Mil: Cost-guided Minimum Makespan Scheduling for Applications of Multiple LLMs

Jingzhi Fang
HKUST
jfangak@connect.ust.hk

Yanyan Shen
Shanghai Jiao Tong University
shenyy@sjtu.edu.cn

Yue Wang
Shenzhen Institute of Computing Sciences
ywangby@connect.ust.hk

Lei Chen
HKUST, HKUST(GZ)
leichen@cse.ust.hk

ABSTRACT

Multi-LLM applications calling multiple LLMs per request are emerging. An important scenario is running these applications offline on a request set. This work aims to minimize the offline inference makespan of these applications to save time and cost. Specifically, we study minimum-makespan scheduling of multi-LLM applications (the MLAS problem), which requires GPU allocation, LLM parallelism selection, and LLM execution orchestration. MLAS is NP-hard, and it differs from existing multi-model frameworks and job scheduling problems due to LLMs' unique properties (e.g., high memory demand, complex inference behavior), the offline inference setting, and relaxed execution precedence constraints. There is no existing work on MLAS and simple rules cannot handle all the problem instances. We propose a framework, Mil, for MLAS with three major components: (1) processing functions estimating LLM processing rates by output length sampling, inference process simulation, and per-generation-iteration latency estimation; (2) a greedy method that finds a good schedule with a theoretical guarantee on a simplified problem instance; (3) a runtime adjustment mechanism reducing GPU idleness. Experiments on various applications (ensembling, routing, chain summary, mixed) show that Mil can achieve up to 3.4× end-to-end speedups over current practice.

PVLDB Reference Format:
Jingzhi Fang, Yanyan Shen, Yue Wang, and Lei Chen. Mil: Cost-guided Minimum Makespan Scheduling for Applications of Multiple LLMs. PVLDB, 19(9): 1893 - 1906, 2026.
doi:10.14778/3819518.3819522

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/puddingfjz/Mil.

1 INTRODUCTION

Multi-LLM applications calling multiple LLMs per request are emerging (e.g., ensembling [21], routing [19], summarization [3]). A key scenario is offline inference, where requests are pre-prepared [28, 29, 34, 36, 37, 39, 43, 56], and the goal is to minimize makespan, not per-request latency [30]. Request counts range from hundreds to hundreds of thousands [3, 19, 21, 28, 43], and running time from minutes to days. Even if a small set of requests may take only minutes, the application may run repeatedly on different data (e.g., in AI development cycles), so the accumulated time and cost savings will be significant, given the high expense of GPUs.

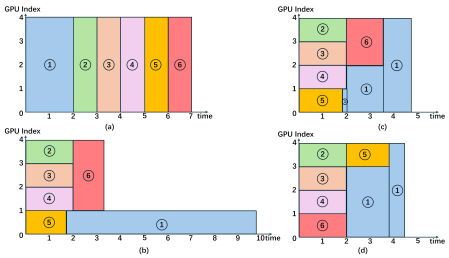


Figure 1: Different schedules: (a) Max-heuristic, (b) non-preemptive Min-heuristic, (c) preemptive Min-heuristic, (d) considering the LLM processing rates.

In this paper, we aim to minimize the multi-LLM application makespan on multiple GPUs in the offline scenario. A multi-LLM application can be run in different ways: (1) assigning different GPU counts and parallelism strategies to LLMs, (2) loading LLMs onto GPUs to compute sequentially or concurrently in any order (subject to data dependencies), and (3) interrupting a running LLM and changing its parallelism strategy. Hereafter, we call such a way an **application schedule**, which determines GPU allocation, parallelism strategy selection, and execution orchestration; we call the parallelism strategy for an LLM a **model execution plan**, which also specifies the number of GPUs required. Since the **processing rate** of a model does not necessarily scale linearly with its GPU count, different schedules lead to different makespans. Therefore, the problem is to find the minimum-makespan application schedule. This multi-LLM application scheduling (**MLAS**) problem is a variant of the well-studied job scheduling problem, where each LLM is a job, and is NP-hard.

1.1 Differences from Existing Problems

(1) Existing frameworks to run multiple models. LLMs differ from traditional DNNs in (1) higher GPU memory demand (may exceed a single GPU), (2) complex inference behavior due to uncertain

Proceedings of the VLDB Endowment, Vol. 19, No. 9 ISSN 2150-8097.
doi:10.14778/3819518.3819522

output lengths and different batching strategies like continuous batching [55], where the batch size and batched sequence lengths vary, unlike traditional DNNs that assume a fixed batch size and uniform batched sequence lengths. Thus, existing multi-DNN inference frameworks cannot be applied to MLAS. For multi-LLM frameworks, to the best of our knowledge, existing works [11, 30, 31, 35] focus on (1) model training whose running process is relatively simple and consistent (fixed batch size, known batch number, and hence known forward and backward pass count), or (2) request serving which minimizes request latency (e.g., average latency) under given arrival rates, assuming LLMs always reside on GPUs ([30, 31] do not even select model execution plans). Works on multi-LLM applications often focus on accuracy rather than scheduling efficiency (e.g., running LLMs sequentially or via online APIs).

(2) Existing job scheduling problems. MLAS differs from other job scheduling variants in its model **processing function**, which describes the processing rates of a model under different conditions. Specifically, in existing job scheduling problems, the processing rate (1) depends only on the job execution plan and hardware, unrelated to the job processing stage [32], (2) or depends on the processing stage, assuming a job has multiple phases with different parallelism degrees [12, 17, 41, 47]. In MLAS, the jobs cannot be simply divided into discrete phases; the processing rate depends on the remaining requests of an LLM, which is more complex. For example, on a GPU, an LLM may start with many unfinished requests in a batch but later have only one left due to long output length, underutilizing the GPU and lowering throughput (i.e., processing rate) far below its initial level.

Besides, traditional job scheduling has strict precedence constraints: a job cannot start until all its dependencies finish. In MLAS, to explore more parallelism, an LLM can run concurrently with its input models, which we call **model-level pipeline parallelism**, i.e., pipelining multiple models.

1.2 Prior Art Limitations

(1) Simple rule limitations. Commonly used rules include Max-heuristic and Min-heuristic [35]. Max-heuristic schedules the LLMs one by one, allocating all the GPUs to each LLM. Min-heuristic allocates the minimum number of GPUs to each LLM while making all the GPUs occupied to maximize the LLM concurrency. Min-heuristic has 2 variants: preemptive Min-heuristic can interrupt running LLMs and change their execution plans to avoid idle GPUs, while non-preemptive Min-heuristic cannot. To compare these rules, suppose we run an application to ensemble 6 LLMs on a 4-GPU machine. For simplicity, we only consider data parallelism for each LLM, which distributes requests over n replicated LLM instances, each running on one GPU. Assume the processing rate ρ_j of each LLM M_j under an execution plan P is fixed during inference, and ρ_1 of M_1 linearly scales with the allocated GPU number n, while other ρ_js are concave to n (i.e., the marginal improvement decreases with n). Figure 1 (a)-(c) show the respective schedules by Max-heuristic, non-preemptive Min-heuristic, and preemptive Min-heuristic. These rules all ignore the LLM processing function differences. In fact, if all the processing functions are linear or convex to n, Max-heuristic can generate the optimal schedule. However, since the ρ_js except ρ_1 are concave to n, there is another schedule in Figure 1 (d), where more GPUs are assigned to M_1 from time 2 to 3 because of its linear scalability, achieving shorter makespan than the above rules.

(2) Processing function modeling limitations. To get the processing functions, profiling-based methods are often used, e.g., Saturn [35], which focuses on model training, profiles the minibatch iteration time as training is iterative and consistent. However, this method cannot work directly on MLAS, because LLM output lengths are unknown in advance, making the batch information of each inference iteration unknown if we use continuous batching [55]. Without batch information, we cannot predict the iteration latencies and hence the processing rate. Therefore, to get the processing functions for MLAS, the key is to estimate the LLM output lengths. Existing works use a model, even an LLM, to predict the request output lengths [18, 23, 46, 58], but this introduces additional running time cost, which contradicts with the objective of MLAS.

(3) Scheduling algorithm limitations. Given our unique processing functions that depend on the remaining requests, many existing job scheduling methods cannot be adapted to MLAS. Specifically, the existing job scheduling methods often use linear programming (LP) [4, 32, 35] or various heuristics. However, although MLAS can be written in MILP in principle, considering preemption, the model-level pipeline parallelism, and our processing functions would make it extremely complex to solve. Heuristics like round robin and Equipartition [17, 41, 47] ignore the computation workload difference and the processing function difference of LLMs, so they may not work well on MLAS (round robin and Equipartition are dominated by the Max-heuristic and Min-heuristic we tested in Section 5). As a result, this work tries to adapt throughput-aware greedy heuristics to MLAS.

1.3 Our Proposed Approach

To solve MLAS, we propose a framework Mil, which consists of three major components.

(1) Processing rate estimation. Given the input requests and the execution plan, we propose a method to estimate the average processing rate of an LLM in any time interval. To do this, it is necessary to have the request output length information as aforementioned. However, since we aim to estimate the inference process for a model to complete a set of requests rather than predict the time to complete each individual request, we do not need to know the exact output length of each request for an LLM. Instead, it is sufficient for us to know how the output lengths of these requests are distributed as a whole. In fact, we observed that the output length of an LLM generally follows a distribution irrelevant to concrete requests, except that there are some special instructions in the request or in the inference settings (e.g., asking the LLM to generate the answer to a multiple-choice question without extra output or setting the maximum output length limit to 900). Therefore, we can obtain the empirical cumulative distribution function (eCDF) for an LLM in advance by collecting the output lengths of a large set of requests. Then, we can randomly sample the output lengths for the given requests accordingly. After that, given the request scheduling policy (e.g., first come first served), we can simulate the inference process of an LLM to obtain the running batch information for each generation iteration. The latency of each generation

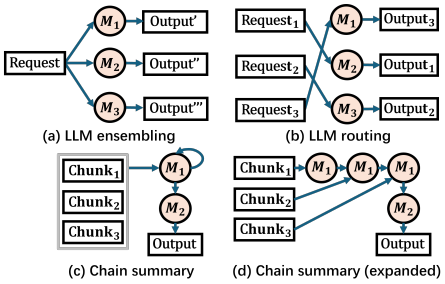


Figure 2: The computation graphs of different applications.

iteration is estimated using a set of linear functions. In this way, we can predict the running batches in any time interval and get the average processing rate for it.

