MFS: An Efficient Model Family Serving System for LLMs

Yunxuan Zhang
The Hong Kong University of Science and Technology
Hong Kong, Hong Kong, China

Hao Wang
The Hong Kong University of Science and Technology
Hong Kong, China

Han Tian
University of Science and Technology of China
Hefei, China

Liu Yang
The Hong Kong University of Science and Technology
Hong Kong, China

Xudong Liao
The Hong Kong University of Science and Technology
Hong Kong, China

Wenxue Li
The Hong Kong University of Science and Technology
Hong Kong, China

Ping Yin
Inspur
Jinan, China

Bowen Liu
The Hong Kong University of Science and Technology
Hong Kong, China

Kai Chen
The Hong Kong University of Science and Technology
Hong Kong, China

Abstract

LLM serving providers typically offer a suite of structurally similar models, known as model families, such as the open-source Llama2 series featuring 7B, 13B, and 70B models. While numerous optimizations for LLM serving have been proposed, the potential for leveraging synergies between models within the same family has not been thoroughly explored. This paper introduces MFS, an innovative multi-tiered LLM model family serving system to exploit the structural similarities and parameter redundancies across different scales of models within a family. By utilizing a novel fine-tuning technique called Knowledge Precipitation, MFS restructures the largest model in a family to encapsulate smaller models within its architecture, enabling a unified multi-tiered serving pipeline. Based on the multi-tiered model, MFS realizes a highly parallelized tiered-level batching approach, significantly enhancing system efficiency. It also enables the sharing of intermediate features and KV-cache between models and facilitates multi-level sampling techniques during the inference phase. Experimental results demonstrate that MFS achieves substantial improvements over existing methods, including a 56.1% reduction in end-to-end token generation latency and a 47.8% decrease in GPU memory footprint without compromising the quality of generated content.

This work is licensed under a Creative Commons Attribution 4.0 International License.
EUROSYS '26, Edinburgh, Scotland Uk
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2212-7/26/04
https://doi.org/10.1145/3767295.3769355

CCS Concepts: • **Computing methodologies** → **Machine learning**; • **Computer systems organization** → *Distributed architectures.*

Keywords: Large Language Models, Model Serving, Distributed Systems, Inference Optimization

ACM Reference Format:
Yunxuan Zhang, Hao Wang, Han Tian, Liu Yang, Xudong Liao, Wenxue Li, Ping Yin, Bowen Liu, and Kai Chen. 2026. MFS: An Efficient Model Family Serving System for LLMs. In *European Conference on Computer Systems (EUROSYS '26), April 27–30, 2026, Edinburgh, Scotland Uk.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3767295.3769355

1 Introduction

Generative artificial intelligence is widely used today, e.g., ChatGPT [2] and DALLE [3]. These applications are powered by large language models (LLMs), e.g., GPT [12], Llama 2 [34], and Gemma [33]. LLM service providers package such pre-trained models into applications to deliver LLM serving.

To accommodate diverse quality-of-service requirements and reduce development costs, LLM providers often offer a series of models with similar architectures but different sizes, referred to as a model family.

For example, OpenAI allows users to choose between GPT-3.5 and GPT-4. The open-source Llama 2 family [34] also offers models of varying sizes, e.g., 7B, 13B, and 70B.

Recent work has introduced batching and caching to optimize LLM serving; for example, Orca [42] proposes selective batching, and KV-cache [27] reuses computational results from previous attention by saving the attention keys and values. However, the above optimization techniques apply only to a single model at a time. For batching, variations in model architecture and computational load violate the

homogeneity required for efficient GPU parallelism, making it challenging to batch requests across multiple models. KV-cache stores intermediate results that are specific to a model's architecture and parameters; therefore, sharing KV-cache across requests for different-sized models from the same family is difficult.

This paper takes a first step toward optimizing LLM serving for model families. We make three observations.

1. First, the Transformer architecture, which is the foundation of LLMs, exhibits redundancy. Prior work has shown that discarding some attention heads or layers does not significantly affect performance. We discuss this in detail in Section 2.4.
2. Second, models from the same family, despite differences in tensor shapes (e.g., number of layers, head size, and hidden size), share a common structure of stacked Transformer layers. This uniformity makes batching and caching feasible across multi-tier models.
3. Third, current LLM service providers promote a modelless paradigm, wherein users specify performance requirements (e.g., latency and quality) rather than a specific model.

Based on these observations, we design MFS, an efficient multi-tier LLM serving system for model families. At its core is the largest model in the family, fine-tuned to encapsulate all smaller models, establishing a multi-tier structure. This design saves GPU memory by sharing model parameters and enables batch processing across requests targeting different model sizes.

To present MFS, we first introduce how to transform a model family into a multi-tier structure, a process we refer to as "Knowledge Precipitation". Starting from a pre-trained model, we perform full-parameter fine-tuning on a small dataset and terminate computation early at intermediate layers. Our experiments show that Knowledge Precipitation achieves performance comparable to the corresponding baseline models in the family. Subsequently, building on the fine-tuned multi-tier model, we design a computationally and memory-efficient tier-level batching algorithm and implement KV-cache sharing across requests targeting different model sizes. We do not prescribe a specific algorithm for choosing a model size per request; rather, we provide an inference architecture that is memory- and computation-efficient. Various state-of-the-art scheduling algorithms, such as speculative sampling, can be integrated into this architecture. Additionally, users can manually specify the desired model size when submitting a request, as supported by most LLM service providers. We further explore scheduling policies for our multi-tier model in the system evaluation.

We evaluate the generation quality of MFS as well as its job completion time (JCT) and GPU memory footprint when serving a model family. Our experiments show that, as measured by metrics such as MMLU, the Knowledge Precipitation algorithm employed by MFS maintains generation quality across model families such as Llama 2 and Gemma. Furthermore, for serving model families, MFS reduces token-generation latency by 56.1% in batching scenarios compared to Orca [42], decreases GPU memory usage by 47.8% with the KV-cache sharing technique enabled, and improves GPU utilization by 35.9% over a state-of-the-art speculative sampling baseline [7].

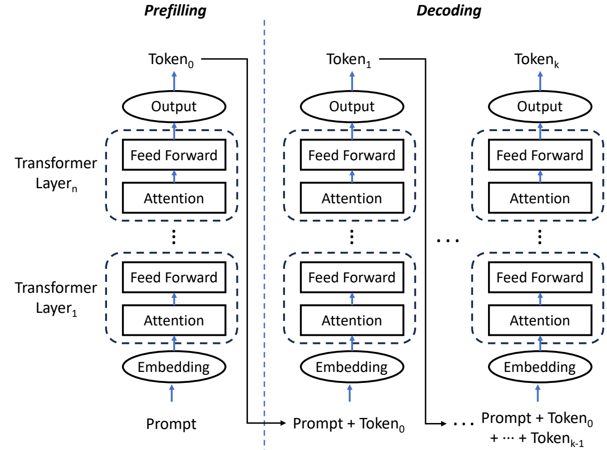


Figure 1. Illustration of the LLM inference procedure (w/o KV-cache). Some details of the Transformer structure, e.g., residual connections, layer normalization, and positional encoding, are omitted from the figure.

2 Background and Motivation

In this section, we first introduce the workflow of LLM inference and the key optimization techniques proposed in prior work. We then introduce model families and the current challenges in serving systems for model families. Finally, we present the theoretical foundations that create opportunities to utilize and optimize model families in a serving system.

2.1 LLM Inference

Prefilling and decoding. The inference process of an LLM can be divided into two stages: prefilling and decoding (Figure 1). In the prefilling stage, the model receives the entire input sequence (the prompt), which may include contextual cues or specific user queries, and fully processes it to establish an initial context state. Specifically, all input tokens are passed through the model's Transformer layers to produce contextualized embeddings that condition subsequent generation. The process then enters the decoding stage, in which tokens are generated sequentially. For each new token, the model conditions on both the original prompt and all

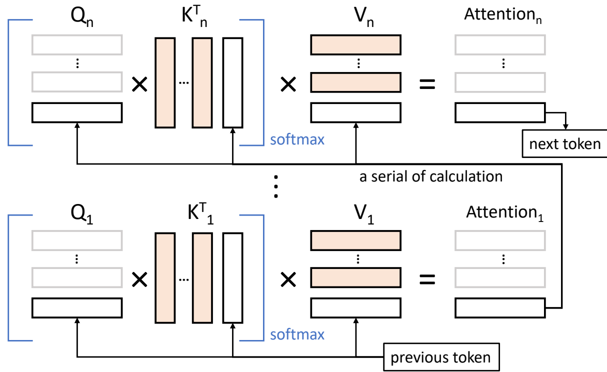


Figure 2. Illustration of the KV-cache. The orange areas indicate results that have been cached. To generate the (n+1)-th token, we first extract the query/key/value vectors ($Q_n/K_n/V_n$) for the n-th token from the last iteration. Together with the cached key/value vectors of previous tokens ($1, 2, \ldots, n{-}1$), we can efficiently compute the n-th attention output and generate the next token.

previously generated tokens to ensure coherence and contextual relevance. After each token is produced, it is immediately incorporated into the attention computations across Transformer layers to generate the next token. Decoding continues until (i) a user-specified or system-imposed maximum sequence length is reached, or (ii) the model emits an end-of-sequence (< *eos* >) token.

