SCIENCE CHINA
Information Sciences

• **RESEARCH PAPER** •

MICO: Efficient Query Scheduling for Multi-Cloud Deployed LLM Inference Service

Peizhuang Cong[1], Tong Yang[1*], Yuchao Zhang[2*], Wendong Wang[2] & Ke Xu[3]

[1]*National Key Laboratory for Multimedia Information Processing,*
School of Computer Science, Peking University, Beijing 100871*, China;*
[2]*Beijing University of Posts and Telecommunications, Beijing* 100876*, China;*
[3]*Tsinghua University, Beijing* 100084*, China*

Abstract Given the powerful capabilities of Large Language Models (LLMs), many tech companies make LLM inference a service for users, which may be deployed in multiple clouds to provide better service. Computational overhead and cloud workload are crucial metrics in cloud computing task scheduling. However, the autoregressive nature of LLMs makes these metrics difficult to measure. Specifically, LLMs require multiple iterations of computation to process a single query, and there is significant differentiation in the number of iterations needed for different queries. Moreover, batch-wise model inference exacerbates the gap between allocated and actual computational loads for each cloud due to these variations, ultimately affecting computational resource utilization and the throughput of inference service query processing. To this end, we propose Mico, which includes a query scheduling strategy based on response length prediction to achieve token-granularity workload distribution across clouds, and an inference framework that supports the flexible insertion of queries into the processing batch, eliminating unnecessary computation introduced by the iteration differentiation of queries in batch-wise inference. We conducted experiments based on two GPT series models, and the results show that Mico can reduce *KV-Cache* resource consumption by 44.89% during inference and increase the query processing throughput of the service system by up to 2.2×.

Keywords query scheduling, cloud computing, prediction, LLM inference, cloud service optimization

1 Introduction

Large Language Models (LLMs) [1–4] exhibit powerful capabilities for handling various tasks, including text generation, summarization, question-answering, *etc.* Many technology companies, including OpenAI, Microsoft, Google, Alibaba, and so on, have deployed their proprietary large language models (LLMs) on cloud platforms, offering LLM inference as a service to users. Notable applications include ChatGPT[1)], Copilot[2)], Gemini[3)], Qwen[4)], *etc.* To enhance response efficiency and reliability of service, LLM services are typically deployed on more than one cloud platform globally, and each cloud is capable of providing inference services independently. For example, Gemini is deployed in multiple Google Cloud locations around the world [5]. Users' queries will be forwarded to one of these clouds, where the LLM inference engine processes them and returns the corresponding response [6].

Transformer-based LLMs differ significantly from traditional deep neural network (DNN) models in terms of inference workflows. A DNN model requires only a single forward propagation computation to obtain the final inference result. For example, in an image classification task, an input image can be processed through one forward pass to produce a classification result [7]. This one-pass inference inherently enables the efficient parallel processing of multiple images in a batch. Each forward propagation of an LLM, also referred to as an iteration, produces only one token. The token is the fundamental unit of the final response sequence, which, for convenience, can be understood as a word forming a sentence.

[*] Corresponding author (email: yangtong@pku.edu.cn, yczhang@bupt.edu.cn)

1) https://openai.com

2) https://copilot.microsoft.com

3) https://gemini.google.com

4) https://qianwen.aliyun.com

In each iteration, the intermediate state data, such as *Keys* and *Values*, will be cached (denoted as *KV-Cache*), and the output token will be appended to the input, which is then re-input into the model for the subsequent iteration. This process, known as autoregressive inference, continues until the model generates a predefined end-of-sequence symbol (⟨EOS⟩) or when the cumulative tokens reach the predetermined maximum limit. Once inference concludes, all output tokens are combined to construct the final response sequence [8].

The response sequence lengths associated with different queries vary, affecting the batch-wise inference efficiency of autoregressive LLMs. The inference of a batch continues until all queries satisfy the inference end condition. In other words, the completion of a batch is decided by the query with the longest response sequence length. The variation in response sequence lengths between different queries within a batch leads to unnecessary idle computing. Specifically, although queries that finish inference early can return the response, they will still occupy resources and iterate with other incomplete queries according to current LLM inference frameworks [9–11]. According to the model's configuration, once a query completes inference, the LLM will output ⟨EOS⟩ for that query in subsequent iterations; these repeating ⟨EOS⟩ have no actual contributions to the final response. Resources consumed by *Keys* and *Values* of a query could be released directly upon completion of the inference. However, it diminishes the overall parallelism of the batch-wise inference from the perspective of the service system. Additionally, replenishing a new query into the processing batch to sustain high parallelism is challenging. This is because parallel computation on the GPU requires the involved matrices to be aligned in the specified dimensions. Nevertheless, the dimensions of the involved matrices change according to iterations, *e.g.*, the length of *KV-Cache* in *seq_length* dimension.

The work [12] eliminates necessary alignment requirements of batch-wise inference by joining all queries' vectors along the token dimension in linear computation layers, and by duplicating self-attention layers to address the nonlinear computations for each query independently. Based on this work, [13, 14] optimizes the policy in choosing a more suitable query to insert into the processing batch. However, this series of methods inherently suffers from resource underutilization, since duplicating all self-attention layers introduces a large number of additional parameters, increasing resource consumption. And the additional resource consumption scales up linearly with the set size of the processing batch. Moreover, the existing works focus only on the optimization of single inference engine scenarios, and thus fail to meet the requirements of practical multi-cloud deployed LLM services [12, 13, 15].

To this end, in this paper, we propose an efficient query scheduling scheme for multi-cloud deployed LLM services, named Mico. It mainly involves two components: (1) response length prediction-based query granularity scheduling across multiple clouds, and (2) iteration granularity scheduling for LLM inference. Specifically, Mico consists of three components, the response length *Predictor*, the *Query granularity scheduler*, and the *Iteration granularity scheduler*, which are respectively responsible for the following functions: 1) Predicting the length of the response sequence of each query to facilitate scheduling; 2) Scheduling the queries to the most suitable cloud based on the predicted results, which considers the inference characteristics of queries; 3) Scheduling queries at the iteration granularity in the inference engine, enabling flexible query insertion into the processing batch. Finally, achieving efficient resource utilization and high-throughput LLM inference services.

The contributions of this paper are summarized as follows:

- We developed a query scheduling scheme based on the prediction of response sequence length for multi-cloud deployed LLM services, achieving token-level workload distribution across multiple service clouds.
- We designed an inference framework that allows flexible query insertion into the processing batch, enabling successive batch-wise inference at the query granularity.
- We conducted extensive experiments, demonstrating that Mico can improve resource utilization and enhance the query processing throughput for LLM inference.

This paper is organized as follows. We review preliminaries and motivations in §2. In §3, we introduce the overview and the details of Mico. We then conduct extensive evaluations and show the results in §4. We review related work in §5. We conclude the paper and present the future work in §6.

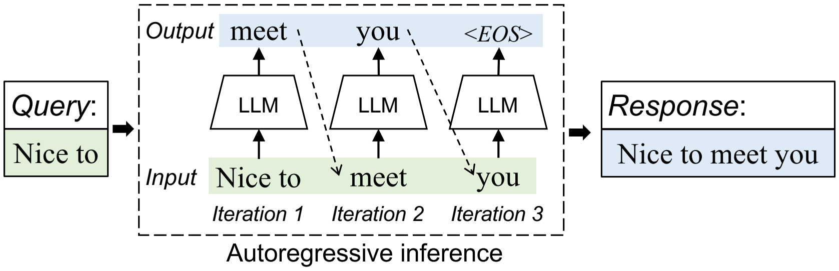


Figure 1 KV-Cache-enabled inference of text generation task

2 Preliminary and motivation

In this section, we first introduce the preliminaries about the workflow characteristics of transformer-based LLMs and then present the motivation of this work based on research actuality and actual practice demands.

2.1 Preliminary of LLMs inference

LLM inference workflow. Current commercial or open-source LLMs, *e.g.*, GPT and Llama, are mainly based on the transformer architecture or its variations [16, 17]. These architectures input a sequence of tokens, where each token is a representation of a word or a phrase encoded by the model's tokenizer. For simplicity, it is possible to recognize the model's input as a text that consists of a sequence of words. Each forward propagation calculation of the model will generate a new token, which will then be added at the end of the existing sequence and input to the model again for the next forward propagation. This workflow demonstrates the autoregressive characteristic of the LLMs. The inference process continues until generating an end-of-inference symbol, ⟨EOS⟩, or reaching the set maximum length. All tokens will be decoded into text by the tokenizer, forming the final output of the inference. The process from inputting tokens into the model to outputting a new token can be called an iteration. For example, if the initial input for the first iteration is "Nice to" and the model outputs "meet", then the input for the next iteration is "Nice to meet" and the output is "you". Similarly, the output "⟨EOS⟩" from the third iteration indicates that the inference is finished.