(2) Greedy search. Given the processing functions, we propose a greedy method for MLAS. The main idea is to maximize the total processing rate of the running LLMs at any time point, so as to minimize the makespan. In addition, LLMs with long running time should be started early to avoid becoming the bottleneck due to late starts. Therefore, we use a candidate model set to ensure models with long running times to be started at the beginning. We iteratively allocate GPUs to the LLMs in the candidate set with an execution plan that brings the highest per-GPU increase to its throughput (i.e., its average processing rate) with respect to the shortest alive time interval of the running LLMs (possible preemption cost is considered), until there is no GPU left. We then update the remaining request information for all the LLMs at the time point where the first running LLM finishes, and reconduct the above GPU allocation process, until all models can be finished. This greedy method is used for proof of concept. It is possible to adapt more advanced greedy heuristics to MLAS.

(3) Runtime schedule adjustment. When running the multi-LLM application according to the schedule we find, the LLM finishing order may differ from what we predict due to estimation errors. Therefore, we design a rule-based mechanism to dynamically adjust the application schedule by bringing forward the running of the waiting models in the schedule, to avoid idle GPUs.

We evaluate Mil on various applications (ensembling, routing, chain summary, and mixed applications) against five competitors. Mil achieves up to 3.4× end-to-end speedup. We also study the performance of Mil's components separately.

2 PROBLEM FORMULATION

Precedence Constraint. The dependencies among the LLMs in an application can be expressed by a computation graph. Figure 2 shows the computation graphs for LLM ensembling, LLM routing, and chain summary (including document summarization and summary quality evaluation) respectively. The computation graph may contain **self-loops**. For example, in chain summary, a long document will be split into multiple chunks, and the summary will be updated as we step through the document chunk-by-chunk [3], as shown by Figure 2 (d) where the document contains 3 chunks and hence there are 3 summarization steps (3 M_1 nodes); while we can also use a self-loop of M_1 to express the summarization loop over the chunks, as shown by Figure 2 (c). In this paper, we keep the self-loops in an application without expanding it (so for chain summary, we adopt the computation graph of Figure 2 (c)), to avoid too many nodes in the graph. Each edge in the graph, except the self-loops, corresponds to a precedence constraint.

Model Execution Plan. Different parallelism strategies can be combined to be an execution plan. We currently consider two commonly used parallelism strategies for each LLM: data parallelism that distributes input requests over replicated model instances, and tensor parallelism that distributes the computation of individual model layers over multiple GPUs. Let dp denote the data parallelism degree, i.e., the number of model replicates, and tp denote the tensor parallelism degree, i.e., the number of intra-layer model partitions. An execution plan can be represented by $P = (\text{dp}, \text{tp})$. Given N GPUs, generally, $\text{dp} \in \{k | k \in \mathbb{N}^+, k \leq N\}$, $\text{tp} \in \{2^q | q \in \mathbb{N}, 2^q \leq N\}$ (tp is a power of 2 for better GPU efficiency). P is valid for an LLM if two constraints are satisfied: (1) the total number of GPUs required by both parallelisms ($\text{dp} \cdot \text{tp}$) does not exceed N; (2) the GPU memory given P is enough for the LLM weights and at least one sequence's KV cache (as this work currently does not consider offloading model weights or KV cache to the CPU side).

Processing Function. As introduced in Section 1, given a model M_j, an execution plan P, a time interval $[t, t + \Delta t]$, the unfinished requests with input lengths I_j^t at time t, the time points H_j^t at which the requests will be ready (as the requests may be output by some precedent LLMs), the request output lengths O_j^t are estimated based on the output length sampler, and the processing function $\rho_j(I_j^t, H_j^t, P, \Delta t)$ returns the average processing rate of M_j in the time interval $[t, t + \Delta t]$ based on the estimated O_j^t.

Definition 1 (MLAS problem). *Given the following input $(N, \mathcal{M}, I, \mathcal{P}, \rho, E)$:*

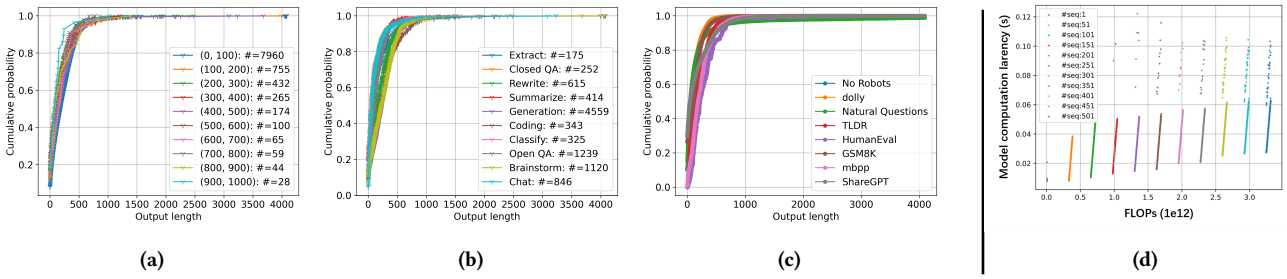
- *N, the total number of GPUs.*
- *$\mathcal{M}$, the set of LLMs.*
- *I maps each $M_j \in \mathcal{M}$ to I_j^0, its initial request input lengths.*
- *$\mathcal{P}$ maps each $M_j \in \mathcal{M}$ to $\mathcal{P}_j$, where $\mathcal{P}_j$ includes the valid execution plans of M_j and an empty plan Λ (i.e., allocating no GPU to M_j).*
- *ρ maps each $M_j \in \mathcal{M}$ to ρ_j, the processing function of M_j.*
- *E, the precedence constraints among $\mathcal{M}$.*

The output is a scheduling assignment function $\mathcal{A}(M_j, t)$ over LLM $M_j \in \mathcal{M}$ and time $t \geq 0$, which represents the execution plan selected from $\mathcal{P}_j$ for M_j at t, with preemption allowed. The schedule should satisfy the following constraints:

- *Capacity constraint: At any time $t \geq 0$, the total number of allocated GPUs should not exceed N.*
- *Demand constraint: For each $M_j \in \mathcal{M}$, all its requests can be completed in the end ($I_j^t = \emptyset$ at its completion time t).*
- *Precedence constraint: if $(M_i, M_j) \in E$, then $\mathcal{A}(M_j, t) \neq \Lambda$ if and only if M_i is completed ($I_i^t = \emptyset$) or $\mathcal{A}(M_i, t) \neq \Lambda$ (allowing model-level pipeline).*

The optimization objective is to minimize the makespan of the schedule, i.e., the largest completion time of models in $\mathcal{M}$.

Theorem 1. *The MLAS problem is NP-hard.*



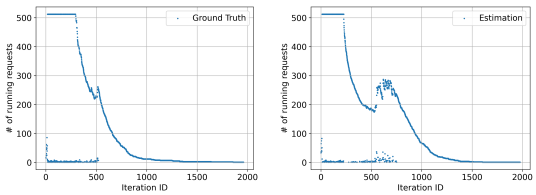
(a) (b) (c) (d)

Figure 3: (a)-(c): Output length eCDFs of vicuna-13b-v1.5 for (a) different input length regions of No Robots, (b) different request categories of No Robots, and (c) different datasets. (d): Model computation latency in a generation iteration for different batch sizes (# seq) VS FLOPs. "#=n" in the legend of (a)&(b) means the number of requests in the corresponding group is n.

Proof sketch. The minimum makespan scheduling problem for malleable jobs (i.e., jobs that can run in parallel on several processors (GPUs)) allowing preemption without precedence constraints is strongly NP-hard for an arbitrary number of processors [10], which is a special case of MLAS. Therefore, MLAS is NP-hard.

3 PROCESSING FUNCTION

Request output length sampler. The insight is that we can regard the output length of a request given an LLM as a random variable, and we find that its distribution is similar for different request input lengths and different request categories, except when the request includes instructions about the output length or when the output length limit is set in the inference settings (e.g., asking the LLM to generate the answer to a multiple-choice question without extra output or forcing the inference to stop after generating 900 tokens in the output). This insight is obtained from our experiment observation. Specifically, we use the No Robots dataset [40], which contains 10 different categories of requests, including Generation, Open QA, Brainstorm, Chat, Rewrite, Summarize, Coding, Classify, Closed QA, and Extract. We send the 10000 requests from the No Robots dataset to different LLMs and collect their output lengths, and then we draw the empirical cumulative distribution functions (eCDFs) of these output lengths. Figures 3a and 3b shows some eCDFs for an LLM, vicuna-13b-v1.5 [6], from which we can see that the eCDFs are similar regardless of the request length (Figure 3a) or the request content category (Figure 3b). We further validate the generalizability of the output length distributions by comparing the eCDFs of different datasets and testing more LLMs. Specifically, we test 3 LLMs (vicuna-13b-v1.5 [6], chatglm3-6b [15], WizardLM-13B-V1.2 [54]) on 8 datasets (sampling 10k requests from each): No Robots [40] and dolly-15k [9] (instruction following), Natural Questions [50] (QA), TLDR [51] (summarization), GSM8K [7] (math), HumanEval [5] and mbpp [2] (coding), ShareGPT [6] (chat). For each LLM, we computes its average output length distribution of the 8 datasets, $\bar{Q} = \sum_k Q^{(k)}/8$. We use the average Jensen–Shannon divergence $\sum_k \mathrm{JSD}(Q^{(k)} \parallel \bar{Q})/8$ to measure the distribution difference. JSD is bounded: $0 \le \mathrm{JSD} \le \ln 2$. The average JSDs for vicuna-13b-v1.5, chatglm3-6b, and WizardLM-13B-V1.2 are 0.11, 0.13, 0.13, indicating the distributions of different datasets are similar. Figure 3c plots the output length eCDFs of vicuna-13b-v1.5, which are close. All these results show that the output length distribution of an LLM does not depend on concrete requests.



(a) (b)

Figure 4: Running request number of each iteration in (a) the real inference process and (b) the simulated inference process, respectively.

Therefore, we construct eCDFs of output lengths for different LLMs using the No Robots dataset [40] in this work and estimate the request output lengths by sampling. Let $F_j(x)$ denote the output length eCDF of LLM M_j and l^j_{max} denote the maximum sequence length that M_j supports. Given an input request of length l_{in} for M_j, its output length is estimated by $l_{out} = \min(X, l^j_{max} - l_{in})$, $X \sim F_j(x)$. If we explicitly limit the maximum output length to y, then $l_{out} = \min(X, y, l^j_{max} - l_{in})$, $X \sim F_j(x)$.

Inference process simulator. Given the input lengths and the sampled output lengths of a set of requests, and the time points at which the requests are ready, as long as we know the request scheduling policy and the available KV cache blocks (using PagedAttention [6]) for an LLM (e.g., vLLM [25] uses the first come first served policy), we can simulate the LLM inference process without actually running the LLM on GPUs. Based on the simulation, we can estimate the running batch information for each generation iteration, i.e., the current total length (the initial input length plus the generated output length) of each request in the running batch. Figure 4 compares how the batch size changes across generation iterations in the real inference process and the simulated inference process for the LLM vicuna-13b-v1.5 [6], given 1000 requests from the ShareGPT dataset [6]. Figure 4 shows that the simulation captures the main curve pattern of the real inference process, despite some differences. For an application, we need to perform the simulation for the LLMs in it in the topological order.

