Sparsity-Aware Scheduling for Efficient MoE-Based LLM Serving on Edge Servers

Abstract

To support real-time and domain-customized edge intelligent applications, LLMs are increasingly deployed on edge computing servers. However, we observe that current MoE-based LLM serving systems, inherited from the cloud-based architectures, face a critical sparsity-vanishing problem when serving at the edge: tokens within an input batch often activate a large fraction of experts, which incurs significant I/O overhead from frequent expert swapping between CPU and GPU memory, ultimately degrading system throughput. In this work, we propose `Saber`, a sparsity-aware scheduling framework optimized for efficient MoE-based LLM serving on edge servers. The core idea of `Saber` is to introduce a group-level batching strategy that dynamically identifies requests with similar activated experts, and coalesces them to construct sparsity-preserving input batches. To ensure efficient online execution, activation patterns are precomputed and clustered offline, enabling fast request grouping through a lightweight classifier at runtime. Additionally, `Saber` incorporates a utility-driven priority scheduling mechanism to maximize inference throughput while maintaining low latency by intelligently adjusting group priorities. Experiments on representative MoE-based LLMs demonstrate that `Saber` significantly outperforms existing sparsity-agnostic schedulers, achieving up to 3.8× lower batch execution time and 3.27× higher throughput at comparable latency levels.

1 Introduction

Large language models (LLMs) empower a variety of emerging edge intelligent applications, such as personal AI assistant [39, 59], smart sensing [49, 64], autonomous driving [25], and healthcare [43, 44]. To deliver responsive and domain-customized services, it is imperative to run LLM inference directly at the edge [32, 46, 65], typically on on-premises edge servers, which can be gateways or strong compute nodes within local edge networks. For example, in smart homes, an LLM deployed on the central hub can offer inference services to laptops, tablets, and IoT-enabled appliances; in healthcare, edge servers hosting LLMs can process data streams from medical IoT devices to provide real-time analysis and alerts. To facilitate such deployments, edge hardware vendors have presented specific hardware and development kits, such as Azure Stack Edge [4] and AWS Outposts Server [3].

However, edge servers are typically provisioned with limited compute resources (*e.g.*, weak GPUs), which cannot afford to deploy complete LLMs in GPU memory for inference. A common architecture to reduce resource requirements is applying sparsely-activated mixture-of-experts (MoE) models [14, 37, 48] which activates only a small subset of parameters, namely experts, for each input token during inference. As such, the bulk model weights can be offloaded to the CPU memory or disk, and further be loaded into GPU memory for inference on demand, thereby significantly reducing both runtime computational cost and memory requirements.

Efficiency bottleneck of MoE serving at the edge. Edge servers must handle concurrent requests from multiple edge devices. Sequentially processing each request incurs large queuing delays, and thus existing LLM serving systems typically employ batched inference, processing multiple requests simultaneously to improve computation efficiency and throughput. However, we observe a critical efficiency bottleneck arising due to the *sparsity-vanishing phenomenon*: While each token individually activates only a few experts, the aggregation of tokens from multiple requests in a batch can collectively activate a large fraction of experts, undermining the sparse activation property. Consequently, without a much larger GPU memory to accommodate this expanded set of active experts, the system must perform frequent I/O operations to swap experts between CPU and GPU memory. These excessive I/O operations introduce additional latency, severely reducing inference speed and system throughput.

Limitations of existing LLM serving systems. Prior works optimized batch scheduling strategies and inference pipelines to improve inference efficiency, but failed to handle this sparsity vanishing problem. (1) *Batch scheduling optimizations* [9, 13, 26, 70] focused on runtime request scheduling according to deployment characteristics. However, these strategies primarily optimize for serving generic LLMs on cloud servers, which are less effective to serve MoE models on resource-constrained edge servers due to their sparsity-agnostic designs. (2) *Inference pipeline optimizations* [17, 29, 34, 69] attempted to maximize the temporal overlap between computation and I/O processes to reduce inference latency through predicting and preloading active LLM parameters. However, the sparsity-vanishing problem incurs excessive I/O overhead, which is difficult to fully overlap with computation. This blocks the subsequent computation tasks, leading to increased inference latency and degraded throughput.

Motivation. The above limitations motivate us to develop `Saber`, a sparsity-aware scheduling framework designed to mitigate excessive I/O overhead caused by the sparsity vanishing issue, and improve LLM serving efficiency and throughput on edge servers. We observe that the sequential

batching strategy, though simple and fair, leaves tokens with similar activation patterns scattered across different batches, leading to frequent memory swapping for the same experts. Our key idea is to identify the tokens with similar activation patterns, manage requests as similarly-activated groups, and batch them in a way that activates only a small subset of experts. This approach optimizes the utilization of expert loading, while maintaining the computational efficiency through batched inference for MoE-based LLM serving.

Design challenges and proposed solutions. To realize Saber, we address three key design challenges arising at the level of token, request, and request group, respectively.

• *Efficiently identifying **tokens** with similar activation patterns.* Online scheduling must efficiently and accurately determine the tokens that activate similar experts (*i.e.*, follow similar activation paths). Predicting the activation path for a given token is complex due to the need to capture the contextual semantics of the input sequence. While large models provide accurate predictions, they are computationally expensive; small models are more efficient but less accurate, leading to unexpected expert loading during inference. *To address this challenge,* Saber *shifts this complexity to an offline profiling stage.* Leveraging the sparse and structured nature of activation path distributions in a given LLM, we identify valid activation paths, perform path clustering, and train a lightweight DNN to classify tokens into path clusters directly. This substantially reduces prediction complexity, thereby enabling efficient and accurate online execution.

• *Effectively grouping **requests** in different inference stages.* Each request goes through two stages during inference: prefilling and decoding. Unlike decoding stage that input only one token in each iteration, prefilling stage involves processing multiple tokens in the input prompt. Processing all the prompt tokens in a batch maximizes computation efficiency but suffers from sparsity vanishing, while processing tokens sequentially preserves sparsity but leads to excessive inference iterations and high latency. *To tackle this challenge, we leverage the temporal locality of expert activations*, *i.e.*, adjacent tokens tend to activate the same experts, and introduce an adaptive prompt splitting approach to split prompts into smaller chunks with similar activation patterns. Then, we employ hybrid batching that enables scheduling prefilling chunks with similarly-activated decoding requests, thereby preserving activation sparsity while minimizing latency.

• *Scheduling **groups** for high throughput and low latency.* Scheduling the grouped requests to optimize throughput without sacrificing latency is non-trivial. Throughput-oriented scheduling prioritizes large groups to maximize throughput, but potentially causes a high tail latency, as the groups associated with infrequent activation paths may experience starvation. In contrast, latency-oriented scheduling mitigates starvation by prioritizing long-waiting groups, but reduces throughput due to unsaturated batches. Saber *addresses this challenge with a priority scheduling mechanism that jointly considers throughput and latency through a unified utility metric.* This metric favors large groups while dynamically increasing the priority of long-waiting groups to prevent starvation. Furthermore, Saber enhances the throughput-latency trade-off by adaptively merging groups based on their activation path similarities, allowing flexible adjustments between batch size and I/O overhead.

Contributions of this work are summarized as follows:

• We identify the sparsity-vanishing problem as a key efficiency bottleneck in MoE-based LLM inference serving systems on resource-constrained edge servers.

• We propose Saber, a sparsity-aware scheduling framework to support efficient MoE model inference on edge servers. By intelligently identifying, grouping, and batching requests with similar activation paths, Saber preserves activation sparsity under batched inference, thereby reducing I/O overhead and improving inference efficiency.

• We implement and evaluate Saber using three types of MoE-based LLMs: Switch Transformers, LLaMA-MoE, and Qwen1.5-MoE. Experiment results show that Saber enables high-throughput, low-latency LLM services, significantly outperforming sparsity-agnostic scheduling policies. Specifically, Saber reduces I/O overhead by up to 1.95× per input batch, and achieves up to 3.8× lower batch execution time and 3.27× higher throughput at the same level of latency.

2 Background and Motivation

2.1 Edge Computing for LLM Services

Deploying LLMs on on-premises edge servers is increasingly important for supporting responsive and context-aware intelligent applications in local environments, such as smart home hubs for local automation (*e.g.*, Xiaomi Miloco [6, 38]), intelligent factory floors for real-time industrial AIoT (*e.g.*, NVIDIA Jetson Orin [7]), and healthcare clinics for on-premises monitoring (*e.g.*, Azure Stack Edge [4] with medical AI). Notably, edge deployment offers several key advantages: (1) *Low latency*: eliminates long-distance cloud communication for fast, stable responses. (2) *Customization*: enables fine-tuned LLMs for domain-specific accuracy and privacy. (3) *Multi-device service*: a shared server supports heterogeneous edge devices without requiring per-device LLMs. (4) *Resource efficiency*: reduces bandwidth and cloud token budgets.

To balance compute capability, power efficiency, and deployment cost, edge servers are commonly equipped with a single consumer- or edge-grade GPU for LLM inference. For instance, the Premio LLM-1U [8] and smart home platforms like Home Assistant [5] (*e.g.*, via an RTX 3060) adopt this setup for local LLM automation, while NVIDIA Jetson devices can support smart retail and industrial applications. These servers commonly serve requests from nearby devices

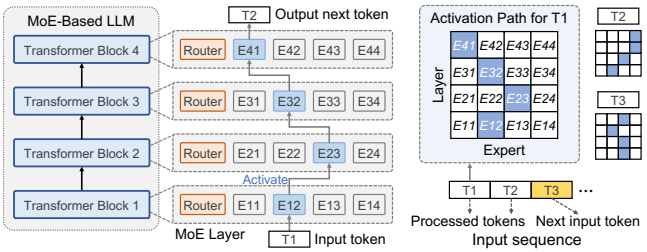

Figure 1: The architecture of MoE-based LLMs (left), and the activation path for an input token (right).

(*e.g.*, tens of sensors or user terminals). Edge devices transmit compact prompts to the local LLM and receive rapid responses, enabling scalable and efficient edge intelligence.

2.2 MoE-Based Large Language Models

Sparsely-activated MoE models. Given the high resource demands of LLM inference, sparsely-activated MoE models [37, 48] are particularly suitable for resource-limited edge servers. As illustrated in Figure 1, the standard feed-forward networks (FFNs) within the LLM are replaced with MoE layers. Each MoE layer comprises multiple FFNs, termed as experts, and a token router. For an input token, the router computes routing scores across all experts, and only activates top-k experts with the highest routing scores. The activated experts across MoE layers for an input token forms an activation path (*e.g.*, $\{E12, E23, E32, E41\}$ for $T1$) which engages only a small subset of the LLM's parameters, thereby enabling edge platforms to deploy LLMs without requiring resources proportional to the full model size.

MoE model inference. Similar to general generative LLMs, MoE-based LLMs follow an autoregressive inference paradigm. Given a prompt represented as a sequence of tokens, the LLM generates a series of new tokens as output. A complete generation process goes through two stages: prefilling and decoding. Specifically, in the prefilling stage, the LLM processes all tokens in the prompt parallelly in one iteration (*i.e.*, one forward pass of the LLM), and generates a new token next to the prompt. Then, in the decoding stage, the LLM takes the token generated in the last iteration as input for the current iteration, and generates the next token. This process is repeated for multiple iterations until a special token "<EOS>" is generated indicating the end of generation, or the sequence length has reached a maximum number.

