Qiao, Yonghui Wu, and Mingxuan Wang. Dapo: An open-source llm reinforcement learning system at scale, 2025. 12

[99] Shaoxun Zeng, Minhui Xie, Shiwei Gao, Youmin Chen, and Youyou Lu. Medusa: Accelerating serverless llm inference with materialization. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, ASPLOS '25, page 653–668, New York, NY, USA, 2025. Association for Computing Machinery. 9

[100] Biao Zhang, Ankur Bapna, Rico Sennrich, and Orhan Firat. Share or not? learning to schedule language-specific capacity for multilingual translation. In *Ninth International Conference on Learning Representations 2021*, 2021. 1

[101] Chengliang Zhang, Minchen Yu, Wei Wang, and Feng Yan. {MArk}: Exploiting cloud services for {Cost-Effective},{SLO-Aware} machine learning inference serving. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 1049–1062, 2019. 12

[102] Haoyu Zhang, Jianjun Xu, and Ji Wang. Pretraining-based natural language generation for text summarization, 2019. 1

[103] Hong Zhang, Yupeng Tang, Anurag Khandelwal, and Ion Stoica. SHEPHERD: Serving DNNs in the wild. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 787–808, Boston, MA, April 2023. USENIX Association. 11, 12

[104] Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, et al. Opt: Open pre-trained transformer language models. *arXiv preprint arXiv:2205.01068*, 2022. 9

[105] Justin Zhao, Timothy Wang, Wael Abid, Geoffrey Angus, Arnav Garg, Jeffery Kinnison, Alex Sherstinsky, Piero Molino, Travis Addair, and Devvret Rishi. Lora land: 310 fine-tuned llms that rival gpt-4, a technical report. *arXiv preprint arXiv:2405.00732*, 2024. 1

[106] Yihao Zhao, Xin Liu, Shufan Liu, Xiang Li, Yibo Zhu, Gang Huang, Xuanzhe Liu, and Xin Jin. Muxflow: Efficient and safe gpu sharing in large-scale production deep learning clusters. *arXiv preprint arXiv:2303.13803*, 2023. 4

[107] Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Joseph E. Gonzalez, and Ion Stoica. Alpa: Automating inter- and intra-operator parallelism for distributed deep learning. *CoRR*, abs/2201.12023, 2022. 5

[108] 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. In A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang, editors, *Advances in Neural Information Processing Systems*, volume 37, pages 62557–62583. Curran Associates, Inc., 2024. 12

[109] Yaowei Zheng, Richong Zhang, Junhao Zhang, Yanhan Ye, Zheyan Luo, Zhangchi Feng, and Yongqiang Ma. Llamafactory: Unified efficient fine-tuning of 100+ language models. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 3: System Demonstrations)*, Bangkok, Thailand, 2024. Association for Computational Linguistics. 9

[110] 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, 2024. 9

[111] Han Zhou, Xingchen Wan, Ivan Vulić, and Anna Korhonen. Autopeft: Automatic configuration search for parameter-efficient fine-tuning. *arXiv preprint arXiv:2301.12132*, 2023. 2

[112] Zhe Zhou, Xuechao Wei, Jiejing Zhang, and Guangyu Sun. {PetS}: A unified framework for {Parameter-Efficient} transformers serving. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 489–504, 2022. 1, 2

[113] Kan Zhu, Yilong Zhao, Liangyu Zhao, Gefei Zuo, Yile Gu, Dedong Xie, Yufei Gao, Qinyu Xu, Tian Tang, Zihao Ye, Keisuke Kamahori, Chien-Yu Lin, Stephanie Wang, Arvind Krishnamurthy, and Baris Kasikci. Nanoflow: Towards optimal large language model serving throughput, 2024. 9

[114] Barret Zoph and Quoc V Le. Neural architecture search with reinforcement learning. *arXiv preprint arXiv:1611.01578*, 2016. 2