Per-generation-iteration latency estimator. The insight is that a generation iteration mainly consists of a list of tensor operators on the GPUs, and the latency of an operator is related to the number of computation FLOPs and the GPU kernel parallelism. Specifically, we divide a generation iteration into three major parts: (1) model computation (computing the results of all model layers), (2) input preparation before model computation, and (3) output token sampling after model computation.

For part 1 (model computation), the number of computation FLOPs can be estimated by

$$\text{FLOPs}_{\text{prefill}} = L(c \cdot Bs + 2Bhs^2/\text{tp}) \quad (1)$$

$$\text{FLOPs}_{\text{decode}} = L(c \cdot B + 2hS/\text{tp}) \quad (2)$$

Equations (1) and (2) compute the FLOPs in a prefill iteration and the FLOPs in a decode iteration, respectively, where L is the number of layers, B is the batch size, s is the maximum request length of all running requests, h is the hidden dimension, tp is the tensor parallelism degree, S is the total request length, and c is a constant computed by summing up the sizes of all the model weight matrices that are related to matrix multiplication operations. The GPU kernel parallelism is related to the batch size B. We profile the part 1 latency (Figure 3d) in the inference process of Llama 7B [49] for a given set of requests, running on 1 GPU with vLLM [25], as an example. Figure 3d shows that, for each batch size B (#seq in the figure), this part of latency increases linearly with the number of FLOPs (there are some noise points sparsely distributed, which we can ignore). This conclusion also holds for other LLMs because they use the same GPU kernels.

For part 2 and 3, we use the padded total length of the running requests ($B \cdot s$) and the total length of the running requests without padding (S) to represent the number of computation FLOPs, respectively. We also profile these two parts of latency in the same inference process as for part 1. The results show that we can also use linear functions associated with different request numbers B to model these two parts of latency (the figures are not shown here due to the space limit).

Based on the above observations, we estimate the per-generation-iteration latency of an LLM M_j given an execution plan P by $t = t_{\text{comp}} + t_{\text{prep}} + t_{\text{samp}}$, where $t_{\text{comp}}, t_{\text{prep}}, t_{\text{samp}}$ are the respective time for model computation (comp), input preparation (prep) and output sampling (samp) and they are all in the form of linear functions:

$$a_{j,\text{phase}}[P, B] \cdot x_{j,\text{phase}} + b_{j,\text{phase}}[P, B] \quad (3)$$

where $a_{j,\text{phase}}[P, B], b_{j,\text{phase}}[P, B]$ denote constants specific to M_j, its execution plan P, the phase (comp, prep, samp) and the batch size B, and $x_{j,\text{phase}}$ is FLOPs for comp, $B \cdot s$ for prep, and S for samp. We profile the three parts of latency with different inference request batches to compute the constants in Equation (3).

Put them all together. For an LLM M_j, given an exection plan P and the unfinished requests at time t with input lengths I_j^t and ready time points H_j^t, we first sample the output lengths O_j^t for the requests using the output length sampler, then run the inference process simulator to estimate the running batch information for each generation iteration in the whole inference process, and finally predict the latency for each generation iteration using the per-generation-iteration latency estimator. Let $\text{FLOPs}_k, \text{t}_k$ denote the computed FLOPs and the latency of iteration k, respectively. Given any time point $t' = t + \Delta t$, we can get the maximum iteration involved in it, denoted by i, and then the average processing rate during $[t, t + \Delta t]$ is $\rho_j(I_j^t, H_j^t, P, \Delta t) = \sum_{k=1}^{i} \text{FLOPs}_k / \Delta t$.

Besides, every time M_j begins to run or changes its execution plan, we need to consider the model loading time (the time to load model weights from CPU to GPUs and prepare the communication environment if tensor parallelism is used), which can be profiled for each model and each execution plan in advance and stored in a cost table. In this case, the completed workload during $[t, t + \Delta t]$ and hence the average processing rate are also affected.

We test the whole estimation method by estimating the time for vicuna-13b-v1.5 [6] to complete the 1000 requests from ShareGPT [6], whose simulated inference process is shown in Figure 4. The estimated running time is 98 seconds, while the real one we measured is 92 seconds, with an error ratio of 6.5%. Considering the model loading time, our estimation is 128 seconds, while the real running time is 127 seconds, which is almost the same as the estimation.

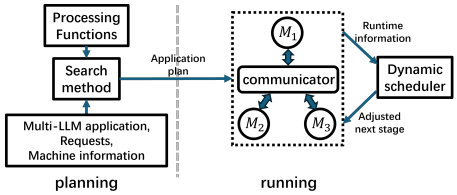


Figure 5: The framework overview.

4 THE MIL FRAMEWORK

Figure 5 shows our framework overview. Given a multi-LLM application, its input requests, and machine information, we first search for the minimum-makespan schedule for the application using a greedy method, based on the processing functions. Then, we run the application according to the schedule. We launch one process for each LLM to do the inference. The dotted box area in Figure 5 illustrates how the LLM processes are organized in our framework. As there may be data dependency among LLMs, we use a separate process as the communicator among the LLMs to receive the LLM outputs, process them (e.g., filling the outputs into templates), and send them to the corresponding LLM processes upon request. When an LLM is interrupted, we will record the generated tokens for each request and discard the KV cache. We also provide a rule-based mechanism to dynamically adjust the schedule according to the runtime information to reduce GPU idleness.

4.1 Search method

Solving the MLAS problem is NP-hard, therefore we propose a greedy method here for it. Algorithm 1 shows the pseudocode. The main idea is maximizing the total processing rate at any time point and scheduling the LLMs with long running time early.

Therefore, at time t (t starts from 0), we maintain a candidate model set J_{tot} and we consider allocating more GPUs to the models in it. The models are added to the candidate set in the order of their running time given their unfinished requests and their minimum-GPU execution plans, with the precedence constraints are satisfied

Algorithm 1: Greedy Search

Data: The MLAS problem input instance $(N, \mathcal{M}, I, \mathcal{P}, \rho, E)$
Result: A schedule $\mathcal{A}$

1 $\mathcal{A}(j, t') = \Lambda, \forall M_j \in \mathcal{M}, t' \geq 0$; $t \leftarrow 0$; *// current time point*
2 $\alpha_j \leftarrow +\infty, \forall M_j \in \mathcal{M}$ *// the check order of* $M_j \in \mathcal{M}$
3 **while** $\mathcal{M}$ *has unfinished LLMs* **do**
4 $S^* \leftarrow \{\}, \phi^* \leftarrow$ None; *// S^*: best current LLM execution plan group*
5 $J_{\text{tot}} \leftarrow []$;
6 **while** *True* **do**
7 $J_{\text{cand}} \leftarrow \{j | M_j$ is ready w.r.t. $E, j \notin J_{\text{tot}}\}$;
8 $t_j \leftarrow$ the min-GPU completion time of M_j $(j \in J_{\text{cand}}, \alpha_j = +\infty)$ w.r.t. S^*, ρ_j;
9 sort J_{cand} by $(\alpha_j, -t_j)$ in ascending order;
10 no_cand $\leftarrow (|J_{\text{cand}}| == 0)$;
11 **while** $|J_{\text{cand}}| > 0$ *or* no_cand **do**
12 move $J_{\text{cand}}[0]$ to J_{tot} if no_cand == False;
13 $\Phi \leftarrow [], S_0^* = S^*$;
14 **for** $j \in J_{\text{tot}}$ **do**
15 $\Phi = \Phi + \{(j, S) | P \in \mathcal{P}_j, S = S^*.\text{update}((j, P)), S^*.\#\text{gpu} < S.\#\text{gpu} \leq N\}$;
16 **end**
17 $\Delta n_\phi = (\phi.S.\#\text{gpu} - S^*.\#\text{gpu}), \phi \in \Phi$;
18 $\Delta t^* = \min_{s \in S^*}(s.\text{completion\_time})$;
19 $\theta_j^* = \rho_j$ value w.r.t. $S^*, [t, t + \Delta t^*]$;
20 $\Delta t_\phi = \min_{s \in \phi.S}(s.\text{completion\_time}), \phi \in \Phi$;
21 $\theta_\phi = \rho_{\phi.j}$ value w.r.t. $\phi.S, [t, t + \Delta t_{\phi.j}], \phi \in \Phi$;
22 **if** $\Phi == \emptyset || \max_{\phi \in \Phi}(\theta_\phi - \theta^*_{\phi.j}) < 0$ **then**
23 stop = True; break;
24 **end**
25 $\phi^* = \arg\max_{\phi \in \Phi}(\theta_\phi - \theta^*_{\phi.j})/\Delta n_\phi, S^* = \phi^*.S$;
26 move $J_{\text{tot}}[-1]$ back to J_{cand} if $|S_0^*| == |S^*|$ and no_cand == False;
27 **end**
28 break if stop == True;
29 **end**
30 $\mathcal{A}(s.j, t') = s.P, t' \in [t, \Delta t_{\phi^*}]$ for $s \in S^*$,;
31 $t = t + \Delta t_{\phi^*}$;
32 update I_j^t for $M_j \in \mathcal{M}$;
33 **end**

(line 7-9, 12). We iteratively allocate GPUs to the LLM in J_{tot} and its execution plan with the highest per-GPU increase of its throughput (i.e., its average processing rate) with respect to the shortest alive time interval of the running LLMs (possible preemption cost is considered), assuming no running model will be interrupted (line 13-25), the selection is stored in ϕ^*. The alive time interval of any M_j given execution plan P is the shortest time to complete its remaining FLOPs, w, at t, i.e., $\arg\min_{\Delta t} \rho_j(I_j^t, H_j^t, P, \Delta t) \cdot \Delta t = w$. Only when all the models in J_{tot} have been allocated at least one GPU can we add new models into J_{tot} (line 26), so the long-running-time models will be started at the beginning. When all the GPUs are allocated or when we cannot improve the throughput of any running model by allocating more GPUs to it (line 22), suppose the shortest completion time of the running models is Δt_{ϕ^*}, then we update the application schedule $\mathcal{A}$ for the running models and the time interval $[t, t + \Delta t_{\phi^*}]$ to the corresponding selected execution plans in ϕ^* (line 30). The time point t is also updated to $t + \Delta t_{\phi^*}$ (line 31). The remaining request information at the new time point t will be updated for all the models (line 32), and we will reconsider the application schedule at t, during which the order of the models being added to the candidate set will not be changed (line 2,9). This process is repeated until all the models can be finished.