2.3 Serving MoE Models on Edge Servers

Figure 2 illustrates a typical LLM inference serving system, which comprises two main components: a *request scheduler* and an *inference engine*. In each inference iteration, the scheduler selects a batch of requests from the request waiting pool, creates an input data batch, and sends it to the inference engine. The inference engine then performs LLM inference to generate new tokens, and returns the results to the scheduler.

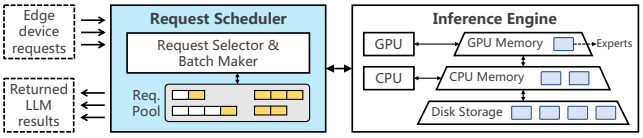

Figure 2: Illustration of an edge LLM inference serving system using expert offloading technique.

These results are either queued for future inference iterations or sent back to users if the generation is complete. To efficiently handle requests from multiple devices, LLM serving systems commonly employ batched inference, which combines multiple requests into a single large input tensor for inference computation. This method improves computation efficiency and system throughput, as LLM inference relies on tensor operations, and inference engines are optimized for processing large tensors using parallel computation units.

Expert-offloading-based LLM inference engine. Due to limited resources, edge servers can not deploy complete LLMs in GPU memory for inference. Fortunately, leveraging the sparse activation property of MoE, it is feasible to serve MoE-based LLMs on edge platforms through expert offloading [11, 29, 34, 68, 69]. As illustrated in Figure 2, this technique maintains only the active expert weights in GPU memory, while offloading inactive experts to CPU memory or SSDs. During inference, the required experts are dynamically loaded into GPU memory on demand. Since the experts constitute the majority of an LLM's weights, the experts offloading technique significantly reduces the GPU memory footprint. For instance, for a Switch Transformer [21] with 32 experts per layer, the sparse MoE layers account for 89.7% of the total model weights, and expert offloading can reduce memory usage for model weights by up to 7.63×.

2.4 Limitations of Existing Systems

Although current LLM serving systems are highly optimized for cloud environments, they fall short of serving MoE-based LLMs on edge servers due to their sparsity-agnostic designs, suffering from the following *sparsity-vanishing problem*.

Sparsity-vanishing problem. This problem arises when multiple tokens are processed in parallel, which occurs during prompt prefilling and batched decoding. As illustrated in Figure 3, each input token activates only a small subset of experts (an activation path), while tokens from different requests in a batch tend to activate diverse subsets, causing a large fraction of experts to be activated for the batch. As a result, the sparsity property diminishes as more tokens are processed in parallel, leading to excessive data transfer overhead for swapping in and out expert weights between CPU and GPU memory. We use Switch Transformers to exemplify this issue. As shown in Figure 4 (left), the number of activated experts grows rapidly with the number of input tokens. For instance, in Switch-64, 32 tokens activate over 10 experts per layer, incurring over 1 GB of data transfer. This

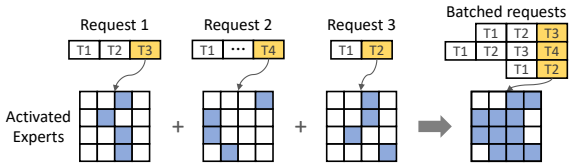

Figure 3: Illustration of the sparsity-vanishing problem when processing multiple tokens in parallel.

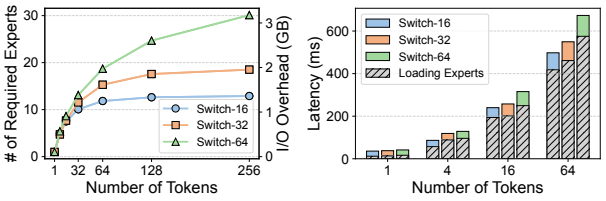

Figure 4: Impact of sparsity vanishing. Expert weights are offloaded to CPU memory and loaded into GPU on demand. Switch-n denotes n experts per MoE layer.

overhead further escalates to 3 GB for 256 tokens, which is 30.1× higher than the single-token sparse-activation case. Furthermore, Figure 4 (right) indicates that expert loading dominates inference latency (*e.g.*, averaging 84.5% with 64 tokens), making it a critical bottleneck for edge LLM serving.

Limitations of existing schedulers. Various request schedulers have been developed to improve throughput and reduce latency in LLM serving systems [9, 26, 35, 70]. For example, Orca [70] proposes iteration-level scheduling to eliminate redundant computation and reduce the waiting time inherent in request-level scheduling. Recognizing the distinct characteristics between prefilling and decoding, Sarathi-Serve [9] and DeepSpeed-FastGen [26] divided prefilling prompts into uniformly sized chunks to balance batch computation loads and mitigate pipeline bubbles. Despite these advances, existing schedulers typically select requests sequentially from the waiting pool. Their constructed batches result in sparsity vanishing, thereby degrading throughput due to excessive I/O costs when deployed on edge servers.

Limitations of existing inference engines. Another line of work optimizes inference pipelines to mitigate the performance degradation due to expert loading [29, 50, 52, 66, 68, 69], employing techniques such as expert preloading and efficient compute-I/O parallelism. The primary goal is to overlap computation and data transfer temporally to accelerate inference (Figures 5 (a)-(b)). For instance, Pre-gated MoE [29] and EdgeMoE [69] predict the required experts for the next MoE layer, and preload their weights before they are needed for computation. However, these methods remain less effective for batched inference, as sparsity-vanishing results in massive expert weights to load. This data transfer process is time-consuming, which is hard to overlap with computation (Figure 5 (c)). Consequently, this blocks the subsequent computation tasks, and degrades system performance [11].

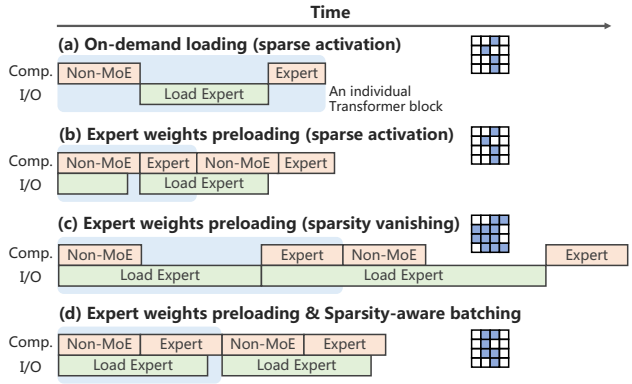

Figure 5: The inference pipeline under different expert loading strategies and activation patterns.

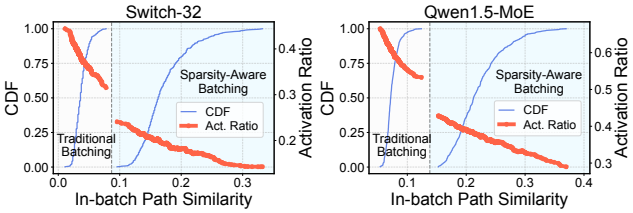

Figure 6: CDF of path similarity and the corresponding activation ratios for $batch\_size = 16$ and $pool\_size = 48$.

Our motivation. The limitations of existing systems motivate the design of a sparsity-aware framework for high-throughput and low-latency inference at the edge. Saber introduces a sparsity-aware scheduling strategy that explicitly considers the active experts required by each request. Instead of batching requests sequentially, the scheduler groups requests with similar activation paths in each iteration. Such sparsity-preserving batches significantly reduce expert loading overhead, thereby improving inference efficiency (Figure 5 (d)). To quantify the grouping potential, we analyze the in-batch path similarity (defined as average pairwise overlapping coefficient) and corresponding activation ratios for 1k batches constructed by traditional and sparsity-aware strategies. As shown in Figure 6, the sparsity-aware strategy substantially increases path similarity, and reduce the number of activated experts. For example, in Qwen1.5-MoE, traditional batching yields a low in-batch path similarity of 0.074, and activates 60.7% of experts per batch. In contrast, sparsity-aware batching increases the similarity to 0.22, and reduces activation ratio to 37.7%, resulting in a 6.1 GB I/O overhead reduction per batch. This trend is consistent across various workloads (Appendix A).

3 Design Challenges and Opportunities

The goal of Saber is to preserve activation sparsity during batched inference while balancing throughput and tail latency. To achieve this, we address three key challenges.

***Challenge 1**: How to efficiently identify tokens with similar activation paths under complex semantic-path correlations?*

Opportunity 1*: Cluster-level token classification (§5.1).* To address this challenge, we propose an alternative to exact per-token path prediction by exploiting the structured nature of activation patterns in LLMs. Specifically, the activation paths of an LLM are not uniformly distributed across the full path space, but exhibit strong regularities reflected in: (1) the space of valid activation paths is highly sparse. For example, in Switch-16, only approximately 2% of all possible activation paths are observed to be valid. This sparsity arises because experts are trained to collaborate on specific subtasks [76]. Thus, not every subset of experts can form a valid combination for processing input tokens. (2) Expert activations are often correlated. Certain experts are frequently co-activated, forming stable clusters of similar activations paths that correspond to particular domains or input structures.

Leveraging these insights, we can profile the LLM offline to identify and cluster commonly occurring activation patterns. At runtime, instead of predicting precise activation paths, incoming tokens are classified into these precomputed clusters using lightweight models. This approach substantially reduces the complexity of runtime scheduling and maintains high accuracy with minimal overhead, enabling scalable and efficient inference in MoE-based systems.

Challenge 2*: How to handle the expanded expert activation in prefilling requests that have multiple prompt tokens?*

Opportunity 2*: Leveraging temporal locality for prompt chunking (§5.2).* An opportunity arises from the observation that expert activation patterns in MoE models exhibit strong temporal locality: adjacent tokens in a prompt tend to activate the same set of experts during inference [18, 28, 30]. This property enables efficient prompt chunking, wherein the input prompt of a prefilling request is divided into contiguous token segments that are likely to share similar activation paths. By scheduling and processing each chunk in an input batch, the system can retain activation sparsity while avoiding the inefficiencies of token-by-token execution. This approach reduces redundant memory access and expert loading, thereby improving computation efficiency without compromising latency.

Challenge 3*: How to schedule similarly-activated request groups while balancing throughoput and latency?*

Opportunity 3*: Utility-based priority scheduling (§5.3).* Given the identified request groups, we can estimate key statistics for each group, including the number of requests, required experts, and memory demands (I/O overhead). These statistics allow us to formulate online scheduling as a multi-objective optimization problem aiming to maximize throughput while minimizing latency and avoiding request starvation. Recognizing the dynamic nature of incoming workloads, we design a unified utility function that synthesizes these statistics to guide scheduling decisions efficiently. This utility-driven mechanism enables adaptive prioritization: it ensures that smaller, latency-sensitive request groups are not indefinitely delayed, while still favoring large, compute-efficient batches when conditions permit. Ultimately, this provides a scalable solution for maintaining high responsiveness within sparsity-aware MoE inference systems.

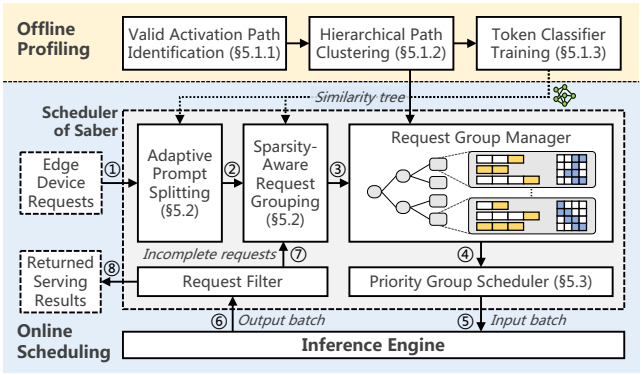

Figure 7: Overview of the Saber framework.