Algorithm 3 Dynamic Temporal Sharing

Require: q, b, a, c
$Q[]$, $B[]$, r_a, r_c
s, f_p, d
procedure SCHEDULER_STEP(q, b, a, c)
    $r_a \leftarrow r_a + a$
    $r_c \leftarrow r_c + c$
    Q.append(q)
    B.append(b)
    $s \leftarrow s - 1$
    if $s \leq 0$ **then**
        $d \leftarrow d + 1$
        if $d \geq 3$ **then**
            $s \leftarrow$ COMPUTE_NEXT_INTERVAL
            $d \leftarrow 0$
        else
            $s \leftarrow \min(512, f_p \times 1.1)$
        RESET_STATS
        return True ▷ switch to finetuning
    return False ▷ continue inference
procedure COMPUTE_NEXT_INTERVAL
    if Q.empty() **then**
        return 64
    $\bar{q} \leftarrow \text{mean}(Q)$
    $q_{max} \leftarrow \max(Q)$
    $\bar{b} \leftarrow \text{mean}(B)$
    $\lambda \leftarrow r_a / |Q|$
    $\mu \leftarrow r_c / |Q|$
    $p_q \leftarrow \min(1.0, \bar{q}/20.0)$
    $p_s \leftarrow \min(0.5, q_{max}/25.0)$
    $p_b \leftarrow \max(0.0, (\lambda - \mu)/8.0)$
    $p \leftarrow p_q + p_s + p_b$
    if $p \leq 0.8$ **then**
        $f \leftarrow 64$
    else if $p \geq 2.0$ **then**
        $f \leftarrow 512$
    else
        $p_n \leftarrow (p - 0.8)/1.2$
        $f \leftarrow 64 + p_n \times 0.6 \times (512 - 64)$
    $f \leftarrow f \times 1.35$ ▷ stabilization adjustment
    $f_s \leftarrow (f + 2 \times f_p)/3$
    $f_p \leftarrow f_s$
    $f_s \leftarrow \max(f_s, 64 + 16)$
    return clamp(f_s, 64, 512)

A Dynamic Temporal Sharing

Dynamic Temporal Sharing (DTS) is an adaptive baseline that dynamically adjusts the frequency of switching between inference and finetuning phases based on real-time system conditions. The algorithm (Appendix A) continuously monitors queue lengths, batch sizes, arrival rates, and completion rates to compute a multi-dimensional pressure metric comprising queue pressure (avg_queue/20.0), spike pressure (max_queue/25.0), and backlog pressure ((arrival_rate − completion_rate)/8.0). Based on the total pressure, DTS adaptively selects finetuning intervals between 64-512 steps using pressure thresholds (pressure ≤ 0.8 for minimum frequency, pressure ≥ 2.0 for maximum frequency) and linear interpolation with a $0.6\times$ scaling factor for intermediate values. To ensure system stability, the algorithm incorporates hysteresis through weighted historical averaging ((frequency $+ 2 \times$ prev_frequency)/3), applies stabilization adjustments to computed frequencies, and implements decision delays where frequency recomputation occurs only every third scheduling decision to prevent oscillatory behavior under varying workloads.

B Coserving Eviction Rates

Table 1 shows the percentage of inference requests experiencing a KV cache eviction during the evaluation of co-serving under the end-to-end experiment described in Section 8. Overall, the eviction rates are negliglible. The eviction rate for the Qwen2.5-32B (TP=4) model is 0.29% for when the inference requests arrive at a rate of 16req/s, and reaches a maximum of 1.20% when the arrival rate is 20req/s. In all other scenarios, no evictions occurred.

Table 1: Percentage of requests experiencing an eviction

Model	QPS=4	QPS=8	QPS=12	QPS=16	QPS=20
Llama-3.1-8B	0.00%	0.00%	0.00%	0.00%	0.00%
Qwen2.5-14B	0.00%	0.00%	0.00%	0.00%	0.00%
Qwen2.5-32B	0.00%	0.00%	0.00%	0.29%	1.20%

C Preventing Tenant Interference with Virtual Token Counter

FlexLLM can prevent tenant interference (e.g., noisy neighbor problems) thanks to the Virtual Token Counter (VTC) algorithm [77], which can be combined with our co-serving scheduler. This integration ensures fair resource allocation across tenants while maintaining high GPU utilization.

In multi-tenant PEFT serving, tenants may submit requests at vastly different rates. Without fairness controls, aggressive tenants can monopolize resources and degrade service for others. VTC addresses this by tracking per-tenant service consumption and prioritizing underserved tenants.

