UNIVERSITY OF CALIFORNIA
RIVERSIDE

Towards Efficient and Adaptive LLM System: From Uncertainty-Aware Inference to Hybrid Training Integration

A Dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Electrical Engineering

by

Yufei Li

September 2025

Dissertation Committee:
- Dr. Cong Liu, Chairperson
- Dr. Yue Dong
- Dr. Hyoseung Kim

The Dissertation of Yufei Li is approved:

Committee Chairperson

University of California, Riverside

Acknowledgments

I am grateful to my advisor, Dr. Cong Liu, without whose help, I would not have been here. His forward-looking research ideas, support for my long-term development, and belief in me have been instrumental throughout my PhD journey. His expertise and character left a lasting impact not only on my academic development but also on how I approach challenges in life and research. In times of uncertainty, his mentorship gave me clarity and renewed motivation. His support extended far beyond academic matters—he genuinely cared about my well-being and helped me maintain a healthy work-life balance. I am especially grateful for his dedication, encouragement, and the example he set as both a scholar and mentor. I will always hold deep appreciation for the significant role that he has played in my personal and professional growth. Since our first meeting in 2022, it has been an honor to work under his guidance. I will strive to make him proud in the years ahead as I continue to grow and contribute.

I would like to extend my appreciation to my incredible labmates, whose companionship and collaboration made my PhD journey not only intellectually stimulating but also emotionally fulfilling. Working alongside such talented, kind, and open-minded individuals transformed our lab into more than just a research environment—it became a community of shared growth, warmth, and mutual respect. I deeply value the engaging academic discussions, the generous exchange of ideas, and the many moments when we supported one another through both breakthroughs and setbacks. Equally meaningful were the casual, heartfelt conversations about life, family, and our diverse cultural backgrounds. These moments—often spontaneous and full of laughter—reminded me that research is as

much about people as it is about ideas. Though our time together may have been brief, the bonds we formed will stay with me for a lifetime. I will always carry fond memories of our shared meals, late-night work sessions, and spontaneous coffee breaks. To Zexin, Ziliang, Yingfan, Shahab, Aritra, and Yifan—thank you for creating an environment where I felt both intellectually challenged and emotionally supported. Your presence made the lab not just a space for academic pursuit, but a home filled with friendship, trust, and countless cherished memories.

I want to give my heartfelt thanks to my wife, Dr. Yanghong Guo. Without her unwavering support, thoughtful advice, and companionship, none of this journey would have been possible. From the very beginning, we walked this path together—applying for Ph.D. programs side by side, facing the uncertainty of the pandemic, and navigating the challenges of graduate school as a team. Through every decision, big or small, she stood by me—not only as a life partner but also as a trusted advisor and my strongest pillar of strength. During these transformative years, we celebrated life's most meaningful milestones together—we got engaged, married, and began building our future, hand in hand. Her resilience, empathy, and belief in me carried me through the most difficult moments and made every success more joyful and meaningful. And to our lovely dog, Bubble, thank you for your healing presence and unconditional love. Your companionship brought peace and joy to our lives during the most stressful times. You reminded me to stay grounded, playful, and grateful through it all. Together, this chapter of my life wouldn't be the same without you. Thank you for being my family and my constant source of strengths.

To my parents for all the support.

ABSTRACT OF THE DISSERTATION

Towards Efficient and Adaptive LLM System: From Uncertainty-Aware Inference to Hybrid Training Integration

by

Yufei Li

Doctor of Philosophy, Graduate Program in Electrical Engineering
University of California, Riverside, September 2025
Dr. Cong Liu, Chairperson

Large language models (LLMs) have rapidly advanced in capability but pose significant challenges for deployment, particularly under tight latency constraints and limited hardware resources. This dissertation presents three systems for adaptive scheduling and resource management, each addressing a critical layer of complexity in vectorizing LLM inference and retraining.

In the first part of the dissertation, I tackle the unpredictability of autoregressive decoding latency induced by input uncertainty. By quantifying correlations between input queries and response length, our scheduler dynamically prioritizes and consolidates inference workloads—further leveraging CPU offloading when beneficial, which reduces average response time and improves throughput during heavy traffic periods. In the second part of the dissertation, I extend the ecosystem to multi-GPU, distributed settings, enabling colocated inference and retraining. Through offline profiling, execution prediction, node partitioning, and runtime scheduling, our system significantly boosts resource utilization and inference accuracy without sacrificing service-level objectives. In the third chapter, I

propose a hybrid runtime GPU-local system that jointly optimizes iteration-level scheduling, prefix cache reuse, and KV pruning. This enables adaptive allocation between prefill, decoding, and fine-tuning phases. Compared to both periodic and continuous retraining strategies, our system reduces inference latency, sustains throughput, achieves alignment accuracy up, and maintains GPU utilization.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Efficient and Resource-Aware LLM Systems

Large language models (LLMs) have rapidly evolved from research prototypes to the backbone of real-world applications such as conversational agents, code assistants, and decision-support systems. Despite their success, deploying LLMs efficiently in practical systems remains a formidable challenge. Their enormous parameter sizes and autoregressive decoding characteristics lead to high computational demand, memory pressure, and unpredictable latency under dynamic workloads. Moreover, as LLMs are increasingly required to adapt to new data and user preferences in real time, system design must jointly consider inference efficiency and continual training.