4 Saber Overview

Figure 7 illustrates the overview of the Saber framework. Specifically, Saber operates across three levels: (1) *Offline token-level classification (§5.1).* Saber profiles the activation patterns of the target LLM by performing valid activation path identification, hierarchical path clustering, and token classifier training. As a result, the offline stage yields a lightweight token classifier and a similarity tree, which are subsequently used to guide online scheduling. (2) *Online request-level grouping (§5.2).* Utilizing the token classifier, Saber's scheduler splits input prompts into smaller chunks, and then organizes both the prefilling chunks and decoding requests into groups based on their classified path clusters. Requests within the same group share similar activation patterns and are batched together to maintain sparsity and computation efficiency. (3) *Online group-level scheduling (§5.3).* In each iteration, a priority group scheduler selects a group for execution, aiming to dynamically balance throughput and latency. Moreover, leveraging the similarity tree that encodes the cluster similarity structure, Saber supports adaptive group merging to trade off between batch size and I/O overhead.

We describe the online workflow of Saber as follows.

- *Request pre-processing.* Before an iteration, Saber splits the prompts of newly arrived requests into smaller chunks (①), and then organizes them along with decoding requests into similarly-activated groups, waiting for scheduling (②-③).
- *Group-level request scheduling.* In each iteration, the group scheduler selects the group with the highest priority, considering both throughput and latency, to form an input batch. This batch is then sent to the inference engine (④-⑤).
- *LLM inference.* The engine performs inference with the batched input for one iteration. The output generated tokens are returned to the request scheduler (⑥).

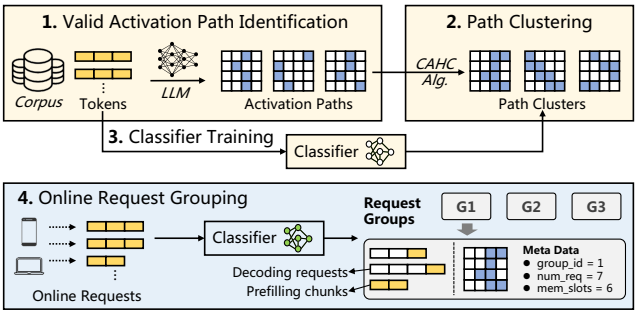

Figure 8: Workflow of offline activation pattern profiling and online request grouping.

• *Request post-processing.* The scheduler reassigns incomplete requests to the grouping component (⑦), waiting for scheduling in future iterations, while completed requests are removed from the batch and returned to edge devices (⑧).

5 Saber Design

5.1 Token-Level Activation Profiling

Figure 8 illustrates the overall workflow of offline profiling and the rationale of request grouping. Specifically, we first identify valid activation paths out of all possible paths to vastly reduce the operation space (§5.1.1). Then, we cluster the valid activation paths according to their similarity to form a set of path clusters under the given memory budget. Each cluster contains a set of similar activation paths, with their union still preserving sparse activation property (§5.1.2). With these well-defined path clusters, we train a lightweight but accurate classifier, which only needs to classify a given token to a specific path cluster rather than predicting its exact activation path (§5.1.3). This approach substantially reduces the problem difficulty compared to the exact path prediction according to the theoretical analysis in [71], and thus can be accurately handled by a lightweight model.

5.1.1 Valid Activation Path Identification. An activation path refers to a subset of experts across the MoE layers in an LLM that are simultaneously activated when processing a specific input token. Formally, given an LLM with L MoE layers and N experts per layer, the activation path (or path for short) for a token t is represented by a binary matrix $\boldsymbol{p}^{(t)} = [p_{ln}] \in \{0,1\}^{L\times N}$, where $p_{ln} = 1$ indicates expert n in the l-th MoE layer is activated, and 0 otherwise. The complete set of possible activation paths forms a path space $\mathcal{U}$, which grows exponentially with $O(N^L)$.

To constrain this space, we identify a set of valid path set $\mathcal{V} \subset \mathcal{U}$ for a given LLM. In the offline stage, we leverage a public corpus containing data from a wide range of tasks and domains to construct this valid activation path set $\mathcal{V}$. Specifically, starting from an empty set, we sample sentences in the corpus to feed into the LLM, and record the activation paths. The activation path $\boldsymbol{p}$ of each token is added to $\mathcal{V}$, with its frequency $Freq(\boldsymbol{p})$ being recorded or updated as well. After the construction of $\mathcal{V}$, the empirical distribution over paths is estimated as $Pr(\boldsymbol{p}) = Freq(\boldsymbol{p})/\sum_{\boldsymbol{p}'\in\mathcal{V}} Freq(\boldsymbol{p}')$. To reduce redundancy, the paths with activation probability below τ (*e.g.*, 10^{-4}) are discarded, further shrinking $\mathcal{V}$ while preserving performance-critical paths.

5.1.2 Similarity-Based Path Clustering. Given the set of valid activation paths $\mathcal{V}$, we formulate a constrained clustering problem to group similar paths into a set of clusters:

$$\begin{aligned} \min_{C} \quad & L(C) = \sum_{i=1}^{|\mathcal{V}|} \min_{\boldsymbol{c}\in C} d(\boldsymbol{c}, \boldsymbol{p}^{(i)}), \text{ (Clustering Loss)} \\ \text{s.t.} \quad & \sum_{l=1}^{L}\sum_{n=1}^{N} c_{ln} \le B, \forall \boldsymbol{c} \in C, \text{ (Memory Constraint)} \\ & \sum_{\boldsymbol{p}\in\boldsymbol{c}} Pr(\boldsymbol{p}) \le P, \forall \boldsymbol{c} \in C, \text{ (Act. Prob. Constraint)} \\ & |C| \le N_c. \text{ (Cluster Cardinality Constraint)} \end{aligned} \tag{1}$$

The clustering objective minimizes the total distance from each activation path $\boldsymbol{p}^{(i)}$ to its nearest cluster center $\boldsymbol{c}$ among the set of cluster centers C. The first constraint ensures that the total number of memory slots[1] occupied by any cluster does not exceed a predefined budget B, thereby preventing excessive expert loading during LLM inference. The second constraint regulates the cumulative activation probability of paths assigned to a cluster, encouraging a balanced distribution of request traffic during deployment. This balance is also beneficial for the training of the request classifier (Section 5.1.3). Finally, the third constraint limits the total number of clusters to an upper bound N_c.

Here, the distance function $d(\boldsymbol{p}^{(i)}, \boldsymbol{p}^{(j)})$ is defined as:

$$d(\boldsymbol{p}^{(i)}, \boldsymbol{p}^{(j)}) = \sum_{l,n} \max\{p_{ln}^{(i)}, p_{ln}^{(j)}\} - \max\{\sum_{l,n} p_{ln}^{(i)}, \sum_{l,n} p_{ln}^{(j)}\}, \tag{2}$$

which quantifies the increased number of memory slots when processing both paths $\boldsymbol{p}^{(i)}$, $\boldsymbol{p}^{(j)}$ in a single batch compared to processing them separately. A cluster center $\boldsymbol{c} \in \{0,1\}^{L\times N}$ is defined as the element-wise logical OR over all paths assigned to this cluster[2]: $\boldsymbol{c} = \vee_{\boldsymbol{p}\in\boldsymbol{c}}\boldsymbol{p}$, representing the union of experts needed for processing these paths in one batch. This center can be interpreted as a "wider" activation path, and Equation 2 applies equally to calculate the distance between cluster centers and paths: $d(\boldsymbol{c}, \boldsymbol{p})$ and $d(\boldsymbol{c}^{(i)}, \boldsymbol{c}^{(j)})$.

While solving this constrained integer programming provides high-quality clustering results, it has prohibitively high complexity for numerous activation paths in practice. In addition, online dynamic workloads further expect the awareness of the similarity between the clusters to enable flexible

[1] A memory slot refers to the memory space occupied by an expert.

[2] For notation simplicity, we write $\boldsymbol{p} \in \boldsymbol{c}$ to indicate that path $\boldsymbol{p}$ is assigned to the cluster whose center is $\boldsymbol{c}$; formally, this implies that $\boldsymbol{c} = \arg\min_{\boldsymbol{c}'\in C} d(\boldsymbol{c}', \boldsymbol{p})$.

trade-offs between batch size and memory cost (we detail this scheduling in Section 5.3).

Constrained agglomerative hierarchical clustering. To address this, we propose a hierarchical clustering-based algorithm [16, 41] that constructs path clusters at multiple granularities while satisfying both memory and probability constraints. This algorithm progressively optimizes the objective function within the feasible region by iteratively selecting and merging the most similar clusters—those yielding the minimal increase in clustering loss, thus promoting high intra-cluster similarity throughout the clustering process.

As detailed in Algorithm 1, this algorithm begins by initializing each path as a cluster and constructing a cost matrix that quantifies the merging cost for each pair of clusters (Line 1). To steer the clustering toward low-loss and maintaining balanced activation probabilities, we apply a penalty term in the cost definition of m_{ij}, where the first term captures the similarity between clusters, and the penalty term discourages merging already large clusters, weighted by a coefficient α to control its impact. At each iteration, the algorithm selects the cluster pair with the lowest merging cost that satisfies the memory and probability constraints (Line 4), merges them into a new cluster (Line 12), and updates the cluster set and cost matrix accordingly (Lines 13-14). This merging process continues until no further feasible merges can be found, at which point the resulting cluster C^* is recorded (Lines 5-7). Importantly, the algorithm does not terminate after identifying C^*. Instead, it continues merging remaining clusters, regardless of constraints, to construct a similarity tree $\mathcal{T}$ (Lines 9-11), which is later used for inter-group scheduling detailed in Section 5.3.

5.1.3 Lightweight Token Classifier Training. Next, we train a token classifier that maps tokens to the clusters in C^*. This classifier should be both lightweight for efficient online execution and accurate to avoid unexpected data transfer during online inference.

To achieve this goal, we employ a lightweight DNN with a partial attention mechanism. The model first applies a standard attention layer to extract the contextual information of a sequence, followed by a two-layer MLP to classify input tokens into specific groups. We observe that the primary inference overhead in this classifier is the attention layer, whose computational cost grows quadratically with the sequence length. Prior studies have revealed that a small subset of tokens typically encodes the critical context information of a sequence [17, 33, 62], indicating an opportunity to simplify attention calculation. In particular, a token primarily attend to nearby tokens and several initial tokens (*i.e.*, attention sink [63]). Leveraging this property, we employ a sink and window attention mechanism that only selects n_s tokens out of the total n tokens in the sequence, containing n_i sink tokens and n_r most recent tokens indexed from $(n - n_r)$ to n, as the attention input. Our experiments confirm that this method preserves contextual information for accurate token classification while reducing computation overhead.