KV-cache. Within the Transformer architecture, the attention mechanism requires three intermediate vectors for each token: query (Q), key (K), and value (V). To generate the next token, the model must compute (or retrieve) Q, K, and V for all preceding tokens (Figure 2). The KV-cache stores the K and V vectors of previous tokens to avoid recomputation. Instead of recalculating K/V for all prior tokens, the model retrieves them from the cache, leveraging past computation to reduce latency for subsequent tokens and maintain responsiveness during long generations.

Batching. Batching is a foundational technique in LLM inference. It groups multiple inference requests so that the model processes them as a single batch rather than individually. Traditionally, requests are collected until a predefined batch size is reached and then processed together through the model's computational pipeline.

Recent work [42] has introduced continuous batching to account for the varying lengths of LLM responses. With continuous batching, individual requests can join or leave the batch at the end of each iteration (corresponding to the generation of one token). As a result, incoming requests experience minimal delay and are processed as soon as slots become available. This technique enables simultaneous processing of requests of different lengths without padding, significantly reducing wait time and improving throughput.

Sampling. Because the output of LLM inference is a probability distribution over tokens, sampling methods are used to generate coherent response sequences. In one approach, models generate multiple candidate tokens per iteration and select a sequence with high joint probability to enhance semantic coherence. A recent and efficient technique is speculative sampling [7], which employs a dual-model approach: a small draft model quickly proposes multiple tokens sequentially, and a larger target model verifies and refines them in parallel. This approach accelerates generation while maintaining the quality of the larger model.

2.2 Model Serving and Model Families

Model families. In the context of LLMs, a model family refers to a series of pre-trained models that share a similar architecture but differ in parameter size—specifically, the number of layers, the number of attention heads, and the hidden size.

Model/Parameter	7B	13B	70B
Layer Number	32	40	80
Head Number	32	40	64
Hidden Size	4096	5120	8192

Table 1. Specification of the Llama 2 model family.

As shown in Table 1, within the Llama 2 family, scaling from Llama 2–7B to Llama 2–70B increases the number of layers/heads/hidden size from 32/32/4096 to 80/64/8192, respectively. The sizes of the fully connected layers are adjusted proportionally. At the same time, all Llama 2 models use the same Transformer variant, which applies techniques such as pre-normalization, RMSNorm [44], SwiGLU [31], and grouped-query attention. Model families allow service providers to offer tailored services based on user needs while leveraging similarities between models to reduce development and maintenance costs.

Model serving system. A model serving system manages and executes model inference at scale, especially where low latency and high throughput are crucial. Such systems provide the infrastructure to handle variable workloads, dynamically manage resources, and efficiently execute multiple requests simultaneously. Related work includes INFaaS [28], Tabi [37], and Cocktail [16]. INFaaS automatically selects appropriate models for different devices and batch sizes, adapting quickly to workload changes. Cocktail improves performance through ensemble learning, using multiple small models in parallel and dynamically adjusting ensembles to optimize cost. Tabi features a multi-level inference engine that simplifies queries with smaller models before processing them with larger models; however, it targets discriminative rather than generative models. Systems such as Clockwork [15], BatchMaker [14], Orca [42], and vLLM [22] further optimize serving with techniques such as execution time prediction, batching, and GPU memory management.

2.3 Challenges of Serving Model Families

2.3.1 GPU Memory Challenges.

The growth in model size far outpaces the increase in GPU memory. From CNNs (e.g., ResNet and AlexNet) to LLMs, model sizes have grown from tens or hundreds of millions to tens of billions of parameters. In contrast, GPU memory has increased more modestly (e.g., from 32 GB in the V100 to 80 GB in the H800). Consequently, GPU memory footprint is a critical concern for LLM serving. Although inference, unlike training, does not require storing gradients and optimizer states (which consume most GPU memory), it still demands substantial memory to hold the model parameters themselves. Moreover, serving systems for model families may need to host multiple LLMs concurrently, further exacerbating memory pressure.

The KV-cache also significantly consumes GPU memory. For example, a Llama 2–70B model with a 512-token input requires at least 1.25 GB for its KV-cache. This is estimated by doubling the hidden size (8192) to account for key and value vectors, multiplying by 80 layers, and using 2 bytes per element in FP16. The extensive memory usage of KV-cache intensifies pressure on limited GPU resources, thereby limiting the capacity for concurrent requests.

For model families, storing and managing KV-caches for multiple models adds complexity. In addition, KV-cache cannot be shared across models, further taxing GPU memory resources.

2.3.2 Underutilization of GPU Compute Resources.

In LLM inference, the decoding stage underutilizes GPU compute resources. Because KV-cache avoids redundant computation, each iteration processes only the vectors related to the most recent token. Meanwhile, only a single token is generated per iteration, leading to poor utilization of GPU parallelism. Although batching multiple requests can improve utilization, existing methods do not support batching across different models within a family, limiting efficiency in model-family serving scenarios.

2.3.3 Cost per LLM Query Matters.

Efficiently managing the cost per LLM query is crucial for the financial viability of LLM services. As models become larger and more complex, their computational requirements increase, raising operational costs. To meet strict latency demands, service providers often overprovision compute resources, particularly GPUs, which can reduce cost effectiveness. Balancing computational expense with performance is essential to ensure each query delivers maximum value at minimal cost. In the context of model families, this balance is even more challenging because different models require different resource allocations, further complicating cost management and efficiency.

2.4 Observations and Opportunities

We next discuss key observations and opportunities in serving model families of LLMs. We focus on the structural uniformity within a model family, the redundancy of Transformer architectures, and the shift toward model-less inference systems. These insights are crucial for improving deployment efficiency and performance.

Structural uniformity within a model family. Models in the same series often share an identical architectural framework with variations primarily in scale. While specific parameters (e.g., the number of layers, hidden size, and number of attention heads) differ, the overall design—such as the use of stacked Transformer layers and the connectivity pattern among them—remains the same. For instance, the Llama 2 series includes Llama 2–7B, Llama 2–13B, and Llama 2–70B, all of which employ a common Transformer variant. This consistency provides a foundation for applying uniform optimizations such as batching and KV-cache strategies across models in the same series.

Redundancy within Transformers. Prior studies show that some Transformer layers and attention heads can be pruned without significantly affecting performance, indicating redundancy.

To exploit this redundancy, researchers propose structured pruning [17], which reduces model complexity by removing entire components (e.g., neurons or channels). In Transformers, structured pruning commonly targets heads and layers. Head pruning addresses redundancy among attention heads: not all heads contribute equally to performance. In practice, a small subset often plays critical and linguistically interpretable roles, such as encoding positional, syntactic, or rare-word information. For example, pruning up to 38 out of 48 heads in an encoder resulted in only a 0.15 BLEU decrease on the English–Russian WMT dataset [35]. Layer pruning selectively removes layers to reduce network depth and improve inference efficiency. Because layers execute serially, reducing depth can substantially decrease latency. Layer pruning can be implemented straightforwardly; for instance, LayerDrop applies structured dropout during training, enabling dynamic depth adjustment under computational or latency constraints.

This redundancy presents opportunities to share model parameters across tasks and models, potentially reducing GPU memory usage and increasing computational efficiency, leading to more resource-efficient deployments of LLMs.

Shift toward model-less inference systems. Inference serving systems increasingly adopt a model-less approach, where the focus is on meeting specified performance metrics rather than invoking predefined models. This shift opens opportunities to adapt model architectures for specific operational goals. By tailoring model structure to task requirements instead of adhering to a fixed architecture, service providers can achieve greater efficiency and effectiveness.

Opportunity. We explore the feasibility of employing a single model to encapsulate multiple tiers within a series. For instance, can Llama 2–70B be structured to also perform the functions of Llama 2–13B and Llama 2–7B? Achieving this would require adapting pre-trained parameters to enable such flexibility. While this approach challenges conventional pre-training practices, it could streamline model management by reducing the number of distinct models that must be maintained and deployed.

Exploring the potential of multi-tier models for LLM serving reveals several advantages. Within a series, batching becomes more feasible due to shared architectural elements (e.g., identical layers or parameters at certain depths). This commonality also facilitates KV-cache sharing among different requests, further optimizing memory usage and reducing redundant computation. Additionally, in approaches such as speculative sampling, draft and target models can share partial computation results and KV-caches. This synergy improves token-generation efficiency, reduces latency, and increases resource utilization across the serving system. Such multi-tier configurations can increase throughput and scalability in LLM serving.

