BrownoutServe: SLO-Aware Inference Serving Under Bursty Workloads for MoE-Based LLMs

Jianmin Hu, Minxian Xu, *Senior Member, IEEE*, Kejiang Ye, *Senior Member, IEEE*, and Chengzhong Xu, *Fellow, IEEE*

***Abstract*—In recent years, the Mixture-of-Experts (MoE) architecture has been widely applied to large language models (LLMs), providing a promising solution that activates only a subset of the model's parameters during computation, thereby reducing overall memory requirements and allowing for faster inference compared to dense models. Despite these advantages, existing systems still face issues of low efficiency due to static model placement and lack of dynamic workloads adaptation. This leads to suboptimal resource utilization and increased latency, especially during bursty requests periods. To address these challenges, this paper introduces BrownoutServe, a novel serving framework designed to optimize inference efficiency and maintain service reliability for MoE-based LLMs under dynamic computational demands and traffic conditions. BrownoutServe introduces "united experts" that integrate knowledge from multiple experts, reducing the times of expert access and inference latency. Additionally, it proposes a dynamic brownout mechanism to adaptively adjust the processing of certain tokens, optimizing inference performance while guaranteeing service level objectives (SLOs) are met. Our evaluations show the effectiveness of BrownoutServe under various workloads: it achieves up to 2.46× throughput improvement compared to state-of-the-art systems and reduces SLO violations by up to 90.28%, showcasing its robustness under bursty traffic while maintaining acceptable inference accuracy.**

***Index Terms*—MoE, LLM inference serving, SLO, brownout, requests burst.**

Received 6 July 2025; revised 12 November 2025; accepted 11 January 2026. Date of publication 20 January 2026; date of current version 13 March 2026. This work was supported in part by the National Natural Science Foundation of China under Grant 62572462; in part by Guangdong Science and Technology Cooperation Project under Grant 2025A0505020065; in part by Guangdong Basic and Applied Basic Research Foundation under Grant 2024A1515010251 and Grant 2023B1515130002; in part by Guangdong Special Support Plan under 2021TQ06X990; in part by the Key Research and Development and Technology Transfer Program of Inner Mongolia Autonomous Region under Grant 2025YFHH0110; and in part by Shenzhen Basic Research Program under Grant JCYJ20220818101610023, Grant KJZD20230923113800001, and Grant JCYJ20240813155810014. Recommended for acceptance by Y. Hu. (*Corresponding author: Minxian Xu.*)

Jianmin Hu is with the Southern University of Science and Technology, Shenzhen 518055, China, and also with Shenzhen Institutes of Advanced Technology, Chinese Academy of Sciences, Shenzhen 518055, China.

Minxian Xu and Kejiang Ye are with Shenzhen Institutes of Advanced Technology, Chinese Academy of Sciences, Shenzhen 518055, China (e-mail: mx.xu@siat.ac.cn).

Chengzhong Xu is with the State Key Lab of IOTSC, University of Macau, Macau 999078, China.

Digital Object Identifier 10.1109/TC.2026.3655019

I. Introduction

MOE [1] architecture has garnered widespread attention in the field of LLMs, with a surge of research and applications in recent years, such as Mixtral [2], Qwen3 [3], DeepSeekV3 [4], and their variants. MoE architectures significantly reduce training and inference costs by activating only a subset of model parameters (experts) during computation [5]. However, efficient inference for MoE models remains a challenging problem in both industry and academia.

LLM inference is divided into two stages: prefill and decoding [6]. During the prefill stage, the model processes the user's input to generate the first output token, caching the generated key-value pairs in the key-value (KV) cache to avoid redundant computation. This stage involves processing a large number of tokens, activating almost all experts, and often becomes a performance bottleneck. In the decoding stage, each request processes only one token at a time, but when the model is under high load (i.e., the current batch size approaches the maximum batch size), the MoE module can still become a bottleneck. Studies [5], [7], [8] have shown that due to the varying activity levels of different experts, only a few experts handle a large number of tokens (hot experts), while most experts handle very few tokens (cold experts). This imbalance prevents these cold experts from fully utilizing GPU parallelism, leading to decreased resource utilization. Fig. 1 illustrates that in the prefill stage, the latency of the MoE accounts for around 81.23% of the total latency of the transformer layer (including attention and MoE), and in the decoding stage, the MoE latency accounts for about 93.89% of the total latency of the transformer layer. Therefore, reducing the latency of the MoE module is crucial to reducing the overall inference latency.

Existing MoE inference systems, such as vLLM [9] and DeepSpeed-MoE [10], have introduced techniques like tensor parallelism and expert parallelism to accelerate inference. While these systems incorporate certain dynamic scheduling mechanisms (e.g., continuous batching [11]), they lack fine-grained, model-level adaptation specifically designed to handle the internal dynamics of MoE architectures under bursty loads. Consequently, they struggle to mitigate sudden load spikes and request fluctuations effectively, often leading to SLO violations. In low-resource GPU clusters, with a limited number of GPUs, restricted memory capacity, and constrained communication bandwidth, MoE inference faces several key challenges: uneven expert load causing computational bottlenecks, high cross-GPU

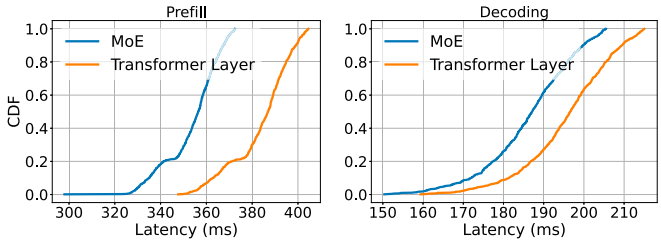


Fig. 1. Cumulative distribution function (CDF) of latency for the prefill and decoding stages of the MoE module and transformer layer using the Qwen1.5-MoE-A2.7B model with a batch size of 64 on the alpaca dataset, utilizing four A100-40GB-PCIE GPUs. The left plot illustrates the latency distribution for the prefill stage, and the right plot shows the latency distribution for the decoding stage.

communication costs exacerbated by parallelism techniques, and large model size making it costly and inflexible to scale up (e.g., by changing the tensor parallelism degree, which requires service restart) in response to bursty requests. These challenges significantly limit the usability and practicality of MoE models in small-to-medium-scale deployment environments with limited resources.

To address these challenges, we introduce the concept of united experts and brownout approach to reduce inference latency of MoE module. United experts integrate the knowledge of multiple experts into a single expert of equivalent parameter size, reducing the times of expert access during inference, increasing the average number of tokens processed by each expert, and fully leveraging GPU parallelism to reduce latency and improve throughput. The brownout approach is inspired by the brownout strategies in power systems. This approach has also been applied in cloud computing environment that can dynamically activate or deactivate optional application components based on different workloads [12]. In this work, the brownout approach is used to determine which tokens are processed by the original experts and which are processed by the united experts with accuracy and efficiency trade-offs. To mitigate this, based on the brownout approach, we propose the SLO-Aware Latency Control algorithm, which can dynamically adjust the brownout configuration to minimize accuracy loss while ensuring inference latency meets SLO, even under fluctuating request rate and bursty workloads.

In this paper, we present BrownoutServe, an efficient MoE inference serving framework designed for small-to-medium-scale GPU clusters. The core design of BrownoutServe includes two key mechanisms: First, the united expert model, which integrates the knowledge of multiple experts into a united expert to reduce expert access during inference. Second, the dynamic brownout mechanism (brownout approach and SLO-Aware Latency Control algorithm), which dynamically routes a subset of tokens to the united experts under resource constraints or bursty workloads, reducing expert access overhead and helping maintain SLO attainment. Our evaluations demonstrate the effectiveness of BrownoutServe in various workloads. Our contributions are as follows:

- We have developed an efficient inference framework for MoE-based LLMs, featuring the united experts-based MoE module and an SLO-Aware Latency Control algorithm-driven scheduler.
- We have designed the concept of united experts, the brownout approach and the SLO-Aware Latency Control algorithm, all of which can be flexibly adapted in scenarios with bursty workloads, and elaborate on three brownout strategies: zero-brownout, full-brownout, and partial-brownout.
- We have comprehensively evaluated BrownoutServe's performance across various realistic workloads. Compared to state-of-the-art systems, BrownoutServe achieves up to 2.46× higher throughput and reduces SLO violations by up to 90.28%.

II. Related Work

In this section, we discuss the related work on inference optimization for LLMs. Current work can be categorized into two classes as follows:

Optimization for MoE-based LLMs. As MoE-based architectures evolve and model parameters expand exponentially, a multitude of optimization approaches have emerged. Representative work includes GShard [13] which significantly improves computational efficiency and scalability by distributing expert modules of the MoE layer across multiple devices. Switch Transformer [8] simplified routing by activating only one expert per token. Lina [14] uses dynamic resource scheduling and expert popularity estimation to balance the workload across devices and improve inference efficiency. MPMoE [15] introduces adaptive pipeline parallelism and memory-reuse strategies to accelerate training and reduce memory overhead. EfficientMoE [16] uses dynamic scheduling and capacity adjustment to optimize the training efficiency of MoE models. Tutel [17] presents a highly scalable stack design and implementation for MoE, featuring dynamically adaptive parallelism and pipelining. Pre-gated MoE [18] introduces a novel pre-selection mechanism (pre-gating function), which predicts the set of experts to be activated before inference. This allows for preloading necessary parameters, thereby reducing the overhead of dynamic scheduling during inference. MoE-Lightning [19] implements CPU-GPU-I/O pipeline scheduling and a Hierarchical Roofline Model, enabling high-throughput MoE inference on memory-constrained GPUs. Although these optimization techniques have accelerated inference to some extent, they lack the ability to handle bursty requests, thereby causing SLO violations. Our work addresses these limitations by introducing novel mechanisms such as united experts and the brownout approach to dynamically optimize inference performance and maintain service reliability under fluctuating workloads.