This dissertation addresses the central question of how to design resource-aware LLM systems that balance accuracy, latency, and adaptability across a spectrum of deployment scenarios. We approach this challenge incrementally:

- Starting from single-GPU real-time inference, we develop mechanisms to handle workload variability and latency service-level objectives (SLOs).
- We then scale to multi-GPU distributed environments, where joint scheduling of inference and training tasks becomes critical to maximize utilization and maintain throughput.
- Finally, we extend to single-node systems where inference and training co-exist, tackling the fundamental tension between fast response time and frequent retraining for alignment.

Through these stages, the dissertation provides a coherent exploration of how system-level scheduling, resource management, and model-level adaptation can be co-designed to enable efficient, reliable, and adaptive LLM serving.

1.2 Dissertation Outlines and Contributions

1.2.1 Chapter 2: RT-LM: Uncertainty-Aware Resource Management for Real-Time Inference of Language Models

We begin with the problem of efficient inference serving on a single GPU/CPU. RT-LM proposes an uncertainty-aware resource manager that dynamically adjusts batching and scheduling to meet strict latency SLOs while maintaining model accuracy. By leveraging uncertainty estimation, RT-LM adapts resource allocation under variable request patterns, improving both responsiveness and throughput. This chapter establishes the foundation of SLO-aware inference scheduling.

1.2.2 Chapter 3: LeMix: Unified Scheduling for LLM Training and Inference on Multi-GPU Systems

We then extend to distributed multi-GPU clusters, where both inference and retraining must be orchestrated together. LeMix introduces a unified scheduler that coordinates GPU resources between training and inference at the node level. It accounts for heterogeneous resource requirements, network contention, and synchronization delays. By co-optimizing training and serving, LeMix demonstrates that retraining can be performed continuously without significantly degrading inference performance. This chapter highlights the feasibility of co-locating inference and training in distributed systems.

1.2.3 Chapter 4: MACE: A Hybrid LLM Serving System with SLO-aware Continuous Retraining Alignment

Finally, we focus on the runtime dynamics of a single GPU when inference and training jobs compete for fine-grained resources. Unlike prior node-level approaches, our hybrid iteration-level scheduler integrates priority assignment (PA), iteration-level batching (IB), and cache management (CM) to reduce fragmentation and latency interference. The system aligns retraining with inference workloads while preserving SLOs, offering a practical pathway toward real-time "learning while serving". This chapter represents the culmination of our exploration, demonstrating that hybrid GPU-local scheduling can bridge inference efficiency and continual alignment.

Chapter 2

RT-LM: Uncertainty-Aware Resource Management for Real-Time Inference of Language Models

2.1 Introduction

The recent surge in the development and dissemination of language models (LMs) such as ChatGPT has significantly reshaped the landscape of natural language processing (NLP) [14, 23, 24, 83]. This advancement holds immense promise for a multitude of applications, including multi-lingual robots and voice control devices integral to the future of smart homes [61, 72, 162]. Despite the impressive capability to generate human-like

responses, these state-of-the-art LMs present a formidable challenge when attempting to deploy them on various devices due to their complex computational behaviors and unpredictable real-time inference capabilities [148,150]. With the increasing demand for real-time language processing, server-backed systems, such as online chatbots (e.g., ChatGPT manages over 10 million daily queries) and live-translation services, exemplify the need for devices that can efficiently process simultaneous requests from multiple users, especially during peak times.

A set of recent works seek to enhance the inference latency of on-device LMs by crafting an array of model optimization techniques, including quantization [160], pruning [44, 56], and distillation [133]. These techniques aim at decreasing model complexity (thus the computational demand) while preserving their accuracy. Nonetheless, a knowledge gap persists in understanding and exploring the correlation between an input text and the corresponding inference latency within a given LM from a system-level perspective.