KV-Cache mechanism. The attention mechanism is a fundamental component of the transformer architecture [18], which focuses on computing a weighted average of selected tokens, ensuring that each token in the sequence is contextually linked to every other token. It processes three input vectors: query, key, and value. For a given token, the attention mechanism calculates the dot products between its query and keys corresponding to the tokens of interest. These dot products are then normalized using a Softmax function to obtain weights, which are used to perform a weighted average on the associated values. Attention relies on accessing the keys and values from all previous tokens, recomputing keys and values for all tokens in the sequence at each iteration of the naive stateless inference method is highly inefficient. To this end, some studies suggest using incremental decoding, a technique that retains previously computed keys and values for future iterations, which is called the KV-Cache mechanism [19].

Consequently, for inference utilizing the KV-Cache mechanism, it is only the first input that requires the entire query tokens, and subsequent iterations take the new token generated in the last iteration as model input. This is because the attention calculations for the existing tokens are repetitive, and the results are already cached. For the instance above, using the KV-Cache will change the process, as depicted in Figure 1, the first iteration inputs the full query tokens, the second iteration inputs "meet", and the third iteration inputs "you", and so forth.

Batch-wise inference. Model inference in batch-wise is illustrated in Figure 2. Since the dimensions of the calculated vectors need to be consistent, when the input queries are not uniform, *i.e.*, with different token lengths, the shorter ones will be padded with specific symbols. A 0-1 mask vector is generated to distinguish between original tokens and padded tokens, allowing the model to ignore the padded tokens during inference. After the first iteration of inference, each subsequent input uses the new token output from the last iteration. If any query in the batch is completed, represented by the output of an ⟨EOS⟩, the model will continue to generate an ⟨EOS⟩ for that query directly in subsequent inferences. This process

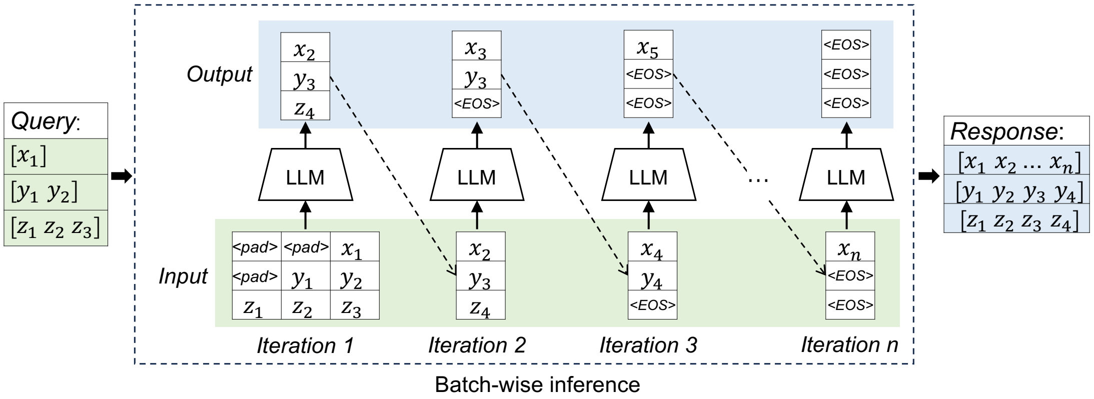


Figure 2 Batch-wise inference of LLM

continues until all queries in the batch have finished the inference, thus completing the inference of this batch.

Inference details. In the inference of a KV-Cache mechanism-enabled LLM model, each iteration involves three primary variables: *token vector*, *attention mask*, and *KV-Cache*. The first iteration of inference, also referred to as the profiling phase, differs from subsequent iterations, which are known as the decoding phase. In the prefilling phase, the *token vector* contains all tokens from each query and aligns them using padding operations. The original tokens and padding symbols will be distinguished by a 0-1 vector, known as the *attention mask*, with dimensions of $batch\_size \times l$, where $l = \max(query_i)$ represents the number of tokens of the longest query within the batch. The initial *KV-Cache* is empty. At the end of the first inference iteration, the *KV-Cache* is updated and the model outputs the latest token for each query.

In contrast to the prefilling phase, where the keys and values for all tokens in the entire context need to be computed during each inference iteration, the decoding phase, enabled by the KV-Cache mechanism, only requires computing the key and value for the latest input token in each iteration. Consequently, the *KV-Cache* is updated by appending the key and value of the input token after each iteration, facilitating its use in subsequent inference iterations. Specifically, using the GPT or Llama series model as an example, the structure of *KV-Cache* is as follows. *KV-Cache* is a multi-dimension tensor with [*layer*, 2, *batch_size*, *mul_head*, *seq_length*, *embed_length*], where *layer* identifies the number of model's transformer layers, 2 denotes the Keys and Values, *mul_head* denotes the number of multi-head, seq_length refers to the length of the processed sequence, *i.e.*, the number of tokens for which the corresponding keys and values have already been computed, and *embed_length* denotes the embedding dimension. Among these, only the length of seq_length increases with inferring iteration, while the other dimensions remain unchanged. Therefore, for convenience in this paper, the length of the *KV-Cache* refers to seq_length specifically.

Similarly, the latest output token with dimensions of $batch\_size \times 1$ will be treated as the new *token vector*. Correspondingly, the *attention mask* is updated by appending a column of 1, increasing its dimensions to $batch\_size \times (l + 1)$, where l represents the number of tokens already processed. Given the role of the *attention mask*, the lengths of the three input variables to the model always satisfy the condition that the length of the *attention mask* equals the sum of the lengths of the *token vector* and the *KV-Cache*. The autoregressive iteration continues until the model outputs the $\langle \mathrm{EOS} \rangle$ for a query.

2.2 Motivation

In contrast to traditional deep learning, the computational cost of the same model in inferring the same type of task is constant, *e.g.*, for an image classification task, it requires only one forward propagation computation for different images as inputs to get the result. However, for LLMs, even for the same type of task with the same model, there still are differences in inference for different inputs; that is, the number of tokens of responses for different queries varies, resulting in different iterations of computations. For example, in a question-and-answer system, the sentence lengths of answers vary for different questions. Therefore, it is inappropriate to directly employ the query granularity scheduling policy that ignores the iteration differences among queries for the LLM inference service.

Assuming that the number of inference iterations for each query is pre-estimated, the scheduling policy

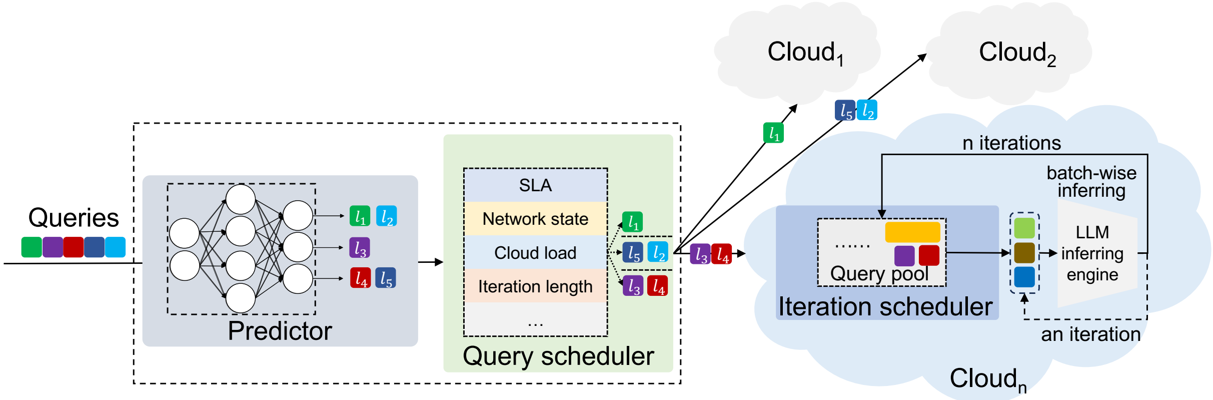


Figure 3 Mico overview

can no longer be limited to the network state, geographic location, query-granularity evaluated workload, *etc*. To simplify the description, constraints such as network, load, *etc*, are not considered. In the query granularity scheduling, balancing the number of queries that need to be processed by each inference engine (*i.e.*, cloud) can be achieved by round robin or hashing. However, such scheduling policies will cause load differences at the token granularity (*i.e.*, the actual counting) among different clouds. From the system perspective, the imbalance of processed tokens among clouds will underutilize the computational resources and reduce the system throughput.

When multiple queries are inferred in a batch, different queries may require different inference iterations, which will cause computational idling for the earlier completed queries. That is, although the total number of tokens to be processed by each cloud has been scheduled appropriately, traditional batch-wise inference will still result in unnecessary idle computation. Despite returning inference results immediately for completed queries and releasing the resources occupied by the corresponding KV-Cache, adding new queries to the processed batch is still challenging, given the strict requirement of vector dimension consistency in GPU matrix operations. Overall, the computational idling caused by the different iterations among queries in batch-wise inferring affects the resource utilization and processing throughput.