Algorithm 1: Constrained AHC for path clustering

Input: Activation path set $\mathcal{V}$, cost coefficient λ
Output: Cluster set C^*, cluster similarity tree $\mathcal{T}$

/* 1. Initialize the cluster set and the cost matrix. */
1 Initialize $C \leftarrow \mathcal{V}$, and $\boldsymbol{M} \in \mathbb{R}^{|C|\times|C|}$,
  where $m_{ij} = d(\boldsymbol{c}^{(i)}, \boldsymbol{c}^{(j)}) + \alpha(Pr(\boldsymbol{c}^{(i)}) + Pr(\boldsymbol{c}^{(j)}))$;
2 $save\_clusters\_flag = True$;
3 **while** $|C| > 1$ **do**
  /* 2. Select the lowest-cost cluster pair with their merging satisfying the constraints in Eq. (1). */
4 $(i, j) \leftarrow \arg\min_{i,j} \boldsymbol{M}_{ij}$ s.t. MergeFeasible$(\boldsymbol{c}^{(i)}, \boldsymbol{c}^{(j)})$;
5 **if** (i, j) *is None* **then**
6 **if** $save\_clusters\_flag == True$ **then**
7 $C^* \leftarrow C$; // Save target clusters
8 $save\_clusters\_flag \leftarrow False$;
9 **else** // Continue and record similarity structure
10 $(i, j) \leftarrow \arg\min_{i,j} \boldsymbol{M}_{ij}$;
11 $\mathcal{T} \leftarrow \mathcal{T} \cup (\boldsymbol{c}^{(i)}, \boldsymbol{c}^{(j)}, \text{Merge}(\boldsymbol{c}^{(i)}, \boldsymbol{c}^{(j)}))$;
  /* 3. Merge the selected pair of clusters. */
12 $\boldsymbol{c}^{\text{new}} \leftarrow \text{Merge}(\boldsymbol{c}^{(i)}, \boldsymbol{c}^{(j)})$;
  /* 4. Update the clusters and the cost matrix. */
13 Update cluster set: $C \leftarrow C \setminus \{\boldsymbol{c}^{(i)}, \boldsymbol{c}^{(j)}\} \cup \{\boldsymbol{c}^{\text{new}}\}$;
14 Update cost matrix $\boldsymbol{M}$;
15 **return** $C^*, \mathcal{T}$

To train this classifier, we construct a synthetic dataset based on a public corpus containing various tasks and domains. Each data sample is a pair of $(feature, label)$, where the feature consists of a target token and $(n_s - 1)$ contextual tokens extracted from the original sequence following the sink and window attention mechanism. The label is the corresponding cluster index assigned to the activation path of the target token. For sequences whose lengths are smaller than n_s, we pad them to have a consistent input shape. For a given LLM, this process only needs to be performed once.

5.2 Request-Level Grouping

In the online stage, Saber organizes requests into groups to facilitate subsequent scheduling. Since the token classifier operates at the token level, while each request consists of multiple tokens, we next address the request grouping strategy. For decoding requests, which input only one token per iteration, grouping is straightforward: each request is classified based on its next input token using the token classifier. For prefilling requests, which involve multiple input tokens, we adopt an adaptive prompt splitting strategy.

Adaptive prompt splitting. The key of prompt splitting is identifying the optimal division points. The guiding principle is that tokens within a chunk should have similar activation paths, thereby maintaining activation sparsity

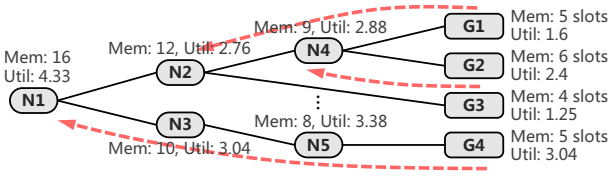

Figure 9: Illustration of the similarity-based scheduling tree and bottom-up request collection.

when processed together. Conversely, tokens with divergent activation paths should be separate into distinct chunks. Fortunately, the path clusters derived in the offline stage effectively capture these similarities, naturally grouping tokens with similar activation paths, thereby making them well-suited for guiding prompt splitting. Specifically, when a new request arrives, Saber assigns a group ID to each token using the token classifier. The prompt is then split sequentially into smaller chunks based on these group IDs, with each chunk routed to the corresponding group for scheduling.

As a result, both prefilling chunks and decoding requests are organized into groups with similar activation patterns. As shown in Figure 8, each group is associated with metadata containing essential information such as the number of requests, the required experts, and the maximum waiting time. This metadata informs the scheduler's algorithmic decisions.

Hybrid batching strategy. Requests within each group are scheduled collectively. We adopt a hybrid batching strategy as in [9, 26], which enables the joint execution of prefilling and decoding requests within a single batch. During hybrid batch execution, non-batchable operations [9, 70] (*e.g.*, attention operations), are executed separately for prefilling and decoding requests, while token-level operations are executed in parallel to maximize computation efficiency. Notably, a prefilling chunk can only be scheduled once all its preceding chunks have completed inference, ensuring that a token can attend to its prior context. Additionally, smaller chunks could be combined with adjacent chunks from other groups to form larger chunks when memory constraints permit, thereby enhancing efficiency (Section 5.3).

5.3 Group-Level Priority Scheduling

With the request groups, we now determine an appropriate group to schedule in each iteration. To maximize system throughput while preventing request starvation, Saber employs a priority scheduling algorithm that integrates throughput, latency, and memory into a unified utility function to determine the priorities of the candidate groups.

Priority definition. As throughput primarily depends on batch sizes, we define the throughput utility of group $\boldsymbol{G}_i$ as the number of tokens that can be processed in parallel per memory slot: $Util_{tp}(\boldsymbol{G}_i) = |\boldsymbol{G}_i|/Mem(\boldsymbol{c}^{(i)})$, where $\boldsymbol{c}^{(i)}$ is the path cluster center associated with group $\boldsymbol{G}_i$, and $Mem(\boldsymbol{c}^{(i)}) = \sum_{l,n} c_{ln}$. Selecting the group with the largest $Util_{tp}$ prioritizes throughput[3], maximizing GPU utilization under the memory budget. To prevent high tail latency due to starvation, we further enhance this utility with a latency expansion factor. This results in the overall utility function:

$$Util(\boldsymbol{G}_i) = \frac{|\boldsymbol{G}_i|}{Mem(\boldsymbol{c}^{(i)})} \times \left(\frac{t_{max}}{T}\right)^{\mathbb{I}(T \le t_{max}) \times \lambda}, \tag{3}$$

where T is a predefined latency threshold, t_{max} represents the current maximum latency of requests in group $\boldsymbol{G}_i$, and $\mathbb{I}(x)$ is an indicator function that outputs 1 if x is true, and 0 otherwise. Requests that have waited longer than T, potentially resulting in high latency, will have their utility expanded by a factor controlled by λ, whereas the utilities of non-delayed requests remain unaffected, as they do not exceed the acceptable latency threshold.

Adaptive inter-group scheduling. During deployment, we observed that while group-level scheduling effectively preserves activation sparsity, this single group scheduling fails to consistently yield the optimal computation efficiency under varying request traffic patterns. For example, under low system load, each group may receive only a few requests, resulting in underfilled batches and reduced throughput. In such cases, executing several small, sparsely activated batches across multiple iterations can be less efficient than merging them in a single larger batch within one iteration despite a slight increase in I/O overhead. This highlights the need for an adaptive inter-group scheduling mechanism that dynamically balances batch size and I/O overhead.

To address this, we leverage the *similarity-based scheduling tree* $\mathcal{T}$ constructed in Algorithm 1, and employ a *bottom-up request collection* strategy. As illustrated in Figure 9, this similarity tree captures the hierarchical similarity structure of path clusters. Leaf nodes correspond to the path clusters C^*, while non-leaf nodes represent progressively merged groups with increasing memory cost and size. Each node is annotated with metadata, including its utility and associated memory cost. To avoid overestimating memory requirements under low system load, we calibrate the memory cost of a node $\boldsymbol{N}_i$ as $Mem'(\boldsymbol{N}_i) = \min\{Mem(\boldsymbol{p}) \times |\boldsymbol{N}_i|, Mem(\boldsymbol{c}^{(i)})\}$, where $\boldsymbol{c}^{(i)}$ is cluster center of node $\boldsymbol{N}_i$, and $\boldsymbol{p}$ represents the activation path of an individual token. This hierarchy provides multi-granularity groups, offering flexible scheduling options. Specifically, during online scheduling, Saber performs a bottom-up traversal of this tree to collect requests across groups. This process continues until the memory cost of the current node exceeds a predefined maximum budget. For example, in Figure 9, given a memory budget of 8 slots, Saber selects node N5 rather than node G4 to improve computation efficiency. Once a suitable node is identified, Saber batches its requests to form the next input batch.

[3]This can be applied to "back-of-house" tasks (*e.g.*, benchmarking) which focus on inference throughput but are less sensitive to latency [11, 50].

6 Implementation

We have implemented Saber on top of PyTorch [2] and the HuggingFace's transformers library [60] with support for the sparsity-aware request batching scheduler. The LLM implementation and pretrained weights are obtained from HuggingFace models [1]. For experimental environments, we use a Linux server equipped with a 10-core 2.4GHz Intel Xeon Silver 4210R CPU, and an NVIDIA RTX 3090 GPU as the edge computing server for LLM serving.

Pipeline scheduling. Saber utilizes a preload-compute pipeline to overlap inference computation and expert weights loading, effectively hiding communication time during computation. Specifically, Saber's scheduler operates on the CPU. When a request group is scheduled, a preloading thread loads the necessary experts into GPU memory according to the group metadata, while simultaneously sending the input batch to the GPU for inference computation. For expert cache eviction, we employ a least-recently-used (LRU) policy, which retains frequently used experts in memory, thereby reducing loading overhead.

KV cache management. By default, Saber stores KV cache in GPU memory, with the option to offload it to CPU memory under stringent memory constraints. This is based on the observation that KV cache rarely becomes a bottleneck in edge scenarios, as edge applications typically involve short input sequences (*e.g.*, compact data formats) to ensure low-latency responses. Moreover, domain-specific edge LLMs are often fine-tuned on refined contexts to accelerate inference. In cases where long sequences introduce substantial KV cache, widely adopted sparse attention mechanisms [36, 63] can be deployed to reduce cache size. Saber can also be integrated with advanced KV cache management techniques such as PagedAttention [35] for enhanced efficiency.

7 Evaluation

7.1 Experimental Setup

Datasets and workloads. We use several real-world datasets commonly adopted for LLM evaluation, including MT-Bench (multi-turn conversation) [72], Alpaca (instruction following) [55], XSUM (text summarization) [42], and GLUE [58] (language understanding, such as SST-2 and MNLI). We consider two workload settings: (1) *Synthetic workload*: Following [70], we synthesize a request trace from multiple edge devices. Request arrivals follow a Poisson process, where p_λ controls the average arrival rate. (2) *Real trace-driven workload*: we use the Azure Trace [10, 45], a real-world LLM inference trace, and implement edge clients that send requests with inter-arrival intervals modeled from this trace. Request prompts are sampled from the above datasets.

MoE-based LLMs. We evaluate three MoE-based LLMs with different model sizes and configurations: *Switch Transformers* [21], *LLaMA-MoE* [56], and *Qwen1.5-MoE* [57], with their hyperparameters detailed in Table 1. Consistent with most modern LLMs, LLaMA-MoE and Qwen1.5-MoE adopt a decoder-only architecture, while we fine-tune and utilize only the decoder of the original encoder-decoder Switch Transformers for consistency. We denote Switch-*n* as the Switch Transformer with *n* experts per MoE layer.

MoE-based LLMs	Parameters		#L	Experts/Layer	
	Total	Active		Total	Active
Switch-16/32/64 [21]	0.5/1.0/1.9B	0.16B	6	16/32/64	1
LLaMA-MoE [56]	6.7B	3.0B	32	16	2
Qwen1.5-MoE [57]	14.3B	2.7B	24	60	4

Table 1: The configuration of LLMs in the experiments.

