StageML: Partial Evaluation for Multi-Tenant MoE-LoRA Inference

Xavier Adettu[0009-0006-5986-7380] and Tian Zhao[0000-0001-6456-9763]*

University of Wisconsin – Milwaukee, Milwaukee, WI, 53211, USA
{xadettu,tzhao}@uwm.edu

Abstract. Multi-tenant large language model platforms increasingly serve heterogeneous Low-Rank Adaptation (LoRA) modules over shared Mixture-of-Experts (MoE) backbones. Existing serving systems treat multi-tenant LoRA inference as a pure runtime scheduling problem, ignoring the temporal boundaries between deployment, adapter loading, tenant request admission, expert routing, and token generation.
This paper introduces StageML, a partial evaluation (PE) system that exploits these temporal boundaries using a binding-time lattice. StageML precomputes adapter computations that depend only on early-stage values, leaving token-dependent routing dynamic. A memory-aware planner precomputes hot adapters under VRAM budgets. On Mixtral 8×7B with 2GB budget, StageML removes 52.34% of adapter computation from token time under a Zipfian workload (where a few popular items get most of the traffic), achieving 1.105 times speedup. When residualized plans run through an optimized fused-experts backend, the expert computation phase sees 2.806 times speedup, an upper bound for this bottleneck. The results show that PE complements low-level kernel optimization.

Keywords: binding-time analysis · partial evaluation · mixture of experts · LoRA · tensor compilation

1 Introduction

Large language model (LLM) deployment has increasingly shifted toward multi-tenant serving. A single frozen base model is shared by many users while task-specific behavior is provided through small adaptation modules. Low-Rank Adaptation (LoRA) has become a standard method for this setting because it trains compact low-rank matrices while leaving the base model frozen [7]. This reduces training and storage cost, but it creates a serving problem when many adapters must be active over the same base model.

Modern frontier models have also moved toward Mixture-of-Experts (MoE) architectures. MoE layers route each token to a sparse subset of expert networks, which increases model capacity while limiting per-token computation [3,9,15]. When LoRA is combined with MoE, adapter state may become expert-specific. The serving system must then coordinate tenant adapter selection with token

expert routing. This creates a product space over tenants, adapters, experts, ranks, layers, and tokens.

Existing multi-adapter serving systems have made important progress. Punica introduced specialized kernels for multi-tenant LoRA serving [2]. S-LoRA introduced unified paging and custom kernels to support many concurrent adapters [16]. LoRAServe showed that heterogeneous adapter rank causes significant interference under production workloads and designed a rank-aware placement and routing system [8]. InfiniLoRA showed that MoE models greatly increase LoRA memory pressure and proposed a disaggregated serving architecture [1]. Recent vLLM work further shows that fused MoE-LoRA support requires careful handling of kernel specialization and hardware-specific tuning [18].

These systems mainly treat the issue as a runtime systems problem. StageML asks a different question. Can the interaction between base model topology, adapter weights, tenant identifiers, expert routing, and token activations be expressed as a multi-stage program transformation problem? If so, a compiler can make explicit which computations are known at deployment time, adapter load time, tenant request time, routing time, or token time.

The key observation is that not all values in MoE-LoRA inference become available at the same moment. The base expert layout is static after deployment. Adapter weights and ranks are known when adapters are loaded. Tenant adapter identifiers are known when a request is admitted. Expert routing is known only after the router processes token activations. Token activations and KV cache state are dynamic during generation. Current tensor graph compilers often optimize local operators but do not expose these temporal boundaries as first-class facts for residual program generation.

StageML formalizes these temporal boundaries using a binding-time lattice. The compiler then produces residual execution plans. Some computations remain dynamic because they depend on token activations or routing. Other computations can be residualized because they depend only on base model or adapter stage values. The purpose is to remove adapter-dependent computations from token time when it is safe and profitable under memory constraints.

This paper makes the following contributions:

1. A workload-aware partial evaluation formulation for multi-tenant MoE-LoRA inference that separates base, adapter, tenant, request, routing, and token stages, making tenant reuse visible to the compiler.
2. A binding-time lattice and residualization rules that classify each adapter-expert pair for precomputation, dynamic execution, or fallback.
3. A memory-aware planning rule that selects hot adapter-expert pairs for precomputation under a VRAM budget.
4. A prototype compiler with an MoE-LoRA intermediate representation and a vLLM fused-experts bridge, evaluated on Mixtral 8×7B with controlled multi-tenant workloads.

This work connects to the logic-oriented themes of this conference through several dimensions. First, the core optimization is a *program transformation* that systematically rewrites tensor expressions into residualized forms, a classic logic

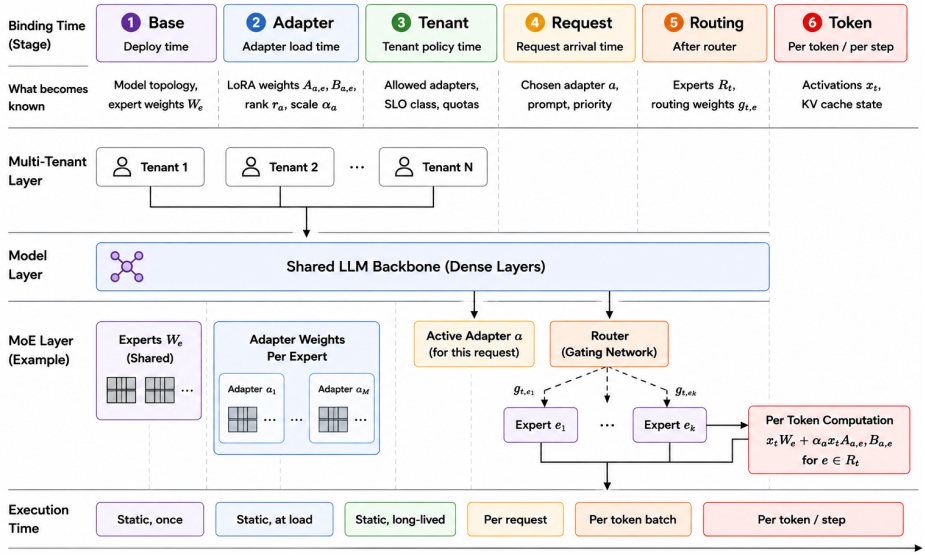

Fig. 1. StageML view of multi-tenant MoE-LoRA inference showing the six binding stages and which computations can be precomputed.