Time complexity. Algorithm 1 can be computed in $O(C|\mathcal{M}|N^2 \min(|\mathcal{M}|, N))$ time, where C is the time of one estimation process of the processing rate estimation, which is related to the inference process complexity (e.g., the request number, request lengths, the GPU memory size, etc.). To make the search faster, our inference process simulator runs for different execution plans in parallel.

Approximation analysis. Algorithm 1 has an approximation guarantee for the simplified problem instance: **(1)** we assume that given a model M_j and its execution plan P, the processing rate can change over the execution process but only depends on the number of remaining requests to answer, **(2)** given the same remaining request number W_j, the processing function of M_j regarding the GPU number n is concave (for each GPU number n, we only consider the best processing rate of M_j of all its possible execution plans).

Theorem 2. *Let $t_{\mathcal{A}}$ be the makespan of our schedule $\mathcal{A}$, $t_{\mathcal{A}^*}$ be the makespan of the optimal schedule $\mathcal{A}^*$, N be the total GPU number, C_j be the maximum preemption cost of M_j (of all the possible execution plans), and $|\mathcal{M}|$ be the number of models. Let α satisfy that, the processing rate of M_j when using all the GPUs, denoted by $\rho_{\text{all}j}$, is at least α ($\alpha \geq 1$) times higher than the processing rate of M_j when using the minimum possible number of GPUs, denoted by $\rho_{\text{min}j}$, i.e., $\rho_{\text{all}j} \geq \alpha\rho_{\text{min}j}$. Let β satisfying that $\max_j C_j \leq \beta t_{\text{min}j}, \forall j$, where $t_{\text{min}j}$ is the time to finish M_j with the minimum possible number of GPUs. We have $t_{\mathcal{A}} \leq Nt_{\mathcal{A}^*}/\alpha + |\mathcal{M}| \max_j C_j \leq (\frac{1}{\alpha} + \beta)Nt_{\mathcal{A}^*}$.*

Proof. Given a schedule $\mathcal{A}'$, $\text{AREA}(\mathcal{A}')$ denotes the total non-idle time of all the GPUs. We have $\text{AREA}(\mathcal{A}^*) \geq \sum_j t_{\text{min}j}$, and hence $t_{\mathcal{A}^*} \geq \text{AREA}(\mathcal{A}^*)/N \geq \sum_j t_{\text{min}j}/N$.

In our schedule $\mathcal{A}$, there may be some intervals where some GPUs are idle due to the preemption cost. Let Δ be such an interval and M_j be the model that finishes at the interval end. Consider the execution plan P' to assign all the idle GPUs to M_j in this interval. Let t_1 be the interval length, and t_2 be the time to complete M_j since the interval beginning under P'. We know $t_2 \geq t_1$, otherwise it means we can improve the throughput of M_j by assigning more GPUs to it, which contradicts the schedule $\mathcal{A}$. Therefore, if we select P' for M_j, let n be the original number of idle GPUs, and we have $\text{AREA}(\Delta) \leq Nt_1 \leq (N - n)t_1 + nt_2 = \text{AREA}(\Delta')$, where Δ' is the new interval where M_j runs for t_2 to complete and the other models still run only for t_1.

By doing such modifications to all the intervals, denoted by $\{\Delta_i\}$, we can obtain a new set of intervals $\{\Delta_i'\}$ ($\Delta_i = \Delta_i'$ if Δ has no idle GPUs). There are at most $|\mathcal{M}|$ intervals in $\mathcal{A}$, and each interval has at most $\max_j C_j$ time for preemption cost. We have $\text{AREA}(\mathcal{A}) = \sum_i \text{AREA}(\Delta_i) \leq Nt_{\mathcal{A}} \leq \sum_i \text{AREA}(\Delta_i') \leq \text{AREA}(\mathcal{A}_{\text{Max}}) + N(|\mathcal{M}| \max_j C_j - \sum_j C_j) = Nt_{\mathcal{A}_{\text{Max}}} + N(|\mathcal{M}| \max_j C_j - \sum_j C_j)$.

Therefore, $t_{\mathcal{A}} \leq t_{\mathcal{A}_{\text{Max}}} + |\mathcal{M}| \max_j C_j - \sum_j C_j$. Since $\rho_{\text{all}j} \geq \alpha \rho_{\text{min}j}$, we have $t_{\mathcal{A}_{\text{Max}}} - \sum_j C_j \leq \sum_j t_{\text{min}j}/\alpha \leq N t_{\mathcal{A}^*}/\alpha$. Therefore, we have $t_{\mathcal{A}} \leq N t_{\mathcal{A}^*}/\alpha + |\mathcal{M}| \max_j C_j$. Further, we have $|\mathcal{M}| \max_j C_j = \sum_j \max_j C_j \leq \beta \sum_j t_{\text{min}j} \leq \beta N t_{\mathcal{A}^*}$. Therefore, $t_{\mathcal{A}} \leq (\frac{1}{\alpha} + \beta) N t_{\mathcal{A}^*}$.

□

In practice, when the number of requests is large enough, (1) data parallelism can almost linearly increase the processing rate with the assigned GPUs, making α close to N, and we can roughly regard the processing rate under an execution plan unchanged during the model inference because the GPUs are saturated most of the time; (2) the inference time will be long enough to make β close to 0 as C_j is a constant on the order of tens of seconds (e.g., 30s for chatglm3-6b [15], 60s for Llama-2-70b-chat-hf [49]). In this case, the above bound will be tight. Our experiments show that α varies with the model workloads, e.g., for alpaca-13b, as the number of requests increases from 1000 to 10000, α increases from 2.4 to 6.4, so N/α decreases from 3.3 to 1.25 ($N = 8$). Otherwise, the bound can be loose, and Mil may fail.

4.2 Runtime adjustment mechanism

As our processing functions are not perfectly accurate, the LLM finishing order may differ from what we predict. For example, suppose the application schedule we find is $\mathcal{A}$ and the makespan of it is t. According to Algorithm 1, $[0, t]$ is split into multiple time intervals: $[t_0, t_1]$, $[t_1, t_2]$, ..., $[t_{k-1}, t_k]$, where $t_0 = 0, t_k = t$, and each t_i ($i = 0, ..., k$) is the time point where some LLM is started, changes its execution plan or finishes. Suppose for $[t_0, t_1]$, the running LLMs and their execution plans are $[(M_1, P_1), (M_2, P_2), (M_3, P_3)]$, and according to $\mathcal{A}$, M_1 will finish at t_1, i.e., M_1 finishes earlier than M_2, M_3. However, in the real running process, M_2 may finish earlier than M_1. In this case, if we do not adjust the schedule and wait until M_1 finishes to run the models with the execution plans specified by $\mathcal{A}$ for $[t_1, t_2]$, the GPU resources will be severely wasted.

Schedule adjustment. The principle is to bring forward the waiting execution plans of models in $\mathcal{A}$ if there are newly released GPUs. For the above example, the adjustment mechanism works as follows. For each unfinished model M_j with execution plan P_j in $[t_0, t_1]$ (i.e., (M_1, P_1) or $(M_3, P3)$). If $[t_0, t_1]$ is the last interval that M_j is involved in, then we will keep M_j running with P_j until it finishes. Otherwise, if $\mathcal{A}(j, t') = P_j$ for $t' \in [t_1, t_2]$, then we will keep M_j running, which is equivalent to scheduling M_j according to the $[t_1, t_2]$ part of $\mathcal{A}$. If $\mathcal{A}(j, t') \neq P_j$ for $t' \in [t_1, t_2]$, then we will first consider running the models M_q with $\mathcal{A}(q, t') = P_q \neq \Lambda$ ($t' \in [t_1, t_2]$) on the GPUs with P_q. After this, the previously unfinished M_j will finally be considered for running with P_j if there are still available GPUs. If all GPUs are occupied, we will no longer consider scheduling this (M_j, P_j) pair from the $[t_0, t_1]$ part of $\mathcal{A}$. When the (M, P) pairs from the $[t_i, t_{i+1}]$ part of $\mathcal{A}$ have all been scheduled or the related models have finished, we can consider the next part $[t_{i+1}, t_{i+2}]$ of $\mathcal{A}$ for the possible adjustment later. The above description has covered all the possible cases at runtime.

GPU selection. When selecting GPUs for an LLM, we follow the principle of minimizing LLM reloading costs with all the NVlink connection requirements of the execution plans satisfied (tensor parallelism should be run on GPUs fully connected with NVlinks to ensure efficiency). For example, if there is a newly started model M_1 with tensor parallelism degree 2, and the NVlinks in a 4-GPU machine only connect (GPU 0, GPU 1) and (GPU 2, GPU 3), then M_1 can only be loaded on GPU 0-1 or GPU 2-3. In this case, we may need to move some models if they occupy the GPUs required by M_1, and we want to choose the reloading solution with the minimum reloading cost. To do this, we heuristically sort the GPUs by their possible reload costs if the running models on them need to be relocated, and assign the GPUs to the models in order.

5 EXPERIMENTS

Machine. We use vLLM [25] as the engine to run an LLM. The experiments are conducted on a ubuntu machine with 8 A100-80G NVIDIA GPUs (where every 2 GPUs are grouped and connected by NVLink), 2 24-core CPU, and 1510 GB RAM.

Competitors. All the methods use our processing functions (except LightRule), allow preemption (except HybridFlow), and are supported by our runtime adjustment mechanism.

(1) **Max-heuristic (MAX)** [35]: it schedules LLMs on all the GPUs sequentially, maximizing each LLM's processing rate.

(2) **Min-heuristic (MIN)**: adapted from [35], it allocates all the GPUs to as many LLMs as possible every time there are idle GPUs; the GPUs are partitioned as evenly as possible among the running LLMs. When selecting execution plans, MIN maximizes the total processing rate of the running LLMs.

(3) **MIN-R**: A variant of MIN, which (1) randomly samples up to N (the GPU number) ready LLMs to consider concurrent running every time and (2) only considers minimum-GPU execution plans for LLMs when the unfinished LLMs are not fewer than the GPUs.

(4) **Saturn***: adapted from Saturn [35], Saturn* formulates the scheduling as an MILP (to be solved by Gurobi [16]), assuming there is no preemption; when there is a model expected to finish, Saturn* will solve a new MILP according to the new model states. To consider model-level pipeline parallelism, due to the complexity of modeling this in linear constraints (Section 1.2), we (a) estimate the running time of an LLM assuming its input models (if any) are finished before it starts and (b) add the constraints that a model M_1 cannot start before its input model M_2 starts, and M_1 will not release its GPUs before M_2 finishes.

