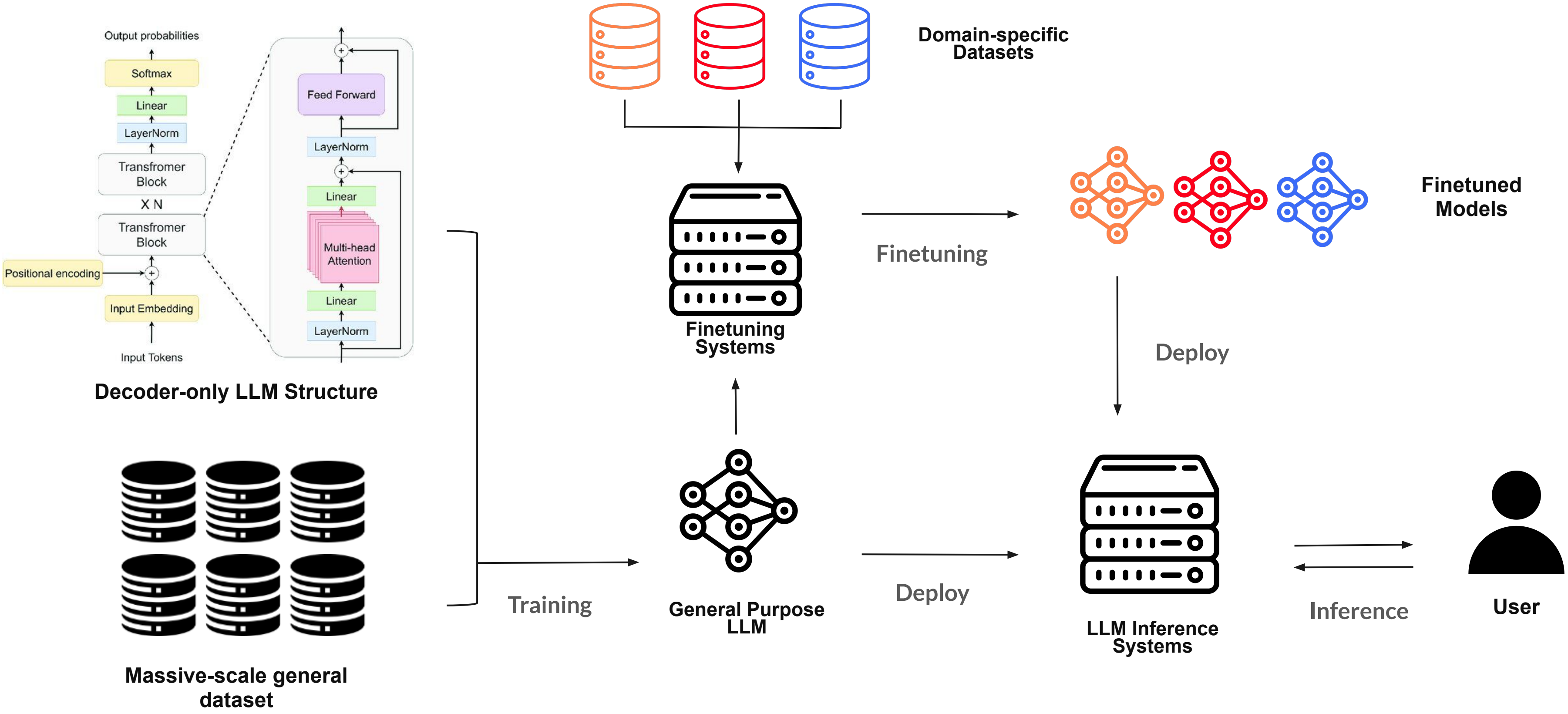
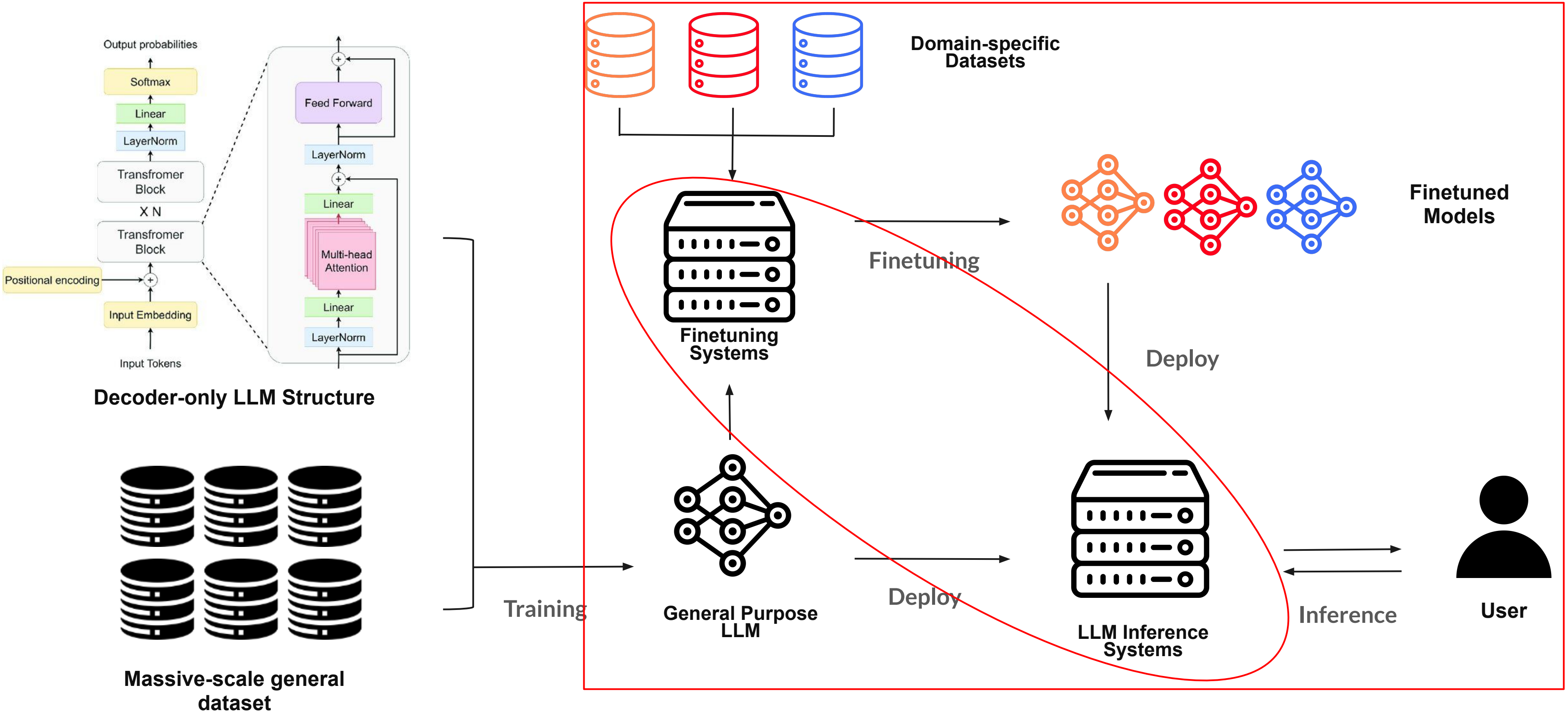
Inference and Fine-Tuning Co-serving for LoRA-Adapted LLMs