and program analysis problem. Second, the binding-time analysis and residualization rules are presented as *declarative inference rules* (Section 4.2) that can be read as logical specifications of when a transformation is sound. Third, the tensor calculus core language (Section 4.1) serves as a formal object for reasoning about program equivalence and optimization correctness, with preservation proofs in the appendix demonstrating that the transformation is sound with respect to the language semantics. This declarative, rule-based approach to compiler optimization aligns with the conference's emphasis on logic-based program analysis and transformation.

2 Background and Motivating Problem

This section introduces the serving setting that StageML targets: multi-tenant LLM serving with LoRA adapters and MoE routing. We show why their interaction naturally creates multiple binding times that existing runtime systems ignore. Figure 1 summarizes the execution structure.

2.1 Multi-Tenant LLM Serving

LLMs are often deployed as shared infrastructure. A single base model, trained at great cost, is exposed to many tenants who each want task-specific behavior. For example, a cloud provider might serve one base model to hundreds of customers, each fine-tuned for their specific application.

The challenge is that each tenant's customization must not interfere with others. Naively replicating the entire model per tenant is prohibitive: Mixtral 8×7B requires over 80GB per copy, so 100 tenants would need 8TB. Instead, production systems keep one base model and load tenant-specific *adapters* alongside it.

2.2 Low-Rank Adaptation (LoRA)

LoRA [7] is a parameter-efficient fine-tuning method that has become the standard for multi-tenant serving. Instead of modifying all weights, LoRA inserts small trainable matrices into each layer while keeping the original weights frozen.

For a dense layer with input $x \in \mathbb{R}^{1\times d}$, frozen weight $W \in \mathbb{R}^{d\times k}$, LoRA adds two low-rank matrices $A \in \mathbb{R}^{d\times r}$ and $B \in \mathbb{R}^{r\times k}$ where $r \ll d, k$:

$$y = xW + \alpha \cdot xAB$$

Here α is a scaling factor. Because r is small, storing A and B costs a fraction of storing a full copy of W.

For a single static adapter, the adapter can be *folded* into the base weight:

$$W' = W + \alpha AB$$

Then inference becomes simply $y = xW'$, with no runtime LoRA overhead.

2.3 Mixture-of-Experts (MoE)

LLMs like Mixtral [9] use MoE layers to increase capacity without proportionally increasing computation. In an MoE layer, multiple *experts* exist in parallel. For each token, a *routing mechanism* selects a sparse subset of experts and computes a weighted sum.

For experts indexed by $e \in \{1, \ldots, E\}$, each expert has weight matrix W_e. The router computes routing weight $g_{t,e}$ for token x_t. The final output is:

$$y_t = \sum_{e \in R_t} g_{t,e} \cdot (x_t W_e)$$

where R_t is the set of selected experts. Expert selection depends on the token's content and is therefore only known at generation time.

2.4 Combining LoRA with MoE

When LoRA and MoE are combined, each adapter may have expert-specific matrices $A_{a,e}$ and $B_{a,e}$. The adapted computation becomes:

$$y_{t,a} = \sum_{e \in R_t} g_{t,e} \cdot (x_t W_e + \alpha_a \cdot x_t A_{a,e} B_{a,e})$$

Different pieces of this computation become known at different times:

- Base weights W_e: deployment time
- LoRA matrices $A_{a,e}, B_{a,e}$ and scales α_a: adapter load time
- Adapter identifier a: request arrival time
- Expert set R_t and routing weights $g_{t,e}$: router execution time
- Token activation x_t: generation time

2.5 Runtime-Only Serving Is Costly

A runtime-only serving system must handle all these sources of variation at token time. For each token and selected expert, it must:

- Fetch base expert weight W_e
- Fetch adapter matrices $A_{a,e}$ and $B_{a,e}$
- Compute $x_t A_{a,e}$ (matrix-vector multiply)
- Compute $(x_t A_{a,e})B_{a,e}$ (vector-matrix multiply)
- Scale and add to $x_t W_e$

This per-token, per-expert overhead adds up. Moreover, different adapters may have different ranks r, and co-serving adapters with mismatched ranks can cause interference [8]. MoE models exacerbate the problem because each adapter may have separate matrices for each expert, multiplying memory pressure [1].

This motivates a compiler-based approach that moves computations depending only on early-stage values out of the token-time critical path.

3 Approach

StageML makes residualization decisions by constructing an execution plan from stage information, adapter metadata, workload reuse, and a memory budget. The architecture is shown in Figure 1.

The core insight is that if we could precompute the folded weight $W'_{a,e} = W_e + \alpha_a A_{a,e} B_{a,e}$ for hot adapter-expert pairs, token-time computation would reduce from $x_t W_e + \alpha_a (x_t A_{a,e}) B_{a,e}$ to $x_t W'_{a,e}$. StageML's compiler decides, based on workload observations and memory budget, which adapters to precompute and which to leave dynamic.

3.1 Compiler Input and Residual Plan Output

The input to StageML is

$$\mathcal{I} = (E, A, \Gamma, T, M).$$

Here E is the set of MoE experts and base weights W_e. A is the set of loaded adapters and their expert-specific matrices $A_{a,e}$, $B_{a,e}$, scales α_a, and ranks. Γ is the initial binding-time environment. T is a request trace used for planning. M is the precomputation memory budget.