(5) **LightRule**: The running time of an LLM is estimated by the request number multiplied by the average output length and divided by its average throughput (generated # of tokens/s). For each model and its execution plan, we profile on the No Robots dataset [40] and use the average throughput of the maximum batch size during the inference as the average throughput (when GPUs are likely to be saturated). We estimate the average output length by the minimum between the length expectation based on the output length eCDF (Section 3) and the maximum output length limit (if any). When the remaining models are more than the GPUs, LightRule selects the minimum-GPU execution plan for each LLM, sorts the models by their topological order and their estimated running time (assuming its input models are finished), and schedules the models using list scheduling; otherwise, LightRule first selects the minimum-GPU execution plan for each model and then greedily allots GPUs to the LLM with the maximum makespan decrease per increased GPU.

Table 1: LLM selection frequency

ID	Model	# Request	Ratio
0	Llama-2-70b-chat-hf	408	0.06
1	Mixtral-8x7B-Instruct-v0.1	1267	0.18
2	WizardLM-13B-V1.2	2068	0.30
3	CodeLlama-34b-Instruct-hf	456	0.07
4	Mistral-7B-Instruct-v0.2	2657	0.39
	Total:	6856	1.00

When a model is expected to finish, the remaining models are rescheduled using the above rule.

(6) **LightRule***: a variant of LightRule which replaces the cost model of LightRule with our processing functions.

(7) **HybridFlow** [44]: designed for Reinforcement Learning from Human Feedback (RLHF), with all LLMs held on GPUs together, it partitions the LLMs and GPUs into sets (a GPU set is assigned to an LLM set) and finds the best schedule via enumeration.

(8) **HybridFlow***: it runs the same search algorithm as HybridFlow but does not consider the total memory constraint for colocated models, because it loads LLMs on GPUs one by one and discards them when they finish.

Applications. We test 4 types of applications.

(1) **LLM ensembling** [21] has no model dependency, and each model has the same number of requests. **Models** are chosen from LLM-Blender [21]: vicuna-13b-v1.5 [6], oasst-sft-4-pythia-12b-epoch-3.5 [26], alpaca-13b [48], baize-v2-13b [53], koala-13B-HF [14], dolly-v2-12b [8], mpt-7b-chat [33], chatglm3-6b [15], stablelm-tuned-alpha-7b [45]. **Dataset**: the MixInstruct dataset [21]. We set the maximum request output length limit to 256 as done in LLM-Blender [21]. We change the inference workload by varying the number of requests from 1000 to 10000. The request input length in this dataset ranges from 5 to 127, with an average of 21.

(2) **LLM routing** [19] also has no LLM dependency. Each request is assigned to one LLM, so different LLMs have different numbers of requests. **Models** are 5 open-source LLMs from ROUTERBENCH [19]: Llama-2-70b-chat-hf [49], Mixtral-8x7B-Instruct-v0.1 [1], WizardLM-13B-V1.2 [54], CodeLlama-34b-Instruct-hf [42], Mistral-7B-Instruct-v0.2 [22]. **Dataset**: the ROUTERBENCH dataset [19]. Table 1 shows the corresponding request numbers. The dataset provides the response to each request by each LLM, so we can obtain the exact output lengths.Therefore, we also do an experiment with known output lengths (w/ outlen), and make LLMs generate the exact number of output tokens for the requests as in the dataset. The input length varies from 9 to 577 and the average is 310; the output length varies from 3 to 1585 and the average is 199. In the experiment with unknown output lengths (w/o outlen), as ROUTERBENCH does not mention the maximum output length limit when they obtain the responses, we set the limit to a relatively large value 4096. We change the number of requests by replicating the dataset for different number of times, to vary the LLM workloads.

(3) **Chain summary** [3] has model dependencies (Figure 2 (c)): the evaluator LLM (M_2) depends on the summary LLM (M_1) and the M_1 has a self-loop. Besides, given a set of documents with different lengths, from the expanded computation graph Figure 2 (d), we can see the workloads of the summary steps (the M_1 nodes) will decrease from left to right. These factors can lead to GPU idleness. **Models**: vicuna-13b-v1.5 as the summary model and Llama-2-70b-chat-hf as the evaluator model. **Dataset**: we sample documents from two datasets to summarize: BOOOOKSCORE [3] and BookSum [24]. Each document is split into chunks of size 2048 using the code in BOOOOKSCORE [3]. The document length distribution is very skewed: when the sample size is 100, there is one extremely long document with 60 chunks, while the median length is 3 chunks. A summary may need to be judged from multiple aspects [20]. We have 3 dataset settings: (1) varying the document number from 100 to 500, with the maximum output length limit of 900 and the number of evaluation times of 4; (2) varying the number of evaluation times from 2 to 8, with the document number of 300 and the maximum output length limit of 900; (3) varying the maximum output length limit from 100 to 900, with the document number of 400 and the number of evaluation times of 4.

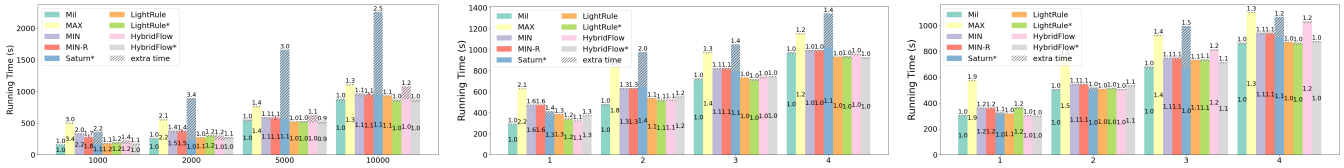
(4) **Mixed application**. To explore whether there are additional time-saving opportunities when we run multiple application together, we also test mixed applications. **Models**: a mixture of chain summary and LLM ensembling. **Dataset**: we set the number of evaluation times in chain summary to 4, the maximum output length limit of chain summary and LLM ensembling to 900 and 256, respectively; we fix the number of LLM ensembling requests to 5000 and vary the number of documents to summarize from 100 to 500, to simulate different workload combinations. Besides, we also use one of the above settings with 200 documents as a base setting, and replicate the base-setting mixed applications twice and four times (getting 4 and 8 apps) for the scalability test.

Metric. For each method, its *end-to-end running* time consists of 2 parts: (1) *extra time*, i.e., the time to generate the schedule, and (2) *inference time*, i.e., the time to run the application.

5.1 End-to-end results

Figure 6 shows the time for each method to run single applications given different dataset settings, Figure 7 shows the results on the mixed application, and Figure 8 shows the scalability test on the mixed application. In Figures 6 to 8, each bar contains two parts, the inference time (the lower part) and the extra time (the upper part with slashes). There are two values for each bar: the *normalized inference time* against Mil (in the middle) and the *normalized end-to-end running time* against Mil (on the top). For MIN-R, when the # of models is not fewer than N, the bars report the average inference time and the average end-to-end time of 3 runs on each test case, and the error bar denotes its end-to-end running time range.

5.1.1 Speedups. According to Figures 6 and 7, in terms of the end-to-end running time, Mil is 1.0-3.0× better than MAX, 1.0-2.0× better than MIN, 1.0-1.7× better than MIN-R, 1.0-3.4× better than Saturn*, 1.0-1.7× better than LightRule, 1.0-1.2× better than LightRule*, 1.0-1.4× better than HybridFlow, and 1.0-1.3× better than HybridFlow* except one ensembling case (10% worse); in terms of the LLM inference time itself, we are 1.0-3.4× better than MAX, 1.0-2.2× better than MIN, 1.0-1.8× better than MIN-R, 1.0-1.4× better than Saturn*, 1.0-1.8× better than LightRule, 1.0-1.2× better than LightRule*, 1.0-1.3× better than HybridFlow, and 1.0-1.3× better than HybridFlow* except on some ensembling and mixed applications where we are 10% worse. Comparing Figure 7 against Figures 6a and 6d, we find that for Mil, running the two



(a) Ensembling: vary # requests.

(b) Routing: w/o outlen; vary # replications

(c) Routing: w/ outlen; vary # replications

(d) Chain summary: vary document number.

(e) Chain summary: vary evaluation times.

(f) Chain summary: vary max output length.

Figure 6: Single application end-to-end results

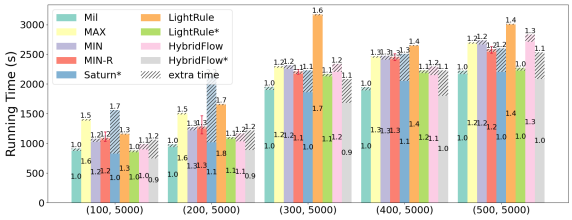


Figure 7: Running time VS request number combination of chain summary and LLM ensembling.

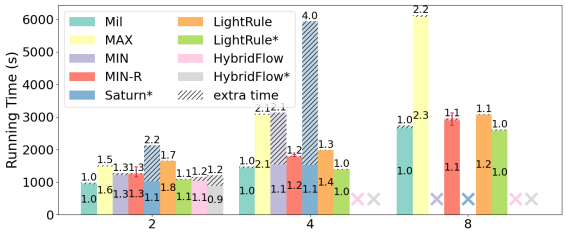


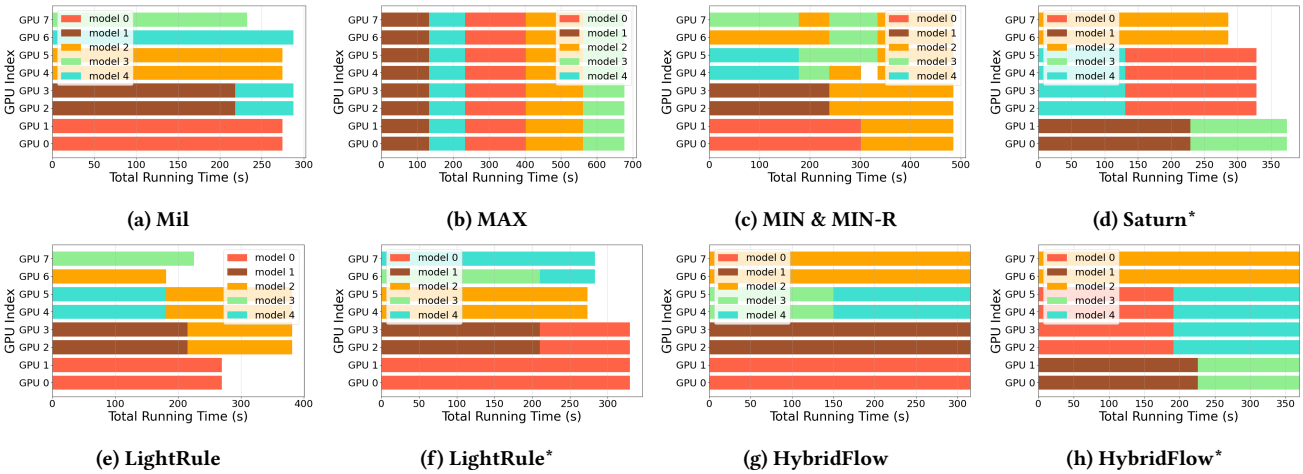
Figure 8: Running time VS application number.

applications separately costs 1.0-1.3× longer end-to-end running time than scheduling them as a whole, because the models in the LLM ensembling can make the idle GPUs busy when only a few long documents are not finished in the chain summary. In terms of inference time, running the mixed application as a whole is not slower than separate scheduling for the competitors as well, except for LightRule (due to its ineffective cost model), LightRule* (when with 300 documents), and HybridFlow (due to model colocation).