For the three aforementioned issues of the existing multi-cloud deployed LLM services, we designed Mico in this paper to improve the processing efficiency and throughput for the LLM inference system.

3 Mico design

In this section, we begin by providing an overview of the three components of Mico from a functional perspective, followed by a detailed design of each function module.

3.1 Mico overview

As shown in Figure 3, the overview of Mico can be mainly divided into three workflows: sentence length predicting, inter-cloud query granularity scheduling, and intra-cloud iteration granularity scheduling. The functions of each workflow are as follows.

Sentence length predicting: Mico sets up a predictor for the scheduler as a precursor. In particular, the predictor is responsible for predicting the sentence length of the corresponding answer for each query, *i.e.*, the predictor evaluates the iteration frequency of the performed autoregressive computations in the model inference. The prediction result will be leveraged in calculating the scheduling policy for user queries, which is demonstrated in the following.

Inter-cloud query granularity scheduling: Mico aims to balance token granularity loads among different clouds. Moreover, for KV-Cache-enabled inference, the occupied GPU memory is proportional to the total tokens of queries and corresponding responses. Therefore, if the total token number of all queries in a batch is substantial, the inference process may lead to GPU memory overflow and, consequently, inference failures. It is thus essential to balance queries with varying token lengths.

Intra-cloud iteration granularity scheduling: Mico facilitates the return of query results immediately after inference completion and enables the reuse of corresponding resources, allowing the insertion

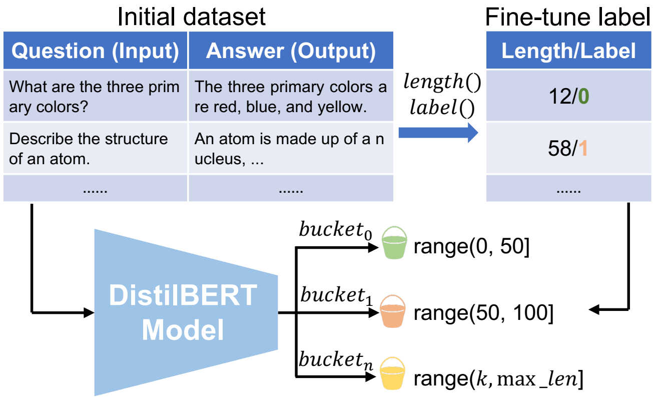


Figure 4 Example of fine-tuning the predictor model

of new queries into the batch without impacting ongoing inference processes. Mico transitions the scheduling from batch-by-batch to an iteration-based, query-by-query update. The specific scheme will be detailed in the subsequent section.

3.2 Response length predictor

In the inference process of generative LLMs, a notable characteristic of models is the iterative computation mechanism: for different queries, the model needs to perform varying iterations to generate the complete response. Specifically, each iteration of the LLM generates one token of the response, so the length of the response (*i.e.*, the number of tokens) directly determines the number of computation iterations required. Based on this property, the problem of predicting the number of computation iterations required for a query can be transformed into predicting the token count of its corresponding response, which simplifies the complexity of the problem, providing an important basis for optimizing query scheduling.

However, due to the temperature parameter or random sampling settings of LLMs, the generated responses exhibit inevitable randomness, making it difficult to precisely predict the exact token count. Therefore, instead of predicting the exact token count, it is possible to predict the length range of the response, which can also meet the scheduling requirements. Based on this consideration, we designed the prediction task as a classification problem, where the response length is divided into discrete categories (*i.e.*, "buckets") with each one corresponding to a specific range of token counts. For instance, responses with fewer than 50 tokens are assigned to $Bucket_0$, those with 50-100 tokens to $Bucket_1$, and so forth. This classification approach not only reduces the complexity of prediction but also improves the robustness and practicality of the model. To achieve efficient prediction, we adopted DistilBERT as the base model, which is a lightweight and efficient variant of BERT, specifically designed for sentence classification tasks. It retains approximately 97% of BERT's efficacy with only 60% of its parameters, while tripling the inference speed.

Moreover, it is necessary to process the dataset for fine-tuning the base model. First, the label of each item (i) in the initial dataset is switched to the number of sentence's token (l_i), and then generating the corresponding $label_i$ mapped to $bucket_i$ based on $label_i = \lfloor \frac{l_i}{l_{max}/N} \rfloor$, where l_{max} is the set maximum sentence length of LLM and N is the set granularity of prediction. This restructured dataset is then utilized to fine-tune the DistilBERT model. The illustrative example of fine-tuning the response sentence length is shown in Figure 4.

Finally, in practical deployments, it is desirable to sample queries and collect the corresponding response lengths to facilitate continuous updates of the prediction model, thereby enabling dynamic adaptation to the evolving characteristics of real queries.

3.3 Query scheduler

In current cloud computing, task scheduling typically considers factors such as network conditions, location, SLA, and the workload of cloud nodes. However, existing scheduling algorithms are not directly applicable to LLM query scheduling due to the highly variable and non-explicitly known processing costs associated with each query. To this end, we proposed the aforementioned *Predictor*, which can estimate the processing load of each query, *i.e.*, predicting the number of response sequence tokens. Hence, based on the *Predictor*, we designed the processing cost-aware query task scheduling for LLM inference service. In the cloud computing environment, task scheduling is typically handled by a dedicated module, which assigns an appropriate cloud to each received task based on the deployed scheduling algorithm. In this work, we focus on the introduction of the scheduling algorithm.

In general, the optimization objective of the proposed scheduling algorithm is to leverage predicted query load information to improve the query processing throughput of the entire inference service. It is important to note that, on the one hand, since the GPU memory consumption of KV-Cache of inference is proportional to the total length of the query and its response, the length of the query itself should also be considered. It helps prevent allocating too many long-sequence queries to one cloud, which may otherwise lead to GPU memory overflow during inference. On the other hand, given that the predicted load information is represented as ranges, it is unnecessary to employ a highly sophisticated and time-consuming method to achieve strict load balancing across clouds. Therefore, a simple yet effective allocation method is as follows, which focuses on balancing the response length of queries while considering the total processed tokens of each cloud.

First, assume that the predicted response length label of query Q_i is L_i, where L_i represents a range (x_1, x_2). At this point, the specific value of its response length l_i is designated as the median of the range, expressed as $l_i = \frac{x_1+x_2}{2}$. All queries are classified into different groups based on their response length l_i, denoted as G_l. Then, queries will be distributed evenly group by group in descending order of l of G across cloud C, meaning that queries with longer responses are assigned first. Meanwhile, the sum of the distributed query token length of each C_k will be maintained, denoted as M_k. If strict even distribution within a group is not possible, queries from the next group can be borrowed, thereby ensuring flexibility rather than enforcing an overly rigid balancing constraint. The borrowed queries can be allocated to different clouds using a round-robin approach, which can theoretically compensate for load differences caused by the borrowing mechanism. The M_k is used to balance the processed tokens of queries across all clouds as evenly as possible. The complexity of this scheduling algorithm is O(n) based on the above description. If each cloud node has varying computing power or existing workloads, the allocation proportions can be adjusted flexibly rather than being distributed evenly. The specifics will not be elaborated further here.

Backoff solution for corner case: To address potential extreme cases caused by prediction inaccuracies, such as excessive workload allocation to a specific cloud node, resulting in resource imbalance and increased overall query processing efficiency degradation, we present a backoff solution. Specifically, after a cloud node completes its allocated queries, it can notify other cloud nodes. The cloud with pending queries can offload some to this cloud. To prevent further load imbalance, the offloading policy can be dynamically adjusted through inter-cloud negotiation; however, we do not elaborate on the specific mechanisms here. Theoretically, it is preferable to estimate whether the transmission overhead is lower than the queuing delay, which can be assessed based on both the query queue and the network conditions. However, given that LLM inference is computationally intensive, inference latency is typically much higher than transmission latency, making task offloading to idle devices a more effective strategy.

Discussion on integrating with network state: Based on the aforementioned query scheduling, we provide a discussion on integrating network state considerations. Given that LLM inference is computationally intensive, the latency caused by computation significantly outweighs, by orders of magnitude, the latency associated with data transmission. Therefore, for tasks aimed at minimizing query processing time, the scheduling method proposed in this paper can be prioritized. For tasks with specific requirements, such as those focusing on the latency of the first token of the response, the cloud node with lower end-to-end network transmission latency can be prioritized, and it is necessary to activate priority-enabled query processing in the inference engine. Since this paper primarily focuses on computation load-oriented scheduling, the integration with network states will be further studied as part of our future work.