3 Design

In this section, we introduce MFS, a serving system targeting LLM model families. We first illustrate how to transfer a pre-trained LLM model family into a multi-tiered model, then we describe how to integrate the multi-tiered model into LLM serving systems and enforce further optimization.

3.1 Building Multi-tiered Structure

Now we explore how to encapsulate a series of LLMs with only one model and introduce the system-aware optimization.

Design goal. MFS aims to develop a universal method to transform any pre-trained decoder-based LLM model family into a multi-tiered model, where each tier corresponds to a specific model from the family. The requirements are: 1) The quality of generated content and inference timing of each tier must be equivalent to its corresponding original independent model. 2) Minimize GPU computation power consumption and time cost. 3) Bringing opportunities to optimize the batching and KV-cache sharing among different tiers.

Strawman model. In previous work, the design of early exiting matches the multi-tiered structure. Therefore, we first implement a strawman design based on early exiting. We first evenly divide the largest model in a model family into several segments by layer, with the number of segments matching the number of models in the family. Since it is a generative task in our case, we add an output layer (a fully connected layer followed by a softmax function). However, our experiment on Llama2-chat family shows that this approach does not work. The lower-tier model generates meaningless words that do not even form grammatically correct sentences.

We further apply parameter-efficient fine-tuning (PEFT) [20], which incorporates additional adapters (LoRA) [21] into each layer to refine model performance. However, it still does not work. This observation aligns with prior research findings [39], which highlight limitations in leveraging shallow layers for complex tasks.

Early exiting. A straightforward approach is to train, from scratch, a multi-tiered model for a given model family. This can be realized by attaching an independent classification head to every Transformer layer and imposing a loss at each layer, thereby endowing each layer with full language-modeling capability. Early-exit training has been applied to Transformers in DeeBERT [40], demonstrating its effectiveness.

However, this approach is prohibitively expensive: training Llama2-70B [34] requires approximately 1.7 million A100 GPU hours. Moreover, our system does not require every layer to achieve independent language-modeling capability; a small number of consecutive blocks suffices.

In addition, state-of-the-art LLM training also requires access to trillion-token, high-quality corpora (often non-public), which are typically unavailable to system designers.

Deep pruning. Deep pruning reduces model size by removing parameters. However, it removes parameters in structurally discontinuous ways, which makes advanced optimizations—such as incremental computation and KV-cache sharing—impractical. For example, pruning a middle layer in a Transformer invalidates the KV-caches of all subsequent layers. Although deep pruning can efficiently produce smaller models without additional fine-tuning, our system requires a specific model structure that preserves KV-cache compatibility for memory efficiency, rendering deep pruning unsuitable.

Knowledge precipitation. Given the high resource cost of early exiting and the low generation quality achieved by the strawman and PEFT variants, we adopt full-parameter fine-tuning to balance efficiency and accuracy.

We adopt Knowledge Precipitation [19], a post-training fine-tuning technique. It uses a larger model to guide the optimization of a smaller model, effectively distilling knowledge from the former into the latter. This enables the smaller model to attain competitive performance with substantially less training compute than directly training it on task data.

In this work, we use Knowledge Precipitation as a preparatory step for our inference system because it naturally yields a usable multi-tier model, which is pivotal to our design. It also enables a key system insight: KV-cache sharing. In multi-model autoregressive generation, requests often switch between small and large models because token difficulty varies (e.g., technical terms versus function words). In conventional systems where the models are independent, when a token is

generated by the small model and the next token requires the large model, the large model cannot reuse the small model's KV-cache due to parameter mismatch, incurring extra computation and memory. By contrast, when the small model is structurally embedded in the large model (as produced by Knowledge Precipitation), KV-caches remain compatible across switches and can be reused end-to-end.

In practice, we create tiers from the largest model in a model family by dividing it according to layers and heads, corresponding to all other models in the family. These tiers are nested and share parameters, forming a multi-tiered structure. We utilize only the parameters from the largest model because models in a family typically follow a common training procedure and use identical datasets, ensuring consistency and efficiency in leveraging pre-trained knowledge. The division of layers and heads is based on the corresponding model's size, with each tier's layer and head count proportionally adjusted to match or not exceed the model's size, considering variations in hidden size.

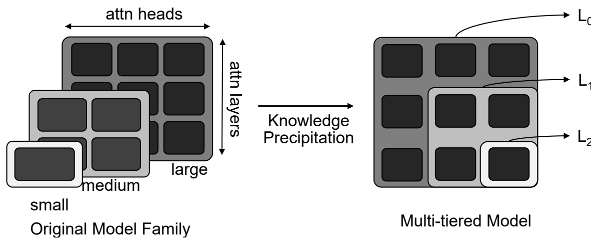


Figure 3. Workflow of knowledge precipitation.

Figure 3 illustrates an example workflow. The left side shows the original model family, and the right side depicts the multi-tiered model created through knowledge precipitation. In this example, the model family consists of three models, i.e., large, medium, and small. The large model has three layers, with each layer containing three heads. The multi-tiered model is structured into three hierarchical tiers, each corresponding to one of the family's models.

Each tier defines an independent loss L_i. We train all tiers jointly by minimizing a weighted sum:

$$L = L_0 + \lambda_1 L_1 + \cdots + \lambda_i L_i \ .$$

This objective encourages every tier to acquire full language-modeling capability. Moreover, joint backpropagation allows gradients from higher tiers to guide parameter updates of smaller tiers—the core idea of knowledge precipitation.

Furthermore, for the knowledge precipitation of multi-tier models, step-by-step fine-tuning is necessary. The full-parameter fine-tuning process has multiple steps according to the number of tiers, each focusing on precipitating the knowledge from the higher tiered models down to the next. For a n-tier structure, we need $n-1$ steps of fine-tuning. The loss functions for step i is $L_0 + \lambda_1 L_1 + \cdot + \lambda_i L_i$, where L_k is the loss of tier k, with each tier featuring an additional output layer. This can be intuitively understood as knowledge precipitating from the largest model to the lower model tier step by step. Our experiments show that if the loss functions of all tiers are directly added together without step-by-step fine-tuning to perform the knowledge precipitation of all model tiers at once, the performance of the lower-level model tiers will become uncontrollable.

System-aware optimization. Based on the above multi-tiered structure, we consider further optimizations tailored to the requirement of the serving system. Instead of splitting both layers and heads, we opt to divide only the layers. The reason is that: 1) Prior work [39] has shown that the number of layers has a greater impact on inference latency, since layers operate serially while heads can be calculated in parallel. 2) The computation of attention involves all heads in each layer, requiring that each tier has the same heads to achieve consistent KV-cache results in each layer. However, splitting only layers to match each tier's size with its corresponding model can lead to insufficient layers in lower tiers. For example, Llama2-70B has 80 layers. To keep the sizes consistent, the tier corresponding to Llama2-7B will only have 8 layers, note that the original Llama2-70B has 32 layers.

We observe that model size is not exactly proportional to the inference latency. Therefore, by measuring the inference latency of smaller models in the model family, we identify the corresponding layer number of the largest model. For example, we find that the inference latency of Llama2-7B (32 layers) and Llama2-13B (40 layers) correspond to 20 and 25 layers of Llama2-70B, when the prompt size is 1024.

3.2 Model Family Serving System

Here we introduce the serving system for model family with the above multi-tiered model. We first show the system overview, then we describe the procedure of batching and caching optimization. Finally, we introduce the possibility of integrating several multi-model scheduling algorithms into our architecture and discuss several design details.

3.2.1 System Overview. MFS system design consists of a front-end, request pool, tier-level scheduler, multi-tiered model, and KV-cache manager. As shown in Figure 4. The front-end provides a user interface or API, handling the receipt of user requests, i.e., prompts and the required model, and delivers the system-generated responses back to the users. The request pool is responsible for storing pending requests. The tier-level scheduler acts as the control center for the entire model family serving, determining the order of execution of each request's inference. It uses several queues to schedule requests of different tiers and decides which unprocessed requests to take out from the request pool. The execution of inference is carried out by the multi-tiered model, which replaces the model family to handle requests with varying QoS requirements. The KV-cache manager stores

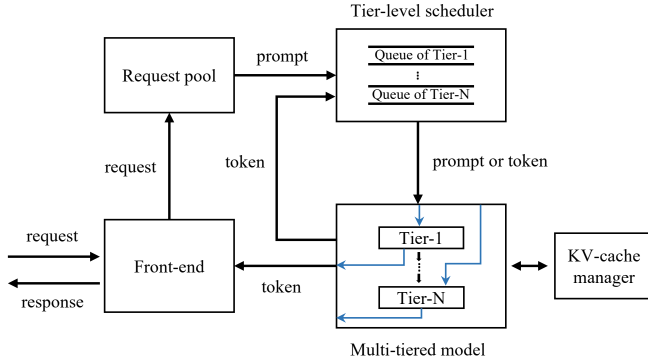


Figure 4. Overview of MFS. We replace the model family with the multi-tiered model. A request first comes to the front-end to create a user session or reuse an existing session. The front-end sends the request to the request pool.