The output is a residual plan

$$\mathcal{P} = (C, D, F, V).$$

The component C is a cache of precomputed residual expert weights. D is a request-time dispatch map from adapters and experts to execution choices. F is the dynamic fallback path. V is a validation record containing stage checks, shape checks, memory accounting, removed work estimates, and benchmark environment assertions.

The optimization boundary is deliberately narrow. StageML does not precompute the full MoE output

$$y_t = \sum_{e \in R_t} g_{t,e} x_t W'_{a,e},$$

because R_t, $g_{t,e}$, and x_t are routing or token stage values. StageML only moves the adapter-dependent construction

$$W'_{a,e} = W_e + \alpha_a A_{a,e} B_{a,e}$$

out of token time when the construction is safe and worth precomputing.

3.2 Stage Separation

StageML refines the traditional static/dynamic distinction using a six-stage binding-time lattice:

$$\mathsf{Base} \sqsubseteq \mathsf{Adapter} \sqsubseteq \mathsf{Tenant} \sqsubseteq \mathsf{Request} \sqsubseteq \mathsf{Routing} \sqsubseteq \mathsf{Token}.$$

For adapter-load-time specialization, the six stages are classified as:

$$\underbrace{\mathsf{Base} \sqsubseteq \mathsf{Adapter}}_{\text{static}} \;\Big|\; \underbrace{\mathsf{Tenant} \sqsubseteq \mathsf{Request} \sqsubseteq \mathsf{Routing} \sqsubseteq \mathsf{Token}}_{\text{dynamic}}.$$

The Base stage contains deployment-fixed expert weights and model topology. The Adapter stage contains loaded LoRA matrices, ranks, and scales. Therefore,

$$W'_{a,e} = W_e + \alpha_a A_{a,e} B_{a,e}$$

is static relative to adapter-load-time specialization and may be precomputed before request execution.

The Tenant, Request, Routing, and Token stages are dynamic relative to adapter-load-time specialization because their values are resolved after the shared adapter-level plan has been generated. They are not all dynamic in the same way: Tenant values are known at onboarding, Request values at HTTP arrival, Routing values after router execution, and Token values during generation. This refined view enables potential deeper specialization in future work, though StageML focuses on the adapter-load specialization point as a practical sweet spot.

3.3 Residual Plan Records

StageML represents each adapter-expert decision as a plan record

$$r_{a,e} = \langle a, e, \mathsf{kind}, \mathsf{stage}, \mathsf{shape}, \mathsf{bytes}, \mathsf{benefit}, \mathsf{backend} \rangle.$$

The field kind is dynamic, precomputed, or fallback. The stage field records the binding time of the residual weight expression. The shape field records the matrix shape required by the backend. The bytes field records the memory cost of precomputing the residual expert weight. The benefit field estimates dynamic LoRA work removed from token time on the planning trace. The backend field records whether execution uses the PyTorch dynamic path, the PyTorch precomputed path, or the vLLM fused-experts bridge.

3.4 Residual Plan Construction

The StageML pass constructs the residual plan in four decisions.

First, it performs a stage safety check. For adapter a and expert e, the exact residual candidate is

$$\Delta_{a,e} = \alpha_a A_{a,e} B_{a,e}, \qquad W'_{a,e} = W_e + \Delta_{a,e}.$$

The candidate is safe only when

$$\mathsf{BT}(W_e) \sqsubseteq \mathsf{Base}, \quad \mathsf{BT}(A_{a,e}), \mathsf{BT}(B_{a,e}), \mathsf{BT}(\alpha_a) \sqsubseteq \mathsf{Adapter},$$

and the matrix dimensions agree. The resulting $W'_{a,e}$ is adapter stage, not base stage, because it is specific to adapter a. This avoids the unsound interpretation that all tenants share the same folded weight.

Second, StageML estimates memory and benefit. The memory cost of precomputing adapter a is

$$\mathcal{C}(a) = \sum_e \mathrm{bytes}(W'_{a,e}).$$

Let $N_T(a, e)$ be the number of routed token-expert pairs in trace T that use adapter a and expert e. Let $\mathcal{C}_{\mathrm{lora}}(a, e)$ be the estimated dynamic cost of the adapter path $x_t A_{a,e} B_{a,e}$. The estimated benefit is

$$\mathcal{B}(a) = \sum_e N_T(a, e) \mathcal{C}_{\mathrm{lora}}(a, e).$$

This makes the decision workload-aware. Uniform traffic gives lower concentration and therefore lower removable work. Zipf and bursty traffic create hot adapters that are better candidates for precomputation.

The benefit estimate is necessary because binding-time safety alone is not enough. Many adapter-expert weights may be safe to precompute, but only a subset should be stored when GPU memory is limited.

Third, StageML chooses a memory-bounded precomputation set. The implemented planner uses the density rule

$$score(a) = \mathcal{B}(a)/\mathcal{C}(a)$$

and adds adapters in decreasing score order while the cumulative cost remains below the memory budget M. The resulting selection problem is knapsack-like; the greedy implementation is simple and provides a reasonable approximation for the memory-bounded knapsack problem in practice.

Fourth, StageML assigns each surviving candidate to a backend. A candidate may be stage-safe and shape-safe but still unsupported by a particular execution backend because of layout, precision, or kernel constraints. If the backend supports the precomputed residual weight, StageML records the PyTorch precomputed path or the vLLM fused-experts bridge in the backend field. Otherwise, the candidate is assigned to the fallback path. This separates semantic safety from backend availability.

This construction rejects candidates that are too late in the stage lattice, malformed in shape, unsupported by the backend, or too expensive for the memory budget. It precomputes only the selected residual expert weights and records fallback dispatch for all other adapters.