3.4 Iteration scheduler

Since the computations for queries within a batch are independent of each other, the values of the *token vector*, *attention mask*, and *KV-Cache* corresponding to a completed query within the batch can be overwritten or modified by a new one, which allows seamless replenishment of the processing batch. This is the core of iteration granularity scheduling, primarily using padding and assigning negative infinity (-inf) to align these three variables for a newly inserted query without affecting the execution of subsequent

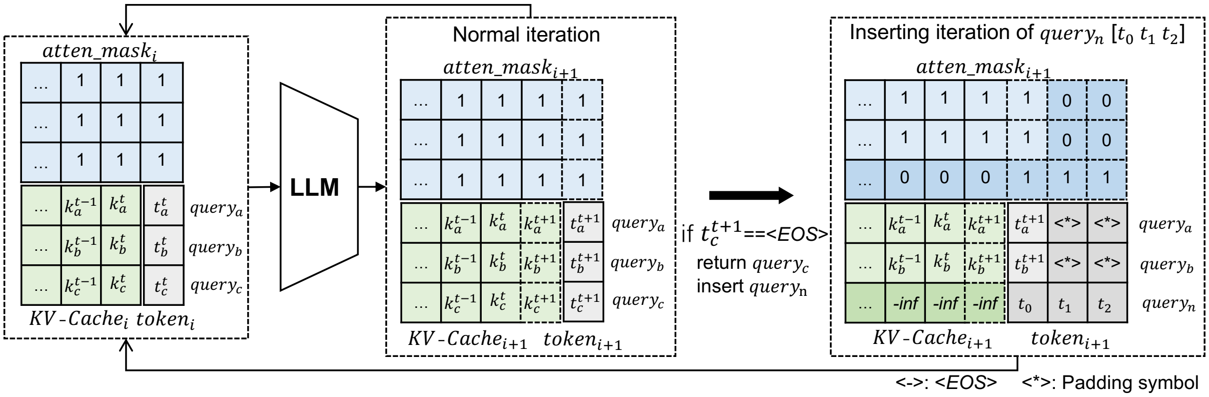
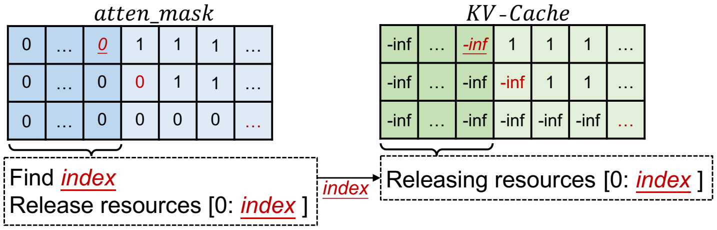


Figure 5 Example of iteration processing



$atten\_mask$
KV-$Cache$
-inf
Find *index*
Release resources [0: *index*]
index
Releasing resources [0: *index*]

Figure 6 Example of resource releasing

computations. Specifically, as shown in Figure 5, assuming the inference for $query_c$ has completed, the modification of *token vector*, *attention mask*, and *KV-Cache* for inserting $query_n$ is conducted as described below:

1) For *token vector*$_{i+1}$, the next inference iteration is effectively the first iteration for $query_n$, meaning that all tokens of $query_n$ need to be input into the model. In contrast, for the incomplete queries ($query_a$ and $query_b$), only their latest single tokens need to be processed. To this end, padding can be applied to align the two single tokens of $query_a$ and $query_b$ with the full sequence of $query_n$, similar to the standard padding operation in batch-wise inference. Assuming the length of $query_n$ is l_n, the dimension of the modified *token vector*$_{i+1}$ becomes $batch\_size \times l_n$.

2) For *attention mask*$_{i+1}$, the value corresponding to $query_c$ is first set to 0, indicating that the existing *KV-Cache* for $query_c$ is invalid for $query_n$. In a normal iteration, a column of ones is appended to the mask, indicating that each query's input token in the next iteration is not a padding symbol. In contrast, based on the padding modification of the *token*$_v$*ector*, a sub-*attention mask* of length l_n is appended instead. Specifically, the sub-*attention mask* vector for $query_n$ is $[1, \ldots, 1]$, indicating that none of its tokens are padding symbols, while the sub-*attention mask* vector for each incomplete query is $[1, 0, \ldots, 0]$, indicating that all tokens except for the first one are padding symbols.

3) For *KV-Cache*$_{i+1}$, the values corresponding to incomplete queries remain unchanged. In contrast, the values associated with the completed $query_c$ should not impact the computation of the new $query_n$. To ensure the independence of $query_n$'s calculation, all *Keys* and *Values* of $query_c$ are set to $-inf$. Additionally, after this iteration, the *KV-Cache* will be appended to the latest *Keys* and *Values* with a length of l_n, as the length of the input *token vector* is l_n, which is similar to the process in the prefilling phase of inference.

When inserting a new query, the front modification parts of both the corresponding *attention mask* and *KV-Cache*—essentially a form of padding—are uninfluential to the model inference result. Therefore, after each position of the batch has been updated for a round, the resources occupied by these padding parts of the *KV-Cache* and *attention mask* of each query overlap can be released. Since the positions in the *KV-Cache* set to $-inf$ align with the positions in the *attention mask* that are set to 0 during query insertion, it is possible to find the longest prefix with a value of 0 of each vector in the *attention mask*, denoted as *index*. As illustrated in Figure 6, the *attention mask*$[0:index]$ and the *KV-Cache*$[0:index]$ along the *seq_length* dimension can then be released simultaneously. After the aforementioned operations, when a new query is inserted into the current processing batch, the three variables involved in the inference not only satisfy the model's dimensional requirements but also ensure that the calculations for all queries remain unaffected.

To provide further clarification, the operations mentioned above are not restricted to a specific model, as the KV-Cache mechanism is universally supported by current mainstream open-source LLMs. For example, the interfaces for models such as the GPT series, Llama series, and Qwen series remain consistent in the Transformers framework. Moreover, regarding model accuracy, the computation of the next token in the LLM primarily relies on the *KV-Cache* and the current input token. The iteration scheduler of Mico does not modify the already cached *Keys* and *Values*, thereby preserving both the logical and numerical consistency of the computations. This ensures that the inference process with the task iteration scheduler maintains the same result in the generated responses as the inference without it.

3.5 Optimizations

3.5.1 *Mitigating padding implications*

Inserting a new query introduces additional padding for aligning to incomplete queries within the batch, with the padding size of the new query's length. To mitigate this issue, selecting shorter queries from the query pool can reduce the padding overhead. A trade-off can be made between the remaining iterations of incomplete queries and the padding introduced by the new query to decide whether to insert. It is possible to further employ the prefilling and decoding phases decoupling scheme [21, 22]. Briefly, this scheme first conducts the first inference iteration for the query, storing the *attention mask*, *token vector*, and *KV-Cache*; then, it inserts the prefilled query into the processing batch. This allows the *token vector* and *KV-Cache* to be aligned directly without introducing additional padding operations. This scheme has performance gains in scenarios with very long queries.

3.5.2 *Adaptive batch size*

A small batch size will reduce the parallelism of inference. Since the volume of *KV-Cache* increases gradually with the inference iterations, a larger batch size may cause GPU memory overflow, leading to inference failure and affecting the system's throughput. Given that the response length predictor, it is possible to adjust the batch size adaptively according to the status of the resources, *e.g.*, using the greedy algorithm to select as many queries as possible to compose a batch, on the premise of ensuring that there will be no resource overflow during inferring. In addition, in the context of iteration granularity scheduling, the adjustment of batch size can also be executed during the inference process, *i.e.*, when the resource occupation reaches a set threshold, it allows storing the involved three variables of some being inferred queries to RAM and releasing the occupied resource to downscale the batch size; Conversely, it is also possible to add multiple queries to upscale the batch size to improve the parallelism of inference when the occupied memory resources are released.

4 Evaluation

In this section, we introduce the experiment setting first, and exhibit the improvements of Mico by comprehensive experimental results.

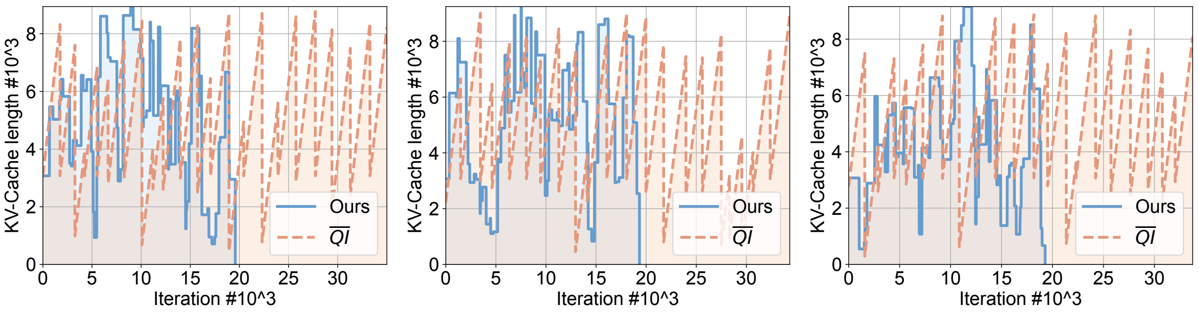
4.1 Experimental settings

4.1.1 *Predictor settings*

The Predictor's base model employs the pre-trained DistilBERT [20], consisting of 66 million parameters. The model employs a cross-entropy loss function as the training objective, uses the *AdamW* optimizer with a linear learning rate decay strategy during training. The learning rate is 5×10^{-5}, batch size is 32, and epoch is 3. And two datasets, *Alpaca* [5], and *Math-Word-Problems* [6], which respectively comprise

[5] https://huggingface.co/datasets/tatsu-lab/alpaca

[6] https://huggingface.co/datasets/microsoft/orca-math-word-problems-200k



(a) (b) (c)

Figure 7 *KV-Cache* length of each cloud. (a) Could 1; (b) Could 2; (c) Could 3.