Baselines. We compare Saber with the request scheduling strategies used in two representative LLM serving systems. (1) *Orca* [70] employs prefill-prioritizing scheduling with iteration-level first-come-first-served (FCFS). It eagerly inserts prefilling requests into ongoing batches, while ensuring earlier requests executes no fewer iterations than later ones. (2) *Sarathi* [9] also adopts FCFS, but splits prompts into smaller chunks processed across multiple iterations, and supports hybrid batching of prefilling and decoding requests. For inference engine, we employ two strategies using a vanilla inference backend based on HuggingFace: (1) *Load On-Demand*, which loads experts into GPU memory when needed, and (2) *Preload*, which predicts and preloads experts required for the next MoE layer while the current layer executes.

Furthermore, to evaluate Saber with advanced inference systems, we integrate it with MoE-specialized backends based on *MoE-Infinity* [68], which provides optimized expert caching and prefetching pipelines. We then compare the end-to-end serving performance of Saber against two representative systems: *MoE-Infinity* and *EdgeMoE* [69].

Metrics. We report three key metrics for serving performance: (1) *throughput*, measured as completed requests per second (*rps*), (2) *latency*, defined as the time elapsed between a request's arrival and completion, and (3) *batch execution time*, reported to assess the efficiency of inference execution.

Configurations. For the vanilla HF-based inference engine, the maximum cached expert ratio is set to 20% for Switch-32, and 50% for both LLaMA-MoE and Qwen1.5-MoE. During serving, the maximum *bs* is set to 32 for Switch-32, and 16 for LLaMA-MoE and Qwen1.5-MoE. For experiments with advanced inference engines, we only specify a maximum memory budget for each model, while runtime batch sizes are dynamically determined based on the available memory and system load. In the priority scheduling, the threshold is set to $T = 2s$, with an expanding factor $\lambda = 1.5$.

7.2 Overall Performance Evaluation

We first evaluate the system using the synthetic workload and the vanilla HF-based inference backend to isolate the impact of scheduling strategies without backend-specific

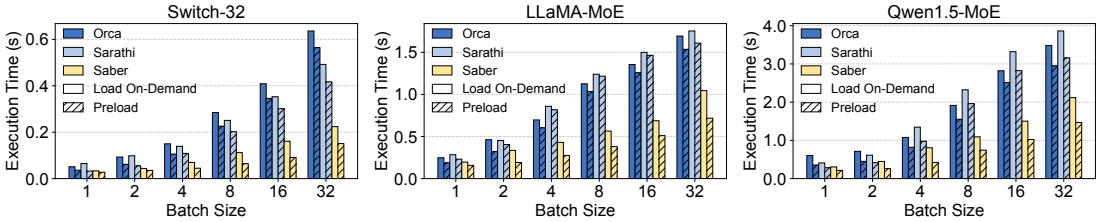

Figure 10: Average execution time for a batch of requests using different scheduling and expert loading strategies.

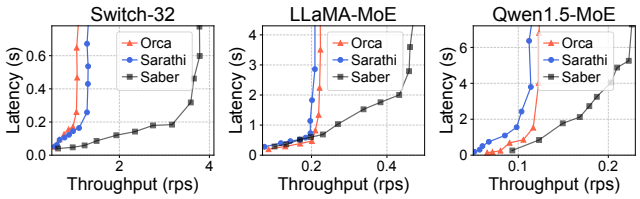

Figure 11: Overall LLM serving performance across different schedulers (lower right is better). Latencies are normalized by the number of generated tokens.

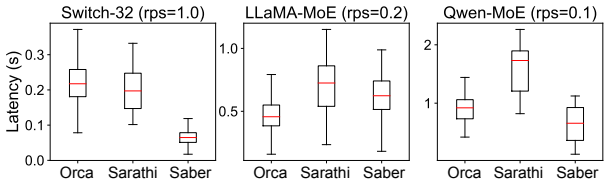

Figure 12: Serving latency breakdown. Each box illustrates the 25%, median, and 75% latencies, and whiskers plot the 1% and 99% non-outlier values.

optimizations. Figures 10 and 11 report the average batch execution time and throughput-latency performance.

Batch execution time. Saber substantially reduces batch execution time across different batch sizes. First, under *Load On-Demand*, Saber achieves up to 2.84× and 2.20× reductions in batch time compared with Orca and Sarathi, respectively. The improvement stems from activating fewer experts per batch, which significantly reduces expert-loading overhead. For instance, in Qwen1.5-MoE with $bs = 16$, Saber reduces this overhead by 50.4% and 47.3% compared with Orca and Sarathi. Second, with reduced I/O overhead, Saber also benefits more from expert preloading. Under *Preload*, Saber achieves up to 3.80× and 3.31× reductions in batch time compared with Orca and Sarathi. For the baseline schedulers, expert preloading yields an average batch time reduction of 21.1% for Orca and 19.5% for Sarathi, but the benefit diminishes at larger batch sizes due to sparsity vanishing, where more activated experts reduces the potential of preloading. For example, Orca's speedup drops from 31.7% at $bs = 1$ to 12.0% at $bs = 32$. In contrast, Saber reduces I/O overhead, and enables greater overlap between expert loading and computation, achieving a 33.2% reduction even at $bs = 32$.

Throughput and latency. Benefitting from reduced batch execution time, Saber achieves significantly higher serving throughput and lower latency compared to the baselines.

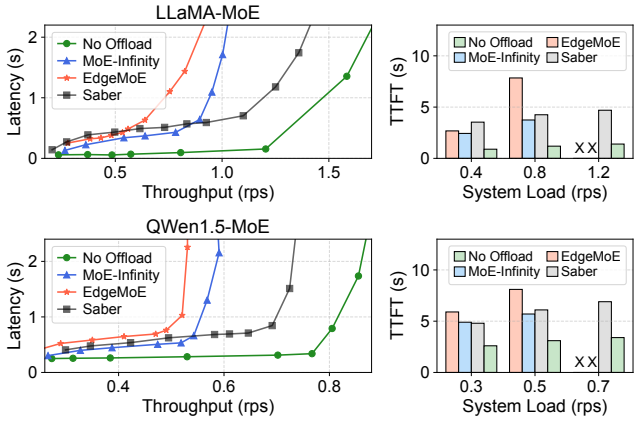

Figure 13: LLM serving performance on the systems using advanced MoE inference backends.

Under the same latency constraints, Saber sustains up to 3.27×, 1.98×, and 1.66× higher load than Orca for Switch-32, LLaMA-MoE, and Qwen1.5-MoE, respectively, and 2.86×, 2.15×, and 1.78× higher load than Sarathi. Nevertheless, when the request arrival rate is low, Saber performs similarly to the baselines, because all systems operate with small batch sizes and are less affected by sparsity vanishing. Furthermore, we analyze latency distributions and present the results in Figure 12. At the same throughput levels, Saber achieves lower or comparable average and tail latencies compared with the baselines. This behavior is largely attributed to the priority scheduling strategy, which prevents requests starvation, and helps maintain acceptable tail latency.

Performance with modern MoE inference systems. To evaluate performance under realistic conditions, we further use the Azure Trace workload, and compare Saber with state-of-the-art MoE inference systems. Experiments are conducted on LLaMA-MoE and QWen1.5-MoE, with memory budget set to 12 GB and 24 GB [4], respectively (Figure 13).

Even with optimized inference backends, Saber consistently sustains higher throughput. For example, under a decoding latency constraint of 1*s*, Saber achieves 1.35×, 1.27× higher throughput than EdgeMoE and MoE-Infinity. Notably, Saber may exhibit slightly higher latency under light system load. In this regime, systems operate with small batch sizes where activation sparsity is naturally preserved, and

[4]Due to GPU memory limitations, the "No Offload" upper bound is estimated by using only cached experts to eliminate expert loading overhead.

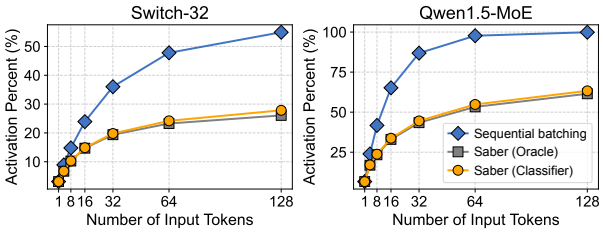

Figure 14: The average expert activation percent of input batches under different batch sizes. "Saber (Oracle)" assumes the activation paths of candidate requests are known when grouping and batching requests.

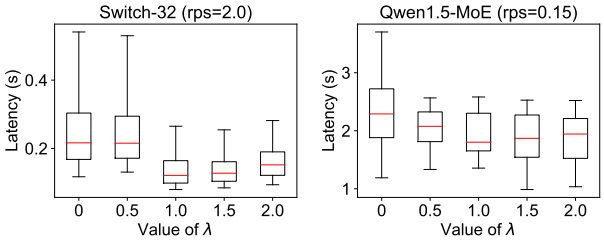

Figure 15: Impact of the value of λ on tail latency.

modern backends are already highly optimized in such cases. Moreover, to improve sparsity, Saber may schedule requests in non-consecutive iterations, which can slightly increase latency, including both decoding latency and time-to-first-token (TTFT). As system load becomes heavier, the advantages of Saber's sparsity-aware batching become more pronounced, allowing it to deliver higher throughput with modest latency increases due to I/O overhead savings, effectively narrowing the gap with the no-offloading upper bound.

7.3 Performance Breakdown Analysis

In this subsection, we dive into the performance of Saber's key components on two representative LLMs, Switch-32 (smaller and has a moderate number of experts) and Qwen1.5-MoE (larger and has a greater number of experts), to provide deeper insights of the performance improvements.

Sparsity-aware batching. To evaluate the effectiveness of Saber's offline profiling and online batching strategies, we vary the batch size (measured by the number of tokens) and report the average expert activation ratio during inference. As shown in Figure 14, with sequential batching, the number of activated experts increases rapidly with batch size. For example, in Qwen1.5-MoE, a batch with $bs = 8$ activates 41.6% experts on average, while this number increases to 97.7% for $bs = 64$. In contrast, Saber's sparsity-aware batching activates only 23.8% and 54.8% of experts for $bs = 8$ and 64, respectively, yielding up to 1.95× reduction, and effectively mitigating the sparsity-vanishing problem. Moreover, we observe that the performance of "Saber (Classifier)" closely matches that of "Saber (Oracle)", with less than 2% error, demonstrating the effectiveness of the lightweight token classifier in accurately grouping requests.

Model	Offline Profiling (*min*)			Online Scheduling (*ms*)	
	Ident.	Clust.	Train Clf.	Cls. (bs=1)	Cls. (bs=16)
Switch-32	2.8	2.1	15.5	0.64	0.70
Qwen1.5-MoE	36.6	5.4	29	0.61	0.85

Table 2: Overhead analysis for Saber's components.

Model	Method	XSUM	Alpaca	MT-Bench	MNLI	SST2
Switch-32	CLS Acc (%)	91.53	88.41	87.17	87.20	90.38
	AR-Saber (%)	20.74	19.13	20.87	20.99	18.43
	AR-Oracle (%)	20.59	17.80	19.69	20.00	17.74
Qwen1.5-MoE	CLS Acc (%)	87.96	92.77	83.23	84.18	85.85
	AR-Saber (%)	37.27	34.28	36.12	37.60	35.83
	AR-Oracle (%)	36.99	32.88	32.81	34.64	29.23

Table 3: Classifier's accuracy (CLS Acc) and expert activation ratio (AR, $bs = 16$) under distribution shifts.