3.5 Runtime Use of the Residual Plan

At runtime, the concrete adapter identifier a becomes known at request admission. During router execution, the system obtains R_t and $g_{t,e}$. For each routed token-expert pair, the dispatch map chooses between the precomputed residual path and the dynamic fallback path:

$$z_{t,e} = \begin{cases} x_t C[a,e] & \text{if } D[a,e] = \mathsf{precomputed} \\ x_t W_e + \alpha_a (x_t A_{a,e}) B_{a,e} & \text{otherwise} \end{cases}$$

The final token output remains $y_t = \sum_{e \in R_t} g_{t,e} z_{t,e}$.

3.6 Example: Source and Residual Program

To illustrate the transformation, consider the following simplified StageML representation of one MoE-LoRA layer for a request using adapter a:

source(x_t, a) :
  $R_t, g_t = \mathsf{route}(x_t)$
  $y = 0$
  for $e \in R_t$ **do**
    $z = x_t W_e + \alpha_a (x_t A_{a,e}) B_{a,e}$
    $y = y + g_{t,e} z$
  return y

The binding-time environment assigns:

$$\mathsf{BT}(W_e) = \mathsf{Base}, \quad \mathsf{BT}(A_{a,e}) = \mathsf{BT}(B_{a,e}) = \mathsf{BT}(\alpha_a) = \mathsf{Adapter},$$

$$\mathsf{BT}(a) = \mathsf{Request}, \quad \mathsf{BT}(R_t) = \mathsf{BT}(g_{t,e}) = \mathsf{Routing}, \quad \mathsf{BT}(x_t) = \mathsf{Token}.$$

StageML cannot precompute the whole source program because routing and token activation are not available until later. It can only precompute the adapter-expert residual weight $W'_{a,e} = W_e + \alpha_a A_{a,e} B_{a,e}$ for selected pairs. The residual program keeps routing and token execution dynamic but replaces selected LoRA computations with cache lookups:

residual(x_t, a, C, D) :
  $R_t, g_t = \mathsf{route}(x_t)$
  $y = 0$
  for $e \in R_t$ **do**
    if $D[a, e] = \mathsf{precomputed}$ **then**
      $z = x_t C[a, e]$
    else
      $z = x_t W_e + \alpha_a (x_t A_{a,e}) B_{a,e}$
    $y = y + g_{t,e} z$
  return y

The residual program is not just a folded tensor. It contains a dispatch table D, a residual-weight cache C, and a fallback path. Routing remains dynamic because R_t and $g_{t,e}$ depend on x_t. Token application remains dynamic because every expert output still depends on the runtime activation x_t. Formal residualization rules are provided in the appendix.

4 StageML Core Language

4.1 Syntax

StageML uses a small tensor calculus to model adapter inference code e:

$$e \in c \mid x \mid W \mid A \mid B \mid e_1 \otimes e_2 \mid e_1 \oplus e_2 \mid \mathsf{fold}(e) \mid \mathsf{route}(e)$$

where c denotes a scalar constant, x denotes a runtime activation, W denotes a base expert weight, and A and B denote adapter matrices. The operator $\otimes$ denotes matrix multiplication. The operator $\oplus$ denotes elementwise addition. The form $\mathsf{fold}(e)$ requests static evaluation when it is safe. The form $\mathsf{route}(e)$ represents token-dependent expert routing.

4.2 Types and Stages

Each expression has a tensor type with a shape and binding time:

$$\Gamma \vdash e \in \mathsf{Tensor}(m, n, s)$$

where m and n denote shape and s denote binding time stages. The meaning of the stages is shown in Table 1, which explains why an adapter weight can be specialized before token execution, while routing decisions and token activations must remain dynamic.

Table 1. StageML binding times and their meaning for MoE-LoRA serving

Stage	Examples	Available at
Base	Topology and expert shapes	Deployment
Adapter	LoRA weights and ranks	Adapter load
Tenant	Allowed adapters and Service-Level-Objective class	Tenant policy
Request	Selected adapter identifier and prompt length	Request arrival
Routing	Expert identifiers and routing weights	Router execution
Token	Activation and KV cache state	Generation

The join operation $s_1 \sqcup s_2$ returns the later stage. Tensor operations propagate the latest dependency among their operands. Matrix multiplication requires the inner dimensions to agree:

$$\frac{\Gamma \vdash e_1 \in \mathsf{Tensor}(m, k, s_1) \quad \Gamma \vdash e_2 \in \mathsf{Tensor}(k, n, s_2)}{\Gamma \vdash e_1 \otimes e_2 \in \mathsf{Tensor}(m, n, s_1 \sqcup s_2)}$$

Elementwise addition requires both operands to have the same shape:

$$\frac{\Gamma \vdash e_1 \in \mathsf{Tensor}(m, n, s_1) \quad \Gamma \vdash e_2 \in \mathsf{Tensor}(m, n, s_2)}{\Gamma \vdash e_1 \oplus e_2 \in \mathsf{Tensor}(m, n, s_1 \sqcup s_2)}$$

4.3 Binding-Time Soundness

A static transformation is allowed only when the expression does not depend on future stage values. For adapter-stage residualization, the rule is

$$\frac{\Gamma \vdash e \in \mathsf{Tensor}(m, n, s) \quad s \sqsubseteq \mathsf{Adapter}}{\Gamma \vdash \mathsf{fold}(e) \in \mathsf{Tensor}(m, n, \mathsf{Adapter})}$$

This rule means that an adapter-dependent expression may be evaluated before token execution if all its inputs are known by adapter load time. The result is represented as an adapter-stage residual constant in the residual program. It is not represented as a base-stage constant, because the folded value is specific to adapter a. The shared base model remains unchanged.

5 Experimental Evaluation

5.1 Setup

Hardware. All experiments run on an NVIDIA H100 NVL GPU with 80GB memory using PyTorch 2.11.0 with CUDA 13.0.