Jiaxuan Chen, Oana Balmau

LLM Lifecycle



LLM Lifecycle

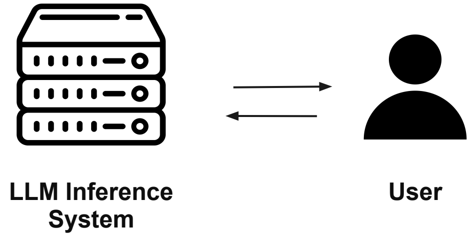


Inference & Finetuing Co-serving

Why?

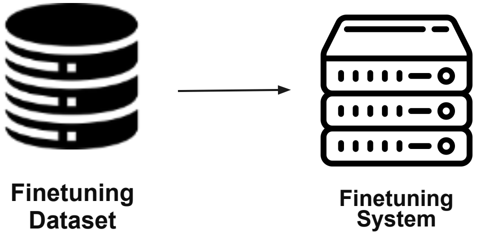
Inference:

- User-centric
- Latency-critical
- SLO compliance required
- Unpredictable traffic



Finetuning:

- Data-driven
- More latency-tolerant
- Throughput & accuracy prioritized
- Steady, predictable workload



Inference & Finetuing Co-serving

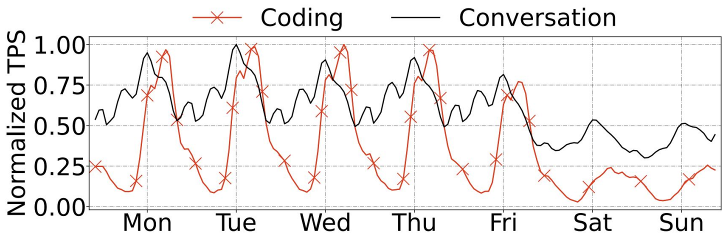
Why?

Inference:

- User-centric
- Latency-critical
- SLO compliance required
- Unpredictable traffic

Finetuning:

- Data-driven
- More latency-tolerant
- Throughput & accuracy prioritized
- Steady, predictable workload



Load over a week for Coding and Conversation LLM **inference workloads** (1)

GPU Underutilization at low traffic period

→ Use free cycles for finetuning?

(1) Jovan Stojkovic, Chaojie Zhang, Íñigo Goiri, Josep Torrellas, & Esha Choukse. (2024). DynamoLLM: Designing LLM Inference Clusters for Performance and Energy Efficiency.

Project Goal: Inference & Finetuing Co-serving

Objectives:

- **A unified, scalable runtime** that co-serves inference and fine-tuning on the same cluster
- Fine-tuning is **low-overhead and transparent** to users
- Maintains inference performance **on par with dedicated inference-only systems**
- **Maximize GPU utilization** by scheduling finetuning during inference idle periods

Project Goal: Inference & Finetuing Co-serving

How to co-serve inference & finetuning?

- Shared forward pass in inference & finetuning

  Inference uses iterative **forward pass**

  Finetuning requires one **forward pass** and one backward pass

  → **Can we use the same forward pass for inference and finetuning forward?**

- But finetuning updates parameters, inference does not.

  → Finetuning weight update should not interfere inference

  → **Can we keep the update separate from the base model?** LoRA Adapter!

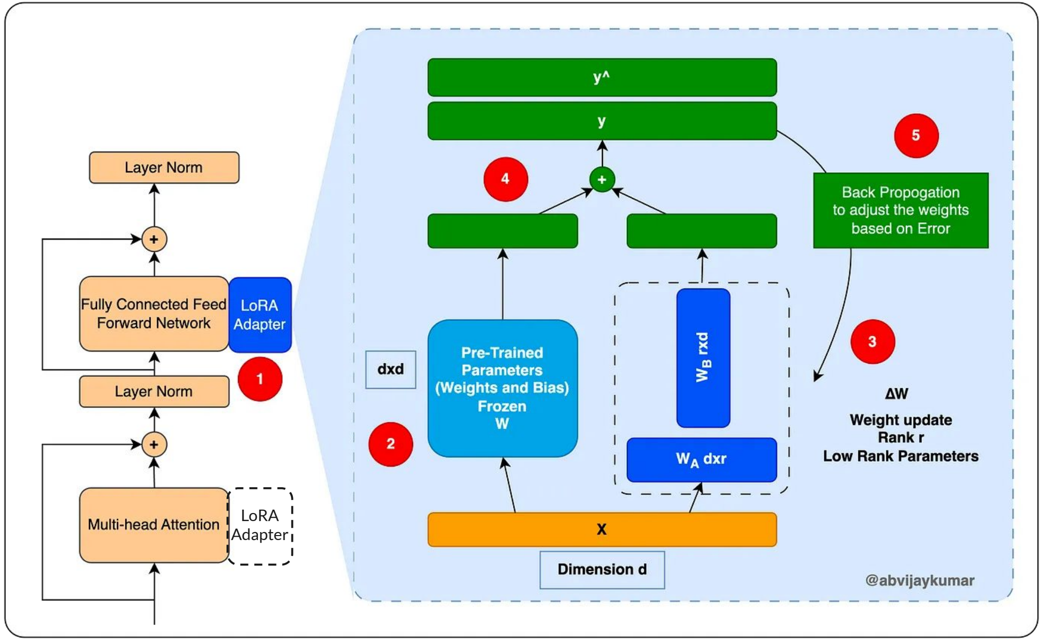
Low-Rank Adaptation (LoRA)