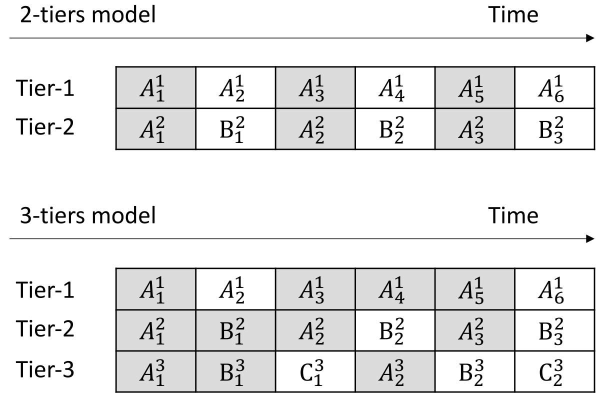


Figure 5. Group batching of multi-tiered model (using First-Come-First-Serve). The first case is a 2-tiers model and the second is 3-tiers. Grey squares represent batch-processable for the concurrent requests. A,B,C represent different tiers of the model. For simplicity, we assume the corresponding model part for each tier runs on a separate GPU and only one token was generated for each request within this time snippet.

and manages the KV-cache of the multi-tiered model during the inference.

3.2.2 Optimization on Batching and Caching.

The multi-tiered structure presents opportunities to optimize batching and caching within the model family serving system. Essentially, the inference processes for requests across different tiers share the same portions of the model. This commonality facilitates the batching of overlapping parts and enables the possibility of KV-cache sharing.

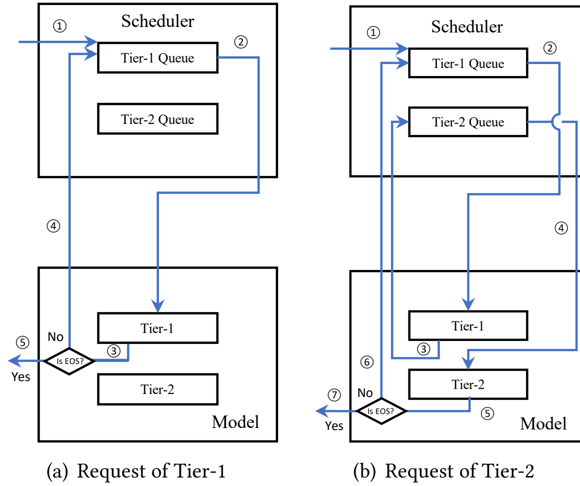
MFS employs three techniques to optimize batching and caching. First, unlike traditional inference batching which is limited to identical and complete models, we introduce group batching that maximizes parallel processing for common model parts across different tiers. Considering that each inference iteration in the decoding stage processes and generates only one token, leading to low GPU parallel utilization, we propose attention fusion to enable parallel execution of attention calculations across different requests. Finally, to enable KV-cache sharing among requests from different tiers, we introduce a shareable KV-cache management method.

Group Batching. Recall that in a multi-tiered model, the front layers of a higher-tier model correspond to a lower-tier model. When processing a higher-tier request within these shared layers, it can be batched together with requests from the lower tier. We call this procedure as group batching, where parts of requests that utilize the same model components are batched together. Figure 5 illustrates two examples of group batching. The first case is running six tier-1 requests and three tier-2 requests in a 2-tier model. At time-1, we simultaneously serve the first requests of tier-1 (A_1^1) and tier-2 (A_1^2), which share the same model part and are batch-processable. At time-2, we run the second request of tier-1 (A_2^1) on GPU-1 and the remaining of the first request of tier-2 (B_1^2) on GPU-2. The second case shows running six tier-1, three tier-2 and two tier-3 requests on 3-tier models. As we can see, as the number of tiers increases, it leads to more batch processing opportunities.

Attention fusion. For generative tasks, the varying lengths and number of tokens generated per request prevent direct parallel processing. Previous methods use padding to uniform lengths or employed selective batching to batch only certain operations. Padding significantly increases latency for shorter requests (thus not a consideration in our design), while selective batching reduces GPU parallelism as it does not allow batching for certain operations, primarily attention. When serving a model family, the average batch size tends to be smaller compared to serving a single model, which can reduce GPU parallel efficiency. To compensate, we propose attention fusion, which allows the attention computations of different requests to be batch processed. Attention fusion processes the attention of all requests within a batch by first concatenating the QKV (Query, Key, Value) matrices of each request. It then performs matrix multiplication to compute the attention between different positions. Although this approach introduces some redundant computations, such as the attention between tokens of different requests, it does not increase computational time due to the parallel processing capabilities of GPUs.

Shareable KV-cache. The multi-tiered structure inherently facilitates more dynamic and extensive KV-cache sharing across different levels of the model due to the uniform structure and redundancy inherent in the transformer layers. Our design utilizes the inherent structural similarities and redundancies within the transformer architecture to enable smart KV-cache sharing. Each tier in the multi-tiered model

setup corresponds to a different model size and complexity, making it possible to tailor cache utilization according to the processing needs of specific tiers. For instance, higher tiers, which handle more complex queries, can dynamically borrow computation results from lower tiers when similar requests are processed concurrently. This not only reduces the overall computational load but also speeds up the inference time by avoiding redundant calculations.



(a) Request of Tier-1 (b) Request of Tier-2

Figure 6. Example of tier-level scheduling.

3.2.3 Request Scheduling on Multi-tier.

As shown in Figure 6, our scheduling system operates at a more granular level by focusing on individual tiers of the model architecture. This approach allows for a more detailed control over resource allocation, ensuring that each tier receives the necessary computational resources based on its current load and computational requirements. By managing resources at the tier level, our system can more effectively handle the dynamic nature of LLM requests, which vary significantly in complexity and urgency.

To enhance the responsiveness of our LLM serving system, we incorporate preemptive scheduling strategies that allow for the interruption of lower-priority tasks in favor of higher-priority ones. This capability is particularly crucial when dealing with a range of request complexities across different model tiers. For example, urgent requests that require immediate attention can preempt ongoing tasks within lower tiers, thus reducing wait times for critical operations.

Our scheduling system also implements priority-based queuing mechanisms. Each request is assigned a priority level based on several factors, including the expected computational load, the urgency of the request, and the service level agreement (SLA) associated with the user or task. This structured prioritization helps in managing the queue more effectively, ensuring that high-priority requests are serviced promptly.

Fairness is a critical component of our scheduling strategy, especially when serving requests across multiple tiered models. We aim to provide equitable access to computational resources for all requests, regardless of the tier they originate from. To achieve this, our system employs fair queuing algorithms that balance resource allocation across different tiers, preventing any single tier from monopolizing the GPU resources.

3.3 Distributed Architecture

As the LLM size grows, model parameters require multiple GPUs for deployment. Existing work, e.g., FasterTransformer and Orca, uses two parallelism strategies for distributed deployment, i.e., intra-layer and inter-layer parallelism. The former slices tensors (linear or attention parameters) within the same layer across different GPUs, while the latter distributes layers on different GPUs. MFS also supports the above two types of distributed execution. Similar to Orca, each worker process is responsible for one inter-layer partition (different partitions may cross servers), and there are multiple GPUs within a worker (same server) to perform intra-layer parallelism.MFS has an inherent advantage in distributed scenarios. Considering the structural characteristics of multi-tiered, when performing inter-layer partitioning, MFS partitions the layers belonging to the same tier together, so that different inter-layer partitions can be asynchronous with each other, avoiding synchronization overhead.

3.4 Design Discussion of Scheduling

As noted earlier, our multi-tier inference architecture does not prescribe a specific algorithm for scheduling requests across models. Scheduling can be treated as a plug-and-play module that integrates with our framework to exploit its memory and computational efficiencies. Below, we discuss two state-of-the-art multi-model scheduling strategies. In Section 5.3, we implement speculative sampling and show how our system accelerates it.

Heuristically determine the query level. Heuristic algorithm automatically identifies the optimal complexity level for processing a query within a multi-tiered model structure. This method uses differences in output distributions from models of varying sizes to intelligently select the smallest model capable of handling the query with minimal accuracy loss.

The core idea here is to use a smaller version of the largest model in the model family to quickly assess the complexity of the incoming query. The smaller model generates preliminary results, and based on certain metrics or thresholds, such as confidence scores or deviation from expected patterns, the system determines whether the query can be satisfactorily resolved at this level or if it needs to be escalated to a more complex model.