Model. The base model is Mixtral 8×7B with 8-bit quantization, which has 8 experts per MoE layer with top-2 routing.
Adapters. Expert-specific LoRA adapters have rank 8. One adapter is trained on a code dataset; it is replicated to create 8 virtual adapters for multi-tenant simulation. The workload uses 16 tenants, each mapped to a subset of adapters.
Workload. The benchmark generates 128 requests of 16 tokens each. Three traffic patterns are evaluated:

- *Uniform*: equal probability for each tenant
- *Bursty*: one hot tenant with probability 0.5, others uniform
- *Zipf*: the arrival traffic is modeled as a Zipfian workload [12,19] with 16 tenants and skewness exponent 1.2 so that top-2 tenants receive 54.2% of the requests, top-4 receive 69.1%, and top-8 receive 85.2%.

Budget. The precomputation memory budget is swept over 2GB, 4GB, and 8GB. Each latency measurement consists of 30 runs after 5 warmup runs.
HTTP serving. For the vLLM HTTP experiment, we use vLLM's OpenAI compatible server with Llama-2-7B and 8 SQL LoRA adapters.

5.2 Request-Level Latency

The main experiment evaluates whether binding-time analysis reduces latency when tenant identities and adapter weights are known before token execution and when the request stream contains reuse.

Table 2. Comparison between dynamic runtime adapters and StageML precomputed cache with Zipfian workload and memory budgets 2GB, 4GB, and 8GB. Time is in ms.

System	Runs	Mean	Std	P50	P95	Speedup
Dynamic	30	409.504	0.805	409.286	410.862	-
StageML 2GB	30	370.734	0.998	370.550	372.431	1.105
StageML 4GB	30	348.701	0.465	348.755	349.335	1.173
StageML 8GB	30	331.321	0.705	331.233	332.556	1.227

Table 2 shows the StageML-only effect on the Zipfian workload. StageML reduces P50 latency from 409.286 ms to 370.55 ms, a 1.105× speedup. Standard deviations are small: 0.805 ms for dynamic, 0.998 ms for precomputed. In this run, StageML precomputes two adapters, using 1792 MiB of memory and removing 52.34% of adapter computation from token time. The speedup of StageML increases to 1.173× and 1.227× when the memory budget increases to 4GB and 8GB respectively.

Table 3 shows that StageML's benefit correlates with tenant reuse. For uniform traffic, where no adapter dominates, and 2GB memory budget, StageML removes only 28.91% of adapter work and yields 1.05× speedup. Bursty and Zipfian traffic create hot adapters so that StageML removes about half of adapter

Table 3. Tests of workload sensitivity of StageML tested with uniform, bursty, Zipfian workload and memory budgets 2GB, 4GB, and 8GB. Time is in ms.

Workload	Removed-work	Dynamic-P50	StageML-P50	StageML-Std	Speedup
Uniform, 2GB	28.91%	402.994	382.331	1.402	1.054
Bursty, 2GB	50.00%	406.512	369.436	0.473	1.100
Zipf, 2GB	52.34%	409.286	370.550	0.998	1.105
Uniform, 4GB	56.25%	403.678	363.091	0.840	1.112
Bursty, 4GB	81.25%	403.910	343.465	0.919	1.176
Zipf, 4GB	78.12%	409.073	348.755	0.465	1.173
Uniform, 8GB	100.00%	405.734	329.527	0.457	1.231
Bursty, 8GB	100.00%	406.180	331.693	0.395	1.225
Zipf, 8GB	100.00%	406.402	331.233	0.705	1.227

work, and yield 1.10× and 1.105× speedups respectively. This is consistent with how StageML exploits workload reuse. When memory budget is 8GB, the difference between workloads disappears since all adapters are precomputed in this case.

5.3 Composition with Optimized Kernels

The vLLM fused-experts kernel is optimized for static expert weights: it loads base model matrices W_e into fast memory and keeps them resident across token batches. Dynamic LoRA forces the kernel to handle two separate computations per token-expert pair: the base expert multiplication $x_t W_e$ and the LoRA contribution $\alpha_a(x_t A_{a,e})B_{a,e}$, which requires additional memory traffic.

StageML's precomputed cache precomputes $W'_{a,e} = W_e + \alpha_a A_{a,e} B_{a,e}$. When these residualized weights are fed into vLLM's fused-experts backend, each precomputed $W'_{a,e}$ appears as a static expert weight, which is indistinguishable from a base model expert. The kernel no longer needs separate LoRA kernels or adapter-specific memory swapping.

Table 4. Comparison of dynamic runtime adapters, StageML precomputed cache, and StageML on vLLM fused-experts backend with Zipfian workload. Time is in ms.

System	Runs	Mean	Std	P50	P95	Speedup
Dynamic runtime adapters	30	405.922	1.407	405.780	407.673	-
StageML precomputed cache	30	367.416	1.352	367.223	369.080	1.105
StageML on vLLM fused backend	30	145.069	3.107	144.631	150.535	2.806

Table 4 compares the performance of StageML with or without fused-experts backend. The dynamic baseline uses standard PyTorch operations. StageML's precomputed cache precomputes hot adapter residuals. The third row executes

the same residualized computation through vLLM's fused-experts backend. Latency drops to 145.069 ms, a 2.806× speedup over dynamic.

The 2.806× speedup exceeds what one would expect from simply removing adapter arithmetic (1.105×). The additional gain comes from memory behavior: residualized weights satisfy the kernel's assumption of static expert weights, eliminating per-adapter kernel launches and memory traffic. This speedup represents an upper bound for the expert computation phase; realistic end-to-end gains in a full serving system would be smaller due to attention, KV cache, and network overhead.