Main idea: Decompose weight update Δ to two low rank matrices

1. LoRA introduce additional layer of weights
2. Original weights (d*d) are **frozen** during finetuning
3. LoRA weights (W_A,W_B) are low-rank vectors (d*r, r*d)
4. Forwarding:

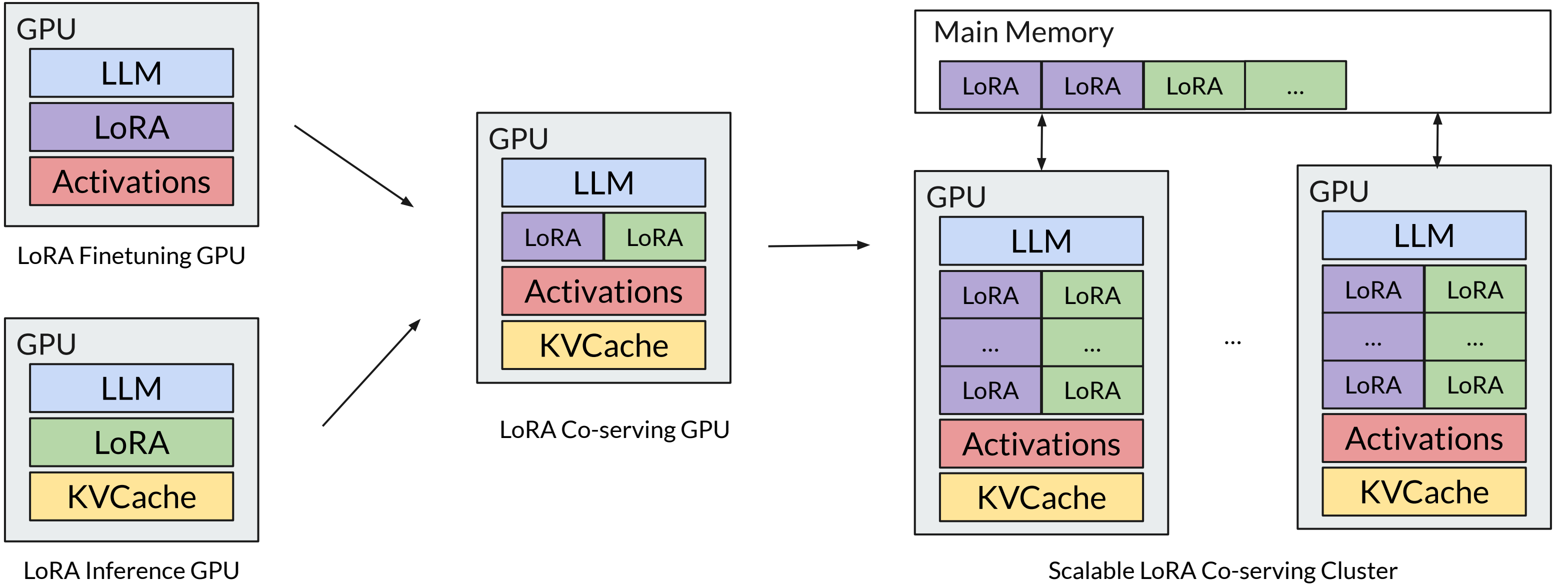
$$h = xW' = x(W + AB)$$
$$= xW + xAB.$$

5. Backpropagation update LoRA weights only



Scenario: Base Model + Adapters

Note: LoRA Layers can be treated as an add-on of the backbone model.