Speculative sampling. Speculative sampling technique utilizes the hierarchical model structure by using lower-tiered models as draft models, which are less computationally demanding and quicker to execute, making them suitable for initial evaluations of multiple potential continuations of an input sequence. This approach not only speeds up initial computations but also provides early insights that help reduce the computational load on more complex target models.

The Speculative sampling architecture can be optimized by our system because sharing computation results and KV-caches between the draft and target models further reduces the computational cost. Once a draft model processes an input, its outputs and associated KV-cache are reused by the target model, eliminating the need to recompute initial sequence stages and conserving GPU resources. This efficient cache integration minimizes redundant calculations and enhances response times.

Additionally, our system implements batch processing for both draft and target models to maximize computational resource utilization. This strategy, combined with our shared KV-cache system, allows simultaneous sequence generation across multiple requests, optimizing GPU use and improving overall throughput. We show in detail the speedup of our system for speculative sampling in section 5.3.

4 Implementation

In our implementation, we applied the concept of "knowledge precipitation" to restructure a pre-trained LLM into a multi-level model. This involved full-parameter fine-tuning, which allowed each layer of the original model to condense its functionalities into discrete submodels, each capable of standalone operation. This hierarchical structuring not only preserves the integrity and depth of the learned representations but also facilitates more granular control over the inference process.

Training infrastructure. We use two servers, each equipped with eight NVIDIA H800 SXM5 GPUs. Each server has 2×56 Intel Xeon Gold CPU cores, and 2TB memory. Two servers are interconnected with eight 400Gbps NDR InfiniBand network interfaces.

Models and datasets. We chose the Llama2 family, a widely-used series of large models that includes three official versions: Llama2-7B, Llama2-13B, and Llama2-70B. Due to computational resource constraints, i.e., 16× NVIDIA H800 GPUs can only support up to Llama2-13B, we only fine-tune Llama2-7B and Llama2-13B (the chat version). To test the generality of our approach, we also evaluate the Qwen model family, which includes Qwen-4B and Qwen-7B. To demonstrate the resilience and time efficiency of knowledge precipitation, we use a small dataset [4] with 9.85k dialogues on Hugging Face for fine-tuning.

Fine-tuning parameters. The model fine-tuning mostly follows the fine-tuning parameters in the LLAMA-2 paper. We use the AdamW optimizer, with a learning rate of 2e-5. The half-periodic cosine learning rate function is used. The weight decay amplitude is 0.1, and we apply gradient clipping of 0.3. With 8 gradient accumulations, the equivalent batch size of training is 64. The input sequence length is 4096 tokens. The model is fine-tuned for one epoch on a generation/dialogue mixed dataset with about 0.66T (665.3 billion) tokens, a total of 2500 iterations, and a total time of about 24 hours.

For inference serving, we have two settings, one is a 2× NVIDIA A100 GPUs server with 2×48 Intel Xeon Gold CPU cores and 512G memory. The second is an 8× NVIDIA 3090 GPUs server with 80 Intel Xeon Gold CPU cores and 256G memory.

5 Evaluation

In this section, we first assess the effectiveness of knowledge precipitation in constructing multi-tiered models. Then, we evaluate the system performance and inference speedup of MFS. Finally, we deep dive into the scenarios of KV-cache sharing and speculative sampling. The key findings are as follows:

- We assess the effectiveness of knowledge precipitation by evaluating the quality of content generated by multi-tiered models. Using ten widely-used LLM metrics, we find that it can achieve performance comparable to that of the original model family.
- We evaluate the system improvement of MFS on batching. Results show that MFS can improve the JCT by up to 31.2% and reduce the response latency by up to 56.1% compared to ORCA.
- Further evaluation on the scenarios of KV-cache sharing and speculative sampling show that MFS can reduce the GPU memory footprint by up to 47.8% and increase the GPU utilization from 23.9% to 59.8%.

5.1 Baselines and Metrics

To evaluate the quality of content generated by the multi-tiered model, we use Llama2-7B-chat and Llama2-13B-chat as baselines, hereafter referred to as Llama2-7B and Llama2-13B, which are publicly available versions of the Llama2 models. We assess performance across ten common tasks, serving as metrics for evaluation. These include MMLU [18], a multi-task benchmark; PIQA [6], which focuses on physical commonsense reasoning; OpenBookQA [25], centered on open-domain question answering; HellaSWAG [43], which predicts sentence completions in a narrative context; BoolQ [8], a yes/no question answering dataset; ARC Easy and ARC Challenge [9], which involve science question answering; and ANLI rounds 1, 2, and 3 [26], which are adversarial NLI tasks designed to test reasoning abilities under progressively challenging conditions. To evaluate MFS, we compare it with Orca [42], the state-of-the-art batching design for

Metric	Llama2-7B	**MFS-7B**	FT-7B	Llama2-3B	**MFS-3B**
mmlu	**0.48**	0.46	0.48	0.24	**0.43**
piqa	**0.78**	0.76	0.77	**0.75**	0.71
openbookqa	0.29	**0.33**	0.32	**0.27**	0.25
hellaswag	0.56	**0.57**	0.59	0.49	**0.49**
boolq	0.75	**0.78**	0.70	0.68	**0.76**
arc-easy	0.68	**0.71**	0.69	**0.69**	0.57
arc-hard	0.39	**0.42**	0.42	**0.34**	0.33
anli-r1	0.35	**0.35**	0.39	0.33	**0.34**
anli-r2	0.34	**0.35**	0.40	0.32	**0.34**
anli-r3	0.37	**0.38**	0.36	0.35	**0.36**

Table 2. Convert Llama2-7B into a two-tier structure.

Metric	Llama2-13B	**MFS-13B**	FT-13B	Llama2-7B	**MFS-7B**
mmlu	0.53	**0.54**	0.53	0.48	**0.51**
piqa	0.78	**0.78**	0.77	**0.78**	0.77
openbookqa	0.35	**0.36**	0.36	0.29	**0.33**
hellaswag	**0.61**	0.60	0.59	**0.56**	0.54
boolq	**0.82**	0.77	0.79	0.75	**0.76**
arc-easy	0.78	**0.79**	0.76	0.68	**0.69**
arc-hard	0.46	**0.49**	0.47	0.39	**0.39**
anli-r1	0.43	**0.47**	0.43	0.35	**0.43**
anli-r2	0.43	**0.46**	0.43	0.34	**0.41**
anli-r3	0.41	**0.46**	0.43	0.37	**0.43**

Table 3. Convert Llama2-13B into a two-tier structure.

LLM serving and measure both average execution time of requests and response latency to generate each token. We also measure the reduction of GPU memory footprint and the improvement in GPU utilization by using multi-tiered model to replace the serial of models in a model family.

We do not compare MFS against state-of-the-art general-purpose serving systems (e.g., vLLM) because the optimization targets fundamentally differ, making a direct comparison inappropriate. vLLM primarily optimizes distributed execution across multi-node, multi-GPU, and heterogeneous environments using tensor, pipeline, and sequence parallelism (TP, PP, SP). It does not modify model architecture or parameter allocation during fine-tuning.

5.2 Quality of Multi-tiered Models

Before deploying MFS, we need to transform the model family into a multi-tiered model, ensuring that each tier matches the corresponding model in terms of inference latency and content quality. In this section, we first evaluate the performance of the multi-tiered models obtained through knowledge precipitation and further assess the impact of system-aware optimizations. To ensure generality, we test various tier configurations and different LLM model families.

Effectiveness of knowledge precipitation. We first assess the quality of content generated by models using knowledge precipitation without system-aware optimization. As shown in Table 2 and Table 3, we convert both Llama2-7B and Llama2-13B into a two-tier structure. For the 7B model, which consists of 32 layers and 32 heads per layer, we transform it into a two-tier model incorporating a 3B model. Since the Llama2 family does not have an official 3B version, we chose a popular open-source 3B version [5] with around 80,000 monthly downloads, featuring 24 layers and 24 heads each. For the 13B model, we convert it into a two-tier structure including a 7B model.

Table 2 shows the scores of the two-tier model based on Llama2-7B across various tasks. We compare its first tier, MFS-3B, with Llama2-3B, and the second tier, MFS-7B, with Llama2-7B. To exclude the impact of the fine-tuning dataset, we also fine-tune the Llama2-7B baseline directly on the same dataset to obtain FT-7B. As shown in the table, compared to Llama2-7B, MFS-7B performs better in eight out of ten metrics, and the remaining two metrics are also comparable. For MFS-3B, the number is 5 out of 10. Additionally, FT-7B and Llama2-7B have similar performances across all metrics, with some minor variations, indicating that the dataset and the fine-tuning process do not significantly influence the results. Table 3 shows the performance metrics for the two-tier model derived from Llama2-13B. MFS-13B/MFS-7B has 8/7 better metrics compared to Llama2-13B/Llama2-7B, respectively. These results all demonstrate that knowledge precipitation successfully produces multi-tiered models that meet the required performance standards.