Priority scheduling. To assess the effectiveness of priority scheduling, we vary the parameter λ and report latency statistics in Figure 15. When $\lambda = 0$, indicating that latency is not considered and scheduling is throughput-oriented, tail latency becomes high (*e.g.*, $0.54s/token$ for Switch-32 and $3.78s/token$ for Qwen1.5-MoE) due to request starvation. This issue worsens with higher requests arrival rates. For instance, Switch-32's tail latency increases to $1.24s/token$ when $rps = 3.0$. Fortunately, by increasing λ to 1.5, Saber's priority scheduling increases the priority scores for long-waiting groups, effectively reducing tail latency by 2.08× for Switch-32 and 1.47× for Qwen1.5-MoE. However, further increasing λ does not consistently improve performance, as excessive prioritization of long-waiting groups can lead to smaller batch sizes and reduced inference efficiency.

Overhead analysis. Table 2 breaks down the overhead of Saber's components. *Offline profiling* typically completes within tens of minutes. The main cost arises from Path Identification (*Ident.*) and Classifier Training (*Train Clf.*), which together account for more than 90% of the total latency. In contrast, Path Clustering (*Clust.*) incurs much lower overhead, benefitting from the compressed path space and the efficient clustering algorithm. For *online scheduling*, we report the primary cost, *i.e.*, Token Classification (*Cls.*) using the trained classifier. Owing to its lightweight design, this latency is less than 1 *ms*. Other scheduling operations, such as updating utility scores and selecting high-priority groups, have $O(n)$ complexity with respect to the number of concurrent requests (up to hundreds), and thus can be efficiently executed with marginal overhead. Since an LLM inference iteration typically takes tens of milliseconds, the scheduling can run in parallel without affecting critical-path latency.

7.4 Robustness Analysis

Robustness to data distribution shifts. Since the performance of the offline-trained classifier is critical to Saber's

scheduling decisions, we evaluate its robustness under distribution shifts. Specifically, we train the classifier on XSUM and test it on a diverse set of tasks listed in Table 3. Our results show that: (1) Despite substantial variation in task types, the classification accuracy degrades only marginally under distribution shifts (within 5% in most cases). We attribute this robustness to the sink and window attention mechanism, which captures transferable linguistic patterns shared across tasks and domains. (2) The expert activation ratio (AR) remains stable even when classification accuracy slightly decreases. For example, for Qwen1.5-MoE on MT-Bench, Saber with 83.23% classification accuracy activates only 3.31% more experts on average compared to the Oracle case (100% accuracy). These results indicate that Saber can effectively operate in dynamic environments with shifting workloads. Moreover, a lightweight online fine-tuning mechanism that leverages a small buffer of recently served requests can be incorporated to further adapt the classifier to gradual distribution changes.

Scalability with large expert counts. As the number of experts per layer increases, the activation path space expands substantially, potentially exacerbating computational overhead. We evaluate Saber on Switch-64 and Switch-128 to assess the scalability of its path compression and clustering design (Figure 16). To control the enlarged path space, Saber adopts a two-stage compression strategy. First, it extracts valid activation paths, whose proportion becomes increasingly sparse as expert counts grow (left). Second, it profiles only high-probability paths based on the long-tailed distribution of path activations (middle). For example, in Switch-128, only 1163 paths with probability greater than 10^{-4} are retained for clustering. This compression ensures the effective path space tractable under the $O(n^2)$ clustering complexity. Despite aggressive pruning, representational coverage remains robust: clustering the top-1k paths captures 89.3% of the remaining paths with similarity > 0.8. Consequently, using these clusters, Saber consistently outperforms sequential batching under larger expert configurations (right).

Due to space limits, additional results on robustness analysis and performance breakdown are provided in Appendix B.

#E per Layer	Valid Ratio
8	0.109
16	0.016
32	2e-4
64	1e-5
128	3e-7

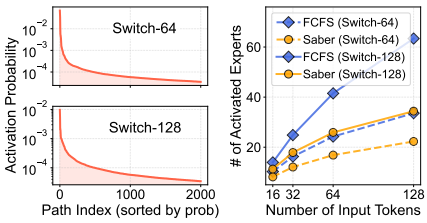

Figure 16: Scalability analysis in terms of expert counts.

8 Related Work

Resource-efficient MoE-based LLMs. MoE architectures leverage sparse activation to scale LLMs without a proportional increase in computational overhead. Despite their strong performance, MoE-based LLMs remain challenging to deploy on resource-constrained edge platforms due to their substantial model sizes. To address this, existing works have focused on optimization techniques such as expert compression [12, 23, 75], quantization [22, 69], sparse-to-dense knowledge distillation [21, 47, 67] to reduce model sizes and memory demands of MoE-based LLMs. Different from these model compression optimizations, Saber is based on expert offloading to address resource limitations without the need to modify the architecture or compress the target LLM.

Serving MoE models on resource-limited platforms. Expert offloading extends traditional parameter offloading by accounting for sparse activation patterns, allowing LLMs to operate even when their memory requirements exceed available hardware capacity. To enhance inference efficiency, prior works have proposed various techniques to mitigate the latency introduced by loading expert weights. These include reducing load sizes through expert quantization and adaptive gating [34, 54, 69, 73], overlapping loading and computation via expert preloading [17, 20, 24, 29, 51, 68, 69], and improving cache hit rate through advanced caching and eviction strategies [18, 28, 34, 54, 68]. Additionally, leveraging CPU to assist computation not only reduces loading overhead, but also exploits available processing power to accelerate inference [11, 31, 52, 54, 66]. As Saber only modifies the input batch and optimizes scheduling without altering LLM architectures, it is compatible with existing methods and can be integrated to further improve inference speed.

Modern LLM serving systems. As the popularity of LLMs grows alongside increasing inference workload, various LLM serving systems have been developed to deliver high-throughput and low-latency inference. These systems are particularly optimized for autoregressive Transformer-based LLMs, employing optimizations including request scheduling [9, 53, 61, 70], memory optimization [15, 19, 35, 36, 40], and prefilling and decoding disaggregation [27, 45, 74], etc. Nevertheless, they are mainly designed for cloud environments with abundant resources, making them not readily deployable on resource-limited edge platforms. In contrast, Saber is specifically tailored to support efficient MoE-based LLM serving on edge platforms.

9 Conclusion

In this work, we present Saber, an efficient MoE-based LLM serving framework tailored for resource-limited edge servers. Leveraging our sparsity-aware scheduling, Saber effectively preserves activation sparsity during batched inference, thereby reducing expert loading overhead and enhancing inference efficiency. Extensive experiments show that Saber achieves a superior throughput-latency trade-off, delivering up to 3.8× batch execution time reduction and 3.27× throughput improvement compared with sparsity-agnostic schedulers.

References

[1] 2024. HuggingFace models. https://huggingface.co/models.

[2] 2024. PyTorch. https://pytorch.org/.

[3] 2025. AWS Outposts. https://aws.amazon.com/outposts/.

[4] 2025. Azure Stack Edge. https://azure.microsoft.com/en-us/services/databox/edge/.

[5] 2025. Building the AI-powered local smart home. https://www.home-assistant.io/blog/2025/09/11/ai-in-home-assistant/.

[6] 2025. Xiaomi Miloco Copilot. https://github.com/XiaoMi/xiaomi-miloco.

[7] 2026. NVIDIA Jetson Orin: Next-level AI performance for next-gen robotics and edge solutions. https://https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-orin/.

[8] 2026. Premio LLM-1U-RPL Series. https://premioinc.com/collections/llm-1u-rpl-series-edge-ai-rackmount-server.

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

[10] Microsoft Azure. 2024. Azure Public Dataset: Azure LLM Inference Trace. https://github.com/Azure/AzurePublicDataset.

[11] Shiyi Cao, Shu Liu, Tyler Griggs, Peter Schafhalter, Xiaoxuan Liu, Ying Sheng, Joseph E. Gonzalez, Matei Zaharia, and Ion Stoica. 2025. MoE-Lightning: High-Throughput MoE Inference on Memory-constrained GPUs. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 715–730.

[12] Tianyu Chen, Shaohan Huang, Yuan Xie, Binxing Jiao, Daxin Jiang, Haoyi Zhou, Jianxin Li, and Furu Wei. 2022. Task-Specific Expert Pruning for Sparse Mixture-of-Experts. *arXiv preprint arXiv:2206.00277* (2022).

[13] Weihao Cui, Han Zhao, Quan Chen, Hao Wei, Zirui Li, Deze Zeng, Chao Li, and Minyi Guo. 2022. DVABatch: Diversity-aware Multi-Entry Multi-Exit Batching for Efficient Processing of DNN Services on GPUs. In *2022 USENIX Annual Technical Conference (ATC)*. 183–198.

[14] Damai Dai, Chengqi Deng, Chenggang Zhao, R.x. Xu, Huazuo Gao, Deli Chen, Jiashi Li, Wangding Zeng, Xingkai Yu, Y. Wu, Zhenda Xie, Y.k. Li, Panpan Huang, Fuli Luo, Chong Ruan, Zhifang Sui, and Wenfeng Liang. 2024. DeepSeekMoE: Towards Ultimate Expert Specialization in Mixture-of-Experts Language Models. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (ACL)*. 1280–1297.

[15] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. 2022. FLASHATTENTION: fast and memory-efficient exact attention with IO-awareness. In *Proceedings of the 36th International Conference on Neural Information Processing Systems (NeurIPS)*. 16344–16359.

[16] Sanjoy Dasgupta. 2016. A cost function for similarity-based hierarchical clustering. In *Proceedings of the Forty-Eighth Annual ACM Symposium on Theory of Computing (STOC)*. 118–127.

[17] Zhixu Du, Shiyu Li, Yuhao Wu, Xiangyu Jiang, Jingwei Sun, Qilin Zheng, Yongkai Wu, Ang Li, Hai Li, and Yiran Chen. 2024. SiDA: Sparsity-Inspired Data-Aware Serving for Efficient and Scalable Large Mixture-of-Experts Models. In *Proceedings of Machine Learning and Systems (MLSys)*. 224–238.

[18] Artyom Eliseev and Denis Mazur. 2023. Fast Inference of Mixture-of-Experts Language Models with Offloading. *arXiv preprint arXiv:2312.17238* (2023).

[19] Jiarui Fang, Yang Yu, Chengduo Zhao, and Jie Zhou. 2021. TurboTransformers: an efficient GPU serving system for transformer models. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*. 389–402.

[20] Zhiyuan Fang, Yuegui Huang, Zicong Hong, Yufeng Lyu, Wuhui Chen, Yue Yu, Fan Yu, and Zibin Zheng. 2025. Klotski: Efficient Mixture-of-Expert Inference via Expert-Aware Multi-Batch Pipeline. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 574–588.

[21] William Fedus, Barret Zoph, and Noam Shazeer. 2022. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity. *The Journal of Machine Learning Research (JMLR)* 23, 1 (2022), 5232–5270.

[22] Elias Frantar and Dan Alistarh. 2023. QMoE: Practical Sub-1-Bit Compression of Trillion-Parameter Models. *arXiv preprint arXiv:2310.16795* (2023).

[23] Ze-Feng Gao, Peiyu Liu, Wayne Xin Zhao, Zhong-Yi Lu, and Ji-Rong Wen. 2022. Parameter-Efficient Mixture-of-Experts Architecture for Pre-trained Language Models. In *Proceedings of the 29th International Conference on Computational Linguistics (COLING)*. 3263–3273.

[24] Xin He, Shunkang Zhang, Yuxin Wang, Haiyan Yin, Zihao Zeng, Shaohuai Shi, Zhenheng Tang, Xiaowen Chu, Ivor Tsang, and Ong Yew Soon. 2024. ExpertFlow: Optimized Expert Activation and Token Allocation for Efficient Mixture-of-Experts Inference. *arXiv preprint arXiv:2410.17954* (2024).