Batched Forward with Hetergeneous LoRA Adapters

Several systems have been developed to leverage this flexibility

S-LoRA: Scalable Serving of Thousands of LoRA Adapters is one of them

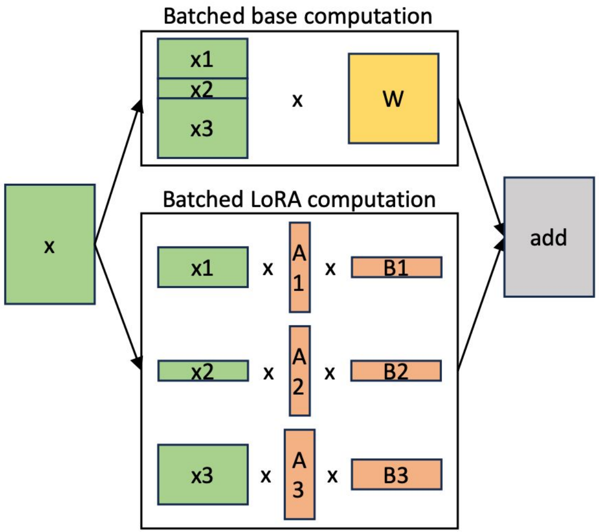
It is able to batch different adapters in **a single forward pass**:

Heterogeneous LoRA batching:

many requests, one backbone, mixed adapters

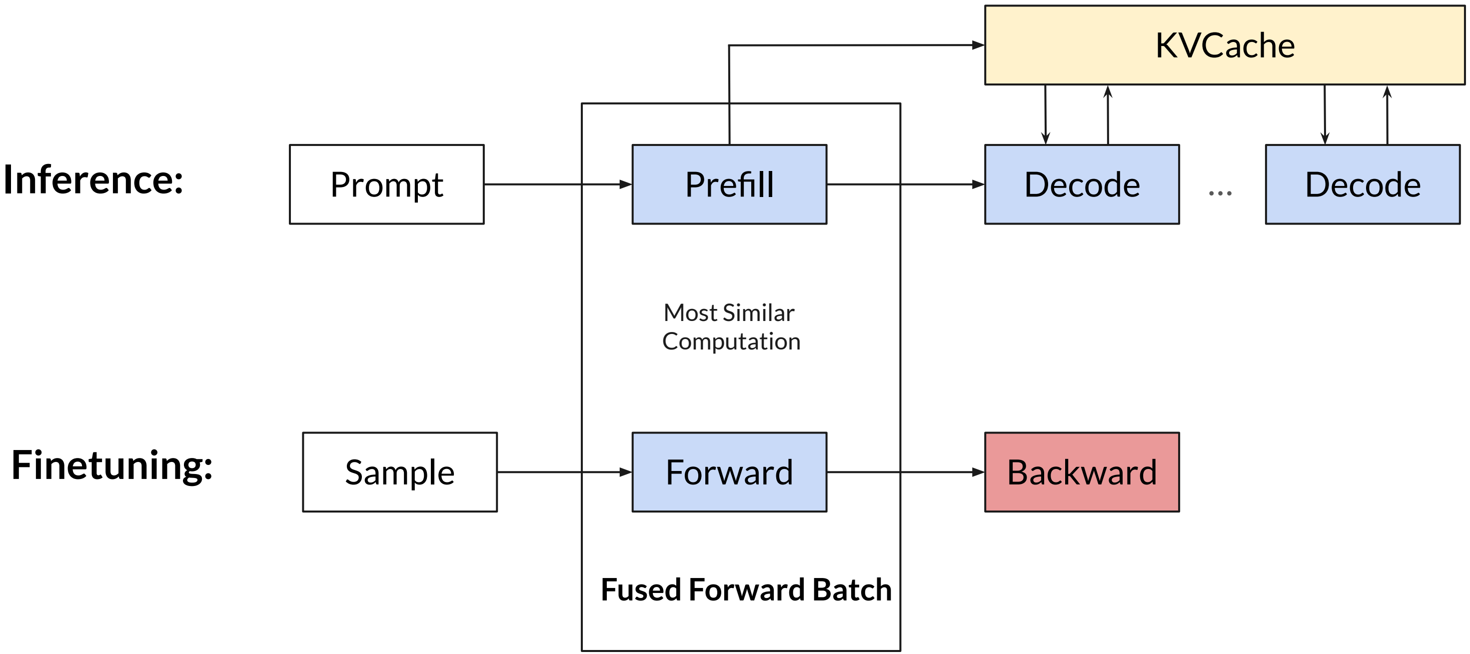
Split compute:

- Pretrained weights computation uses matrix multiplication.
- Different LoRA layers are computed together with **customized CUDA kernel**

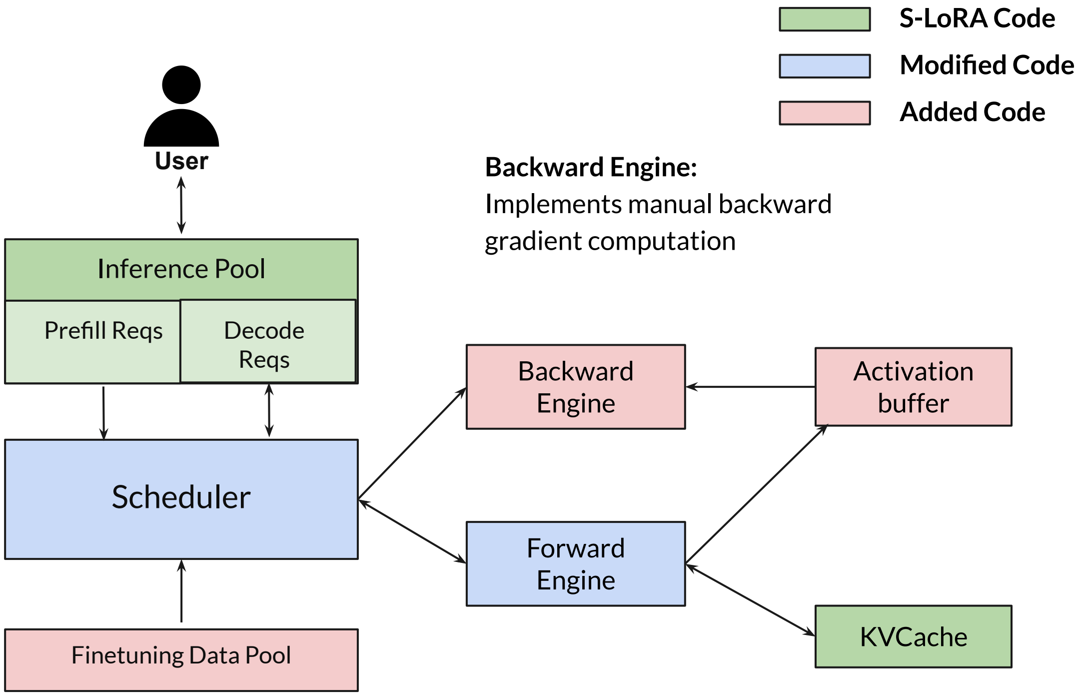


Our system is built on S-LoRA and extend it to support **finetuning**

Workload Breakdown & Fused Batch



System Design



Backward Engine:
Implements manual backward gradient computation

Inference Pool:
Track status of all inference requests

Scheduler:

- Decides when to run forward/backward
- Form fused batch or decode batch
- Tracks finetuning status (eg. epoch, #tokens pending backprop)

Forward Engine:
Implements hetergeneous LoRA Batching, with selective activation saving

Scheduler Design

Inference first, always:
Decode-phase tokens are dispatched immediately to keep user latency minimal.

Opportunistic training:
Finetuning backward runs only when the inference pool is empty, ensuring it never delays live queries.

Inference Prioitized Fused Batch:
Prefill requests are packed alongside training samples, maximizing GPU occupancy.