52k and 200k data items, are employed to fine-tune the prediction model. In terms of model output, we set up 10 buckets for length classification.

4.1.2 *Employed foundation LLMs*

GPT-2 [7] and GPT-Neo [8] are utilized as foundational service models, supporting maximum sequence lengths of 1,024 and 2,048 tokens, respectively. Each inference engine employs an NVIDIA A6000 GPU, which adequately satisfies the inference requirements to validate the performance of MICO and compared methods.

4.1.3 *Comparison methods description*

MICO not only considers query scheduling based on predicted workloads rather than quantity, denoted as Q for clarity, but also introduces scheduling at iteration granularity for inference processing (denoted as I). Based on recent related studies, three comparative methods are employed.

a) Query quantity-based scheduling: As existing task scheduling strategies neglect the inference workload, they can all be grouped as quantity-based methods. This benchmark is denoted as $\overline{QI}$, which schedules queries as evenly as possible for service nodes;

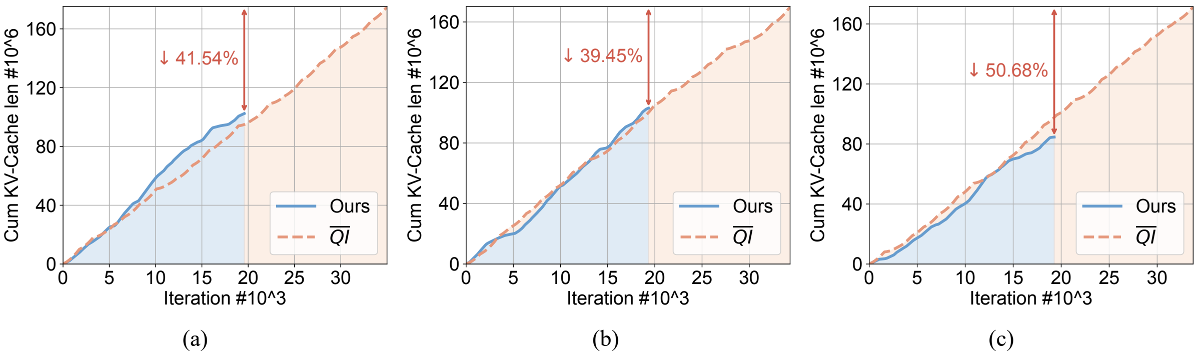
b) Iteration granularity scheduling: As classical works on batch-wise inference optimization, Orca series works [12, 14] with the same tactic to improve the inference efficiency of a single service engine. This comparative method employs quantity-based scheduling, while incorporating iteration granularity scheduling during the inference processing, denoted as $\overline{Q}I$;

c) An ablation method: For comparison, this method conducts workload-based query scheduling according to prediction results but excludes iteration granularity optimization, denoted as $Q\overline{I}$. For ease of presentation, the comparative methods are represented by defined symbols in the evaluation.

In the experiments, a set of queries is randomly selected from the validation dataset of *Alpaca* and processed using both MICO and the SOTA method. Moreover, to directly demonstrate the efficacy of the proposed scheme, differences in network performance from the scheduler to each cloud and the processing capabilities of all clouds were deliberately obscured, assuming uniformity in these metrics.

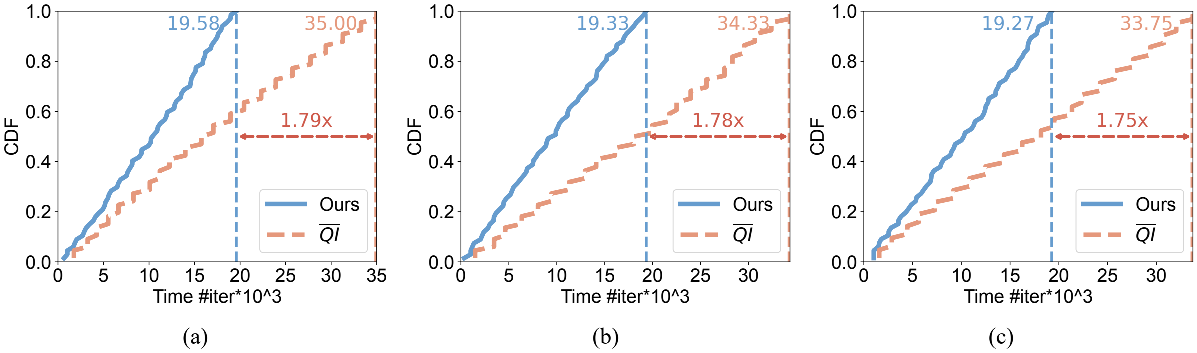
Note that since queries are randomly selected, their length distribution tends to be relatively balanced, enabling the round-robin distribution scheme to provide a small variance in token load across clouds. However, in practice, length distributions are often imbalanced, and it is infeasible to completely prevent load imbalances issue. We conducted an informal experiment with manually configured imbalanced query lengths, which demonstrated that MICO effectively balances the workload. Additionally, in alignment with intuitive expectations, the extent of MICO's relative improvement over existing methods is directly related to the degree of imbalance. Due to the lack of real traces and for the sake of fairness, we eliminate the above imbalance as much as possible by randomly selecting the query strategy in the evaluation experiments.

[7] https://huggingface.co/openai-community/gpt2
[8] https://huggingface.co/EleutherAI/gpt-neo-2.7B



(a) (b) (c)

Figure 8 Cumulative *KV-Cache* length of each cloud. (a) Cloud 1; (b) Cloud 2; (c) Cloud 3.



(a) (b) (c)

Figure 9 Cumulative distributions of completed queries. (a) Cloud 1; (b) Cloud 2; (c) Cloud 3.

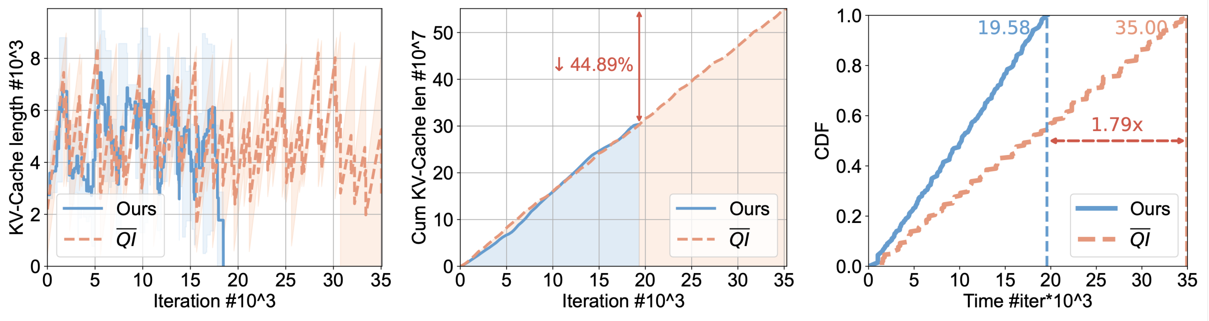
4.2 Results and analyses

4.2.1 *Overall performances*

In the experiments, we compared the performance of Mico and $\overline{QI}$ in terms of GPU memory consumption, query completion time, and throughput of query processing, respectively. The experiment variables involve the number of processed queries, the number of cloud nodes, and the batch size. The GPT-2 model supports a maximum sequence length of 1,024 tokens by default, and it will intercept the inference when the sequence length over the threshold. In addition to fixed model parameters, *KV-Cache* is the primary dynamic memory-consuming part during the model inferring [9], whose size linearly increases with the number of inference iterations along the sequence length dimension. Therefore, it is reasonable to use the length of *KV-Cache* in *seq_length* dimension to evaluate the performance of memory consumption for different methods. Empirical measurements demonstrate that the time overhead for each new query insertion remains consistently below 1 ms, a negligible computational cost when compared to the substantial processing time of a single model iteration, which typically requires approximately 70 ms—an order of magnitude difference. Therefore, for simplicity, it is possible to leverage iteration as the metric of the time dimension.

a) **Individual cloud**. Under the settings with queries of 200, batch size of 3, and cloud nodes of 3, the multi-faceted performances of each cloud are as follows:

- **Memory resource consumption**. The *KV-Cache* lengths of Mico and $\overline{QI}$ are illustrated in Figure 7. During the inference process, the *KV-Cache* length of $\overline{QI}$ exhibits a periodic sawtooth pattern. This pattern emerges because the *KV-Cache* length for each query in the processing batch incrementally increases per token from the beginning, until all resources are simultaneously released upon the completion of the batch inference. However, the *KV-Cache* length of Mico can be maintained relatively high; the reduction in length results from resource release following the insertion of a new query. Additionally, the peak *KV-Cache* lengths of both methods are comparable, yet Mico completes inference tasks earlier. This indicates that Mico can process the queries with higher resource efficiency.