In the scalability test (Figure 8), the end-to-end running time of Mil scales sublinearly with the number of applications, and we remain 1.0-4.0× better than the competitors except HybridFlow* (in the case of 2 applications). Some competitors fail to generate schedules for some tests due to (1) long search time (MIN, HybridFlow*), (2) failure to solve the MILP (Saturn*), or (3) infeasibility of holding all the models on the GPUs together (HybridFlow). The successful cases also show sublinear inference time increase of the competitors (except for MAX), because in these cases, with more models to schedule, GPU idleness and underutilization are relatively alleviated. We compare Mil with the competitors in detail below.

Comparison with MAX: Mil try to assign GPUs to LLMs to improve the overall throughput instead of each single model's throughput. For MAX, since using more GPUs will not bring a linear throughput improvement when the number of requests is not large enough, it may cause severe computation resource waste. For instance, when # Requests is 1000 in LLM ensembling, it takes 122s to run alpaca-13b on 1 GPU and 84s on 8 GPUs, i.e., 8× more GPUs only make the throughput 1.5× higher; while when # Requests is increased to 10000, it takes 721s to run alpaca-13b on 1 GPU and 160s on 8 GPUs, i.e., 4.5× throughput with 8× GPUs. When the model loading time is excluded, 8× GPUs leads to 2.4× throughput for 1000 requests while 6.4× throughput for 10000 requests. Figure 9 also validates this point. Comparing model 1 (Mixtral-8x7B-Instruct-v0.1) in Figure 9a and Figure 9b shows that 4× GPUs do not lead to 4× throughput. Besides, the model loading overhead (ranging from 11s to 70s for a model) of MAX's sequential scheduling may account for a large proportion of the end-to-end running time when the model workloads are not large enough.

As the number of requests increases, the advantage of Mil against MAX becomes smaller (e.g., Figures 6a to 6c and 7). One reason is that, when the number of requests is large enough, under different execution plans, the number of running requests per GPU for an LLM in most of its inference time will be at a similar high level, so the computation throughput per GPU will be similar. Another factor is that the ratio of the time to load models onto GPUs in the overall inference process is smaller when the number of requests is larger. Therefore, as long as we make each GPU have LLMs running on it all the time, we can achieve a good overall throughput.



(a) Mil (b) MAX (c) MIN & MIN-R (d) Saturn*

(e) LightRule (f) LightRule* (g) HybridFlow (h) HybridFlow*

Figure 9: Application schedules for Routing (output lengths unknown, # data replication is 1). LLM IDs are in Table 1.

Comparison with MIN & MIN-R: MIN and MIN-R blindly split all the GPUs evenly among the remaining LLMs all the time, ignoring the model workloads and the model reloading overhead, which damages the application efficiency and makes them perform worse than Mil. In Figure 9, by comparing model 3 (CodeLlama-34b-Instruct-hf) in Figure 9a and Figure 9c, we find that unnecessary execution plan changes waste running time. When the number of requests is large enough, we also observe that the advantage of Mil against MIN (MIN-R) generally becomes smaller (e.g., Figures 6a to 6c), and the reasons are the same as the analysis for MAX. Sometimes MIN-R may not sample the long-running-time models at the beginning, making the inference time longer than that of MIN, e.g., Figure 8, while MIN-R may also beat MIN, e.g., Figure 6a.

Comparison with Saturn*: Although Saturn* uses an industrial-strength MILP solver, Gurobi [16], to solve its MILP problem each time, due to the complexity of the MILP, Gurobi may not find the optimal solution. The simplification we do to the MILP modeling to make Saturn* support model-level pipeline parallelism may also limit its performance. Furthermore, the MILP assumes models cannot be interrupted once started, which is different from the original MLASproblem, so the schedule Saturn* finds may be slower than ours in some cases. Take Figure 9d as an example. Not considering model preemption, Gurobi finds the optimal solution for the initial MILP to minimize the makespan, which allots 4 GPUs to model 4 (Mistral-7B-Instruct-v0.2); then for the later MILPs when there are models expected to finish, considering the extra costs to reschedule models and the remaining workloads, the optimal solutions found by Saturn* are to keep the model execution plans unchanged. However, with model preemption, Figure 9a shows that, we can first allot 1 GPU to model 4 to achieve higher overall throughput (as model throughput does not scale linearly with the GPU number) and then allot more GPUs to it when some models finish, resulting in much shorter makespan.

Comparison with LightRule & LightRule*: While LightRule has a competitive performance on some test cases, its simple cost model can lead to bad schedules, because the cost model uses (1) the average token generation speed at a fixed batch size as the model throughput, ignoring the influence of varying computation workloads, and (2) the number of tokens to generate as the computation workload, ignoring the influence of varying request lengths. In Figure 9e, the inaccurate cost model causes high model reloading overhead (model 2 is rescheduled twice) and GPU idleness. For more data replications, the schedule of LightRule is the same, but the model reloading overhead ratio is smaller, so our advantage over LightRule is smaller. On chain summary, LightRule performs much worse because its cost model ignores the request dependency (1) due to the self-loop (Figure 2) by simply summarizing all the tokens to generate in the loop to predict the running time, and (2) due to model dependency when there is model-level pipeline parallelism. The assumption that model throughput is constant during execution is wrong, especially when there is model dependency. Therefore, LightRule allots only 2 GPUs for summarization but 6 GPUs for evaluation, which causes severe GPU idleness. For the mixed application (Figure 7), scheduling it as a whole is even slower than the independent scheduling when document number is 200 or 300, because LightRule only allots 1 GPU for summarization (the bottleneck) and 4 GPUs for evaluation, making the bottleneck model extremely slow and the other GPUs idle for a long time.

Equipped with our processing functions, LightRule* overall performs much better than LightRule, although it sometimes may make wrong decisions, e.g., increasing the GPUs for model 0 in Figure 9f, as our processing functions are not perfectly correct.

Comparison with HybridFlow & HybridFlow*: The auto device mapping algorithm enumerates all the possible device mapping strategies, making HybridFlow perform well in many test cases. However, not considering model preemption can make it miss some optimization opportunities, e.g., Figure 9g (compared with Figure 9a). On the other hand, the GPU sharing of multiple models in HybridFlow may damage the model efficiency when the memory resource is insufficient, e.g., when the GPU memory is small, the model sizes are large, or the request number (i.e., the KV cache demand) is large. This explains why, with more than 200 documents in Figure 7, the inference time of HybridFlow is longer than scheduling the 2 applications independently. When there are

significantly more models than the GPUs, the GPUs cannot hold all the models together (Figure 8).

HybridFlow* loads models only when they need to run, avoiding the efficiency damage due to model colocation, so in terms of inference time, it beats HybridFlow in many cases (e.g., Figures 6a and 7), and even beats Mil in some ensembling and mixed applications by 10% (because of its enumeration-based search). However, without the total memory constraint of colocated models, there are more possible device mappings and execution plan combinations for HybridFlow* to enumerate, increasing its extra time (Figure 7). It even fails to generate schedules within hours for some mixed applications (the cases with more than 2 applications in Figure 8).

5.1.2 Search efficiency. In the experiments except the scalability test, the respective maximum search time of Mil, MAX, MIN, MIN-R, Saturn*, LightRule, LightRule*, HybridFlow and HybridFLow* is 60s, 29s, 71s, 49s, 1308s, 20s, 48s, 205s, and 451s. In the scalability test, the respective maximum search time of Mil, MAX, MIN-R, LightRule, and LightRule* remains within 100s, and the corresponding ratios against the end-to-end running time are 3.2%, 1.1%, 2.4%, 0.6%, 1.6%. Therefore, these methods are efficient. For Mil, the search time consists of 2 parts: initialization and running time estimation. The initialization time is less than 20s and is the same for all the methods, depending on the model number; while computing one estimation is quite efficient due to the parallel simulation. For example, for the mixed application with workload (500, 5000), the search time is 60s, where initialization takes 18s and estimation takes 39s (computing 550 estimations, so each taking 0.07s).

Due to the complexity of solving the MILP, Saturn* may take thousands of seconds to search (the timeout for each MILP is 5min as in [35]), which hinders the use of Saturn* for MLAS when there are multiple models. HybridFlow is time-consuming because it is enumeration-based. HybridFlow searches for 205s on LLM ensembling with 10000 requests, enumerating 694938 device mappings and estimating model running time for 2526 times (169s); by contrast, Mil only checks 55 execution plan groups and estimates model running time for 462 times (30s). HybridFlow* enumerates more device mappings than HybridFlow, so its extra time can be longer, e.g., on mixed applications. However, as it does not need to consider the different available GPU memory amounts for a model in different colocated sets, it conducts fewer model running time estimations, so its extra time is shorter than that of HybridFlow on LLM ensembling, e.g., 1187524 device mappings and 108 estimations (10s) when with 10000 requests. Although MIN uses a simple rule for scheduling, its search time can be long because it checks all the possible concurrent LLM execution plan groups each time to select the highest-throughput one. For the mixed application (2 apps) with workload (500, 5000), MIN enumerates 290891 groups (21s), keeps 122 groups by the even-partition rule, and computes 83 running time estimations (27s), so the search time reaches 72s (including initialization time). For the mixed application with 4 apps, MIN enumerates 16953281 execution plan groups, taking 1417s. With 8 apps, MIN cannot finish the search within hours.

5.2 Component analysis

W/ VS w/o preemption. Table 2 shows the end-to-end running time of Mil w/ and w/o preemption on LLM ensembling with 10000 requests, LLM routing with unknown output lengths and 4 data replications, and the Mixed application with workload (500, 5000) and 2 apps. The data format is "A+B (C)": A, B, C are inference time, extra time, and end-to-end time normalized to that of W/. For chain summary, Mil w/ and w/o preemption generate the same schedule. Table 2 validates the necessity of allowing preemption in scheduling. W/o preemption the search time of Mil will be shorter because a model's execution plan cannot be changed once selected.

Table 2: End-to-end time w/ VS w/o preemption.

Method	Ensembling	Routing	Mixed
W/	840+48	956+21	2164+60
W/o	1050+37 (1.2×)	1414+18 (1.5×)	2243+41 (1.0×)

Table 3: Noise level VS inference time normalized to Mil.