Algorithm 4 shows the integration of VTC into FlexLLM's token-level scheduling. Key modifications include: (1) per-tenant virtual counters c_i tracking cumulative service, (2) counter lifting when idle tenants rejoin to prevent unfair credit accumulation, (3) fair selection of both inference and finetuning tokens based on minimum counter values, and (4) weighted counter updates reflecting different costs of input (w_p), output (w_q), and finetuning (w_r) tokens.

The integration guarantees bounded fairness across tenants. In the analysis, finetuning requests are treated as a special

Algorithm 4 FlexLLM with Virtual Token Counter for Fair Co-Serving

Input: Input weight w_p, output weight w_q, finetuning weight w_r, max tokens M
Initialize: $B \leftarrow \emptyset$, $c_i \leftarrow 0$ for all tenant i, $Q \leftarrow \emptyset$

▷ Monitoring Stream
while True **do**
  if request r from tenant u arrives **then**
    if $\nexists r' \in Q, tenant(r') = u$ **then**
      if $Q = \emptyset$ **then**
        $l \leftarrow$ the last tenant left Q
        $c_u \leftarrow \max\{c_u, c_l\}$
      else
        $c_u \leftarrow \max\{c_u, \min\{c_i \mid i \in Q\}\}$
    $Q_u \leftarrow Q_u \cup \{r\}$

▷ Execution Stream
while True **do**
  if can_add_new_requests() **then**
    ▷ Fair inference selection
    while memory available for inference **do**
      $k \leftarrow \arg\min_{i \in \{tenant(r) | r \in Q \text{ is an inference task}\}} c_i$
      $r \leftarrow$ the earliest inference request in Q from k
      Add inference request from Q_k to batch B
      $c_k \leftarrow c_k + w_p \cdot \text{input\_length}(r)$
    ▷ Fair finetuning token selection
    $s \leftarrow$ compute_finetuning_window()
    while $s > 0$ and memory available **do**
      $k \leftarrow \arg\min_{i \in \{tenant(r) | r \in Q \text{ is a finetuning task}\}} c_i$
      $t \leftarrow \min(s, \text{available})$
      Process next t tokens from tenant k
      $c_k \leftarrow c_k + w_r \cdot t$
      $s \leftarrow s - t$
  Execute decode step for B
  for each tenant i with tokens generated **do**
    $c_i \leftarrow c_i + w_q \cdot \text{output\_tokens}_i$

Table 2: Decision framework for FlexLLM adoption

Scenario	FlexLLM	Separate Clusters
Bursty inference + high finetuning	✓	
Consistent high inference load		✓
Minimal finetuning requirements		✓
Moderate SLOs (50-100ms TPOT)	✓	
Strict SLOs (<25ms TPOT)		✓
Cost-sensitive deployments	✓	
Operational simplicity priority		✓

case of inference requests, allowing us to directly apply results from the Virtual Token Counter analysis [77]. Let $W_i(t_1, t_2)$ denote the weighted service received by tenant i and M be the maximum number of tokens. The following lemma and theorems formalize the fairness bounds.

Lemma 1. *When $Q \neq \emptyset$,*

$$\max_{i \in Q} c_i - \min_{i \in Q} c_i \leq \max(w_p \cdot L_{input}, \max(w_q, w_r) \cdot M). \quad (1)$$

Theorem 1 (Fairness for overloaded tenants)**.** *For any tenants f and g, for any time interval $[t1, t2)$ in which f and g are backlogged, Algorithm 4 guarantees*

$$|W_f(t_1, t_2) - W_g(t_1, t_2)| \leq 2\max(w_p \cdot L_{input}, \max(w_q, w_r) \cdot M).$$

Theorem 2 (Fairness for non-overloaded tenants)**.** *Let $[t_1, t_2)$ be any time interval, $T(t, t2)$ be the total services received for all tenants during $[t, t2)$, $n(t, t2)$ be the number of tenants that have requested services during the interval, and U is the upper bound from Equation 1.*

Assume a tenant f is not backlogged at t_1 and has requested services less than $\frac{T(t,t2)}{n(t,t2)} - 5U$ during $[t_1, t_2)$. Then, all of the services requested from f during the interval $[t1, t2)$ will be dispatched.