Optimization for generic LLMs There are many related works focused on optimizing inference for generic LLMs. For KV cache optimization, vLLM [9] proposes PagedAttetion for KV cache management, which reduces GPU memory fragmentation. InfiniGen [20] reduces data transfer overhead by dynamically selecting and prefetching only the essential KV

cache entries instead of loading all of them. For parallelism strategies optimization, LoongServe [21] provides an Elastic Sequence Parallelism strategy, which effectively improves resource utilization when serving long-context LLMs. alpaServe [22] leverages model parallelism: even when a model fits on a single device, it can be partitioned across multiple devices to enable statistical multiplexing, allowing better handling of bursty requests. DistServe [6] and Splitwise [23] decouples the prefill and decoding phases to eliminate interference between them and applies separate parallelism strategies for each stage. For scheduling optimization, Orca [11] proposes iteration-level scheduling, a more fine-grained scheduling approach where completed requests are removed from the batch after each iteration, and new requests can be added to the batch for the next iteration. Luminix [24] introduces a dynamic scheduling mechanism inspired by context switching in operating systems, which enables real-time reallocation of requests among multiple model instances, achieving better load balancing and resource isolation. These optimization techniques, while originally designed for generic LLMs, are still applicable to MoE-based LLMs. BrownoutServe is orthogonal to these techniques and can work in conjunction with them. Therefore, BrownoutServe integrates some of these techniques, such as using PagedAttention to optimize KV cache management and adopting iteration-level scheduling to improve the scheduler performance. These integrations help BrownoutServe achieve better overall optimization for MoE-based LLMs.

III. Background and Motivation

In this section, we briefly introduce MoE-based LLMs and the theoretical foundations of service. We then reveal the two main challenges currently faced by MoE-based LLMs in real-world deployment.

A. Mixture-of-Experts

The MoE-based LLMs replace the feed-forward networks (FFNs) in standard Transformer-based [7] LLMs with a gating function and multiple small-parameter FFNs, enabling sparse activation and improved computational efficiency. Compared to traditional dense models [25], [26], [27], MoE offers a solution to reduce computational and memory requirements by activating only a subset of the model's parameters, known as "experts", during computation. This selective activation mechanism can reduce the overall memory footprint and computational load, enabling more efficient use of resources and facilitating the training and deployment of models with trillions of parameters.

The experts in the MoE layer typically consist of two types: shared experts and routed experts. The shared experts are a small, fixed set of experts that process every token, providing a common computational base to preserve essential model capacities and stabilize training. In contrast, the routed experts form the majority and are sparsely activated. For each incoming token, a gating function computes a score against each routed expert, and only the top-k experts with the highest scores are activated to process the token. The end-to-end token routing process within an MoE layer can be summarized in the following steps:

- **Gating Computation** : For a given token, the gating network calculates an affinity score between the token's hidden state and each routed expert.
- **Expert Selection** : The top- k routed experts with the highest scores are selected. The token is then replicated and sent to these experts.
- **Expert Computation** : Each of the selected routed experts processes the token through its local FFN. In parallel, the shared experts (if present) also process the token.
- **Output Aggregation** : The outputs from the activated routed experts and shared experts are combined, typically through a weighted sum based on their gating scores, to form the final output of the MoE layer for that token.

Like dense models, the inference process for MoE typically involves two stages: the prefill stage, where the model generates an initial output token based on the user's input, and the decoding stage, where subsequent tokens are generated one at a time. The MoE architecture is particularly beneficial during these stages as it can adjust the number of active experts based on the input data, leading to more efficient processing and reduced latency compared to dense models. Each expert represents a small portion of the knowledge contained within the model, thus for each incoming token, it first needs to undergo computation by a gating function to determine which experts should process it next.

B. Theoretical Foundations of LLM Service Modeling

Request Modeling: In LLM services, user requests arrive randomly and can be modeled as a Poisson process, where the inter-arrival times follow an exponential distribution. This assumption is widely adopted in network and distributed system analysis. Given that the inference time per request is nearly constant (especially when the batch size is 1), the system can be modeled as a classic $M/D/1$ queue [28], and the total average response time W can be approximately expressed as:

$$W = \frac{\lambda \tau^2}{2(1-\lambda\tau)} + \tau, \tag{1}$$

where λ is the average request arrival rate, and τ is the fixed service time per request. This expression captures both the waiting time (the first term) in the queue and the actual processing time (the second term), and it highlights that as the system approaches saturation (i.e., $\lambda\tau \to 1$) the response time grows rapidly.

System Optimization: According to Amdahl's Law [29], the overall system speedup is limited by the proportion of the workload that is not optimized. The formula is as follows:

$$Speedup = \frac{1}{(1-\alpha) + \frac{\alpha}{K}}, \tag{2}$$

where α represents the fraction of execution time consumed by the bottleneck component in the original system, K denotes the speedup factor achieved by optimizing that same component. This law emphasizes that the overall performance improvement of a system depends on the extent to which the performance

bottleneck is optimized. For example, suppose the bottleneck component accounts for $\alpha = 0.6$ (i.e., 60%) of the total processing time. If we optimize it by a factor of $K = 3$ through parallelization, kernel fusion, or communication optimization, the overall system speedup would be:

$$Speedup = \frac{1}{(1-0.6) + \frac{0.6}{3}} = \frac{1}{0.4 + 0.2} = \frac{1}{0.6} \approx 1.67. \quad (3)$$

This result illustrates that focusing on a single bottleneck can substantially enhance system performance.

C. Challenges in MoE-Base LLMs Inference Serving

Although the MoE architecture significantly reduces computation and memory costs during the training and inference stages of LLMs, it also introduces some systemic challenges during real-world deployment and online serving. These challenges mainly fall into the following two categories:

C1: Computational Pressure with Large Batch Size and Experts Load Imbalance. Compared to dense models, MoE models exhibit a much larger total number of parameters due to the presence of multiple experts, even though each inference path only activates a sparse subset of them. In scenarios with large batch sizes, a high number of tokens are routed to experts simultaneously, potentially activating nearly all experts. This turns the inference process from sparse to quasi-dense, resulting in intense compute and memory bandwidth pressure on the system. Although top-k routing strategies (e.g., Top-2) used in GShard [13] and Switch Transformer [8] aim to limit the number of experts each token activates, higher values of k are often required in practice to maintain model accuracy. This leads to frequent activation of most experts, especially under large batches, causing communication bandwidth bottlenecks in multi-GPU or multi-node deployments. Moreover, token distribution across experts typically exhibits a long-tail pattern: only a small portion of the experts handle more tokens than average, while the majority are severely underutilized. This load imbalance results in idle compute resources and exacerbates performance degradation due to communication overhead and synchronization latency, particularly in distributed setups.

C2: SLO Violations under Bursty Workloads. Workloads bursty is a common scenario across various service types, and MoE-based serving systems struggle to meet SLOs under such conditions, especially in GPU clusters with limited resources. Most mainstream elasticity mechanisms still rely on traditional horizontal scaling methods, such as AWS EC2 instance spawning or Kubernetes [30] HPA-based auto-scaling, to handle bursty workloads. However, these approaches suffer from high cold start latency and substantial GPU memory overhead. Cloud instances typically take 30–60 seconds to initialize. MoE models often require 30–90 seconds to load weights due to their large size (tens of GBs); Additional delays stem from container cold starts, network topology reconstruction, requests forwarding. These latencies can lead to total startup latencies exceeding 1–2 minutes, which is unacceptable for latency-sensitive applications such as chatbot or real-time recommendation systems. This untimely scaling often leads to substantial SLO violations. Although emerging systems such as ServerlessLLM [31] attempt to reduce cold start latency through mechanisms like lazy weight loading and model migration, these approaches often rely on high-speed local caching and high-bandwidth environments, placing greater pressure on GPU memory capacity and network I/O. Additionally, horizontally scaling LLM instances often requires provisioning multiple full copies of the model, leading to memory consumption that is several times higher than the unscaled baseline [32]. This results in considerably increased hardware costs for cloud providers. What exacerbates the problem is that bursty traffic is often short-lived. Once the workloads decrease, the provisioned GPU instances remain underutilized or idle, incurring inefficient resource usage and wasted infrastructure cost. Such elasticity mismatches highlight the limitations of traditional scaling approaches in handling highly dynamic LLM workloads.

The fundamental distinction between the two challenges lies in their nature and origin: C1 represents a static, inherent inefficiency within the MoE architecture itself, primarily characterized by severe load imbalance across experts that leads to poor GPU utilization and limited throughput, even under stable workloads. In contrast, C2 is a dynamic, external service reliability problem, where the system's inability to rapidly scale in response to sudden traffic bursts results in high latency and SLO violations, highlighting a critical shortfall in operational elasticity.

To tackle C1, we propose the concept of the united experts, which merges multiple experts into a single, consolidated expert. This can reduce the number of experts involved in computation, thereby alleviating the computational pressure during inference. Additionally, by combining underutilized experts that receive relatively few tokens, we can mitigate the issue of low GPU utilization, leading to more efficient resource usage and better system throughput.

To tackle C2, we introduce the brownout approach along with the SLO-Aware Latency Control (SALC) algorithm to reduce latency. This mechanism dynamically adjusts a part of tokens processed by united experts based on real-time inference latency. This adaptive token routing helps reduce the overall latency introduced by the MoE module, which is often the bottleneck in the inference, while maintaining acceptable accuracy levels. As shown in Eq. (2), since the MoE module dominates end-to-end latency in typical LLM inference workloads, applying the brownout approach and SALC algorithm effectively lowers total system latency, enabling the system to better meet SLO requirements under bursty traffic conditions.

IV. Design and Implementation of BrownoutServe

In this section, we present BrownoutServe, an efficient and reliable inference framework for MoE-based LLMs, specifically designed to handle bursty workloads while maintaining SLO attainment.