# requests	Exact	0	1e-5	1e-4	1e-3	1e-2	Rand
1000	1.0×	761s	1.1×	1.1×	1.4×	2.2×	2.2×
2000	1.0×	1417s	1.0×	1.0×	1.3×	2.3×	1.3×

Estimation error analysis. The respective average errors of the estimated application inference time for ensembling, routing, chain summary, mixed application in Section 5.1 are 16.0%, 26.1%, 35.6%, 20.0%, while the respective maximum errors are 29.5%, 50.7%, 38.6%, 31.3%. The error has two sources. (1) The first source is the iteration latency estimation, because we only estimate the major computation parts (Section 3), while other operations, like the output processing procedure in vLLM [25], are not considered. This error is relatively stable, e.g., for all the routing test cases, knowing exact output lengths leads to an average overall estimation error of 17.3%, with a maximum of 21.6%. As this error has a similar influence on different models, we can still find good schedules despite it. (2) The second error source is the output length estimation. To show its influence on the final application efficiency, we add uniform noise ranging in $[-a, a]$ to our output length probability, where a is the noise level. We select ensembling for the test and set the maximum output length limit to 4096. Table 3 shows the results. We also test the efficiency when the output length estimation is uniformly sampled from the range between 1 and its maximum possible value ("Rand" in Table 3). Table 3 shows that changing the output length distribution for sampling can significantly damage the application efficiency.

In our experiments, the maximum overall estimation errors of routing and chain summary are relatively large. For routing, the extreme case (50.7%) appears with 1 data replication. In this case, our estimated output lengths are longer than real values, but the relative throughputs of different models are similar to the real case, so we still find a good schedule. As the data replication number increases, the estimation error decreases to 19.5%. For chain summary, as there is one extremely long document in the tests, i.e., many summarization steps have only 1 input chunk, sampling cannot estimate the output length accurately in this case, making our estimation error large; but the summarization steps in the earlier stage have more requests, accounting for a large portion of the workloads, and our estimation is more accurate on them, so we can still find good schedules for chain summary.

Table 4: Inference time of Oracle VS Mil.

Method-# requests	Oracle-1k	Mil-1k	Oracle-2k	Mil-2k
Inference time (s)	84	84	116	120
Estimation error	8.0%	10.7%	0.2%	10.8%

Runtime adjustment analysis. We run experiments with no runtime adjustment for Mil on all the experiments in Section 5.1. The results show that runtime adjustment brings 1.0-1.1× speedup. On many test cases, our predicted model completion order is correct, or there are no models to be rescheduled, so the runtime adjustment mechanism is not activated. For mixed applications, runtime adjustment is effective, because in the early execution phase, our predicted model completion order is wrong and there are waiting models; without runtime adjustment, these models cannot be started ahead of schedule and the released GPUs remain idle.

Greedy VS Oracle. To explore how close Mil is to the optimal solution, we compare Mil with Oracle, i.e., enumerating all the possible schedules to select the best one. It is impossible to run each schedule to get its real running time, due to the large search space, so Oracle uses our processing functions with output lengths known. Experiments show Oracle cannot schedule applications with more than 3 LLMs due to OOM, so we test it on ensembling with 3 LLMs (baize-v2-13b, koala-13B-HF, dolly-v2-12b) and 1000 or 2000 requests (the maximum output length limit is 256). Table 4 shows that in these test cases, since the estimation error of Oracle is small, we can regard Oracle's performance as close to optimal, and hence Mil is also near optimal. In terms of the search time, Oracle takes 525s and 1378s, respectively, while Mil takes 10s and 11s. With 1000 and 2000 requests, Oracle computes 10443 and 20338 running time estimations (taking 368s and 1080s) and checks 61008 and 76354 plan groups.

6 RELATED WORK

Malleable job scheduling. The malleable job scheduling problem considers the resource allocation as well as the job scheduling. There are many variants of it, including allowing preemption or not, minimizing the makespan or the flow-time, having precedence constraints or not, etc. The malleable job scheduling problems are often solved by linear programming (LP), or heuristics like round robin, Equipartition, and greedy methods.

Multi-DNN inference acceleration. Existing works improve multi-DNN inference efficiency by resource sharing [52] or resource scheduling[38]. [52] considers GPU sharing in the online task scheduling scenario, and addresses the GPU fragmentation problem. Optimus [38] dynamically allocates resources and places DL training tasks to minimize job completion time, based on online resource-performance models. Alpaserve [27] uses model parallelism additionally for the statistical multiplexing of multiple devices when serving multiple models in the online serving scenario.

Muti-LLM inference acceleration. There are many works about the multi-LLM serving system [11, 30, 31]. MuxServe [11] is a flexible spatial-temporal multiplexing system for multiple LLM serving. Parrot [30] improves the end-to-end experience of the LLM-based applications by performing data flow analysis. Autellix [31] focuses on dynamic multi-LLM agentic programs. Besides online serving, there are also works about multi-LLM training [35, 44]. Saturn [35] reduces the model selection time by solving the joint problem of selecting parallelism, allocating resources, and scheduling. HybridFlow [44] improves RLHF efficiency.

LLM output length prediction. Some recent works try to predict the exact output lengths of LLMs using LLMs or other trained models [13, 18, 23, 46, 58]. [13] trains a model to rank requests by their output lengths. A concurrent work BlendServe [57], similar to our work, estimates the output lengths by sampling a subset of requests before the GPU running and assumes that the requests sharing prefixes in the prompts have a similar distribution of output lengths. While we obtain the output length distribution from a dataset different from the ones used by the applications.

7 CONCLUSION AND FUTURE WORK

In this work, we formulate the MLAS problem and prove it to be NP-hard. To the best of our knowledge, MLAS is the first to study the makespan minimization of multi-LLM applications in offline inference. We propose Mil for MLAS. Experiments show that Mil can achieve up to 3.4× end-to-end speedup. Mil currently considers data parallelism and tensor parallelism for LLMs, but other parallelism strategies (e.g., pipeline parallelism, expert parallelism) can be supported in the future. Another extension is the support for multi-node inference. If there is high bandwidth between nodes, we can merge the nodes and run Mil on it. Future work can support slow inter-node bandwidth.

ACKNOWLEDGMENTS

Lei Chen's work is partially supported by National Key Research and Development Program of China Grant No. 2023YFF0725100, National Science Foundation of China (NSFC) under Grant No. U22B2060, Guangdong-Hong Kong Technology Innovation Joint Funding Scheme Project No. 2024A0505040012, the Hong Kong RGC GRF Project 16213620, RIF Project R6020-19, AOE Project AoE/E-603/18, Theme-based project TRS T41-603/20R, CRF Project C2004-21G, Key Areas Special Project of Guangdong Provincial Universities 2024ZDZX1006, Guangdong Province Science and Technology Plan Project 2023A0505030011, Guangzhou municipality big data intelligence key lab, 2023A03J0012, Hong Kong ITC ITF grants MHX/078/21 and PRP/004/22FX, Hong Kong ITC TC-SKLCRCC26EG01, Zhujiang scholar program 2021JC02X170, Microsoft Research Asia Collaborative Research Grant, HKUST-Webank joint research lab, 2025 HKUST Shenzhen-Hong Kong Collaborative Innovation Institute Green Sustainability Special Fund from Shui On Xintiandi and the InnoSpace GBA, and HKUST(GZ) - CMCC(Guangzhou Branch) Metaverse Joint Innovation Lab under Grant No. P00659. Yanyan Shen is supported by the National Key Research and Development Program of China (No. 2024YFB4505203), National Natural Science Foundation of China (No. 62522211), and Key Research and Development Program of Xinjiang Uygur Autonomous Region (Grant No. 2023B01027, 2023B01027-1). The corresponding author is Yanyan Shen.

REFERENCES

[1] Pranjal Aggarwal, Aman Madaan, Ankit Anand, Srividya Pranavi Potharaju, Swaroop Mishra, Pei Zhou, Aditya Gupta, Dheeraj Rajagopal, Karthik Kappaganthu, Yiming Yang, et al. 2023. Automix: Automatically mixing language models. *arXiv preprint arXiv:2310.12963* (2023).

[2] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. 2021. Program Synthesis with Large Language Models. *arXiv preprint arXiv:2108.07732* (2021).

[3] Yapei Chang, Kyle Lo, Tanya Goyal, and Mohit Iyyer. 2024. BooookScore: A systematic exploration of book-length summarization in the era of LLMs. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net. https://openreview.net/forum?id=7Ttk3RzDeu

[4] Chen Chen, Xiaodi Ke, Timothy Zeyl, Kaixiang Du, Sam Sanjabi, Shane Bergsma, Reza Pournaghi, and Chong Chen. 2019. Minimum makespan workflow scheduling for malleable jobs with precedence constraints and lifetime resource demands. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2068–2078.

[5] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating Large Language Models Trained on Code. arXiv:2107.03374 [cs.LG]

[6] Wei-Lin Chiang, Zhuohan Li, Ziqing Lin, Ying Sheng, Zhanghao Wu, Hao Zhang, Lianmin Zheng, Siyuan Zhuang, Yonghao Zhuang, Joseph E Gonzalez, et al. 2023. Vicuna: An open-source chatbot impressing gpt-4 with 90%* chatgpt quality. *See https://vicuna. lmsys. org (accessed 14 April 2023)* 2, 3 (2023), 6.

[7] Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. 2021. Training Verifiers to Solve Math Word Problems. *arXiv preprint arXiv:2110.14168* (2021).

[8] Mike Conover, Matt Hayes, Ankit Mathur, Xiangrui Meng, Jianwei Xie, Jun Wan, Sam Shah, Ali Ghodsi, Patrick Wendell, Matei Zaharia, and Reynold Xin. 2023. Free dolly: Introducing the world's first truly open instruction-tuned llm.

[9] Mike Conover, Matt Hayes, Ankit Mathur, Jianwei Xie, Jun Wan, Sam Shah, Ali Ghodsi, Patrick Wendell, Matei Zaharia, and Reynold Xin. 2023. *Free Dolly: Introducing the World's First Truly Open Instruction-Tuned LLM.* https://www.databricks.com/blog/2023/04/12/dolly-first-open-commercially-viable-instruction-tuned-llm

[10] Jianzhong Du and Joseph Y-T Leung. 1989. Complexity of scheduling parallel task systems. *SIAM Journal on Discrete Mathematics* 2, 4 (1989), 473–487.

[11] Jiangfei Duan, Runyu Lu, Haojie Duanmu, Xiuhong Li, Xingcheng Zhang, Dahua Lin, Ion Stoica, and Hao Zhang. [n.d.]. MuxServe: Flexible Spatial-Temporal Multiplexing for Multiple LLM Serving. In *Forty-first International Conference on Machine Learning*.

[12] Jeff Edmonds, Donald D Chinn, Tim Brecht, and Xiaotie Deng. 2003. Non-clairvoyant multiprocessor scheduling of jobs with changing execution characteristics. *Journal of Scheduling* 6, 3 (2003), 231–250.