5.4 End-to-End vLLM HTTP Serving Comparison

The fused experts backend bridge isolates the expert computation phase, but it does not measure a full HTTP serving path. To evaluate StageML at the serving boundary, we also compare against vLLM's OpenAI compatible HTTP server. This experiment uses Llama-2-7B with 8 SQL LoRA adapters, bitsandbytes 4-bit loading, 512 requests, concurrency 128, maximum output length 32, and a Zipfian tenant distribution. We used the Llama-2-7B for this live HTTP comparison because the goal of this experiment is to test serving level behavior on vLLM's stable dense LoRA path, not to re-evaluate the MoE-specific planner. In our environment, the Mixtral expert layer LoRA configuration with BitsAndBytes quantization triggered a vLLM fused MoE LoRA backend initialization failure.

The baseline serves the adapters dynamically using vLLM's LoRA path. The StageML specialized configuration merges the selected adapter into the base model before serving, so the HTTP server sees a static residualized model rather than a dynamic LoRA adapter. Both configurations use the same vLLM HTTP serving path and the same client workload.

Table 5. Actual HTTP comparison between vLLM dynamic LoRA and StageML-specialized merged LoRA on Llama-2-7B with 4-bit loading. Time is in ms.

System	Adapters	Requests	Concurrency	P50	P95	Throughput	Speedup
vLLM	8	512	128	1898.772	3755.561	49.126	-
StageML	8	512	128	1265.829	2385.300	76.532	1.500

Table 5 shows that StageML specialization improves end-to-end HTTP serving latency when the adapter can be moved out of the dynamic request path. P50 latency decreases from 1898.772 ms to 1265.829 ms, giving a 1.5x speedup. Throughput improves from 49.126 to 76.532 requests per second.

6 Related Work

6.1 LoRA and Multi-Adapter Serving

LoRA introduced low-rank adaptation for efficient model customization [7]. Subsequent work extended LoRA to multi-tenant serving scenarios. Punica [2] and S-LoRA [16] developed specialized batching, paging, and custom kernels to serve many adapters concurrently over a shared base model. LoRAServe [8] identified rank heterogeneity as a source of interference and designed a rank-aware placement and routing system. InfiniLoRA [1] showed that MoE models exacerbate LoRA memory pressure and proposed disaggregated serving architecture.

StageML differs from these systems in its approach. Rather than optimizing runtime scheduling, it frames adapter serving as a staged program transformation. The compiler identifies which adapter computations depend only on early-stage values and precomputes them before token execution. Existing systems assume all adapter work must happen at runtime; StageML shows that binding-time analysis can move some of that work earlier.

6.2 MoE Serving and Fused Kernels

Mixture-of-Experts models rely on sparse expert routing to increase capacity while controlling computation cost [15,3,9]. Serving MoE models efficiently requires careful kernel design to handle dynamic routing and expert grouping. Recent work on vLLM has added fused MoE-LoRA support for multi-tenant workloads [18], focusing on kernel specialization and hardware-specific tuning.

StageML complements these kernel-level optimizations. While fused kernels make existing operations faster, StageML changes which operations are performed: pre-folding eliminates adapter computation entirely for hot adapters. The two approaches are complementary: StageML's residualized weights can be fed into existing fused kernels, as demonstrated in our vLLM bridge experiment.

6.3 Partial Evaluation and Multi-Stage Programming

Partial evaluation specializes a program with respect to known inputs, residualizing computations that depend only on static data [10]. Binding-time analysis determines which parts of a program can be evaluated early based on when values become available. Multi-stage programming systems such as MetaOCaml [17] and Lightweight Modular Staging [14] provide language support for generating residual code.

Multi-level partial evaluation generalizes the static/dynamic distinction to multiple binding times [5,6,4]. A program can be specialized multiple times as different inputs become available. The key mechanism is *multi-level binding-time analysis*, which annotates each expression with a binding-time level from a lattice. The join operation $\sqcup$ propagates the latest binding time among operands, and expressions at level k evaluated during specialization produce residual code at level $k + 1$.

StageML adapts these techniques to tensor programs and LLM inference. The six-stage binding-time lattice (Section 3.2) corresponds to concrete serving boundaries: Base (deployment), Adapter (load), Tenant (policy), Request (arrival), Routing (execution), and Token (generation). StageML specializes at the Adapter stage, evaluating Base and Adapter expressions while leaving later stages in the residual program. This choice is practical: adapter loading is a natural point to pay a one-time precomputation cost.

The residualization rules are presented declaratively as inference rules (Section 4.2) over a tensor calculus core language (Section 4.1), with preservation proofs establishing soundness. StageML departs from classic multi-level PE by combining binding-time analysis with a cost-benefit model and memory budget: not all safe-to-specialize expressions are precomputed, because GPU memory constraints may prevent storing all residual weights – a practical constraint essential for deployment.

6.4 Compiler Infrastructure

MLIR provides infrastructure for representing and lowering domain-specific compiler intermediate representations [11]. PyTorch offers graph capture and transformation mechanisms used by many machine learning compiler projects [13]. StageML uses these tools to connect residualization with practical tensor graph compilation, though a full lowering to MLIR or end-to-end integration with PyTorch's compiler is left as future work.

7 Discussion

The StageML-only benchmark with 2GB memory budget gives a modest but stable speedup of 1.054× to 1.105× depending on tenant reuse. This is expected because the optimization removes only adapter-specific work, not base expert multiplication or routing. The workload sensitivity (1.054× uniform vs. 1.105× Zipf) confirms that StageML exploits tenant reuse rather than a generic constant-factor optimization.

The vLLM bridge result (2.806×) shows that residualization becomes substantially more valuable when connected to an optimized fused-experts backend. The additional gain beyond arithmetic removal (1.105×) comes from memory behavior: residualized weights satisfy the kernel's assumption of static expert weights, eliminating per-adapter kernel launches and memory traffic. This speedup represents an upper bound for the expert computation phase; realistic end-to-end gains in a full serving system would be smaller due to attention, KV cache, and network overhead.