(a) (b) (c)

Figure 10 Inference service system. (a)*KV-Cache* length; (b) Cumulative *KV-Cache* length; (c) Cumulative distribution of completed queries.

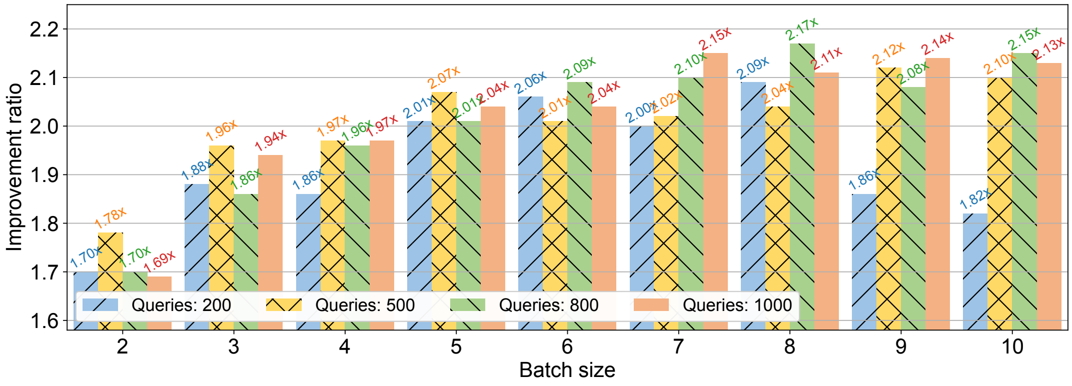


Figure 11 Improvement ratio of system throughput

Furthermore, cumulating the *KV-Cache* lengths of each iteration provides an insight into the memory resource consumption of the inference phase. The cumulative length distributions of three clouds are illustrated in Figure 8, which demonstrates that Mico reduces memory resource consumption by 41.54%, 39.45%, and 50.68% compared to $\overline{QI}$, respectively.

- **Query processing throughput**. For the same set of queries, we respectively measured the query completion times of Mico and $\overline{QI}$. For $\overline{QI}$, the completion of a query is indicated by generating the ⟨EOS⟩ symbol, rather than uniformly obtaining results at the end of the batch inference. The cumulative distributions of completed queries for three clouds are shown in Figure 9. The step-like feature of $\overline{QI}$ is more pronounced than that of Mico, as its batch inference begins synchronously. Conversely, Mico can promptly replenish the batch with new queries to replace those completed, thereby achieving throughput accelerations of 1.79×, 1.78×, and 1.75× for each cloud, respectively.

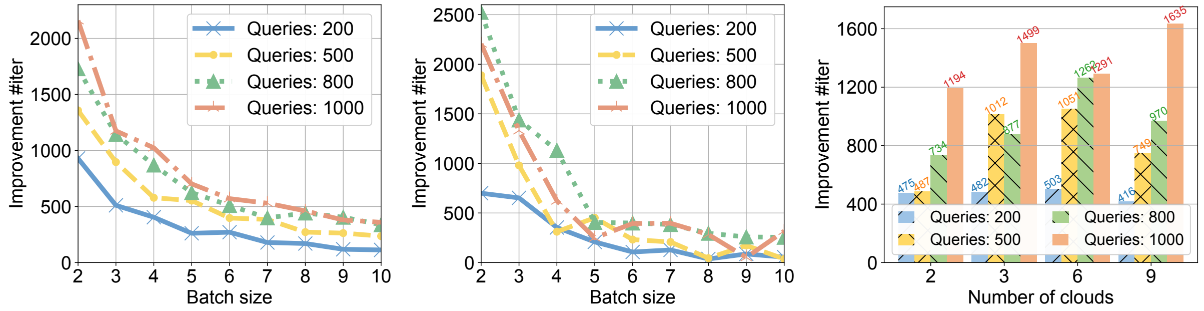
b) **Inference service system**. Under the settings above, we illustrated the inference service system in terms of the *KV-Cache* length (minimum, mean, and maximum of all clouds), cumulative *KV-Cache* length, and the cumulative distribution of completed queried in (a), (b), and (c) of Figure 10, respectively. It can be observed that the predictor of Mico can further balance the actual processing token load among all clouds, even under relatively balanced query length distribution, thereby mitigating the issue that the processing efficiency of the entire system may be hindered by a single cloud. Overall, compared to $\overline{QI}$, Mico reduces memory resource consumption by 44.89% and accelerates query processing throughput by 1.79×.

Additionally, to further evaluate the outperformance of Mico compared to $\overline{QI}$, we conducted the following experiments under various batch sizes and numbers of cloud nodes.

- **Different batch size**. To evaluate the impact of batch size on throughput performance, we conducted experiments with batch sizes ranging from 2 to 10 and queries of 200, 500, 800, and 1,000 settings, respectively. As displayed in Figure 11, compared to $\overline{QI}$, Mico accelerates the query processing throughput of the entire service system by 1.69-2.17×.

Table 1 Throughput improvement ratio under different numbers of clouds

Number	Batch size								
of clouds	2	3	4	5	6	7	8	9	10
2	1.67	1.88	1.98	2.05	2.08	2.10	2.10	2.22	2.17
3	1.70	1.91	2.01	2.04	2.06	2.14	2.10	2.16	2.16
6	1.79	1.94	2.00	2.04	2.07	2.10	2.10	2.09	2.02
9	1.77	1.89	1.98	2.08	2.03	2.02	2.07	2.11	2.14



(a)

(b)

(c)

Figure 12 Throughput gains from Query granularity scheduling (Q). (a) QI vs. $\overline{Q}I$; (b) $Q\overline{I}$ vs. $\overline{QI}$; (c) QI vs. $\overline{Q}I$ under different number of clouds.

- **Number of clouds**. To measure the outperformance of Mico over $\overline{QI}$ under the different number of clouds, we set the cloud nodes to 2, 3, 6, and 9, the batch size to 2 to 10, and queries of 800, respectively. As shown in Table 1, Mico accelerates the system throughput by 1.67-2.22× under the above settings. To explain here, due to the random character of query selection, it is reasonable to expect a slight difference in results with the same parameter settings.

4.2.2 *Ablation evaluations*

a) **Predictor**: The fine-tuned predictor model achieved 85.37% and 80.69% accuracy on *Alpaca* and *Math-Word-Problems*, respectively, with verification sets randomly sampled. The average prediction latency for each query is about 4.5 *ms*. To verify the model's adaptability, we selected about 200 items from the training set of a new dataset [9] to fine-tune the prediction model, and the model achieved 82.5% prediction accuracy in the corresponding validation set, while decreasing the accuracy by about 0.9% on the original dataset. Moreover, we found that the items that are not correctly classified were divided into neighboring buckets with a high probability, which means that the prediction accuracy can be further improved by reducing the number of classification buckets. It is unnecessary to pursue an extremely high accuracy of the Predictor due to the parameters setting of top-k/top-p and temperature for token selecting of LLM will lead to the output results with a certain degree of randomness [23]. Therefore, we demonstrate the superiority of Mico by the comparative performance results with existing methods.

b) **Prediction-based query scheduling (Q)**: In this series of experiments, we concentrate on evaluating the performance of the prediction-based query scheduling Q. Specifically, we compare the system query processing throughput both with and without Q across various query quantities and batch sizes. For the inference engine, queries are randomly selected from the query pool in the initial batch composition and subsequent insertion. To mitigate the effect of randomness, we present the average results from 10 independent experiments.

Firstly, as illustrated in Figure 12 (a) and (b), regardless of whether I is used or not, Q exhibits performance improvements in system throughput compared to $\overline{Q}$. These gains diminish with increasing batch sizes because, given a constant total number of queries, the overall completion time decreases as the batch size scales up.