[13] Yichao Fu, Siqi Zhu, Runlong Su, Aurick Qiao, Ion Stoica, and Hao Zhang. 2025. Efficient LLM Scheduling by Learning to Rank. *Advances in Neural Information Processing Systems* 37 (2025), 59006–59029.

[14] Xinyang Geng, Arnav Gudibande, Hao Liu, Eric Wallace, Pieter Abbeel, Sergey Levine, and Dawn Song. 2023. Koala: A dialogue model for academic research.

[15] Team GLM, Aohan Zeng, Bin Xu, Bowen Wang, Chenhui Zhang, Da Yin, Dan Zhang, Diego Rojas, Guanyu Feng, Hanlin Zhao, et al. 2024. Chatglm: A family of large language models from glm-130b to glm-4 all tools. *arXiv preprint arXiv:2406.12793* (2024).

[16] Gurobi Optimization, LLC. 2024. Gurobi Optimizer Reference Manual. https://www.gurobi.com

[17] Yuxiong He, Wen Jing Hsu, and Charles E Leiserson. 2007. Provably efficient adaptive scheduling for parallel jobs. (2007).

[18] Cunchen Hu, Heyang Huang, Liangliang Xu, Xusheng Chen, Jiang Xu, Shuang Chen, Hao Feng, Chenxi Wang, Sa Wang, Yungang Bao, et al. 2024. Inference without interference: Disaggregate llm inference for mixed downstream workloads. *arXiv preprint arXiv:2401.11181* (2024).

[19] Qitian Jason Hu, Jacob Bieker, Xiuyu Li, Nan Jiang, Benjamin Keigwin, Gaurav Ranganath, Kurt Keutzer, and Shriyash Kaustubh Upadhyay. 2024. ROUTERBENCH: A Benchmark for Multi-LLM Routing System. *arXiv preprint arXiv:2403.12031* (2024).

[20] Yebowen Hu, Kaiqiang Song, Sangwoo Cho, Xiaoyang Wang, Hassan Foroosh, and Fei Liu. 2023. DecipherPref: Analyzing Influential Factors in Human Preference Judgments via GPT-4. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*. 8344–8357.

[21] Dongfu Jiang, Xiang Ren, and Bill Yuchen Lin. 2023. LLM-Blender: Ensembling Large Language Models with Pairwise Ranking and Generative Fusion. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2023, Toronto, Canada, July 9-14, 2023*, Anna Rogers, Jordan L. Boyd-Graber, and Naoaki Okazaki (Eds.). Association for Computational Linguistics, 14165–14178. https://doi.org/10.18653/V1/2023.ACL-LONG.792

[22] Fengqing Jiang. 2024. *Identifying and mitigating vulnerabilities in llm-integrated applications*. Master's thesis. University of Washington.

[23] Yunho Jin, Chun-Feng Wu, David Brooks, and Gu-Yeon Wei. 2023. S^3: Increasing GPU Utilization during Generative Inference for Higher Throughput. In *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*, Alice Oh, Tristan Naumann, Amir Globerson, Kate Saenko, Moritz Hardt, and Sergey Levine (Eds.). http://papers.nips.cc/paper_files/paper/2023/hash/3a13be0c5dae69e0f08065f113fb10b8-Abstract-Conference.html

[24] Wojciech Kryściński, Nazneen Rajani, Divyansh Agarwal, Caiming Xiong, and Dragomir Radev. 2022. BOOKSUM: A Collection of Datasets for Long-form Narrative Summarization. In *Findings of the Association for Computational Linguistics: EMNLP 2022*. 6536–6558.

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

[26] LAION-AI. 2023. Open assistant. https://github.com/LAION-AI/Open-Assistant

[27] Zhuohan Li, Lianmin Zheng, Yinmin Zhong, Vincent Liu, Ying Sheng, Xin Jin, Yanping Huang, Zhifeng Chen, Hao Zhang, Joseph E Gonzalez, et al. 2023. {AlpaServe}: Statistical multiplexing with model parallelism for deep learning serving. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. 663–679.

[28] Zhuoyan Li, Hangxiao Zhu, Zhuoran Lu, and Ming Yin. 2023. Synthetic data generation with large language models for text classification: Potential and limitations. *arXiv preprint arXiv:2310.07849* (2023).

[29] Percy Liang, Rishi Bommasani, Tony Lee, Dimitris Tsipras, Dilara Soylu, Michihiro Yasunaga, Yian Zhang, Deepak Narayanan, Yuhuai Wu, Ananya Kumar, et al. 2022. Holistic evaluation of language models. *arXiv preprint arXiv:2211.09110* (2022).

[30] Chaofan Lin, Zhenhua Han, Chengruidong Zhang, Yuqing Yang, Fan Yang, Chen Chen, and Lili Qiu. 2024. Parrot: Efficient Serving of {LLM-based} Applications with Semantic Variable. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. 929–945.

[31] Michael Luo, Xiaoxiang Shi, Colin Cai, Tianjun Zhang, Justin Wong, Yichuan Wang, Chi Wang, Yanping Huang, Zhifeng Chen, Joseph E Gonzalez, et al. 2025. Autellix: An Efficient Serving Engine for LLM Agents as General Programs. *arXiv preprint arXiv:2502.13965* (2025).

[32] Konstantin Makarychev and Debmalya Panigrahi. 2014. Precedence-constrained scheduling of malleable jobs with preemption. In *Automata, Languages, and Programming: 41st International Colloquium, ICALP 2014, Copenhagen, Denmark, July 8-11, 2014, Proceedings, Part I 41*. Springer, 823–834.

[33] NLP Team MosaicML. 2023. Introducing mpt-7b: A new standard for open-source, ly usable llms.

[34] Zan Ahmad Naeem, Mohammad Shahmeer Ahmad, Mohamed Eltabakh, Mourad Ouzzani, and Nan Tang. 2024. RetClean: Retrieval-Based Data Cleaning Using LLMs and Data Lakes. *Proceedings of the VLDB Endowment* 17, 12 (2024), 4421–4424.

[35] Kabir Nagrecha and Arun Kumar. 2023. Saturn: An optimized data system for multi-large-model deep learning workloads. *Proceedings of the VLDB Endowment* 17, 4 (2023), 712–725.

[36] Avanika Narayan, Ines Chami, Laurel Orr, Simran Arora, and Christopher Ré. 2022. Can foundation models wrangle your data? *arXiv preprint arXiv:2205.09911* (2022).

[37] Shashi Narayan, Shay B Cohen, and Mirella Lapata. 2018. Don't give me the details, just the summary! topic-aware convolutional neural networks for extreme summarization. *arXiv preprint arXiv:1808.08745* (2018).

[38] Yanghua Peng, Yixin Bao, Yangrui Chen, Chuan Wu, and Chuanxiong Guo. 2018. Optimus: an efficient dynamic resource scheduler for deep learning clusters. In *Proceedings of the Thirteenth EuroSys Conference*. 1–14.

[39] Vatsal Raina and Mark Gales. 2024. Question-Based Retrieval using Atomic Units for Enterprise RAG. In *Proceedings of the Seventh Fact Extraction and VERification Workshop (FEVER)*. 219–233.

[40] Nazneen Rajani, Lewis Tunstall, Edward Beeching, Nathan Lambert, Alexander M. Rush, and Thomas Wolf. 2023. No Robots. https://huggingface.co/datasets/HuggingFaceH4/no_robots.

[41] Julien Robert and Nicolas Schabanel. 2007. Non-clairvoyant batch sets scheduling: Fairness is fair enough. In *European Symposium on Algorithms*. Springer, 741–753.

[42] Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, et al. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950* (2023).

[43] Parth Sarthi, Salman Abdullah, Aditi Tuli, Shubh Khanna, Anna Goldie, and Christopher D Manning. 2024. Raptor: Recursive abstractive processing for tree-organized retrieval. In *The Twelfth International Conference on Learning Representations*.

[44] Guangming Sheng, Chi Zhang, Zilingfeng Ye, Xibin Wu, Wang Zhang, Ru Zhang, Yanghua Peng, Haibin Lin, and Chuan Wu. 2025. Hybridflow: A flexible and efficient rlhf framework. In *Proceedings of the Twentieth European Conference on Computer Systems*. 1279–1297.

[45] Stability-AI. 2023. Stablelm: Stability ai language models. https://github.com/stability-AI/stableLM.

[46] Jovan Stojkovic, Chaojie Zhang, Íñigo Goiri, Josep Torrellas, and Esha Choukse. 2024. Dynamollm: Designing llm inference clusters for performance and energy efficiency. *arXiv preprint arXiv:2408.00741* (2024).

[47] Hongyang Sun, Wen-Jing Hsu, and Yangjie Cao. 2014. Competitive online adaptive scheduling for sets of parallel jobs with fairness and efficiency. *J. Parallel and Distrib. Comput.* 74, 3 (2014), 2180–2192.

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

[49] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. 2023. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971* (2023).

[50] Sentence Transformers. [n.d.]. *Natural Questions dataset*. https://huggingface.co/datasets/sentence-transformers/natural-questions

[51] TRL. [n.d.]. *TL;DR Dataset*. https://huggingface.co/datasets/trl-lib/tldr

[52] Qizhen Weng, Lingyun Yang, Yinghao Yu, Wei Wang, Xiaochuan Tang, Guodong Yang, and Liping Zhang. 2023. Beware of fragmentation: Scheduling {GPU-Sharing} workloads with fragmentation gradient descent. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. 995–1008.

[53] Canwen Xu, Daya Guo, Nan Duan, and Julian McAuley. 2023. Baize: An open-source chat model with parameter-efficient tuning on self-chat data. *arXiv preprint arXiv:2304.01196* (2023).

[54] Can Xu, Qingfeng Sun, Kai Zheng, Xiubo Geng, Pu Zhao, Jiazhan Feng, Chongyang Tao, Qingwei Lin, and Daxin Jiang. 2024. WizardLM: Empowering large pre-trained language models to follow complex instructions. In *The Twelfth International Conference on Learning Representations*.

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

[56] Bowen Zhang and Harold Soh. 2024. Extract, Define, Canonicalize: An LLM-based Framework for Knowledge Graph Construction. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*. 9820–9836.

[57] Yilong Zhao, Shuo Yang, Kan Zhu, Lianmin Zheng, Baris Kasikci, Yang Zhou, Jiarong Xing, and Ion Stoica. 2024. BlendServe: Optimizing Offline Inference for Auto-regressive Large Models with Resource-aware Batching. *arXiv preprint arXiv:2411.16102* (2024).

[58] Zangwei Zheng, Xiaozhe Ren, Fuzhao Xue, Yang Luo, Xin Jiang, and Yang You. 2023. Response length perception and sequence scheduling: An llm-empowered llm inference pipeline. *Advances in Neural Information Processing Systems* 36 (2023), 65517–65530.