[25] Yuze He, Chen Bian, Jingfei Xia, Shuyao Shi, Zhenyu Yan, Qun Song, and Guoliang Xing. 2023. VI-Map: Infrastructure-Assisted Real-Time HD Mapping for Autonomous Driving. In *Proceedings of the 29th Annual International Conference on Mobile Computing and Networking (MobiCom)*. 1–15.

[26] Connor Holmes, Masahiro Tanaka, Michael Wyatt, Ammar Ahmad Awan, Jeff Rasley, Samyam Rajbhandari, Reza Yazdani Aminabadi, Heyang Qin, Arash Bakhtiari, Lev Kurilenko, and Yuxiong He. 2024. DeepSpeed-FastGen: High-throughput Text Generation for LLMs via MII and DeepSpeed-Inference. *arXiv preprint arXiv:2401.08671* (2024).

[27] Cunchen Hu, Heyang Huang, Liangliang Xu, Xusheng Chen, Jiang Xu, Shuang Chen, Hao Feng, Chenxi Wang, Sa Wang, Yungang Bao, Ninghui Sun, and Yizhou Shan. 2024. Inference without Interference: Disaggregate LLM Inference for Mixed Downstream Workloads. *arXiv preprint arXiv:2401.11181* (2024).

[28] Haiyang Huang, Newsha Ardalani, Anna Sun, Liu Ke, Shruti Bhosale, Hsien-Hsin S. Lee, Carole-Jean Wu, and Benjamin Lee. 2024. Toward Efficient Inference for Mixture of Experts. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems (NeurIPS)*. 1–27.

[29] Ranggi Hwang, Jianyu Wei, Shijie Cao, Changho Hwang, Xiaohu Tang, Ting Cao, and Mao Yang. 2024. Pre-gated MoE: An Algorithm-System Co-Design for Fast and Scalable Mixture-of-Expert Inference. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. 1018–1031.

[30] Albert Q Jiang, Alexandre Sablayrolles, Antoine Roux, Arthur Mensch, Blanche Savary, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Emma Bou Hanna, Florian Bressand, et al. 2024. Mixtral of Experts. *arXiv preprint arXiv:2401.04088* (2024).

[31] Keisuke Kamahori, Yile Gu, Kan Zhu, and Baris Kasikci. 2024. Fiddler: CPU-GPU Orchestration for Fast Inference of Mixture-of-Experts Models. *arXiv preprint arXiv:2402.07033* (2024).

[32] Mehrdad Khani, Ganesh Ananthanarayanan, Kevin Hsieh, Junchen Jiang, Ravi Netravali, Yuanchao Shu, Mohammad Alizadeh, and Victor Bahl. 2023. RECL: Responsive Resource-Efficient Continuous Learning for Video Analytics. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 917–932.

[33] Sehoon Kim, Sheng Shen, David Thorsley, Amir Gholami, Woosuk Kwon, Joseph Hassoun, and Kurt Keutzer. 2022. Learned Token Pruning for Transformers. In *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD)*. 784–794.

[34] Rui Kong, Yuanchun Li, Qingtian Feng, Weijun Wang, Xiaozhou Ye, Ye Ouyang, Linghe Kong, and Yunxin Liu. 2024. SwapMoE: Serving Off-the-shelf MoE-based Large Language Models with Tunable Memory

Budget. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (ACL)*. 6710–6720.

[35] 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 (SOSP)*. 611–626.

[36] 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)*. 155–172.

[37] Dmitry Lepikhin, HyoukJoong Lee, Yuanzhong Xu, Dehao Chen, Orhan Firat, Yanping Huang, Maxim Krikun, Noam Shazeer, and Zhifeng Chen. 2021. GShard: Scaling Giant Models with Conditional Computation and Automatic Sharding. In *International Conference on Learning Representations (ICLR)*. 1–23.

[38] Jiaze Li, Jingyang Chen, Yuxun Qu, Shijie Xu, Zhenru Lin, Junyou Zhu, Boshen Xu, Wenhui Tan, Pei Fu, Jianzhong Ju, Zhenbo Luo, and Jian Luan. 2025. Xiaomi MiMo-VL-Miloco Technical Report. *arXiv preprint arXiv:2512.17436* (2025).

[39] Yuanchun Li, Hao Wen, Weijun Wang, Xiangyu Li, Yizhen Yuan, Guohong Liu, Jiacheng Liu, Wenxing Xu, Xiang Wang, Yi Sun, Rui Kong, Yile Wang, Hanfei Geng, Jian Luan, Xuefeng Jin, Zi-Liang Ye, Guanjing Xiong, Fan Zhang, Xiang Li, Mengwei Xu, Zhijun Li, Peng Li, Yang Liu, Yaqiong Zhang, and Yunxin Liu. 2024. Personal LLM Agents: Insights and Survey about the Capability, Efficiency and Security. *arXiv preprint arXiv:2401.05459* (2024).

[40] Xupeng Miao, Gabriele Oliaro, Zhihao Zhang, Xinhao Cheng, Zeyu Wang, Zhengxin Zhang, Rae Ying Yee Wong, Alan Zhu, Lijie Yang, Xiaoxiang Shi, Chunan Shi, Zhuoming Chen, Daiyaan Arfeen, Reyna Abhyankar, and Zhihao Jia. 2024. SpecInfer: Accelerating Large Language Model Serving with Tree-based Speculative Inference and Verification. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 932–949.

[41] F. Murtagh and P. Contreras. 2011. Algorithms for hierarchical clustering: an overview. *WIREs Data Mining and Knowledge Discovery* 2 (2011), 86–97.

[42] Shashi Narayan, Shay B. Cohen, and Mirella Lapata. 2018. Don't Give Me the Details, Just the Summary! Topic-Aware Convolutional Neural Networks for Extreme Summarization. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. 1797–1807.

[43] Xiaomin Ouyang, Xian Shuai, Yang Li, Li Pan, Xifan Zhang, Heming Fu, Sitong Cheng, Xinyan Wang, Shihua Cao, Jiang Xin, Hazel Mok, Zhenyu Yan, Doris Sau Fung Yu, Timothy Kwok, and Guoliang Xing. 2024. ADMarker: A Multi-Modal Federated Learning System for Monitoring Digital Biomarkers of Alzheimer's Disease. In *Proceedings of the 30th Annual International Conference on Mobile Computing and Networking (MobiCom)*. 404–419.

[44] Xiaomin Ouyang, Zhiyuan Xie, Heming Fu, Sitong Cheng, Li Pan, Neiwen Ling, Guoliang Xing, Jiayu Zhou, and Jianwei Huang. 2023. Harmony: Heterogeneous Multi-Modal Federated Learning through Disentangled Model Training. In *Proceedings of the 21st Annual International Conference on Mobile Systems, Applications and Services (MobiSys)*. 530–543.

[45] Pratyush Patel, Esha Choukse, Chaojie Zhang, Aashaka Shah, Íñigo Goiri, Saeed Maleki, and Ricardo Bianchini. 2024. Splitwise: Efficient Generative LLM Inference Using Phase Splitting. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. 118–132.

[46] Guanqiao Qu, Qiyuan Chen, Wei Wei, Zheng Lin, Xianhao Chen, and Kaibin Huang. 2025. Mobile Edge Intelligence for Large Language Models: A Contemporary Survey. *IEEE Communications Surveys & Tutorials* (2025), 1–42.

[47] Samyam Rajbhandari, Conglong Li, Zhewei Yao, Minjia Zhang, Reza Yazdani Aminabadi, Ammar Ahmad Awan, Jeff Rasley, and Yuxiong He. 2022. Deepspeed-moe: Advancing mixture-of-experts inference and training to power next-generation ai scale. In *International Conference on Machine Learning (ICML)*. 18332–18346.

[48] Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc Le, Geoffrey Hinton, and Jeff Dean. 2017. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. *arXiv preprint arXiv:1701.06538* (2017).

[49] Leming Shen, Qiang Yang, Yuanqing Zheng, and Mo Li. 2025. AutoIOT: LLM-Driven Automated Natural Language Programming for AIoT Applications. In *Proceedings of the 30th Annual International Conference on Mobile Computing and Networking (MobiCom)*. 1–15.

[50] Ying Sheng, Lianmin Zheng, Binhang Yuan, Zhuohan Li, Max Ryabinin, Beidi Chen, Percy Liang, Christopher Ré, Ion Stoica, and Ce Zhang. 2023. FlexGen: high-throughput generative inference of large language models with a single GPU. In *Proceedings of the 40th International Conference on Machine Learning (ICML)*. 31094–31116.

[51] Xiaoniu Song, Zihang Zhong, Rong Chen, and Haibo Chen. 2025. ProMoE: Fast MoE-based LLM Serving using Proactive Caching. *arXiv preprint arXiv:2410.22134* (2025).

[52] Yixin Song, Zeyu Mi, Haotong Xie, and Haibo Chen. 2024. PowerInfer: Fast Large Language Model Serving with a Consumer-grade GPU. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles (SOSP)*. 590–606.

[53] 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)*. 173–191.

[54] Peng Tang, Jiacheng Liu, Xiaofeng Hou, Yifei Pu, Jing Wang, Pheng-Ann Heng, Chao Li, and Minyi Guo. 2024. HOBBIT: A Mixed Precision Expert Offloading System for Fast MoE Inference. *arXiv preprint arXiv:2411.01433* (2024).

[55] Rohan Taori, Ishaan Gulrajani, Tianyi Zhang, Yann Dubois, Xuechen Li, Carlos Guestrin, Percy Liang, and Tatsunori B. Hashimoto. 2023. Stanford Alpaca: An Instruction-following LLaMA model. https://github.com/tatsu-lab/stanford_alpaca.

[56] LLaMA-MoE Team. 2024. LLaMA-MoE: Building Mixture-of-Experts from LLaMA with Continual Pre-training. https://github.com/pjlab-sys4nlp/llama-moe/blob/main/docs/LLaMA_MoE.pdf.

[57] Qwen Team. 2024. Qwen1.5-MoE: Matching 7B Model Performance with 1/3 Activated Parameters. https://qwenlm.github.io/blog/qwen-moe/.

[58] Alex Wang, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel R. Bowman. 2019. GLUE: A Multi-Task Benchmark and Analysis Platform for Natural Language Understanding. In *International Conference on Learning Representations (ICLR)*. 1–20.

[59] Hao Wen, Yuanchun Li, Guohong Liu, Shanhui Zhao, Tao Yu, Toby Jia-Jun Li, Shiqi Jiang, Yunhao Liu, Yaqin Zhang, and Yunxin Liu. 2024. AutoDroid: LLM-powered Task Automation in Android. In *Proceedings of the 30th Annual International Conference on Mobile Computing and Networking (MobiCom)*. 543–557.

[60] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. 2020. HuggingFace's Transformers: State-of-the-art Natural Language Processing. *arXiv preprint arXiv:1910.03771* (2020).

[61] Bingyang Wu, Yinmin Zhong, Zili Zhang, Shengyu Liu, Fangyue Liu, Yuanhang Sun, Gang Huang, Xuanzhe Liu, and Xin Jin. 2024. Fast Distributed Inference Serving for Large Language Models. *arXiv preprint*

arXiv:2305.05920 (2024).

[62] Guangxuan Xiao, Jiaming Tang, Jingwei Zuo, Junxian Guo, Shang Yang, Haotian Tang, Yao Fu, and Song Han. 2024. DuoAttention: Efficient Long-Context LLM Inference with Retrieval and Streaming Heads. *arXiv preprint arXiv:2410.10819* (2024).