A. Overview

BrownoutServe is an end-to-end MoE-based serving framework and its overview is shown in Fig. 2. BrownoutServe adopts

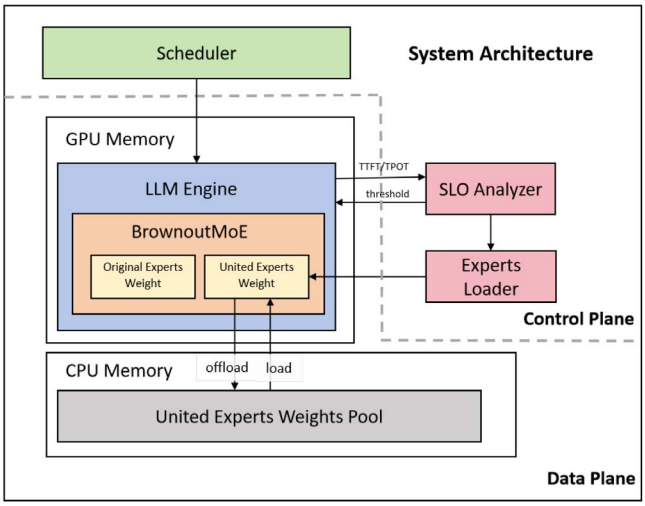


Fig. 2. Overview of BrownoutServe design.

a modular design that divides the entire system into two major parts: the control plane and the data plane. The control plane consists of `Scheduler`, `SLO Analyzer`, and `Experts Loader`, while the data plane is comprised of a highly optimized LLM inference engine that integrates the BrowoutMoE module mentioned in Section IV-D. Additionally, the BrowoutMoE module includes a pluggable fused MoE [33]. It is crucial to note that Fused MoE and our core contribution, United Experts, address orthogonal challenges: Fused MoE optimizes kernel launch overhead, while United Experts reduce the memory and bandwidth pressure by decreasing the volume of expert parameters loaded. They are complementary and can be employed together, as we have done in the experimental evaluation presented in Section V. The LLM engine also incorporates existing optimization techniques, such as FlashAttention [34], PagedAttention [9], and ContinuousBatching [11]. The entire system is implemented using approximately 5.5k lines of Python code.

Control Plane. `Scheduler` uses the first-come first-served (FCFS) algorithm to schedule incoming requests. When the engine reaches its maximum batch processing capacity, excess requests are stored in a waiting queue and processed in sequence. The scheduler supports streaming inputs and outputs, allowing for the dynamic insertion of new requests and the early removal of completed requests. `SLO Analyzer` continuously monitors the LLM engine's inference metrics, including time-to-first-token (TTFT) and time-per-output-token (TPOT), and uses the SLO-Aware Latency Control algorithm to timely adjust the brownout configuration to reduce SLO violations. `Experts Loader` is responsible for loading and unloading the united experts, as well as updating the way of united experts.

Data Plane. The LLM inference engine is primarily built based on PyTorch [35]. For some operations, such as PagedAttention and FlashAttention, they are implemented using Triton [36] language instead of traditional C++/CUDA, because of Triton's seamless compatibility with PyTorch. Compared to PagedAttention proposed in vLLM, we optimize it by moving the block table to the GPU and implement block table operations as kernels (functions executed on the GPU), fully leveraging the parallel computing capability of the GPU to effectively reduce the additional overhead incurred by PagedAttention. This process converts the block table from a two-dimensional list in CPU memory to a two-dimensional tensor on the GPU, where the first dimension indexes sequences and the second dimension stores physical block indices of each sequence. The MoE module introduces the brownout approach, and to further optimize performance, we have also rewritten the MoE-related operators using Triton.

B. United Expert Model

We introduce a novel model in MoE architecture, termed the "united expert". This is an efficient expert model by integrating the knowledge from multiple expert models. This innovative approach alleviates the issue of insufficient GPU computational resources and under-utilization in some unpopular experts, reduces the frequency of expert access, thereby effectively decreasing inference latency and enhancing the efficiency of model inference services.

For each transformer layer, which contains m experts (subsequently called original experts), we divide these m experts into $\lceil \frac{m}{k} \rceil$ groups, where k is the way of united experts (the number of experts in each group). For each group of original experts, we train a united expert via knowledge distillation [37], enabling it to take over their work within the brownout mechanism. During the training process, all original experts in this group serve as the teacher model and this united expert is used as the student model [37]. The output of the original experts (i.e. hidden states) will be used as the training target for the united experts. In this process, by minimizing the difference between the hidden states of the united expert and original experts in this group, the united expert model will learn the comprehensive knowledge of the original experts. Through this method, the united expert model can learn to approximate the knowledge representation of each group of original experts, thereby reducing the times of model access and maintaining the model's inference accuracy across various tasks. We use the mean squared error (MSE) loss function to measure the difference between the united expert and the original experts, which can be mathematically described as follows:

$$\mathcal{L}_{\text{MSE}}^{j} = \frac{1}{k}\left(\sum_{i=0}^{k-1}\left\|H_{u}^{j} - H_{o}^{j\times k+i}\right\|^{2}\right), \tag{4}$$

where $\mathcal{L}_{\text{MSE}}^{j}$ is the difference between the original experts and the united expert for the j -th group. k is the number of original experts in each group. H_{u}^{j} is the hidden states of the j -th united expert, and $H_{\text{o}}^{j\times k+i}$ is the hidden states output by the i -th original expert in the j -th group, since it is grouped according to the expert index, $H_{\text{o}}^{j\times k+i}$ can also be understood as the $(j \times k + i)$-th original expert of the transformer layer. The range of j is from 0 to $\lceil \frac{m}{k} \rceil - 1$. After training, we obtain $\lceil \frac{m}{k} \rceil$ united experts in each transformer layer.

Fig. 3 shows a specific example of training process for united experts. In this example, we have 8 original experts, and we divide them into groups of 2, resulting in $\lceil \frac{8}{2} \rceil = 4$ groups. We

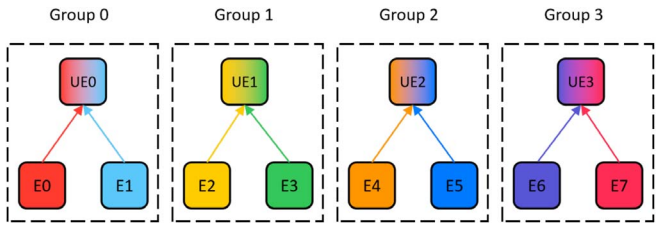


Fig. 3. Training process of four united experts (UE0-UE3), each integrating knowledge from eight original experts (E0-E7), color-coded to represent different token subsets.

can use the method described above to train a united expert for each group, which generally includes the knowledge of all original experts in the group. For instance, for Group 0, in a subsequent inference iteration, if the number of tokens processed by E0 and E1 is relatively small, we can concatenate the tokens to be processed by these two original experts together and provide them to the united expert UE0 of the group for processing, in order to improve GPU computing utilization.

To illustrate the detailed training procedure, the knowledge distillation process for united experts is formalized in Algorithm 1. The algorithm takes as input the set of m original experts E, the set of united experts UE (with one UE for every k original experts), a gating function, the training dataset, and the learning rate. Its goal is to output the trained united experts (lines 1-7). Each united expert and its corresponding optimizer are initialized (lines 8-11). The core training begins by iterating over batches of input tokens X from the dataset D. For each batch, the gating function assigns each token to its top- K original experts (lines 13-14). For each original expert E_i, the algorithm collects all tokens assigned to it. If there are any, E_i processes them to produce the $teacher\_outputs$ (lines 15-18). The algorithm finds the corresponding united expert UE_j responsible for learning from E_i. UE_j then processes the same tokens to produce its $student\_outputs$. The loss is computed as the Mean Squared Error between the $student\_outputs$ and the $teacher\_outputs$, normalized by the group size k to balance the influence of each expert (lines 19-22). The gradients are computed via backpropagation from the loss, and the parameters of the united expert UE_j are updated by the optimizer. This step minimizes the output difference, forcing UE_j to mimic E_i 's functionality (lines 23-25).

Algorithm 1 United Expert Training Process

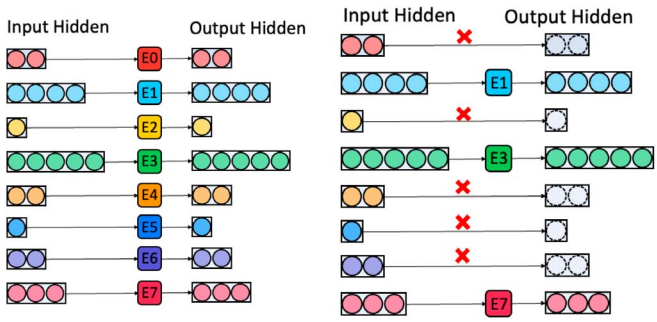
1: **Input:**
2: Original experts set $E = \{E_0, E_1, ..., E_{m-1}\}$ for each layer
3: United experts set $UE = \{UE_0, UE_1, ..., UE_{\lfloor (m-1)/k \rfloor}\}$
4: Gating function Gate$(\cdot)$ for expert routing
5: Training dataset D of input tokens
6: Learning rate η
7: **Output:** Trained united experts set UE
8: **for** each united expert UE_j in UE **do**
9: Initialize UE_j's parameters
10: Set optimizer for UE_j with learning rate η
11: **end for**
12: Set loss function $\mathcal{L}$ to Mean Squared Error (MSE)
13: **for** each training batch of input tokens X in D **do**
14: expert_assign $\leftarrow$ Gate(X)
15: **for** each original expert E_i in E **do**
16: tokens_for_$E_i \leftarrow$ get_tokens$(X,$ expert_assign$, E_i)$
17: **if** tokens_for_E_i is not empty **then**
18: teacher_outputs $\leftarrow E_i$(tokens_for_E_i)
19: united_expert_idx $\leftarrow \lfloor i/k \rfloor$
20: $UE_j \leftarrow UE$[united_expert_idx]
21: student_outputs $\leftarrow UE_j$(tokens_for_E_i)
22: loss $\leftarrow \frac{1}{k} \cdot \mathcal{L}$(student_outputs, teacher_outputs)
23: optimizer.zero_gradients()
24: loss.backward()
25: optimizer.step()
26: **end if**
27: **end for**
28: **end for**
29: **Return** UE

C. Brownout Approach

The application of the brownout approach in MoE-based serving draws an analogy from the brownout mechanisms of the power system. In scenarios where power resources are strained or face a surge in demand, power companies may implement brownout strategies, temporarily disconnecting non-essential facilities to guarantee the power supply to critical infrastructure. Similarly, we propose the brownout approach within the MoE architecture. When computational resources are limited or there is a sudden bursty of user requests, the brownout strategy can be employed to degrade the processing of certain tokens, thereby optimizing the model's inference efficiency and guaranteeing SLOs. In this subsection, we introduce three brownout strategies: zero-brownout, full-brownout, and partial-brownout.