Secondly, with a batch size of 3, we varied the number of cloud nodes (2, 3, 6, and 9) and the number of queries (200, 500, 800, and 1000). The throughput improvement achieved by Q compared

[9] https://huggingface.co/datasets/Vineeshsuiii/Software_Engineering_interview_datasets

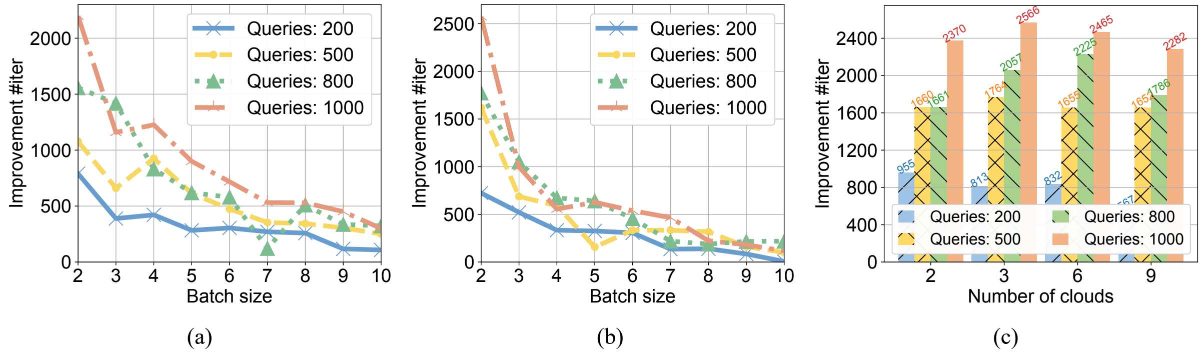


Figure 13 Throughput gains from Iteration granularity scheduling (I). (a) QI vs. $Q\overline{I}$; (b) $\overline{Q}I$ vs. $\overline{QI}$; (c) QI vs. $Q\overline{I}$ under different number of clouds.

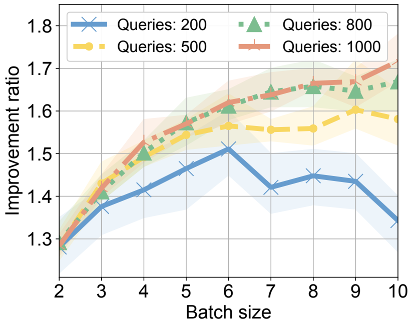


Figure 14 Improvement on GPT-neo model

to $\overline{Q}$ is illustrated in Figure 12 (c). It can be noted that the performance enhancement becomes more pronounced as the number of queries increases. This is primarily due to the system throughput decreasing when queries are distributed unevenly across cloud nodes, especially those involving a large number of tokens.

c) **Iteration scheduling (**I**)**: This set of experiments is specifically designed to evaluate the performance of the iteration scheduling I. Similarly, we compare the system query processing throughput both with and without I across various query quantities and batch sizes. As shown in Figure 13 (a) and (b), the experimental results demonstrate that I enhances system throughput compared to $\overline{I}$, regardless of whether Q is implemented or not. Furthermore, as illustrated in Figure 13 (c), I also improves system throughput across different numbers of clouds.

From the entire ablation experiments, employing either Q or I independently can enhance system throughput across various query quantities, batch sizes, and numbers of cloud nodes. However, the degree of performance improvement is less than when both strategies are employed jointly, which is the scheme proposed in this paper, Mico.

4.2.3 *Different service models*

We conducted the experiments on the GPT-neo model, whose maximum sequence length is 2,048 tokens, i.e., the inference process ends once the total token count reaches 2,048 even if ⟨EOS⟩ has not been generated. We performed 10 independent experiments for different batch sizes and query scales with 3 cloud nodes. As shown in Figure 14, the results indicate that Mico provides a 1.22-1.75× improvement in throughput compared to $\overline{QI}$. This finding is consistent with the aforementioned experimental results.

5 Related work

LLM as-a-service. Recently, the landscape of LLMs has experienced a surge in technological advancements [24–26]. These models have been progressively fine-tuned by adaptive instruction tuning to better align with complex human tasks and are available as services to users now [27]. Some well-known examples include GPT, Llama[10], PaLM[11], ERNIE, and Qwen, which have been efficiently implemented in cloud platforms to facilitate a vast number of LLM-based services[12]. These services efficiently manage millions of daily query inferences, making them integral to modern computational infrastructure. In this rapidly evolving field, improving service quality and significantly reducing inference overhead have become key focuses of current research [12, 15].

High-efficiency model inference. LLM inference services providers can use high-efficiency model inference schemes to provide better quality of service to their users and also reduce operational costs [28, 29]. The technologies involved can be categorized as follows: a) Kernel Customization. [30] designs a hardware-friendly matrix multiplication based on tiled singular value decomposition, which can further utilize GPU resources. [31] divides the softmax layer into multiple layers and modifies the data access mode, which improves the computational efficiency of softmax. [32] reduces the need for large contiguous memory by segmenting the input vector and calculating the attention weights for each segment independently. b) Parallel Computing [33–37]. The pipeline, tensor, and 3D parallelism methods can improve the efficiency of model inference in multi-GPU scenarios. c) Quantization is also an essential technology that can optimize inference processes [38, 39]. d) Some studies try to enhance batch-wise inference [40, 41], *e.g.*, [42] selects queries with similar length for variable length input situations of transformer-based models. e) Distributed inference [43]. [44] introduces a novel approach that partitions and deploys LLM across distributed edge devices and cloud servers in a collaborative edge computing environment, addressing challenges like device heterogeneity, bandwidth limitations, and model complexity through adaptive device selection and model partitioning. Similarly, [45] leverages the Internet to orchestrate geographically distributed devices to run LLMs. PD disaggregation can be regarded as a distributed inference technique [21, 46], which decouples the computationally demanding Prefill phase from the memory-intensive Decode phase during inference. This separation allows these phases to be assigned to different devices or clusters, optimizing resource utilization and enhancing throughput.

However, the aforementioned works, whether focusing on centralized or distributed inference, are dedicated to improving the inference efficiency for a single query. In contrast, our work focuses on allocating queries for multiple inference service-deployed clouds to enhance overall efficiency. In other words, our approach is orthogonal to existing works.

Prediction of response length. Existing works on response length prediction primarily focus on the tasks of non-autoregressive models, where the entire response can be generated in one forward calculation [47]. For example, some straightforward approaches use the length distribution of datasets or predict the number of tokens translated from the input tokens [47, 48], and some studies incorporate special symbols into the encoder or pre-predict the encoder's output to predict the response length [49, 50]. However, such ways are only applicable to translation tasks where there is a strong correlation between the lengths of input and output vectors. [15] utilizes the LLM itself by adding a specific prompter to get the length of the corresponding response before outputting it, but such a method requires completing the profiling phase and a certain number of autoregressive iterations in the inference engine. In this work, we aim to predict the response length of autoregressive LLMs before scheduling the queries, and the methods used will be detailed below.

6 Conclusion

In this paper, we proposed MICO for multi-cloud deployed LLM services, which consists of the token-aware query scheduling scheme based on response length prediction, and the flexible model inference architecture that supports the early return of inference result and dynamic query insertion for being inferred batch. The experimental results show that MICO reduces the resource consumption of *KV-Cache* by 44.89%, and accelerates the query processing throughput by up to 2.22×.

In future work, we will further conduct more experiments to validate the effectiveness of MICO in other series LLMs and integrate attributes such as network state, computing power heterogeneity of cloud nodes, *etc.*, into the scheduling policy to improve the practical deployability of MICO.

Acknowledgements This work was supported in part by the National Key R&D Program of China (No. 2024YFB2906602), and in part by the National Natural Science Foundation of China (NSFC) (No. 62372009 and 62172054), research grant No.

[10] https://www.llama2.ai

[11] https://ai.google/discover/palm2/

[12] https://yiyan.baidu.com

SH2024JK29, and High Performance Computing Platform of Peking University. The work of Ke Xu was supported in part by the National Natural Science Foundation of China for Distinguished Young Scholars under Grant No. 62425201.

References

1 Liu X, Zheng Y N, Du Z X, et al. GPT understands, too. AI Open, 2023. in press

2 Li X, Feng X, Hu S, et al. DTLLM-VLT: Diverse Text Generation for Visual Language Tracking Based on LLM. In: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, Seattle WA, USA, 2024. 7283–7292

3 Kamalloo E, Upadhyay S, Lin J. Towards Robust QA Evaluation via Open LLMs. In: Proceedings of the 47th International ACM SIGIR Conference on Research and Development in Information Retrieval, Washington DC, USA, 2024. 2811–2816

4 Laban P, Kryściński W, Agarwal D, et al. SUMMEDITS: Measuring LLM Ability at Factual Reasoning Through the Lens of Summarization. In: Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing, Singapore, 2023. 9662–9676

5 Google Cloud. Deployments and endpoints Generative AI on Vertex AI. Google Cloud Vertex AI Documentation. Retrieved May 7, 2025.

6 Wu T, He S, Liu J, et al. A Brief Overview of ChatGPT: The History, Status Quo and Potential Future Development. IEEE/CAA J Autom Sin, 2023, 10(5): 1122-1136.

7 Rawat W, Wang Z. Deep Convolutional Neural Networks for Image Classification: A Comprehensive Review. Neural Comput., 2017, 29(9): 2352-2449.