[63] Guangxuan Xiao, Yuandong Tian, Beidi Chen, Song Han, and Mike Lewis. 2024. Efficient Streaming Language Models with Attention Sinks. In *The Twelfth International Conference on Learning Representations (ICLR)*. 1–21.

[64] Huatao Xu, Liying Han, Qirui Yang, Mo Li, and Mani Srivastava. 2024. Penetrative AI: Making LLMs Comprehend the Physical World. In *Findings of the Association for Computational Linguistics (ACL Findings)*. 7324–7341.

[65] Minrui Xu, Hongyang Du, Dusit Niyato, Jiawen Kang, Zehui Xiong, Shiwen Mao, Zhu Han, Abbas Jamalipour, Dong In Kim, Xuemin Shen, Victor C. M. Leung, and H. Vincent Poor. 2024. Unleashing the Power of Edge-Cloud Generative AI in Mobile Networks: A Survey of AIGC Services. *IEEE Communications Surveys & Tutorials* 26, 2 (2024), 1127–1170.

[66] Zhao Xuanlei, Bin Jia, Haotian Zhou, Ziming Liu, Shenggan Cheng, and Yang You. 2024. HeteGen: Efficient Heterogeneous Parallel Inference for Large Language Models on Resource-Constrained Devices. In *Proceedings of Machine Learning and Systems (MLSys)*. 162–172.

[67] Fuzhao Xue, Xiaoxin He, Xiaozhe Ren, Yuxuan Lou, and Yang You. 2022. One Student Knows All Experts Know: From Sparse to Dense. *arXiv preprint arXiv:2201.10890* (2022).

[68] Leyang Xue, Yao Fu, Zhan Lu, Luo Mai, and Mahesh Marina. 2024. MoE-Infinity: Offloading-Efficient MoE Model Serving. *arXiv preprint arXiv:2401.14361* (2024).

[69] Rongjie Yi, Liwei Guo, Shiyun Wei, Ao Zhou, Shangguang Wang, and Mengwei Xu. 2025. EdgeMoE: Empowering Sparse Large Language Models on Mobile Devices. *IEEE Transactions on Mobile Computing (TMC)* (2025), 1–16.

[70] 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)*. 521–538.

[71] Mu Yuan, Lan Zhang, Fengxiang He, Xueting Tong, and Xiang-Yang Li. 2022. InFi: End-to-End Learnable Input Filter for Resource-Efficient Mobile-Centric Inference. In *Proceedings of the 28th Annual International Conference on Mobile Computing And Networking (MobiCom)*. 228–241.

[72] Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric P. Xing, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. 2023. Judging LLM-as-a-judge with MT-bench and Chatbot Arena. In *Proceedings of the 37th International Conference on Neural Information Processing Systems (NeurIPS)*. 46595–46623.

[73] Shuzhang Zhong, Ling Liang, Yuan Wang, Runsheng Wang, Ru Huang, and Meng Li. 2024. AdapMoE: Adaptive Sensitivity-based Expert Gating and Management for Efficient MoE Inference. *arXiv preprint arXiv:2408.10284* (2024).

[74] 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. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 193–210.

[75] Yan Zhuang, Zhenzhe Zheng, Fan Wu, and Guihai Chen. 2024. LiteMoE: Customizing On-device LLM Serving via Proxy Submodel Tuning. In *Proceedings of the 22nd ACM Conference on Embedded Networked Sensor Systems (SenSys)*. 521–534.

[76] Barret Zoph, Irwan Bello, Sameer Kumar, Nan Du, Yanping Huang, Jeff Dean, Noam M. Shazeer, and William Fedus. 2022. ST-MoE: Designing Stable and Transferable Sparse Expert Models. *arXiv preprint arXiv:2202.08906* (2022).

A Path Similarity and Grouping Opportunity

Figure 6 has demonstrated the cross-request path similarity and the potential for request grouping using the hybrid workload, which is constructed using the datasets described in Section 7.1. In this section, we further analyze these properties on individual datasets spanning diverse domains, including conversational, instruction-following, summarization, and natural language understanding tasks.

As illustrated in Figure 17, we consistently observe non-trivial cross-request path similarity across all five individual datasets. Under traditional batching, the average in-batch similarity remains low (reflecting sparsity vanishing due to sequential or random batching), yet a significant fraction of request pairs exhibit moderate-to-high similarity. In contrast, sparsity-aware batching substantially increases the average similarity across all workloads. For instance, for the prefill-dominant task MNLI, the average path similarity increases from 0.076 to 0.256, while for the decode-dominant task MT-Bench, it rises from 0.087 to 0.413.

More importantly, the relationship between path similarity and system efficiency remains monotonic and consistent across workloads. In all evaluated datasets, increasing intra-batch path similarity through sparsity-aware batching leads to a near-proportional reduction in the fraction of activated experts, which in turn reduces expert swapping and I/O overhead. For example, on MT-Bench, an increase of 0.326 in average path similarity corresponds to a 55.2% reduction in the expert activation ratio. This observation holds across heterogeneous workloads and across different MoE architectures, demonstrating that the performance gain mechanism is workload-agnostic rather than benchmark-specific. Therefore, this property exposes a non-trivial optimization space for Saber's scheduling strategy.

Mechanism Analysis. The root cause of this generalization lies in the semantic specialization of MoE routing. Within MoE layers, experts are trained to capture recurring linguistic or functional patterns (*e.g.*, instruction prefixes, conversational structures, sentiment-bearing expressions). Real-world edge workloads, although seemingly heterogeneous, are dominated by a limited set of high-level task patterns and syntactic regularities. Consequently, tokens from different requests could cluster in similar latent regions, leading to overlapping expert selections. While the source of similarity may differ across task types (*e.g.*, template similarity in instruction tuning vs. structural similarity in news summarization), the resulting expert-path clustering phenomenon remains consistent. This allows Saber's performance to generalize across various workloads and MoE-based LLMs.

B Extended Experiments

In this section, we present supplementary results that provide a more granular analysis of the performance of Saber. Specifically, we conduct a detailed analysis of the adaptive prompt splitting strategy, evaluate the robustness of Saber under varying memory budgets, and examine its performance across MoE-based LLMs with different number of experts.

Adaptive prompt splitting. We examine the number of activated experts and the batch execution time of prefilling batches, with results shown in Figure 18. Orca prefills the entire prompt in a single iteration, activating a large number of experts and thus incurring the highest batch latency. Sarathi splits prompts into small chunks, reducing batch latency, but it fails to account for activation sparsity, leading to suboptimal performance. Saber outperforms both baselines, reducing prefilling latency by 3.18× and 2.36× compared to Orca and Sarathi, respectively. This improvement stems from adaptive prompt splitting, which only processes a small, similarly-activated chunk per iteration, and thus preserves sparsity and obtains low batch latency.

Varying memory budgets. Memory budget determines the number of experts that can be cached, directly impacting expert loading overhead and inference efficiency. To evaluate this impact, we vary the cached expert ratio and measure the resulting maximum system throughput. As shown in Figure 19, Saber consistently outperforms baselines across different memory budgets, especially when memory budget is stringent. These improvements stem from both reduced a number of activated experts and a lower cache missing rate. For instance, in Switch-32 with 40% cached ratio and $bs = 32$, the average numbers of activated experts per MoE layer is 12.8 for Orca, 11.3 for Sarathi, and only 6.8 for Saber. Correspondingly, the number of cache-missed experts, which incurs non-overlapping I/O latency, is 6.5 for Orca, 4.8 for Sarathi, and merely 0.24 for Saber. This substantial reduction in cache-missed experts accounts for the superior performance of Saber, which achieves 3.99× and 3.37× higher throughput compared to Orca and Sarathi, respectively.

Varying the number of experts. To evaluate Saber in dealing with different number of experts, we use Switch-16/32/64 and present the results in Table 4. We observe that a larger number of total experts leads to higher activated experts under the same number of input tokens. For example, Switch-32 activates 1.30× more experts than Switch-16, while Switch-64 activates 1.22× more than Switch-32. Additionally, as the total number of experts increases, the cache hit rate declines due to both a greater demand for activated experts and the reduced accuracy in predicting them. This results in higher extra I/O overhead from cache misses, and longer batch execution time accordingly. Notably, Saber mitigates this overhead by up to 8.1×, owing to its sparsity-aware batching (which reduces the number of activated experts) and its accurate expert preloading (which improves cache

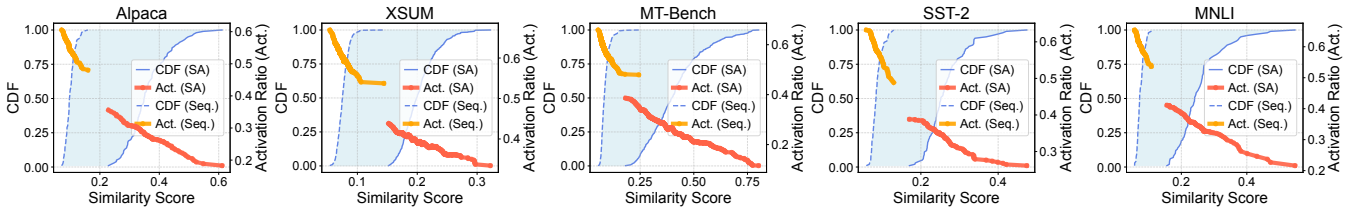

Figure 17: Illustration of the path similarity and grouping opportunity using QWen1.5-MoE across 5 benchmark datasets. The shaded regions indicate the optimization space targeted by Saber. "SA" denotes sparsity-aware batching, whereas "Seq." denotes traditional sequential batching.

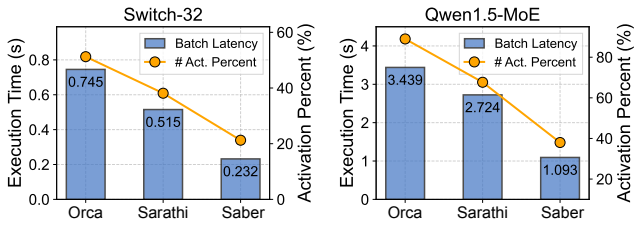

Figure 18: Average batch execution time and expert activation percent of a prefilling batch.

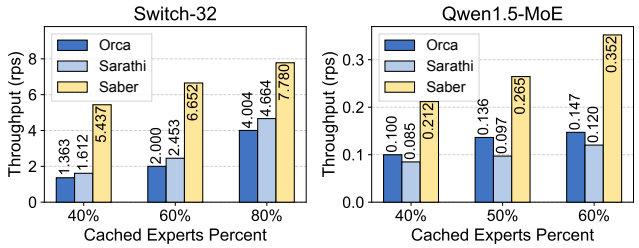

Figure 19: Impact of memory budget on throughput.

Method	Sequential batching			Saber's batching		
Experts per layer	16	32	64	16	32	64
# of act. experts	11.8	15.3	18.7	6.0	7.7	8.9
Cache hit rate (%)	54.2	40.5	27.1	89.3	72.2	68.5
Extra I/O cost (GB)	0.57	0.96	1.44	0.07	0.23	0.30
Batch time (s)	0.26	0.38	0.63	0.09	0.16	0.19

Table 4: Impact of the number of experts on system performance. We fix the number of input tokens as 64 and the number of cached experts as 8 per layer.

hit rate). As a result, Saber reduces batch execution time by 2.9×, 2.4×, and 3.3× compared with the sequential batching method for Switch-16, 32, and 64, respectively.