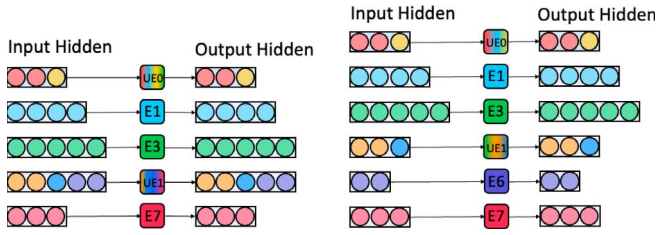
Zero-Brownout. Zero-brownout is a strategy that does not implement any degrading measures, meaning that the number of expert model activations is never reduced under any circumstances. Under this strategy, each input token is processed by all relevant original expert models. This strategy ensures that all tokens are processed by original experts, thereby providing the highest theoretical accuracy in model inference. Existing MoE inference engines can be considered as adopting zero-brownout, suitable for environments with ample computational resources and stable user request loads. Fig. 4(a) shows a simple example of zero-brownout. There are a total of 8 experts and 20 tokens. After passing through the gate unit, the tokens are allocated to the corresponding experts for processing, and no tokens are degraded or dropped out.

Full-Brownout. The idea of full-brownout lies in strictly controlling the number of expert activations in the model. Considering that expert access is the bottleneck of the entire model's inference performance, reducing the times of expert access appropriately can effectively reduce the inference latency to meet user SLO requirements. This mechanism requires setting a $threshold$ $(0 \leq threshold \leq 1)$ in advance, which determines the proportion of tokens that need to be processed by original experts, while other tokens will be ignored in current transformer layer. To minimize inference latency, we need to find



(a) Zero-brownout (b) Full-brownout

Fig. 4. Two brownout strategies illustration with 8 experts (E0-E7) processing 20 tokens.



(a) Common case. (b) Special case.

Fig. 5. Two type cases of partial-brownout strategy.

fewer experts to handle more tokens. For example, if we have 8 experts, 20 tokens, and a threshold of 0.6, as shown in Fig. 4(b), we need to select 12 tokens for their corresponding experts to process, while ignoring the other 8 tokens. Therefore, we will choose E1, E3, E7, these three experts who can handle no less than 12 tokens and meet the established requirements. In this example, we processed 60% of the tokens only accessing 37.5% of the experts.

Partial-Brownout. This is an improved approach based on full-brownout. Unlike full-brownout, which ignores the processing of some tokens, the idea of partial-brownout is delegating these tokens that were originally ignored in the full-brownout to their corresponding united experts, thereby reducing inference latency while ensuring a certain level of inference accuracy. Suppose each transformer layer has m experts, in addition to the parameter $threshold$, we also need the size of the expert group k, as well as $\lceil \frac{m}{k} \rceil$ united experts trained in advance through knowledge distillation techniques. Based on the example mentioned in full-brownout, we assume each transformer layer has 8 experts, E0-E7, with each expert group consisting of 4 experts (i.e., $k = 4$). The number of tokens is 20, and the number of tokens each expert needs to process is 2, 4, 1, 5, 2, 1, 2 and 3 respectively. With a threshold of 0.6, as in the example in Fig. 5(a), 12 tokens need to be processed by the original experts. Thus, we let E1, E3, and E7 handle the corresponding tokens, while E0, E2, E4, E5, and E6 delegate their tokens to the corresponding united experts instead of ignoring them. If each group contains 4 original experts, there are 2 expert groups. UE0 needs to process the tokens of E0 and E2 (a total of 3 tokens), while UE1 needs to process the tokens of E4, E5 and E6 (a total of 5 tokens). This way, we only need to access 5 experts (3 original experts and 2 united experts) to process all tokens, compared to the 8 times of expert access in zero-brownout, reducing the times of expert accesses by 3, and also increasing the batch size that experts can process at one time. Since the united experts integrate the knowledge of all experts in the group, this approach not only enhances inference efficiency, but also effectively mitigates the loss of model accuracy caused by the full-brownout strategy.

Special Case in Partial-Brownout. In the partial-brownout strategy, a special scenario arises when a united expert is designated to process tokens that originate solely from one original expert. Under such circumstances, it becomes more accurate to allow the original expert to handle these tokens directly rather than involving the united expert. This is because the involvement of the united expert does not contribute to reducing the times of expert accesses; on the contrary, it may potentially degrade the inference accuracy. Referring to Fig. 5(b), if we set k =3, with a total of 8 experts, they can be divided into 3 groups. UE0 is required to process tokens from E1 and E2 (a total of 3 tokens), and UE1 is tasked with handling tokens from E4 and E5. However, the tokens from E6 are not delegated to UE2 for processing but are instead processed directly by E6 itself.

Algorithm Design. To decide when to trigger brownout approach, we propose Algorithm 2, which illustrates the specific execution process of the three brownout strategies mentioned above. After passing through the gating unit of the MoE module, we will obtain the number of tokens processed by each expert and their hidden states. The number of original experts, denoted as m, can be obtained from the model configuration. The threshold in the brownout method τ and the parameter $use\_full\_brownout$ (whether to enable full brownout) can be obtained from the brownout configuration, which is set by the user in advance (lines 1-5). After passing through the MoE module, the output of these tokens is stored in the variable $\mathbf{Y}$ (line 6). The input tokens are sorted and then divided into two sets. Tokens in the set $S1$ are processed by the original experts without any accuracy loss, while tokens in the set $S2$ are processed by the united experts, which may incur some accuracy loss (lines 13–20). Tokens in $S1$ are directly processed by the original experts (lines 21–23). For the experts in $S2$ they are first grouped based on m and k. All tokens in each expert group are concatenated and processed by the united expert (lines 24–34), including special case (lines 26–28) and common case (lines 30–32). Clearly, when $threshold$ equals to 1, Algorithm 2 describes the zero brownout process. When $0 \leq threshold < 1$ and $use\_full\_brownout$ is $true$, the algorithm describes the full-brownout process. When $0 \leq threshold < 1$ and $use\_full\_brownout$ is $false$, it describes the partial-brownout process.

Time Complexity Analysis. Algorithm 2 has m original experts and $\lceil \frac{m}{k} \rceil$ united experts. It sorts the input tokens first and divides them into two sets: $S1$ and $S2$, then processes these tokens by original experts and united experts respectively, so the

Algorithm 2 BrownoutMoE Algorithm

Input:
Set of original experts $E = \{e_0, e_1, \ldots, e_{m-1}\}$
Hidden states $\mathcal{H}_i$ and token count $n_i = |\mathcal{H}_i|$ for each expert e_i
United expert group size k, brownout threshold $\tau \in [0, 1]$
Brownout strategy flag $use\_full\_brownout \in \{\text{True}, \text{False}\}$
Output: Output hidden states $\mathbf{Y}$ for all tokens
Initialize $\mathbf{Y} \leftarrow \emptyset$, $S_1 \leftarrow \emptyset$, $S_2 \leftarrow \emptyset$
Expert list: $L \leftarrow [(e_0, n_0, \mathcal{H}_0), \ldots, (e_{m-1}, n_{m-1}, \mathcal{H}_{m-1})]$
Sort L in descending order of n_i
$N_{\text{total}} \leftarrow \sum_{i=0}^{m-1} n_i$
$T \leftarrow \tau \cdot N_{\text{total}}$
$N_{\text{accum}} \leftarrow 0$
for $i \in \{0, 1, 2, ..., m-1\}$ **do**
  if $N_{\text{accum}} \leq T$ **then**
    $S_1 \leftarrow S_1 \cup \{L[i]\}$
  else if not $use\_full\_brownout$ **then**
    $S_2 \leftarrow S_2 \cup \{L[i]\}$
  end if
  $N_{\text{accum}} \leftarrow N_{\text{accum}} + n_i$
end for
for each $(e_i, n_i, \mathcal{H}_i) \in S_1$ **do**
  $\mathbf{Y} \leftarrow \mathbf{Y} \cup \text{FFN}_i^{(r)}(\mathcal{H}_i)$
end for
$G \leftarrow \text{group\_by\_index}(S_2, k)$
for each group $g \in G$ **do**
  if $|g| = 1$ **then**
    Let $(e_i, n_i, \mathcal{H}_i)$ be the element in g
    $\mathbf{Y} \leftarrow \mathbf{Y} \cup \text{FFN}_i^{(r)}(\mathcal{H}_i)$
  else
    $\mathcal{H}_{\text{concat}} \leftarrow \bigcup_{(e_i, n_i, \mathcal{H}_i) \in g} \mathcal{H}_i$
    $j \leftarrow \lfloor i/k \rfloor$
    $\mathbf{Y} \leftarrow \mathbf{Y} \cup \text{FFN}_j^{(u)}(\mathcal{H}_{\text{concat}})$
  end if
end for
Return $\mathbf{Y}$

total time complexity is $O(mlogm + m + m + \lceil \frac{m}{k} \rceil)$, which equals to $O(mlogm + 2m + \lceil \frac{m}{k} \rceil)$.