The algorithm remains work-conserving—idle resources are allocated to active tenants—while preventing any single tenant from monopolizing the system. The overhead is minimal: $O(n)$ counters for n tenants, with updates occurring at FlexLLM's existing token-level granularity. This integration preserves FlexLLM's memory optimizations (dependent parallelization, graph pruning) while adding fairness guarantees essential for multi-tenant deployments.

D Component-wise Memory Breakdown

Figure 14 shows the component-wise memory breakdown for FlexLLM co-serving LLaMA-3.1-8B with LoRA rank 16. For finetuning memory management, FlexLLM preallocates a fixed budget sufficient for the largest supported PEFT configuration (maximum rank with all target modules enabled). This budget covers PEFT weights, gradients, optimizer momentum values, and low-rank activations. The static allocation strategy prevents memory fragmentation during dynamic co-serving while ensuring deterministic memory bounds regardless of the specific PEFT configurations being served.

E Deployment Considerations

FlexLLM excels in scenarios where its co-serving architecture can maximize resource utilization while maintaining inference performance. Table 2 provides a decision framework to help practitioners identify when FlexLLM delivers optimal benefits.

FlexLLM is particularly well-suited for: (1) inference workloads with moderate burstiness and periods of low utilization, (2) environments with substantial and ongoing finetuning demands, and (3) deployments with moderate SLO requirements (50-100ms TPOT). These conditions allow FlexLLM to fully exploit its co-serving capabilities and memory optimizations.

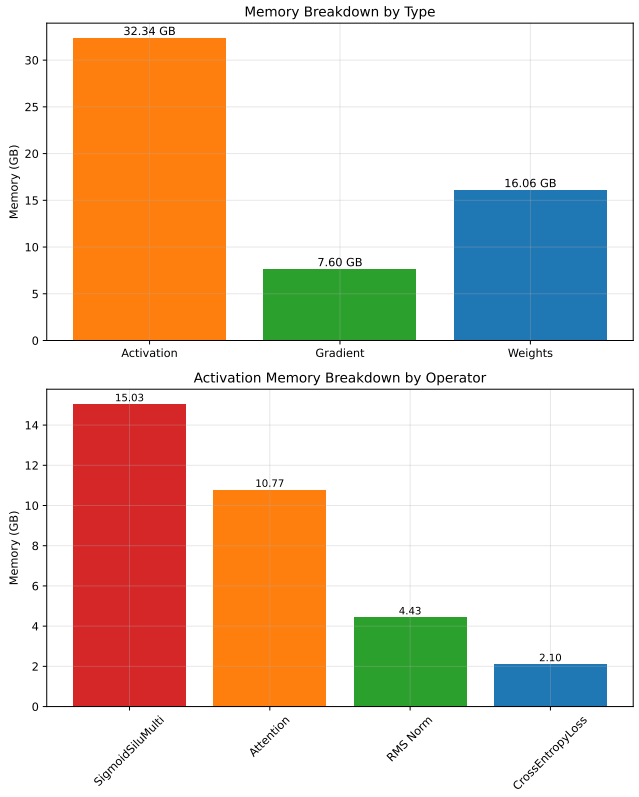


Figure 14: Component-wise memory breakdown for FlexLLM co-serving LLaMA-3.1-8B with LoRA finetuning.

FlexLLM's implementation comprises approximately 9K lines of code distributed across compilation, scheduling, memory management, and kernel integration components. The system leverages advanced techniques in GPU programming, distributed systems, and ML compiler optimizations to achieve its performance gains.

Under stricter SLO requirements, FlexLLM's performance characteristics follow predictable queueing theory principles: co-serving introduces service time variance that can affect tail latencies. This trade-off between resource efficiency and latency guarantees becomes more pronounced as SLO targets approach inherent inference latency bounds, making FlexLLM most effective in moderate SLO environments where it can fully utilize its resource optimization capabilities.

F Artifact

The artifact is available at https://github.com/FlexLLM/artifact. Follow the instructions in the README to build FlexLLM and reproduce the experiments in this paper.