We further observe that the two-tier model derived from Llama2-13B exhibits larger gains over its baseline than the two-tier model derived from Llama2-7B. Larger LLMs generally contain greater representational redundancy, making them more amenable to conversion into multi-tier structures. This finding strengthens the case for the feasibility and robustness of scaling knowledge precipitation to increasingly larger models.

Another notable observation from Table 2 is that the MFS-7B model obtained via knowledge precipitation surpasses the independently fine-tuned 7B baseline (FT-7B) on several metrics. Knowledge precipitation transfers information from a larger model to a smaller one. During training, the objective $L = \alpha L_0 + \beta L_1$—where L_0 and L_1 denote the losses of the large and small tiers, respectively—allows gradients from the large model to guide updates in the small model, effectively distilling its knowledge. This paradigm is widely adopted in practice to train compact, high-performing models; for example, DeepSeek distilled a high-performance 32B model (DeepSeek-R1-Distill) from the 671B DeepSeek-R1 [24].

Impact of system-aware optimization. Considering system batching and caching, we apply system-aware optimization by splitting only by layer. We split the layers based on the inference latency with different numbers of layers. We find that the inference latency of Llama2-7B corresponds to the latency of 24 layers in Llama2-13B. The input prompt size is 1024 in the measurement. Note that the prompt size affects inference latency, when the prompt size is less than 1024, the layer number at the intersection point for the same latency is always greater than or equal to 24. Given that most prompt sizes are below 1024 [34], selecting the first 24 layers of Llama2-13B can meet the inference latency requirements. Table 7 shows the results of the two-tier model obtained by

Metric	**MFS-13B**	Llama2-7B	**MFS-7B**	Llama2-3B	**MFS-3B**
mmlu	0.53	0.48	**0.50**	**0.24**	0.23
piqa	0.79	**0.78**	0.77	**0.75**	0.68
openbookqa	0.35	0.29	**0.30**	**0.27**	0.18
hellaswag	0.60	**0.56**	0.54	**0.49**	0.33
boolq	0.82	0.75	**0.81**	**0.68**	0.64
arc-easy	0.78	**0.68**	0.64	**0.69**	0.52
arc-hard	0.49	**0.39**	0.36	**0.34**	0.29
anli-r1	0.46	0.35	**0.42**	0.33	**0.37**
anli-r2	0.45	0.34	**0.43**	0.32	**0.33**
anli-r3	0.46	0.37	**0.41**	0.35	**0.35**

Table 4. MFS contains 13B, 7B and 3B

Metric	Llama2-13B	MFS-13B	MFS-10B	Llama2-7B	MFS-7B
mmlu	0.53	**0.54**	0.53	0.48	**0.53**
piqa	0.78	**0.79**	0.78	**0.78**	0.73
openbookqa	0.35	**0.36**	0.37	0.29	**0.31**
hellaswag	**0.61**	0.59	0.57	**0.56**	0.52
boolq	**0.82**	0.81	0.84	0.75	**0.82**
arc-easy	**0.78**	0.75	0.74	0.68	**0.68**
arc-hard	0.46	**0.46**	0.42	**0.39**	0.38
anli-r1	0.43	**0.43**	0.46	0.35	**0.44**
anli-r2	0.43	**0.43**	0.45	0.34	**0.45**
anli-r3	**0.41**	0.40	0.45	0.37	**0.44**

Table 5. MFS contains 13B, 10B and 7B

Metric	Qwen-14B	**MFS-14B**	FT-14B	Qwen-7B	**MFS-7B**
mmlu	**0.68**	0.64	0.65	**0.62**	0.58
piqa	0.75	**0.79**	0.76	**0.75**	0.72
openbookqa	**0.36**	0.34	0.36	**0.32**	0.28
hellaswag	**0.62**	0.58	0.59	**0.59**	0.52
boolq	**0.86**	0.84	0.84	**0.84**	0.81
arc-easy	0.72	**0.73**	0.71	**0.68**	0.63
arc-hard	**0.45**	0.44	0.43	**0.43**	0.38
anli-r1	**0.47**	0.45	0.44	**0.42**	0.41
anli-r2	**0.43**	0.42	0.42	**0.40**	0.38
anli-r3	**0.45**	0.43	0.42	**0.41**	0.40

Table 6. Qwen-14B, two-tiers

Metric	Llama2-13B	**MFS-13B**	FT-13B	Llama2-7B	**MFS-7B**
mmlu	0.53	**0.54**	0.53	0.48	**0.53**
piqa	0.78	**0.79**	0.77	**0.78**	0.73
openbookqa	0.35	**0.36**	0.36	0.29	**0.31**
hellaswag	**0.61**	0.59	0.59	**0.56**	0.52
boolq	**0.82**	0.81	0.79	0.75	**0.82**
arc-easy	**0.78**	0.75	0.76	0.68	**0.68**
arc-hard	0.46	**0.46**	0.47	**0.39**	0.38
anli-r1	0.43	**0.43**	0.43	0.35	**0.44**
anli-r2	0.43	**0.43**	0.43	0.34	**0.45**
anli-r3	**0.41**	0.40	0.43	0.37	**0.44**

Table 7. Apply system-aware optimization.

selecting only the first 24 layers of Llama2-13B. We observe that both MFS-13B and MFS-7B achieve comparable or even better performance in terms of generated content quality.

Validation on three-tier model. Building on system-aware optimization, we further evaluate the performance of converting Llama2-13B into different-sized three-tier models. Table 4 displays the results of cutting Llama2-13B into tiers corresponding to sizes of 13B, 7B, and 3B, with layer divisions at 24 and 16 respectively. Table 5 presents the results of cutting layers into 13B, 10B, and 7B tiers, with divisions at 32 and 24 layers. Since Llama2 does not have a 10B version, we only show the results for MFS-10B. The experiments show that each tier achieves content generation quality comparable to the corresponding baselines. Notably, we observe that the performance corresponding to Llama2-3B was significantly inferior, especially in MMLU. Nevertheless, MFS-3B works well in MMLU, as it is fine-tuned from the official Llama2-13B model through knowledge precipitation. Meanwhile, MFS-10B achieves performance between Llama2-13B and Llama2-7B, suggesting that our multi-tiered model can generate a more granular range of models with different QoS, thereby offering more fine-grained services.

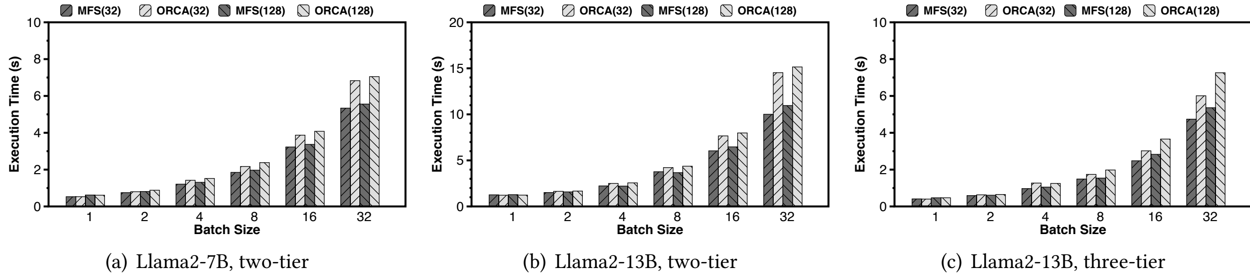
Validation on Qwen model family. In addition to the Llama2 model family, we also validate the effectiveness of knowledge precipitation with system-aware optimization on the Qwen1.5 model family. Due to computational limitations, we select only Qwen-14B and Qwen-7B from the Qwen model family. We transform Qwen-14B into a two-tier model. Table 6 shows the experimental results, the metrics for the corresponding MFS-14B and MFS-7B are comparable to their respective Qwen baseline models. We leave the fine-tuning and validation of more and larger model families as future work.

Generalization of knowledge precipitation in LLM family. Theoretically, the applicability of knowledge precipitation to large language models rests on the stacked, decoder-only Transformer architecture. Because contemporary LLMs (including but not limited to GPT, Gemini, Claude, and DeepSeek) adopt this design, the technique can be readily extended across model families. Moreover, larger models—with deeper stacks of Transformer blocks—tend to exhibit greater optimization stability and representational redundancy. Given that smaller models with fewer layers already train stably and accurately under knowledge precipitation, it is reasonable to expect equal or greater benefits when scaling the technique to larger models.