D. Architecture of BrownoutMoE

Compared to existing MoE architectures such as DeepSeek-MoE, which have been widely applied in the DeepSeek series of models, BrownoutMoE introduces united experts to assist in inference, with the aim of reducing inference latency during emergency periods and guaranteeing SLOs. Fig. 6 illustrates the brownoutMoE architecture and we will provide a detailed introduction to the architecture in this section. $\mathbf{x}_t$ denotes the t-th token input to the MoE module. Then, the output $\mathbf{h}_t$ of the MoE module can be calculated as below:

$$\mathbf{h}_t = \mathbf{x}_t + \sum_{i=1}^{N_s} \text{FFN}_i^{(s)}(\mathbf{x}_t) + \sum_{i=1}^{N_r} p_{i,t} \text{FFN}_i^{(r)}(\mathbf{x}_t) + \sum_{i=1}^{N_u} q_{i,t} \text{FFN}_{f(i)}^{(u)}(\mathbf{x}_t), \tag{5}$$

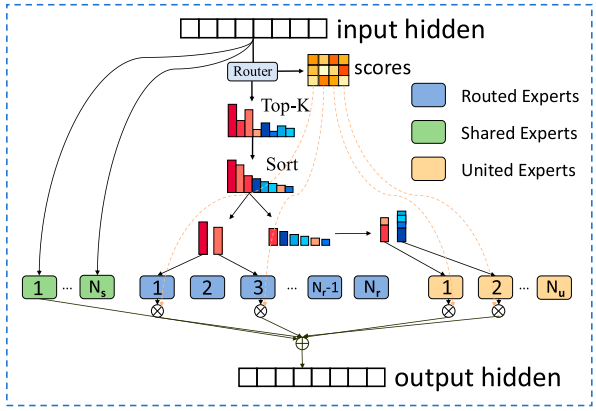


Fig. 6. Illustration of MoE with brownout approach.

where N_s, N_r, N_u are the numbers of shared experts, routed experts (original experts), and united experts, respectively; $\text{FFN}_i^{(s)}(*)$ and $\text{FFN}_i^{(r)}(*)$ denote the i-th shared expert and the i-th routed expert, respectively; $\text{FFN}_{f(i)}^{(u)}(*)$ denotes the united expert corresponding to the i-th original expert. $p_{i,t}$ denotes the score between the t-th token and the i-th original expert if this token belongs to set $S1$. while $q_{i,t}$ refers to the score between the t-th token and the i-th original expert if this token belongs to set $S2$. And they can be described as:

$$p_{i,t} = \begin{cases} g_{i,t}, & \text{if } i \in S1, \\ 0, & \text{otherwise.} \end{cases}, \quad q_{i,t} = \begin{cases} g_{i,t}, & \text{if } i \in S2, \\ 0, & \text{otherwise.} \end{cases} \tag{6}$$

where $S1$ and $S2$ denote the set that needs to be processed by the original experts and that needs to be processed by the united experts, respectively, as shown in Algorithm 2. The $g_{i,t}$ refers to the score between the t-th token and the i-th expert, which can be described as:

$$g_{i,t} = \begin{cases} \frac{\exp(s_{i,t})}{\sum_{j \in \text{TopK}} \exp(s_{j,t})}, & \text{if } i \in \text{TopK}(\{s_{j,t} | 1 \leq j \leq N_r\}, K), \\ 0, & \text{otherwise.} \end{cases}, \tag{7}$$

where $\text{TopK}\ (*, K)$ is the set of scores consisting of the K highest scores among the affinity scores calculated for the t-th token and all routed experts. $s_{i,t}$ is the affinity between the t-th token and the i-th expert, which is calculated by the gate function, and can be described as:

$$s_{i,t} = \mathbf{x}_t^T \mathbf{e}_i, \tag{8}$$

where $\mathbf{e}_i$ is the centroid vector of the i-th routed expert.

E. SLO-Aware Latency Control Algorithm

Tradeoff Analysis. In the brownout configuration, the parameters k and $threshold$ determine the effectiveness of the brownout mechanism. A larger k and a smaller $threshold$ typically yield greater improvements in inference latency. However, they also lead to increased accuracy degradation. Intuitively, a larger k means that each united expert needs to integrate knowledge from more original experts, while its parameter

capacity remains fixed, thus reducing the amount of knowledge it can capture from each individual expert. A smaller $threshold$ implies that more tokens are routed to united experts. Therefore, a trade-off must be made between inference latency and prediction accuracy. A common goal is to maximize the average prediction accuracy while ensuring that each token's latency meets the predefined SLO, as expressed in the following formula:

$$\begin{aligned} &\text{maximize } \frac{1}{n}\sum_{i=1}^{n}\text{Accuracy}^{i}(threshold,k),\\ &\text{subject to Latency}_{j}^{i}(threshold,k) \leq \text{ SLO},\\ &\forall i \in \{1,2,\ldots,n\}, \forall j \in \{1,2,\ldots,m_i\}, \end{aligned} \tag{9}$$

where n is the total number of requests to be processed, $\text{Accuracy}^{i}(*,*)$ denotes whether the i-th request' answer is correct, $\text{Latency}_{j}^{i}(*,*)$ is latency of the i-th request's j-th token, and m_i is the i-th request's output length. In actual inference scenarios, the parameter k for brownout approach changes much less frequently than $threshold$. This is because a change in k implies unloading the current united experts from the GPU memory to CPU memory or disk, and then loading the new united experts back into GPU memory. This incurs non-trivial GPU bandwidth overhead. On the other hand, the overhead introduced by changes in $threshold$ is negligible, and as we will see in the following sections, $threshold$ can be reconfigured at each iteration. Therefore, during a certain inference period, k can be treated as a constant value. $\frac{1}{n}\sum_{i=1}^{n}\text{Accuracy}^{i}(threshold,k)$ is a monotonically increasing function with respect to $threshold$. Consequently, our target is to maximize $threshold$ while ensuring that each token's latency meets the predefined SLO.

Algorithm Design. Given that the factors affecting inference latency are diverse and unpredictable, including fluctuations in the request rate, the variability in request input/output lengths, and interference from other services located on the same machine with the LLM inference service, it is challenging to determine an optimal $threshold$ in advance to satisfy Eq. (9). Therefore, we need to dynamically adjust $threshold$ according to real-time latency feedback during serving. Based on this insight, we design the SALC algorithm, shown as Algorithm 3.

The core idea of SALC is to keep latency slightly below the SLO level, as excessively low latency offers little benefit to the service and may result in a significant loss of prediction accuracy. Therefore, SALC introduces a SLO warning line called $warning\_line$, which is a fraction of the SLO, defined with $warning\_factor$ (line 3). When the recent latency exceeds the SLO, $threshold$ is much reduced to lower the latency below the SLO through multiplying it by $shrink\_ratio$ (lines 7-8). When the latency becomes too low (i.e., below the warning line), we increase $threshold$ linearly (lines 5-6). This ensures that the inference latency is kept within the range between the SLO warning line and the SLO. This design inherently provides robustness against performance interference (e.g., from co-located workloads). Since the SALC algorithm reacts to any latency increase, regardless of its source, it will proactively lower the threshold to mitigate the effects of resource contention, ensuring SLO compliance under unpredictable conditions.

Algorithm 3 SLO-Aware Latency Control Algorithm

1: **Input:** Current brownout threshold $threshold$, requests' SLO slo, SLO warning line factor $warning\_factor$, recent time window tw, the linear increment of threshold $increment$, the multiplicative shrinkage ratio of threshold $shrink\_ratio$.
2: **Output:** updated threshold $threshold$.
3: $warning\_line \leftarrow slo \times warning\_factor$
4: $latency \leftarrow$get_recent_P99_latency($threshold$, tw)
5: **if** $latency < warning\_line$ **then**
6: $\quad threshold \leftarrow threshold + increment$
7: **else if** $latency > slo$ **then**
8: $\quad threshold \leftarrow threshold \times shrink\_ratio$
9: **end if**
10: **Return** $threshold$

Time Complexity Analysis. Algorithm 3 processes n tokens in recent time tw (at line 4). To get P99 latency of all tokens, it needs to sort them first ($O(nlogn)$), and get the value at the 99th percentile ($O(1)$). The time complexity of lines 3, 6 and 8 are $O(1)$. Therefore, the total time complexity is $O(nlogn + 1 \times 4)$, which equals to $O(nlogn)$.

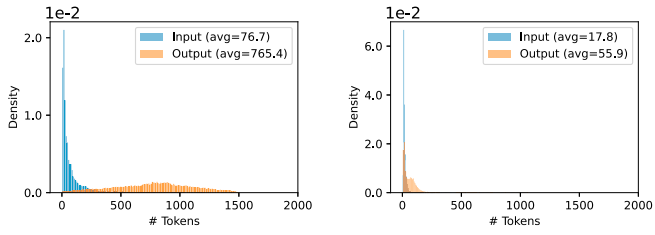
V. Evaluations

In this section, we introduce our experimental settings and evaluate the performance under a variety of workloads.

A. Experimental Setup

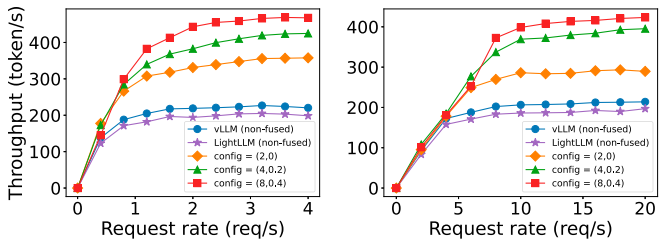
Model and server configurations. We use Qwen1.5-MoE-A2.7B-Chat [38] for evaluations. The model has a total of 14.3B parameters, with 60 experts per transformer layer. Compared to other popular MoE models, such as Mixtral-8x7B, which has only 8 experts per layer, Qwen1.5-MoE-A2.7B-Chat contains more experts in each layer, making the brownout approach more effective for this type of model. All experiments are run on a machine equipped with 4 NVIDIA A100-PCIE-40GB GPUs (40GB memory per GPU) and an Intel Xeon Gold 6238 processor.