7.1 Limitations

StageML is *not* a production replacement for vLLM, FlashInfer, LoRAServe, or InfiniLoRA. Those systems solve serving protocol, request scheduling, batching,

KV cache management, placement, and tuned kernels. StageML solves a complementary compiler problem: given binding-time information, adapter metadata, tenant reuse, and memory budget, it decides what adapter work can be moved out of token time.

The main multi-tenant benchmark is a request-level hidden-state MoE-LoRA benchmark. It uses real Mixtral hidden states and real routing, but it is not a live HTTP server benchmark (except the Llama-2-7B experiment). The vLLM fused backend bridge calls the internal fused-experts path and demonstrates backend compatibility at the MoE execution boundary rather than end-to-end serving throughput. The multi-tenant workload uses virtual adapters derived from one real expert-specific adapter in scale-copy mode.

7.2 Practical Impact and Deployment Scenarios

The practical viability of StageML depends on three key factors.

First, workload reuse determines the benefit: StageML provides the greatest speedup when a small number of adapters receive the majority of requests, as is typical in production environments where a few fine-tuned models dominate traffic. For uniform workloads, the gains are modest ($1.05\times$) and may not justify implementation effort.

Second, the deployment model matters. StageML is most effective in systems that separate adapter loading from request processing, such as those with pre-warmed adapter pools or long-running adapters. The precomputation cost is paid once at adapter load time, so systems with stable adapter sets benefit more than those with ephemeral adapters.

Third, StageML complements existing serving infrastructure rather than replacing it. The residualized weights can be fed into vLLM, FlashInfer, or other serving engines without requiring changes to their core scheduling or kernel logic. This lowers the adoption barrier: StageML can be integrated as a pre-processing step before the serving engine receives requests.

The approach generalizes beyond MoE-LoRA to other staged computations in serving systems, such as prompt pre-computation or KV cache management, wherever temporal separation creates optimization opportunities.

8 Conclusion

StageML reframes multi-tenant MoE-LoRA inference as a workload-aware partial evaluation problem. By separating base, adapter, tenant, request, routing, and token stages, StageML identifies adapter computations that can be moved out of token time while leaving token-dependent routing dynamic. A memory-aware planner precomputes hot adapters under GPU budget. The residualization rules are presented as declarative inference rules over a tensor calculus, with preservation proofs establishing soundness.

Future work will evaluate multiple independently trained adapters, replace the greedy materialization planner with an optimizer that matches brute force

on small cases and scales to larger cases, integrate StageML decisions into a full serving stack, generate tuned Triton kernels for grouped residual experts, and expand quality evaluation to larger language modeling and downstream task suites.

The source code of this paper is available at: `https://github.com/uwm-se/stageML`.

A Meaning Preservation for Folding

We write $[\![e]\!]_\rho$ for the denotation of expression e under environment ρ, where ρ maps tensor variables and model symbols to concrete tensor values. Constants and variables are interpreted by lookup in ρ. Matrix multiplication and addition have their standard tensor meanings:

$$[\![e_1 \otimes e_2]\!]_\rho = [\![e_1]\!]_\rho \otimes [\![e_2]\!]_\rho,$$

$$[\![e_1 \oplus e_2]\!]_\rho = [\![e_1]\!]_\rho \oplus [\![e_2]\!]_\rho.$$

The form $\mathsf{fold}(e)$ evaluates e during specialization and stores the resulting tensor as a residual constant.

Lemma 1 (Fold preservation). *If $\Gamma \vdash e \in \mathsf{Tensor}(m, n, s)$, $s \sqsubseteq \mathsf{Adapter}$, and runtime environment ρ agrees with the specialization environment ρ_s on all base-stage and adapter-stage symbols used by e, then*

$$[\![\mathsf{fold}(e)]\!]_\rho = [\![e]\!]_\rho.$$

Proof. Because $s \sqsubseteq \mathsf{Adapter}$, every free value used by e is available no later than adapter-load time. Specialization therefore evaluates e under ρ_s and records the tensor value $[\![e]\!]_{\rho_s}$ as a residual constant. Since ρ agrees with ρ_s on all symbols used by e, we have $[\![e]\!]_{\rho_s} = [\![e]\!]_\rho$. The residual expression $\mathsf{fold}(e)$ denotes exactly this stored tensor. Therefore $[\![\mathsf{fold}(e)]\!]_\rho = [\![e]\!]_\rho$.

Corollary 1 (Adapter-expert residualization preservation). *Assume that*

$$W'_{a,e} = W_e + \alpha_a A_{a,e} B_{a,e}$$

is produced by folding an adapter-stage expression, and that the runtime environment agrees with the specialization environment on W_e, $A_{a,e}$, $B_{a,e}$, and α_a. Then for every token activation x_t,

$$x_t W'_{a,e} = x_t W_e + \alpha_a (x_t A_{a,e}) B_{a,e}.$$

Proof. By Lemma 1, folding preserves the value of $W_e + \alpha_a A_{a,e} B_{a,e}$, so $W'_{a,e} = W_e + \alpha_a A_{a,e} B_{a,e}$. Multiplying both sides by the same runtime activation x_t gives

$$x_t W'_{a,e} = x_t (W_e + \alpha_a A_{a,e} B_{a,e}).$$

By distributivity of matrix multiplication over addition, this is

$$x_t W_e + \alpha_a (x_t A_{a,e}) B_{a,e}.$$

B Residualization Rules

StageML performs partial evaluation over the tensor calculus by replacing foldable adapter-stage expressions with residual constants and leaving later-stage expressions in the residual program. The key rule is exact adapter-expert folding:

$$\Delta_{a,e} = \alpha_a A_{a,e} B_{a,e}, \qquad W'_{a,e} = W_e \oplus \Delta_{a,e}.$$

If

$$\mathsf{BT}(W_e) \sqsubseteq \mathsf{Base} \quad \text{and} \quad \mathsf{BT}(A_{a,e}), \mathsf{BT}(B_{a,e}), \mathsf{BT}(\alpha_a) \sqsubseteq \mathsf{Adapter},$$

and the shapes are compatible, then StageML may replace

$$x_t W_e \oplus \alpha_a (x_t A_{a,e}) B_{a,e}$$

with

$$x_t W'_{a,e}.$$

This rule is not ordinary constant folding over a closed tensor graph. The residual weight $W'_{a,e}$ is indexed by adapter and expert, selected by a workload-aware planner, and inserted into a residual dispatch program. The dispatch map D determines whether a routed token-expert pair uses the precomputed path or the fallback path.