8 Wolf T, Debut L, Sanh V, et al. Transformers: State-of-the-art Natural Language Processing. In: Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations, Virtual Conference, 2020. 38-45

9 Kwon W, Li Z, Zhuang S, et al. Efficient Memory Management for Large Language Model Serving with Pagedattention. In: Proceedings of the 29th Symposium on Operating Systems Principles, Koblenz, Germany, 2023. 611-626

10 Rasley J, Rajbhandari S, Ruwase O, et al. Deepspeed: System Optimizations Enable Training Deep Learning Models With Over 100 Billion Parameters. In: Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Diego, USA, 2020. 3505-3506

11 Chelba C, Chen M, Bapna A, et al. Faster Transformer Decoding: N-gram Masked Self-attention[J]. arXiv preprint arXiv:2001.04589, 2020.

12 Yu G I, Jeong J S, Kim G W, et al. Orca: A Distributed Serving System for Transformer-Based Generative Models. In: Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation, Carlsbad, CA, USA, 2022. 521–538

13 Jin Y, Wu C F, Brooks D, et al. S^3: Increasing GPU Utilization during Generative Inference for Higher Throughput. In: Proceedings of the Advances in Neural Information Processing Systems, New Orleans, LA, USA, 2023. 18015–18027

14 Wu B, Zhong Y, Zhang Z, et al. Fast Distributed Inference Serving for Large Language Models. arXiv preprint arXiv:2305.05920, 2023.

15 Zheng Z, Ren X, Xue F, et al. Response Length Perception and Sequence Scheduling: An LLM-Empowered LLM Inference Pipeline. In: Proceedings of the 37th International Conference on Neural Information Processing Systems, New Orleans, LA, USA, 2023. 65517-65530

16 Achiam J, Adler S, Agarwal S, et al. Gpt-4 technical report. arXiv preprint arXiv:2303.08774, 2023.

17 Touvron H, Martin L, Stone K, et al. Llama 2: Open Foundation and Fine-tuned Chat Models. arXiv preprint arXiv:2307.09288, 2023.

18 Vaswani A, Shazeer N, Parmar N, et al. Attention is All You Need. In: Proceedings of the Advances in Neural Information Processing Systems, Long Beach, CA, USA, 2017. 5998–6008

19 Strati F, McAllister S, Phanishayee A, et al. DéjàVu: KV-Cache Streaming for Fast, Fault-tolerant Generative LLM Serving. In: Proceedings of the 41st International Conference on Machine Learning, Vienna, Austria, 2024. in press

20 Sanh V, Debut L, Chaumond J, et al. DistilBERT: A Distilled Version of BERT: Smaller, Faster, Cheaper and Lighter. In: Proceedings of the Advances in Neural Information Processing Systems, Vancouver, BC, Canada, 2020.

21 Zhong Y, Liu S, Chen J, et al. DistServe: Disaggregating Prefill and Decoding for Goodput-optimized Large Language Model Serving. In: Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation, Santa Clara, CA, USA, 2024. 193-210.

22 Cong P, Chen Q, Zhao H, et al. BATON: Enhancing batch-wise inference efficiency for large language models via dynamic re-batching. In: Proceedings of the ACM on Web Conference, Sydney NSW, Australia, 2025. 2309-2318.

23 Wan Z, Feng X, Wen M, et al. Alphazero-like Tree-search Can Guide Large Language Model Decoding and Training. In: Proceedings of the 41st International Conference on Machine Learning, Vienna, Austria, 2024. in press

24 Brown T, Mann B, Ryder N, et al. Language models are few-shot learners. In: Proceedings of the Advances in Neural Information Processing Systems, Virtual, 2020. 1877–1901

25 Chowdhery A, Narang S, Devlin J, et al. Palm: Scaling Language Modeling With Pathways. J Mach Learn Res, 2023, 24(240): 1–113

26 Hoffmann J, Borgeaud S, Mensch A, et al. Training Compute-optimal Large Language Models. In: Proceedings of the 36th International Conference on Neural Information Processing Systems, New Orleans, LA, USA, 2022. 30016–30030

27 Ouyang L, Wu J, Jiang X, et al. Training Language Models to Follow Instructions With Human Feedback. In: Proceedings of the 36th International Conference on Neural Information Processing Systems, New Orleans, LA, USA, 2022. 27730–27744

28 Kim S, Hooper C, Wattanawong T, et al. Full Stack Optimization of Transformer Inference. In: Proceedings of the Architecture and System Support for Transformer Models, Oralndo, FL, USA, 2023. 1–6

29 Chitty-Venkata K T, Mittal S, Emani M, et al. A Survey of Techniques for Optimizing Transformer Inference. J Syst Archit, 2023, 144: 102990–103028

30 Choi J, Li H, Kim B, et al. Accelerating Transformer Networks Through Recomposing Softmax Layers. In: Proceedings of the 2022 IEEE International Symposium on Workload Characterization, Austin, TX, USA, 2022. 92–103

31 Li H, Choi J, Kwon Y, et al. A Hardware-Friendly Tiled Singular-Value Decomposition-Based Matrix Multiplication for Transformer-Based Models. IEEE Comput Archit Lett, 2023, 22: 169–172

32 Dao T, Fu D, Ermon S, et al. Flashattention: Fast and Memory-efficient Exact Attention With IO-Awareness. In: Proceedings of the Advances in Neural Information Processing Systems, New Orleans, LA, USA, 2022. 16344–16359

33 Shoeybi M, Patwary M, Puri R, et al. Megatron-lm: Training Multi-billion Parameter Language Models Using Model Parallelism. arXiv preprint arXiv:1909.08053, 2019.

34 Ma R, Yang X, Wang J, et al. HPipe: Large Language Model Pipeline Parallelism for Long Context on Heterogeneous Cost-effective Devices. In: Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Mexico City, Mexico 2024. 1–9

35 Chen Y, Pan X, Li Y, et al. EE-LLM: Large-Scale Training and Inference of Early-Exit Large Language Models with 3D Parallelism. In: Proceedings of the 41st International Conference on Machine Learning, Vienna, Austria, 2024. in press

36 Li Z, Zheng L, Zhong Y, et al. AlpaServe: Statistical multiplexing with model parallelism for deep learning serving. In: Proceedings of the 17th USENIX Symposium on Operating Systems Design and Implementation, Boston, MA, USA, 2023. 663-679

37 Yu C, Wang T, Shao Z, et al. TwinPilots: A New Computing Paradigm for GPU-CPU Parallel LLM Inference. In: Proceedings of the 17th ACM International Systems and Storage Conference, Haifa, Israel, 2024. 91-103

38 Wu X, Li C, Aminabadi R Y, et al. Understanding Int4 Quantization for Language Models: Latency Speedup, Composability, and Failure Cases. In: Proceedings of the International Conference on Machine Learning, Honolulu, Hawaii, USA, 2023. 37524–37539

39 Frantar E, Ashkboos S, Hoefler T, et al. GPTQ: Accurate Post-Training Quantization for Generative Pre-trained Transformers. In: Proceedings of the 11st International Conference on Learning Representations, Kigali, Rwanda, 2023. 1–16

40 Cheng Z, Kasai J, Yu T. Batch Prompting: Efficient Inference with Large Language Model APIs. In: Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing, Singapore, 2023. 792–810

41 Sheng Y, Zheng L, Yuan B, et al. Flexgen: High-throughput Generative Inference of Large Language Models Single GPU. In: Proceedings of the International Conference on Machine Learning, Honolulu, Hawaii, USA, 2023. 31094–31116

42 Fang J, Yu Y, Zhao C, et al. Turbotransformers: An Efficient GPU Serving System for Transformer Models. In: Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Republic of Korea, 2021. 389–402

43 Ma R, Wang J, Qi Q, et al. Poster: PipeLLM: Pipeline LLM inference on heterogeneous devices with sequence slicing. In: Proceedings of the ACM SIGCOMM 2023 Conference, New York City, NY, USA, 2023. 1126-1128

44 Zhang M, Shen X, Cao J, et al. Edgeshard: Efficient llm inference via collaborative edge computing. IEEE Internet Things J., 2024, Early Access.

45 Borzunov A, Ryabinin M, Chumachenko A, et al. Distributed inference and fine-tuning of large language models over the internet. In: Proceedings of the 37th International Conference on Neural Information Processing Systems, New Orleans, LA, USA, 2023, 36: 12312-12331

46 Patel P, Choukse E, Zhang C, et al. Splitwise: Efficient generative llm inference using phase splitting. In: Proceedings of the ACM/IEEE 51st Annual International Symposium on Computer Architecture, Buenos Aires, Argentina, 2024. 118-132.

47 Ren Y, Liu J, Tan X, et al. A Study of Non-autoregressive Model for Sequence Generation. In: Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, Online, 2020. 149–159

48 Sun Z, Li Z, Wang H, et al. Fast Structured Decoding for Sequence Models. In: Proceedings of the Advances in Neural Information Processing Systems, Vancouver, BC, Canada, 2019. 3011–3020

49 Ghazvininejad M, Levy O, Liu Y, et al. Mask-Predict: Parallel Decoding of Conditional Masked Language Models. In: Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing, Hong Kong, China, 2019. 6112–6121

50 Devlin J, Chang M W, Lee K, et al. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In: Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Minneapolis, MN, USA, 2019. 4171–4186