Workloads. To evaluate the prediction accuracy of the model adopting the brownout approach, We use four few-shot tasks: PIQA [39], COPA [40], CEVAL [41], and OBQA [42]. These datasets are widely used as standard benchmarks for model accuracy evaluation in related works [4, 20]. For fine-grained evaluations of model inference performance, we use the ShareGPT [43] and Alpaca [44] datasets, which are also used in the evaluation of vLLM. The ShareGPT dataset is a collection of user-chatbot conversations with ChatGPT. The Alpaca dataset, is an instruction tuning dataset designed for fine-tuning or evaluating language models. Both datasets serve as typical representations of real-world LLM services. As shown in Fig. 7, the input of ShareGPT is approximately 4.3 $\times$ longer than that of Alpaca, and the output of ShareGPT is approximately 13.7 $\times$ longer than that of Alpaca.



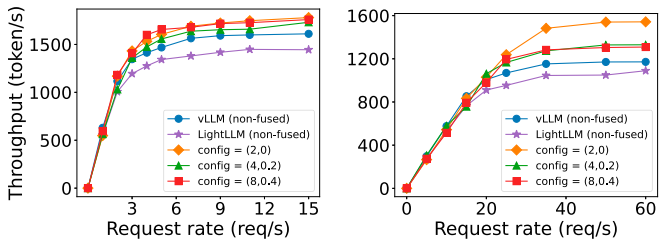
(a) ShareGPT (b) Alpaca

Fig. 7. The input and output tokens length distributions of (a) ShareGPT datasets, (b) alpaca datasets.



(a) ShareGPT (b) Alpaca

Fig. 8. Throughput comparison of models without fused-MoE on ShareGPT and alpaca datasets.



(a) ShareGPT (b) Alpaca

Fig. 9. Throughput comparison of models with fused-MoE on ShareGPT and alpaca datasets.

Baseline: We compare BrownoutServe against two high-performance inference engines: vLLM [9] and LightLLM [45]. Both are designed for high-throughput, low-latency LLM serving. For each baseline, we evaluate two variants to comprehensively assess our approach:

- **Native:** The official release, utilizing fused MoE[1] by default, to measure gains on highly optimized models.
- **Non-fused:** A modified version where the fused MoE module is replaced with a standard implementation, to evaluate effectiveness on models without this specific optimization.

We mainly evaluate the following three key metrics:

- **Throughput:** To provide a more comprehensive and detailed evaluation of throughput, we test three different (way, $threshold$) configurations: (2, 0), (4, 0.2) and (8, 0.4). We compare the throughput against the baseline at various request rates, with a trace that requests continuously sent for 10 minutes.
- **Accuracy:** We evaluate the accuracy loss introduced by using the brownout approach during inference under different way and $threshold$ in brownout configurations.
- **SLO Violations Rate:** It is defined as the proportion of individual tokens that fail to meet the predefined SLO. This is an important metric for assessing service reliability and user experience in real-world production environments.

B. Throughput

According to Eq. (1), we configure clients that continuously send requests to the server for 10 minutes with different request rates, where the request arrival follows a Poisson distribution. Serving throughput is defined as the ratio between the total number of tokens generated and the total time consumed. Given the practical value of configurations (2, 0), (4, 0.2), and (8, 0.4), we evaluate the performance of BrownoutServe in these settings. Each data point was obtained by running the experiment 10 times and reporting the average value, which was done to mitigate the impact of runtime variances and non-deterministic factors, such as fluctuations in GPU memory bandwidth and other system-level noise; The same rationale applies to the experiments presented in Section V.D. We set the maximum batch size to 64 and the maximum sequence length to 2048.

[1]Fused MoE prioritizes computational efficiency, while standard MoE offers greater flexibility and task adaptability.

As shown in Fig. 8, BrownoutServe achieves up to 2.07 × and 2.46 × throughput improvement over vLLM (non-fused) and LightLLM (non-fused), respectively. When equipped with the fused MoE module, as shown in Fig. 9, BrownoutServe achieves up to 1.32 × and 1.53 × throughput improvement over vLLM (native) and LightLLM (native). Compared to the significant improvement over the non-fused versions of vLLM and LightLLM, the gains over their native versions are more modest, primarily because the fused MoE operator is already highly optimized. According to Eq. (2), there is limited room for further performance enhancement within the fused MoE, as it is already near optimal.

C. Accuracy

Fig. 10 shows the accuracy variation on four datasets (PIQA, COPA, CEVAL, OBQA) under different brownout configurations for 5-shot tasks. We designed four brownout united expert modes: 2-way, 4-way, 8-way, and full brownout. For each mode, we varied the threshold from 1 to 0 (with a step size of 0.1), resulting in 44 different brownout configurations (4 modes × 11 threshold values) in total. The accuracy for each configuration was measured once. This is because the inference process for calculating accuracy is deterministic. Given the same model, input data, and configuration, the output is reproducible and does not vary between multiple experiments. When the threshold is set to 1, the brownout approach actually adopts the zero-brownout strategy, which serves as the baseline's accuracy metric (indicated by the gray dashed line in the figure). As seen in the figure, on the CEVAL and OBQA datasets, the accuracy

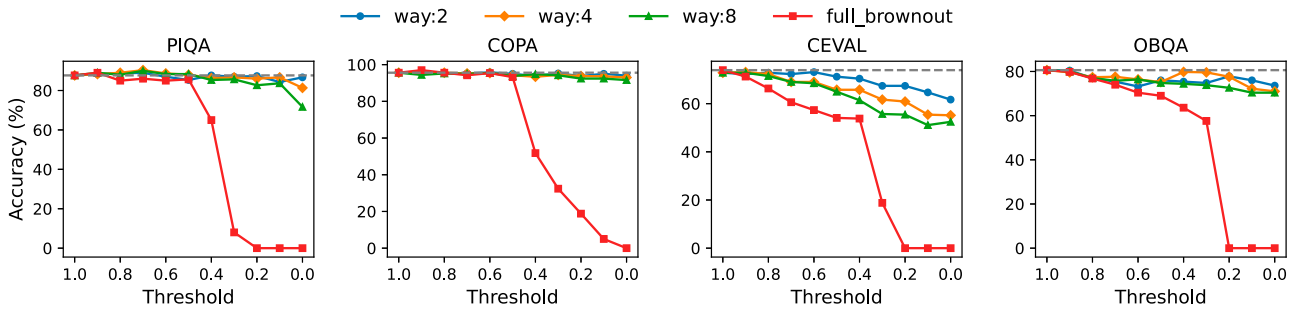


Fig. 10. Accuracy of Qwen1.5-MoE-A2.7B-Chat on 5-shot tasks.

TABLE I
AVERAGE ACCURACY LOSS (%) OF QWEN1.5-MOE-A2.7B-CHAT ACROSS DIFFERENT THRESHOLDS AND WAYS

Threshold	Ways			
	2	4	8	Full Brownout
1.0	0.00%	0.00%	0.00%	0.00%
0.9	0.06%	0.03%	1.01%	0.40%
0.8	0.88%	0.80%	1.59%	4.51%
0.7	1.52%	1.58%	2.29%	7.39%
0.6	2.34%	2.38%	2.66%	9.58%
0.5	2.81%	4.37%	3.78%	11.51%
0.4	2.92%	3.63%	**4.82%**	35.07%
0.3	3.96%	4.43%	7.92%	59.95%
0.2	3.30%	**5.18%**	10.71%	95.08%
0.1	4.23%	9.38%	12.60%	98.69%
0.0	**4.70%**	11.53%	15.77%	100.00%

decreases more significantly as the threshold decreases, compared to the PIQA and COPA datasets. This may be because the problems in these two datasets are more difficult for the model to comprehend, making the model more sensitive to the changes in the brownout strategy when handling these more complex problems. For full-brownout strategy, tasks like PIQA, CEVAL, and OBQA, a threshold above 0.2 results in different degrees of accuracy drop (degradation). However, at or below this threshold, model accuracy drops close to zero, rendering the model non-functional for these tasks. The robustness to token dropping varies by task. COPA, for instance, only fails completely when the threshold is 0 (all tokens dropped), demonstrating a higher tolerance.

Table I presents the average accuracy loss of model Qwen1.5-MoE-A2.7B-Chat across 44 different brownout configurations evaluated on four datasets. We specifically highlight three representative settings: (2, 0), (4, 0.2), and (8, 0.4), which result in average accuracy losses of 4.70%, 5.18%, and 4.82%, respectively, around the 5% level. For short-lived bursty workloads, tolerating an accuracy drop of approximately 5% is a practical trade-off to meet SLO attainment. Therefore, these configurations strike a practical balance between computational savings and model performance, making them valuable for deployment.

D. SLO Violations Rate

We design a 250s request stream to evaluate traffic burst commonly encountered in real-world applications. During this process, we observe the robustness under sudden request pressure and validate the effectiveness of the SALC algorithm by evaluating its improvement over baseline methods in terms of SLO violations rate. We used the ShareGPT and Alpaca datasets as request sources, with an initial request rate of 0.5 requests per second (RPS) for ShareGPT and 1 RPS for Alpaca, and doubled the request rate at the 75s mark to create a bursty scenario. There we use the identical settings of batch size and sequence length in Section V-B.