→ **Priority order:**
Decode ▸ Fused (Prefill ▸Finetune) ▸ Backward

```
while running:
        # 1. Decode work has top priority
        if inference_pool.has_decode():
            batch = inference_pool.take_decode()
            forward_engine.run(batch)
            continue

        # 2. No inference work + enough activations → run backward
        if inference_pool.is_empty()
                and activation_pool.size() >= ACTIVATION_LIMIT:
            batch = activation_pool.take_all()
            backward_engine.run(batch)
            continue

        # 3. Build a fused forward batch (prefill + finetune)
        batch = []
        batch += inference_pool.take_prefill(MAX_BSZ - len(batch))
        batch += finetune_data_pool.take(MAX_BSZ - len(batch))
        if batch:
            forward_engine.run(batch)
```

Scenario: No Inference

1. Scheduler checks finetuning status

2. Scheduler forms a forward batch using only finetuning samples

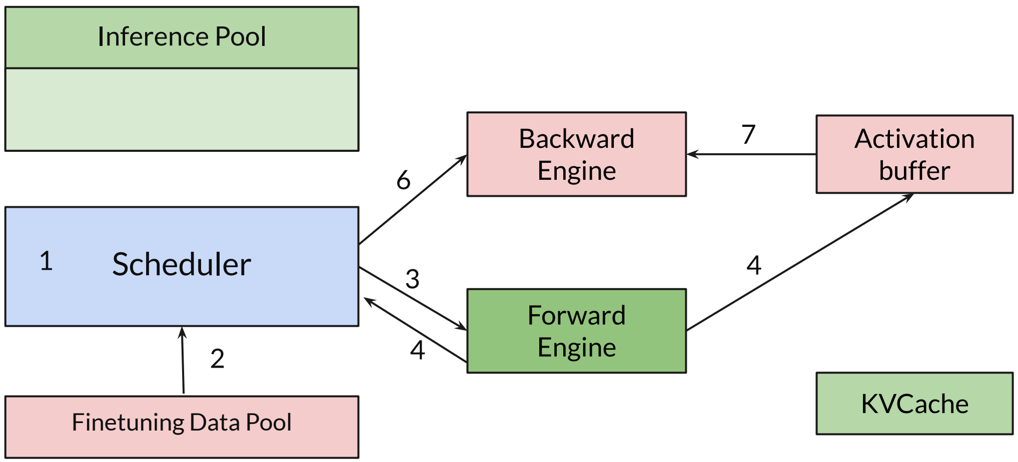
3. Scheduler gives the batch to forward engine

4. Forward engine performs forward pass, saves activations and updates finetuning status

5. Repeat 1-4 until activation limit reached

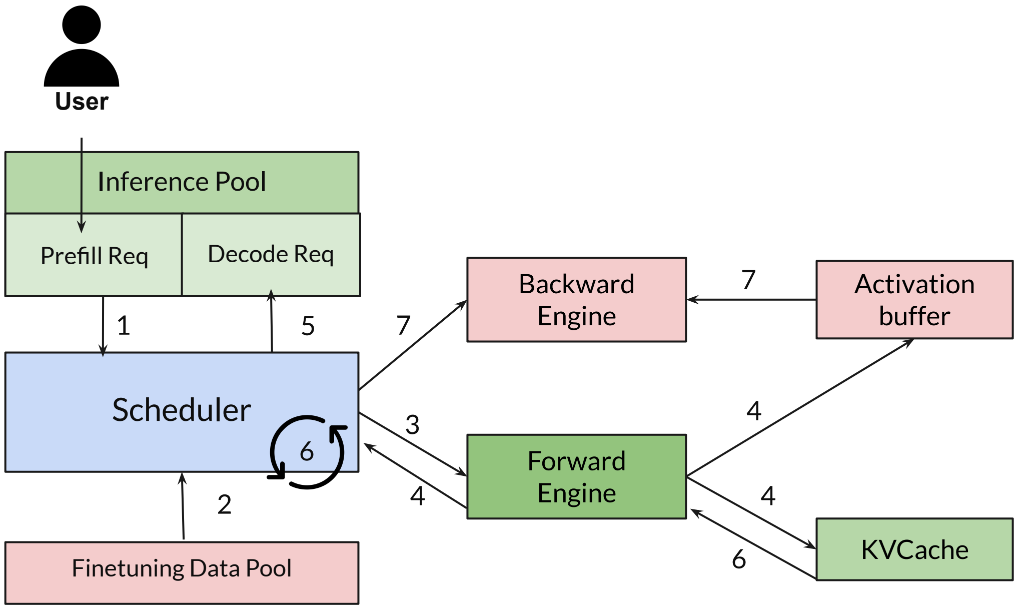
6. Scheduler issue a backward batch to the backward engine