Precomputed rule. If the adapter-expert pair (a, e) is stage-safe, shape-safe, backend-supported, and selected by the planner, StageML emits

$$D[a, e] = \mathsf{precomputed}, \qquad C[a, e] = W'_{a,e}.$$

Fallback rule. If the pair is not selected, exceeds the memory budget, violates a shape constraint, or is unsupported by the backend, StageML emits

$$D[a, e] = \mathsf{fallback}.$$

The residual program then executes the original runtime LoRA expression:

$$x_t W_e + \alpha_a (x_t A_{a,e}) B_{a,e}.$$

The fallback path is important because it lets StageML remain conservative. Unsupported adapters, rare adapters, or backend-incompatible shapes can still execute correctly without being precomputed.

Routing preservation. StageML does not residualize expressions depending on R_t, $g_{t,e}$, or x_t, because these are routing-stage or token-stage values. Therefore the residual program still computes

$$y_{t,a} = \sum_{e \in R_t} g_{t,e} z_{t,e}.$$

The optimization changes only how $z_{t,e}$ is computed for selected adapter-expert pairs.

Precision boundary. For quantized backends, exact folding is performed only when the backend exposes a compatible precision model. In the prototype, StageML treats de-quantization, accumulation precision, and re-quantization as part of the backend contract. If the backend cannot represent $W'_{a,e}$ in the required layout and precision, the pair is rejected and assigned to fallback. Thus binding-time safety and backend validity are checked separately.

Acknowledgment This work is partially supported by NMDSI SS184.

Disclosure of Interests The authors have no competing interests to declare that are relevant to the content of this article.

References

1. Chen, H., Ruan, L., Xu, Z., Li, Y., Chen, X., Leng, J., He, B., Guo, M., Sun, S.: Infinilora disaggregated multi lora serving for large language models. arXiv preprint (2026)
2. Chen, L., Ye, Z., Wu, Y., Zhuo, D., Ceze, L., Krishnamurthy, A.: Punica multi tenant lora serving. arXiv preprint (2023)
3. Fedus, W., Zoph, B., Shazeer, N.: Switch transformers scaling to trillion parameter models with simple and efficient sparsity. Journal of Machine Learning Research **23**(120) (2022)
4. Glück, R.: Efficient multi-level generating extensions for program specialization. In: Programming Languages: Implementations, Logics and Programs. Springer (1995). https://doi.org/10.1007/BFb0026825
5. Glück, R., Jørgensen, J.: Fast binding-time analysis for multi-level specialization. In: Perspectives of System Informatics. pp. 261–272. Springer (1996). https://doi.org/10.1007/3-540-62064-8_22
6. Glück, R., Jørgensen, J.: An automatic program generator for multi-level specialization. LISP and Symbolic Computation **10**(2), 113–158 (1997). https://doi.org/10.1023/A:1007763000430
7. Hu, E.J., Shen, Y., Wallis, P., Allen Zhu, Z., Li, Y., Wang, S., Wang, L., Chen, W.: Lora low rank adaptation of large language models. In: International Conference on Learning Representations (2022)
8. Jaiswal, S., Arun, S., Parayil, A., Mallick, A., Mastorakis, S., Khare, A., Alverti, C., St Amant, R., Bansal, C., Ruhle, V., Torrellas, J.: Serving heterogeneous lora adapters in distributed llm inference systems. arXiv preprint (2025)
9. Jiang, A.Q., Sablayrolles, A., Roux, A., Mensch, A., Savary, B., Bamford, C., Chaplot, D.S., de las Casas, D., Hanna, E.B., Bressand, F.: Mixtral of experts. arXiv preprint (2024)
10. Jones, N.D., Gomard, C.K., Sestoft, P.: Partial Evaluation and Automatic Program Generation. Prentice Hall (1993)
11. Lattner, C., Amini, M., Bondhugula, U., Cohen, A., Davis, A., Pienaar, J., Riddle, R., Shpeisman, T., Vasilache, N., Zinenko, O.: Mlir scaling compiler infrastructure for domain specific computation. In: International Symposium on Code Generation and Optimization (2021)
12. Newman, M.E.J.: Power laws, pareto distributions and zipf's law. Contemporary Physics **46**(5), 323–351 (2005)

13. Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L.: Pytorch an imperative style high performance deep learning library. In: Advances in Neural Information Processing Systems (2019)
14. Rompf, T., Odersky, M.: Lightweight modular staging a pragmatic approach to runtime code generation and compiled dsls. Communications of the ACM (2010)
15. Shazeer, N., Mirhoseini, A., Maziarz, K., Davis, A., Le, Q., Hinton, G., Dean, J.: Outrageously large neural networks the sparsely gated mixture of experts layer. International Conference on Learning Representations (2017)
16. Sheng, Y., Cao, S., Li, D., Hooper, C., Lee, N., Yang, S., Chou, C., Zhu, B., Zheng, L., Keutzer, K., Gonzalez, J.E., Stoica, I.: S lora serving thousands of concurrent lora adapters. arXiv preprint (2023)
17. Taha, W.: A gentle introduction to multi stage programming. In: Domain Specific Program Generation. Springer (2004)
18. vLLM Team, AI, A.: Efficient multi lora inference for moe models. vLLM Blog (2026)
19. Zipf, G.K.: Human Behavior and the Principle of Least Effort. Addison-Wesley (1949)