Latency Trace Analysis. Fig. 11 illustrates a detailed record of the prefill and decoding latency (P99 latency) over a 250s request trace. The red dashed lines represent the SLO (0.25s for prefill and 0.15s for decoding), while the blue dashed lines are the SLO warning lines (0.20s for prefill and 0.12s for decoding, with a warning factor of 0.8). Both systems initially met the SLO requirements during the baseline phase ($t < 75$s). Upon the $2\times$ load surge at t=75s, vLLM and LightLLM exhibited significant latency degradation, peaking at 0.31s (24% over SLO) for prefill and 0.24s (60% over SLO) for decoding. Regulated by SALC ($shrink\_ratio$, in Algorithm 3 is set As 0.8 and $increment$ as 0.1), maintained stable performance within the warning-SLO buffer zone (prefill: 0.20-0.25s; decoding: 0.12-0.15s).

SLO Violation Rate Analysis. We evaluate the SLO violations rate of vLLM under bursty traffic conditions. As shown in Fig. 12, on the ShareGPT dataset, vLLM exhibits SLO violations rate of 73.68% and 98.85% during the prefill and decoding stages, respectively, while maintaining significantly lower rates of 7.14% and 8.57%, corresponding to reductions of 66.54% and 90.28%. On the Alpaca dataset, vLLM's SLO violations rate reaches 94.29% and 98.86%, while BrownoutServe's rates are only 4.55% and 9.14%, with reductions of 89.74% and 89.72%, respectively. As shown in Fig. 11, BrownoutServe also demonstrates consistent advantages over LightLLM. These results collectively demonstrate that BrownoutServe exhibits stronger robustness under bursty loads compared to existing LLM serving systems.

Threshold Variations Analysis. We also record the threshold variations of in this trace. As shown in Fig. 13. During requests burst ($t > 75$s), results show that on the ShareGPT dataset, the prefill phase achieved an average threshold of 0.635 (with $\leq$2.66% accuracy loss at way $= 8$, known from Table I) while the decoding phase averaged 0.524 ($\leq$3.78% accuracy loss). For the Alpaca dataset, the prefill threshold rose to 0.760

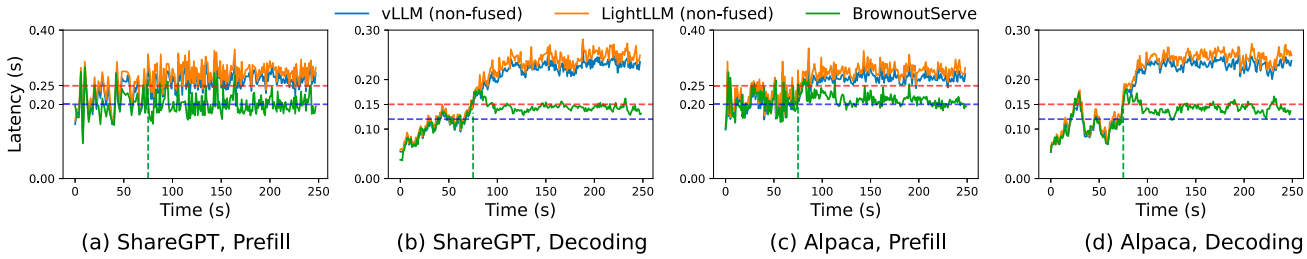


Fig. 11. P99 latency comparison between different LLM serving systems and BrownoutServe under 8-way united experts.

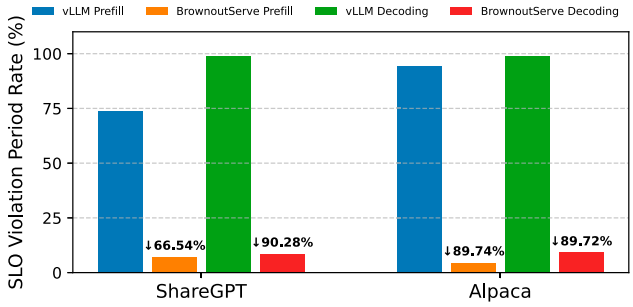


Fig. 12. SLO violations rate comparison between vLLM and across ShareGPT and alpaca datasets.

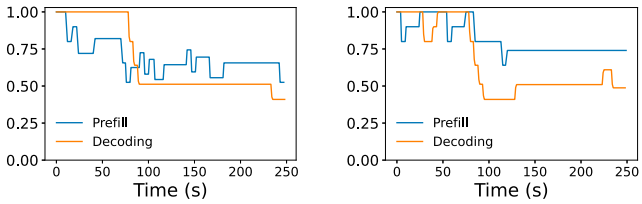


Fig. 13. Threshold variations in BrownoutServe.

($\leq$2.29% accuracy loss) with decoding stabilizing at 0.514 ($\leq$3.78% accuracy loss). These findings demonstrate that the brownout approach effectively mitigates resource overload during traffic bursts through dynamic threshold adaptation while meeting SLO.

To summarize, BrownoutServe achieves up to 2.46 $\times$ higher throughput and reduces SLO violations by up to 90.28% under bursty workloads compared to state-of-the-art systems. This is accomplished by introducing united experts and the dynamic brownout mechanism. And using these techniques, this framework alleviates the limitations of static MoE inference systems and improves serving scalability under bursty requests.

VI. Discussion

In this section, we further discuss the advantages and limitations of BrownoutServe as follows:

Generalizability of BrownoutServe. The design of BrownoutServe is architected for generalizability beyond the specific Qwen1.5-MoE-A2.7B model used in our evaluation. Its core mechanisms are fundamentally decoupled from any particular MoE configuration: the united experts are formed through a general-purpose knowledge distillation process applicable to any expert group, and the brownout mechanism functions as a generic, latency-driven controller for managing computational load. Consequently, the framework is applicable to any MoE-based model where the expert layer constitutes a performance bottleneck.

Trade-off between Hyperparameters. The hyperparameters of our SALC algorithm, namely the increment $increment$ and shrinkage ratio $shrink\_ratio$, entail a trade-off between responsiveness and stability. A smaller $shrink\_ratio$ enables quicker recovery from SLO violations but may induce instability, while a larger $increment$ accelerates the restoration of accuracy at the risk of overshooting the latency target. The values used in our evaluation (e.g., 0.8 and 0.1) were empirically chosen to balance this trade-off for our workloads, yet their optimal setting may vary and is a point for future investigation.

Memory-Constrained Scenario. Our proposed BrownoutServe framework demonstrates particular efficacy in memory-constrained GPU environments. Under such conditions, where the complete set of experts cannot be fully accommodated within GPU memory, the system incurs significant overhead from frequent expert swapping between GPU and CPU memory (or disk). BrownoutServe's united expert mechanism directly addresses this challenge by reducing the total number of distinct experts required during inference. By consolidating multiple original experts into fewer united experts, our approach minimizes the frequency of expensive memory transfer operations, thereby substantially reducing inference latency. This benefit is most pronounced in resource-limited deployment scenarios common in small-to-medium-scale GPU clusters, where memory constraints often become the primary performance bottleneck.

Model Parameters Calibration. BrownoutServe is designed as an inference-serving system, which precludes online calibration of its model during serving. The model remains static with pre-trained weights, as real-time calibration would introduce unpredictable latency that violates our core objective of guaranteed low-latency inference. While certain MoE frameworks support online model updates, BrownoutServe maintains fixed model parameters throughout serving. Any model re-training can be conducted offline and redeployed, ensuring runtime stability and predictable performance.

VII. Conclusion and Future Work

In this paper, we present BrownoutServe, an innovative serving framework for MoE-based LLMs that incorporates the brownout approach to optimize inference efficiency while maintaining service reliability under bursty workloads. Through the

introduction of the united expert models and the brownout mechanism, BrownoutServe effectively reduces inference latency and enhances throughput by dynamically adjusting the allocation of computational resources based on real-time workloads. BrownoutServe's robustness and efficiency have been validated through extensive experiments under various datasets and traffic conditions. As future work, we would like to extend our framework to multiple MoE-based LLMs, such as Mixtral and DeepSeek. Additionally, extending BrownoutServe to multi-node clusters and integrating a prefill-decoding separation architecture will be critical to address system-level bottlenecks and further optimize SLO in production environments.

Software Availability

The codes of BrownoutServe have been open-sourced to https://github.com/beyondHJM/BrownoutServe for research usage.

References

[1] R. A. Jacobs, M. I. Jordan, S. J. Nowlan, and G. E. Hinton, "Adaptive mixtures of local experts," *Neural Comput.*, vol. 3, no. 1, pp. 79–87, 1991.

[2] A. Q. Jiang et al., "Mixtral of experts," 2024, *arXiv:2401.04088*.

[3] A. Yang et al., "Qwen3 technical report," 2025, *arXiv:2505.09388.*

[4] A. Liu et al., "DeepSeek-V3 technical report," 2024, *arXiv:2412.19437*.

[5] M. Lewis, S. Bhosale, T. Dettmers, N. Goyal, and L. Zettlemoyer, "Base Layers: Simplifying training of large, sparse models," in *Proc. Int. Conf. Mach. Learn.,* PMLR, 2021, pp. 6265–6274.

[6] Y. Zhong et al., "DistServe: Disaggregating prefill and decoding for goodput-optimized large language model serving," in *Proc. 18th USENIX Symp. Operating Syst. Des. Implementation (OSDI 24)*, 2024, pp. 193–210.

[7] T. Wolf et al., "Transformers: State-of-the-art natural language processing," in *Proc. Conf. Empirical Methods Natural Lang. Process. : Syst. Demonstrations*, 2020, pp. 38–45.

[8] W. Fedus, B. Zoph, and N. Shazeer, "Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity," *J. Mach. Learn. Res.*, vol. 23, no. 120, pp. 1–39, 2022.

[9] W. Kwon et al., "Efficient memory management for large language model serving with PagedAttention," in *Proc. 29th Symp. Operating Syst. Princ.,* 2023, pp. 611–626.

[10] S. Rajbhandari et al., "DeepSpeed-MoE: Advancing mixture-of-experts inference and training to power next-generation AI scale," in *Proc. Int. Conf. Mach. Learn.,* PMLR, 2022, pp. 18332–18346.