7. Backward engine uses activation to perform backpropagation



Scenario: Light Inference

1. Scheduler pulls the prefill request
2. As space allows, scheduler checks the finetuning status and pulls finetuning samples to form a fused batch
3. Scheduler issue the batch to the forward engine
4. Forward engine performs forward pass, saves activations, KV cache and updates finetuning status
5. Scheduler update inference request status
6. In the following iteration, scheduler forms decode batches using the KV cache, until the request is completed.
7. Sometime in the future, when there is no pending inference requests, and enough saved activations, the scheduler issues backward batch.



Optimizations

Finetuning Interruptibility:

To ensure low-latency serving, the system can **preempt** ongoing fine-tuning tasks—whether in the forward or backward phase—to serve new inference requests immediately. Interrupted backprop tasks are **checkpointed** and can be resumed from the saved state without loss of progress.

Optimizations

Memory Manager: Unified Paging:

The **LoRA adapter weights (d × r)**, **per-sample activations (seq_len × d)**, **and per-request KV-cache entries (seq_len × d)** all share the same hidden-size dimension d. This symmetry lets us treat them as interchangeable “pages” and implement a single, unified paging layer, eliminating fragmentation.

Page Swapping:

Besides, given the high demand of memory in LLM finetuning & serving, the runtime must also handle **oversubscription**. The memory manager should swap pages, freeing space without disrupting computation.

Evaluation Plan

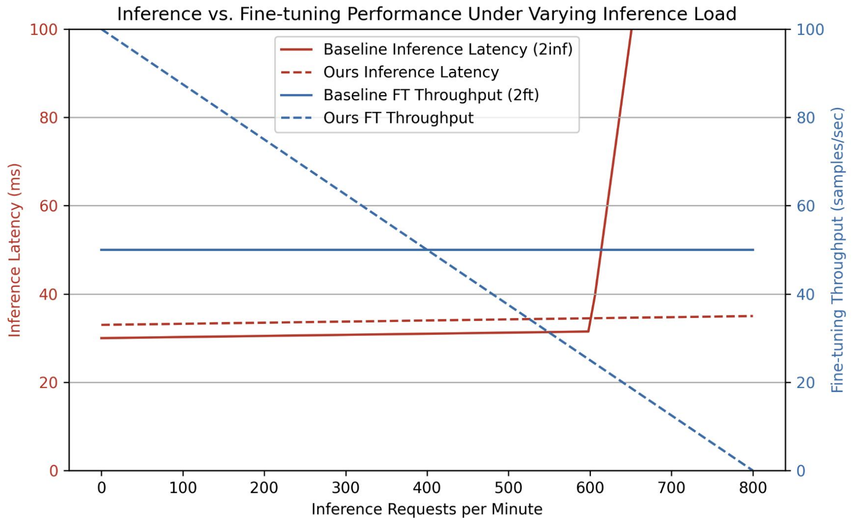
Goal: Compare co-serving vs. static GPU splits

Example Experiment Setup

- Hardware: 4-GPU node
- Baselines: fixed splits → 1/3 • **2/2** • 3/1 (Inf / FT)
- Workload trace: mix of inference + finetune
- Increase inference rate 0 → Max Load req/min

Expect outcome:

- At low inference rate, our system achieves **better throughput at finetuning**
- For inference, our system shows higher capability during high inference rate period



Expected Outcome Trend: Comparing our system to a traditional system with 2 GPUs for finetuning and 2 for inference

Key Takeaways

- **Unified Co-Serving Runtime**
  One software stack handles *both* real-time inference and continuous fine-tuning — no extra GPUs and no changes to model architecture.

- **Latency First, Maximize Utilization**
  Priority scheduling keeps user-facing latency on par with dedicated inference servers while harvesting idle cycles for training, raising overall GPU utilization.

- **Fused Batch with LoRA**
  By heterogeneous LoRA batching, inference prefill, and finetuning forward samples can be fused into one forward pass, eliminating context switch between inference and finetuning.

https://discslab.cs.mcgill.ca