5.3 System Performance of MFS

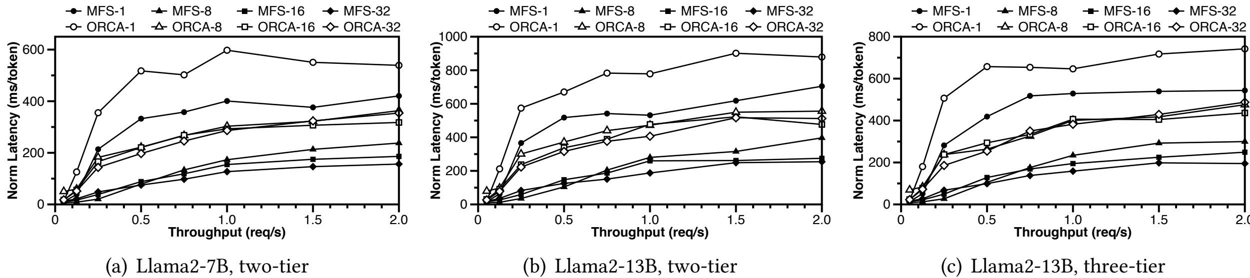
In this section, we integrate the multi-tier models into MFS and evaluate its system performance. We measure the improvements MFS brings to LLM serving in terms of batching, caching, and sampling.

Batching performance. In our evaluation, we initially compare the performance of MFS and Orca in terms of request execution time and token generation latency. We adopt a testing approach based on the experimental settings used in Orca, which are designed to assess the Orca engine's performance without the influence of its iteration-level scheduling. Specifically, we do not activate the Orca scheduler; instead, we inject the same batch of requests into the ORCA engine repeatedly until all requests are processed. This setup aims to



(a) Llama2-7B, two-tier (b) Llama2-13B, two-tier (c) Llama2-13B, three-tier

Figure 7. Median execution time of requests processed by MFS and ORCA. The label "MFS (n)" and "ORCA(n)" indicate results from MFS and ORCA when handling requests with n input tokens.



(a) Llama2-7B, two-tier (b) Llama2-13B, two-tier (c) Llama2-13B, three-tier

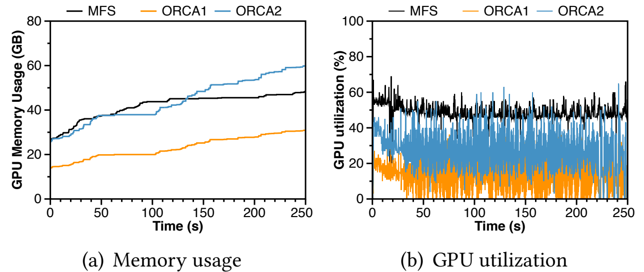
Figure 8. Relation of median end-to-end latency per generated token with throughput. The label "MFS-n" and "ORCA-n" indicate results from MFS and ORCA with n batch size.

emulate the behavior of conventional request-level scheduling, ensuring all requests in the batch have the same number of input tokens and generate the same number of output tokens. We measure the time taken to process the entire batch, not individual requests. Additionally, we simulate an end-to-end performance scenario similar to ORCA's, synthesizing a trace of client requests due to the lack of publicly available request traces for generative language models. Each request in our synthesized trace randomly specifies the number of input tokens and a max generated tokens attribute. For our experiments, we assume that the proportion of each model within a request batch is equal, facilitating a fair comparison across different system configurations and model tiers.

Figure 7 shows the execution time of a batch of requests. When the batch size is 1, the execution time of MFS and ORCA are similar due to the absence of batch processing. As the batch size increases, the improvement of MFS over ORCA becomes more significant.

Figure 8 shows the median end-to-end latency for each token. We can observe that MFS achieves a greater improvement in per-token generation latency. This is because the median latency for each token is primarily determined by the decoding phase, which is memory-bound. Therefore, MFS's efficient batch processing capability can be more fully utilized.

KV-cache sharing. We evaluate KV-cache sharing by configuring two distinct systems to imitate the inferencing process of MFS and Orca. System 1 is MFS, which enables KV-cache



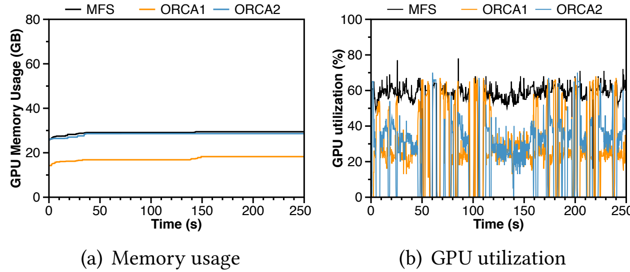
(a) Memory usage (b) GPU utilization

Figure 9. KV-cache sharing.

sharing with a two-tier model where each tier outputs tokens, and the hidden states from the first tier serve as inputs to the second tier. System 2 is Orca without KV-cache sharing, which duplicates the first tier model: one operates independently as a small model, and the other is paired with a second tier to form a larger model. For each inference request, two sub-requests with identical prompts are injected: one requiring only the first tier and the other requiring both tiers.

Tasks are injected following a Poisson distribution. Output data format includes the request ID, total number of tokens generated, the total time from request to completion, and a timestamp at the end of processing, useful for calculating throughput and establishing absolute request timing. The result is shown in Figure 9. For Orca, lines with orange color reflect data from the first tier of model and lines with blue color reflect data from the second tier of model, representing two sub-requests per inference. Under low system loads,

Orca performs comparably to MFS due to ample batch space for handling both tasks, though GPU usage in Orca is about 50% higher. As system load increases, Orca's batches fill up faster than those in MFS, significantly decreasing efficiency and increasing memory utilization.



(a) Memory usage (b) GPU utilization

Figure 10. Speculative sampling.

Speculative sampling. In this section, we evaluate the performance of speculative sampling adapted to our system. Speculative sampling allows our system to leverage the multi-tiered model structure effectively, using lower-tiered models for quick preliminary predictions and higher-tiered models for refining these predictions when necessary.

To measure the effectiveness of speculative sampling, we set up experiments that compare the response times and accuracy of token predictions across different tiers of our model. We also assess the impact of speculative sampling on overall system throughput and latency, particularly focusing on how well it integrates with our batching and caching mechanisms, meanwhile reusing the lower-tiered KV-Cache without recalculating.

In Figure 10, it can be seen that the memory utilization of Orca's large model is roughly the same as the total memory utilization of MFS, but Orca additionally takes up memory space for the small model. This is the extra memory waste caused by the inability to share model parameters and KV-Cache.

Besides, the GPU utilization of MFS consistently remains higher compared to ORCA. In the ORCA system, since the KV cache is not shared between the two-level models, the KV cache becomes invalid when switching models, leading to heavy memory access and decreased GPU utilization.

6 Related Work

KV-cache management. PagedAttention [22] observes the huge and growing KV-cache memory in LLM inference. Meanwhile, the KV-cache storing may introduce fragmentation and redundant duplication in GPU memory. PagedAttention builds a KV-cache management inspired from the paging technique used in operating systems. CachedAttention [13] implements a hierarchical KV caching system for multi-turn conversations of LLMs. InfiniGen [23] implements KV-cache management for long-text generation. These systems are designed to optimize KV-cache storage by minimizing waste and enabling flexible sharing of KV-cache resources both within and across requests, thereby substantially reducing memory utilization.

System-aware model optimization. FlashAttention series [10, 11, 29] speed up the multi-head attention calculation with io-aware operator execution. FasterTransformer [1] improves the GPU utilization for transformers with model-specific computation-aware GPU kernel implementations. Shibo et al. [36] propose a communication-aware decomposition technique to overlap communication with dependent computation.

LLM serving scheduling. FastServe [38] introduces a novel preemptive scheduling technique that allows for token-level preemption, effectively reducing delays caused by head-of-line blocking. DistServe [45] enhances LLM serving by separating the prefill and decoding stages into different computational processes. Llumnix [32] aims to react to heterogeneous and unpredictable requests by online rescheduling across multiple model instances. VTC [30] gives a definition of LLM serving fairness and proposes a Virtual Token Counter scheduling policy to ensure the fair processing of all clients.

7 Conclusion

In conclusion, this paper has explored various enhancements for serving large language models (LLMs), particularly focusing on multi-model architectures. These architectures address the need for higher efficiency and lower latency in AI-generated content applications like ChatGPT. By leveraging the structural uniformity across a series of LLMs, we have demonstrated that features such as batching and KV-cache sharing can be effectively implemented across multi-model systems, thereby optimizing resource utilization.

8 Acknowledgement

We thank the anonymous EuroSys reviewers and our shepherd, Arpan Gujarati, for their insightful comments. This work is supported in part by the Hong Kong RGC TRS T41-603/20R, TACC [41], HKUST-Inspur Cloud joint research project. Kai Chen is the corresponding author.