[11] G.-I. Yu, J. S. Jeong, G.-W. Kim, S. Kim, and B.-G. Chun, "Orca: A distributed serving system for transformer-based generative models," in *Proc. 16th USENIX Symp. Operating Syst. Des. Implementation (OSDI)*, 2022, pp. 521–538.

[12] M. Xu, A. N. Toosi, and R. Buyya, "A self-adaptive approach for managing applications and harnessing renewable energy for sustainable cloud computing," *IEEE Trans. Sustain. Comput.*, vol. 6, no. 4, pp. 544–558, Oct./–Dec. 2021.

[13] D. Lepikhin et al., "GShard: Scaling giant models with conditional computation and automatic sharding," 2020, *arXiv:2006.16668*.

[14] J. Li, Y. Jiang, Y. Zhu, C. Wang, and H. Xu, "Accelerating distributed MoE training and inference with Lina," in *Proc. USENIX Annu. Tech. Conf. (USENIX ATC),* 2023, pp. 945–959.

[15] Z. Zhang et al., "MPMoE: Memory efficient MoE for pre-trained models with adaptive pipeline parallelism," *IEEE Trans. Parallel Distrib. Syst.*, vol. 35, no. 6, pp. 998–1011, Jun. 2024.

[16] Y. Zeng et al., "EfficientMoE: Optimizing mixture-of-experts model training with adaptive load balance," *IEEE Trans. Parallel Distrib. Syst.*, vol. 36, no. 4, pp. 677–l688, Apr. 2025.

[17] C. Hwang et al., "Tutel: Adaptive mixture-of-experts at scale," in *Proc. Mach. Learn. Syst.,* vol. 5, 2023, pp. 269–287.

[18] R. Hwang et al., "Pre-Gated MoE: An algorithm-system co-design for fast and scalable mixture-of-expert inference," in *Proc. ACM/IEEE 51st Annu. Int. Symp. Comput. Archit. (ISCA)*, Piscataway, NJ, USA: IEEE Press, 2024, pp. 1018–1031.

[19] S. Cao et al., "MoE-lightning: High-throughput MoE inference on memory-constrained GPUs," in *Proc. 30th ACM Int. Conf. Architect. Support Program. Lang. Operating Syst.*, 2025, pp. 715–730.

[20] W. Lee, J. Lee, J. Seo, and J. Sim, "InfiniGen: Efficient generative inference of large language models with dynamic KV cache management," in *Proc. 18th USENIX Symp. Operating Syst. Des. Implementation (OSDI)*, 2024, pp. 155–172.

[21] B. Wu, S. Liu, Y. Zhong, P. Sun, X. Liu, and X. Jin, "LoongServe: Efficiently serving long-context large language models with elastic sequence parallelism," in *Proc. ACM SIGOPS 30th Symp. Operating Syst. Princ.,* 2024, pp. 640–654.

[22] Z. Li et al., "AlpaServe: Statistical multiplexing with model parallelism for deep learning serving," in *Proc. 17th USENIX Symp. Operating Syst. Des. Implementation (OSDI),* 2023, pp. 663–679.

[23] P. Patel et al., "SplitWise: Efficient generative LLM inference using phase splitting," in *Proc. ACM/IEEE 51st Annu. Int. Symp. Comput. Archit. (ISCA)*, Piscataway, NJ, USA: IEEE Press, 2024, pp. 118–132.

[24] B. Sun et al., "Llumnix: Dynamic scheduling for large language model serving," in *Proc. 18th USENIX Symp. Operating Syst. Des. Implementation (OSDI)*, 2024, pp. 173–191.

[25] S. Zhang et al., "OPT: Open pre-trained transformer language models," 2022, *arXiv:2205.01068.*

[26] H. Touvron et al., "Llama: Open and efficient foundation language models," 2023, *arXiv:2302.13971*.

[27] J. Achiam et al., "GPT-4 technical report," 2023, *arXiv:2303.08774*.

[28] D. G. Kendall, "Stochastic processes occurring in the theory of queues and their analysis by the method of the imbedded Markov chain," *Ann. Math. Statist.*, vol. 24, no. 3, 1953, pp. 338–354.

[29] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *Proc. Spring Joint Comput. Conf.*, 1967, pp. 483–485.

[30] Kubernetes. (2025) Kubernetes. Accessed: 2025-05-24. [Online]. Available: https://kubernetes.io/

[31] Y. Fu, et al., "ServerlessLLM: Low-latency serverless inference for large language models," in *Proc. 18th USENIX Symp. Operating Syst. Des. Implementation (OSDI)*, 2024, pp. 135–153.

[32] S. S. Shubha, H. Shen, and A. Iyer, "USHER: Holistic interference avoidance for resource optimized ML inference," in *Proc. 18th USENIX Symp. Operating Syst. Des. Implementation (OSDI),* 2024, pp. 947–964.

[33] T. Gale, D. Narayanan, C. Young, and M. Zaharia, "MegaBlocks: Efficient sparse training with mixture-of-experts," in *Proc. Mach. Learn. Syst.,* vol. 5, 2023, pp. 288–304.

[34] T. Dao, D. Fu, S. Ermon, A. Rudra, and C. Ré, "FlashAttention: Fast and memory-efficient exact attention with IO-awareness," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 35, pp. 16344–16359, 2022.

[35] A. Paszke, "Pytorch: An imperative style, high-performance deep learning library," 2019, *arXiv:1912.01703*.

[36] P. Tillet, H.-T. Kung, and D. Cox, "Triton: An intermediate language and compiler for tiled neural network computations," in *Proc. 3rd ACM SIGPLAN Int. Workshop Mach. Learn. Program. Lang.,* 2019, pp. 10–19.

[37] G. Hinton, O. Vinyals, and J. Dean, "Distilling the knowledge in a neural network," 2015, *arXiv:1503.02531.*

[38] Qwen1.5. MoE and A. 27B. Chat, 2025. "Qwen1.5-moe-a2.7b-chat." Accessed: May 24, 2025. [Online]. Available: https://huggingface.co/Qwen/Qwen1.5-MoE-A2.7B-Chat

[39] Y. Bisk, R. Zellers, R. L. Bras, J. Gao, and Y. Choi, "PIQA: Reasoning about physical commonsense in natural language," in *Proc. 34th AAAI Conf. Artif. Intell.*, 2020, pp. 7432-7439.

[40] M. Roemmele, C. A. Bejan, and A. S. Gordon, "Choice of plausible alternatives: An evaluation of commonsense causal reasoning," in *Proc. AAAI Spring Symp. Ser.,* 2011. [Online]. Available: https://people.ict.usc.edu/gordon/publications/Aaai-SPRING11A.PDF

[41] Y. Huang et al., "C-Eval: A multi-level multi-discipline Chinese evaluation suite for foundation models," 2023, *arXiv:2305.08322*.

[42] T. Mihaylov, P. Clark, T. Khot, and A. Sabharwal, "Can a suit of armor conduct electricity? A new dataset for open book question answering," in *Proc. EMNLP*, 2018, pp. 2381-2391.

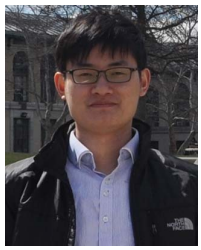
[43] ShareGPT. "ShareGPT." Accessed: May 24, 2025. [Online]. Available: https://sharegpt.com/

[44] R. Taori et al., "Stanford Alpaca: An instruction-following LLaMA model," 2023. [Online]. Available: https://github.com/tatsu-lab/stanford_alpaca

[45] ModelBest Inc., "LightLLM: A high-performance, low-cost LLM inference and serving engine," 2023. Accessed: May 11, 2025. [Online]. Available: https://github.com/ModelTC/LightLLM

Jianmin Hu received the bachelor's degree in software engineering from Harbin Institute of Technology, in 2023. He is currently working toward the joint master's degree with Shenzhen Institutes of Advanced Technology, Chinese Academy of Sciences and Southern University of Science and Technology. His research interests include cloud native and system for large language model, especially for MoE-based LLMs.

Minxian Xu (Senior Member, IEEE) received the Ph.D. degree from the University of Melbourne, in 2019. He is currently an Associate Professor with Shenzhen Institutes of Advanced Technology, Chinese Academy of Sciences. His research interests include cloud-native system for LLM inference and AI infrastructure. He has co-authored 80+ peer-reviewed papers published in prominent international journals and conferences, such as CSUR, IEEE TRANSACTIONS ON SERVICES COMPUTING, IEEE TRANSACTIONS ON MOBILE COMPUTING, TOIT, and TAAS. He is among the World's Top 2% Scientists by Stanford University (2023–2025). He was awarded the 2023 IEEE TCSC Award for Excellence (Early Career Award).



Kejiang Ye (Senior Member, IEEE) received the B.S. and Ph.D. degrees from Zhejiang University. He is currently a Professor and the Director of the Research Center for Cloud Computing, Shenzhen Institute of Advanced Technology, Chinese Academy of Sciences. He was a Postdoctoral Research Associate with Carnegie Mellon University (CMU). His research interests include digital technology and systems (e.g., cloud computing, big data, and industrial internet). He is a Distinguished Member of China Computer Federation (CCF).

Chengzhong Xu (Fellow, IEEE) received the Ph.D. degree in computer science and engineering from the University of Hong Kong, in 1993. He is the Dean with the Faculty of Science and Technology and the Interim Director with the Institute of Collaborative Innovation, University of Macau. His research interests include parallel and distributed computing, with an emphasis on resource management for performance, reliability, availability, power efficiency, and security. His work spans servers and cloud datacenters, wireless embedded devices, and edge AI systems, with applications in smart city and autonomous driving. He has authored two research monographs and more than 600 papers, which have garnered more than 22 K citations with an H-index of 77, and has been cited in over 300 international patents.