References

[1] 2020. Faster Transformer: https://github.com/NVIDIA/FasterTransformer. https://github.com/NVIDIA/FasterTransformer
[2] 2023. OpenAI ChatGPT: https://chat.openai.com. https://chat.openai.com
[3] 2023. OpenAI DALLE2: https://openai.com/dall-e-2. https://openai.com/dall-e-2
[4] 2024. Hugging face dataset guanaco-llama2: https://huggingface.co/datasets/mlabonne/guanaco-llama2. https://huggingface.co/datasets/mlabonne/guanaco-llama2
[5] 2024. Openlm-research Llama2-3B: https://huggingface.co/openlm-research/open_llama_3b_v2. https://huggingface.co/openlm-research/open_llama_3b_v2

[6] Yonatan Bisk, Rowan Zellers, Jianfeng Gao, Yejin Choi, et al. 2020. Piqa: Reasoning about physical commonsense in natural language. In *Proceedings of the AAAI conference on artificial intelligence*, Vol. 34. 7432–7439.

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

[8] Christopher Clark, Kenton Lee, Ming-Wei Chang, Tom Kwiatkowski, Michael Collins, and Kristina Toutanova. 2019. BoolQ: Exploring the surprising difficulty of natural yes/no questions. *arXiv preprint arXiv:1905.10044* (2019).

[9] Peter Clark, Isaac Cowhey, Oren Etzioni, Tushar Khot, Ashish Sabharwal, Carissa Schoenick, and Oyvind Tafjord. 2018. Think you have solved question answering? try arc, the ai2 reasoning challenge. *arXiv preprint arXiv:1803.05457* (2018).

[10] Tri Dao. 2023. Flashattention-2: Faster attention with better parallelism and work partitioning. *arXiv preprint arXiv:2307.08691* (2023).

[11] Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. 2022. Flashattention: Fast and memory-efficient exact attention with io-awareness. *Advances in Neural Information Processing Systems* 35 (2022), 16344–16359.

[12] Luciano Floridi and Massimo Chiriatti. 2020. GPT-3: Its nature, scope, limits, and consequences. *Minds and Machines* 30 (2020), 681–694.

[13] Bin Gao, Zhuomin He, Puru Sharma, Qingxuan Kang, Djordje Jevdjic, Junbo Deng, Xingkun Yang, Zhou Yu, and Pengfei Zuo. 2024. {Cost-Efficient} Large Language Model Serving for Multi-turn Conversations with {CachedAttention}. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*. 111–126.

[14] Pin Gao, Lingfan Yu, Yongwei Wu, and Jinyang Li. 2018. Low latency rnn inference with cellular batching. In *Proceedings of the Thirteenth EuroSys Conference*. 1–15.

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

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

[17] Yang He and Lingao Xiao. 2024. Structured Pruning for Deep Convolutional Neural Networks: A Survey. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 46, 5 (2024), 2900–2919. https://doi.org/10.1109/TPAMI.2023.3334614

[18] Dan Hendrycks, Collin Burns, Steven Basart, Andy Zou, Mantas Mazeika, Dawn Song, and Jacob Steinhardt. 2020. Measuring massive multitask language understanding. *arXiv preprint arXiv:2009.03300* (2020).

[19] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. 2015. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531* (2015).

[20] Neil Houlsby, Andrei Giurgiu, Stanislaw Jastrzebski, Bruna Morrone, Quentin De Laroussilhe, Andrea Gesmundo, Mona Attariyan, and Sylvain Gelly. 2019. Parameter-efficient transfer learning for NLP. In *International conference on machine learning*. PMLR, 2790–2799.

[21] Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, Weizhu Chen, et al. 2022. Lora: Low-rank adaptation of large language models. *ICLR* 1, 2 (2022), 3.

[22] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles*. 611–626.

[23] Wonbeom Lee, Jungi Lee, Junghwan Seo, and Jaewoong Sim. 2024. {InfiniGen}: Efficient Generative Inference of Large Language Models with Dynamic {KV} Cache Management. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. 155–172.

[24] Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, et al. 2024. Deepseek-v3 technical report. *arXiv preprint arXiv:2412.19437* (2024).

[25] Todor Mihaylov, Peter Clark, Tushar Khot, and Ashish Sabharwal. 2018. Can a suit of armor conduct electricity? a new dataset for open book question answering. *arXiv preprint arXiv:1809.02789* (2018).

[26] Yixin Nie, Adina Williams, Emily Dinan, Mohit Bansal, Jason Weston, and Douwe Kiela. 2019. Adversarial NLI: A new benchmark for natural language understanding. *arXiv preprint arXiv:1910.14599* (2019).

[27] Reiner Pope, Sholto Douglas, Aakanksha Chowdhery, Jacob Devlin, James Bradbury, Jonathan Heek, Kefan Xiao, Shivani Agrawal, and Jeff Dean. 2023. Efficiently scaling transformer inference. *Proceedings of Machine Learning and Systems* 5 (2023).

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

[29] Jay Shah, Ganesh Bikshandi, Ying Zhang, Vijay Thakkar, Pradeep Ramani, and Tri Dao. 2024. FlashAttention-3: Fast and Accurate Attention with Asynchrony and Low-precision. *arXiv preprint arXiv:2407.08608* (2024).

[30] Ying Sheng, Shiyi Cao, Dacheng Li, Banghua Zhu, Zhuohan Li, Danyang Zhuo, Joseph E Gonzalez, and Ion Stoica. 2024. Fairness in serving large language models. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. 965–988.

[31] David So, Wojciech Mańke, Hanxiao Liu, Zihang Dai, Noam Shazeer, and Quoc V Le. 2021. Searching for efficient transformers for language modeling. *Advances in neural information processing systems* 34 (2021), 6010–6022.

[32] Biao Sun, Ziming Huang, Hanyu Zhao, Wencong Xiao, Xinyi Zhang, Yong Li, and Wei Lin. 2024. Llumnix: Dynamic Scheduling for Large Language Model Serving. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. USENIX Association, Santa Clara, CA, 173–191. https://www.usenix.org/conference/osdi24/presentation/sun-biao

[33] Gemma Team, Thomas Mesnard, Cassidy Hardin, Robert Dadashi, Surya Bhupatiraju, Shreya Pathak, Laurent Sifre, Morgane Rivière, Mihir Sanjay Kale, Juliette Love, et al. 2024. Gemma: Open models based on gemini research and technology. *arXiv preprint arXiv:2403.08295* (2024).

[34] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. 2023. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288* (2023).

[35] Elena Voita, David Talbot, Fedor Moiseev, Rico Sennrich, and Ivan Titov. 2019. Analyzing Multi-Head Self-Attention: Specialized Heads Do the Heavy Lifting, the Rest Can Be Pruned. *CoRR* abs/1905.09418 (2019). arXiv:1905.09418 http://arxiv.org/abs/1905.09418

[36] Shibo Wang, Jinliang Wei, Amit Sabne, Andy Davis, Berkin Ilbeyi, Blake Hechtman, Dehao Chen, Karthik Srinivasa Murthy, Marcello Maggioni, Qiao Zhang, et al. 2022. Overlap communication with dependent computation via decomposition in large deep learning models. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*. 93–106.

[37] Yiding Wang, Kai Chen, Haisheng Tan, and Kun Guo. 2023. Tabi: An efficient multi-level inference system for large language models. In *Proceedings of the Eighteenth European Conference on Computer Systems*. 233–248.

[38] Bingyang Wu, Yinmin Zhong, Zili Zhang, Gang Huang, Xuanzhe Liu, and Xin Jin. 2023. Fast distributed inference serving for large language models. *arXiv preprint arXiv:2305.05920* (2023).

[39] Ji Xin, Raphael Tang, Jaejun Lee, Yaoliang Yu, and Jimmy Lin. 2020. DeeBERT: Dynamic early exiting for accelerating BERT inference. *arXiv preprint arXiv:2004.12993* (2020).

[40] Ji Xin, Raphael Tang, Jaejun Lee, Yaoliang Yu, and Jimmy Lin. 2020. DeeBERT: Dynamic early exiting for accelerating BERT inference. *arXiv preprint arXiv:2004.12993* (2020).

[41] Kaiqiang Xu, Decang Sun, Hao Wang, Zhenghang Ren, Xinchen Wan, Xudong Liao, Zilong Wang, Junxue Zhang, and Kai Chen. 2025. Design and Operation of Shared Machine Learning Clusters on Campus. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*. 295–310.

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

[43] Rowan Zellers, Ari Holtzman, Yonatan Bisk, Ali Farhadi, and Yejin Choi. 2019. Hellaswag: Can a machine really finish your sentence? *arXiv preprint arXiv:1905.07830* (2019).

[44] Biao Zhang and Rico Sennrich. 2019. Root mean square layer normalization. *Advances in Neural Information Processing Systems* 32 (2019).

[45] Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. 2024. Distserve: Disaggregating prefill and decoding for goodput-optimized large language model serving. *arXiv preprint arXiv:2401.09670* (2024).