The NLP community has recently brought to light various sources of uncertainties [36, 49, 51, 104, 177], which have been shown to negatively impact model's accuracy and may introduce significant variations in the lengths of generated responses. Take, for example, a broad and ambiguous question such as "*Can you tell me the history of art?*". This could prompt a LM to generate lengthier outputs, given that the history of art spans millennia and includes a multitude of cultures, styles, periods, and artistic movements. Intuitively, the longer output a LM generates, the greater the inference latency, as each output token is sequentially generated with negligible computational difference [98, 147]. These sources of uncertainties, often intrinsic to the nature of language understanding and gener-

ation, can stem from varying data distributions [55, 71], intricate model architectures [43], or even the non-deterministic parallel computing behaviors at runtime [111], rendering the induced latency more complex and challenging to manage. Consequently, it is critical to understand and mitigate such uncertainties due to their potential to induce non-trivial inference latency and computational inefficiency, or even hinder the prompt delivery of dialogue generation (DG) due to degraded system performance.

This work is specifically motivated by the following queries: (i) What is the intrinsic correlation between an input text's uncertainty characteristics and the subsequent computational demand (and thus, the inference latency) for a given LM, such as why two syntactically similar inputs may necessitate dramatically different inference latencies? (ii) Is it feasible to devise a lightweight approach to predict an input's computational demand at runtime? and (iii) Can the system-level resource manager exploit these quantified input characteristics to improve latency performance during inference? Understanding the quantifiable correlation between an input text and its computational demand is critical, as it could unveil novel opportunities for system-level optimization, thereby enhancing the performance and efficiency of LMs deployed on embedded devices, e.g., by deferring the execution of inputs with high computational demand thus reducing head-of-line blocking.

Our research attempts to comprehend, quantify, and optimize these uncertainty-induced variations on latency performance in LMs. We propose a cohesive ecosystem that integrates an application-level uncertainty quantification framework with a system-level uncertainty-aware resource manager. The application-level framework aims to precisely quantify task uncertainties and their potential impacts on latency. Simultaneously, the

system-level resource manager utilizes the provided estimations to make informed decisions on resource allocation and task scheduling, thereby mitigating the detrimental effects of uncertainties on system performance.

Contributions. In this paper, we propose an uncertainty-ware resource management ecosystem, namely `RT-LM`, for real-time on-device LMs. Specifically, `RT-LM` features three technical novelties: 1) It first quantitatively reveals how major input uncertainties—well-defined by the NLP community—negatively impact latency. Our findings demonstrate that uncertainty characteristics of an input text may notably increase the output length, i.e., the number of tokens in the generated response; 2) Building on this insight, we develop a lightweight yet effective method that can quickly correlate and quantify the output length for an input text at runtime, considering a comprehensive set of uncertainties defined by the NLP community; 3) Leveraging this quantification as a heuristic of latency, we incorporate the uncertainty information of each input into system-level scheduler that performs several optimizations, including uncertainty-aware prioritization, dynamic consolidation, and strategic utilization of CPU cores.

We implement `RT-LM` mainly on an edge server. We evaluate the response time and throughput across five state-of-the-art LMs[1], namely DialoGPT [167], GODEL [120], BlenderBot [130], BART [80], and T5 [126]. We utilize `RT-LM` four widely-researched benchmark datasets: *Blended Skill Talk* [143], *PersonaChat* [166], *ConvAI2* [33], and *Empathetic Dialogues* [128]. For both the models and datasets, we use the versions released by Hugging Face.

[1]While there are larger models like ChatGPT that offer impressive capabilities, their resource-intensive nature makes them less viable for deployment.

Evaluation results demonstrate that RT-LM achieves: **Efficiency**: RT-LM outperforms all compared methods by a significant margin in most cases, improving the maximum response time by up to 30% and throughput by up to 40% compared to uncertainty-oblivious baselines. **Efficacy across a range of behaviors**: The tested workloads include five LMs with diverse task uncertainty characteristics and varied workload settings. **Robustness under malicious scenarios**: RT-LM is resilient when facing adversarial conditions, effectively mitigating the impact of malicious tasks by resource management. **Runtime overhead**: The design and implementation of RT-LM is efficient, incurring a rather small runtime latency and memory usage.

2.2 Background and Challenges

2.2.1 Dialogue Generation using LMs

Recently, pre-trained LMs such as ChatGPT and GPT-4 [14] have emerged as a dominant force in the field of dialogue generation (DG). These models are characterized by their large size and are often trained on vast amounts of textual data, which demonstrate remarkable capabilities in understanding and generating human-like responses across a wide range of tasks. A key property of these models is the *autoregressive* generation process [150], where output tokens are generated sequentially with each new token being conditioned on the previously generated tokens. Consequently, the output length plays a pivotal role in determining the inference latency of a LM, as generating longer sequences inherently requires more time. Depending on the nature of inputs, a LM may generate outputs of varied lengths. For instance, a query that has clear and concise meanings may elicit a brief

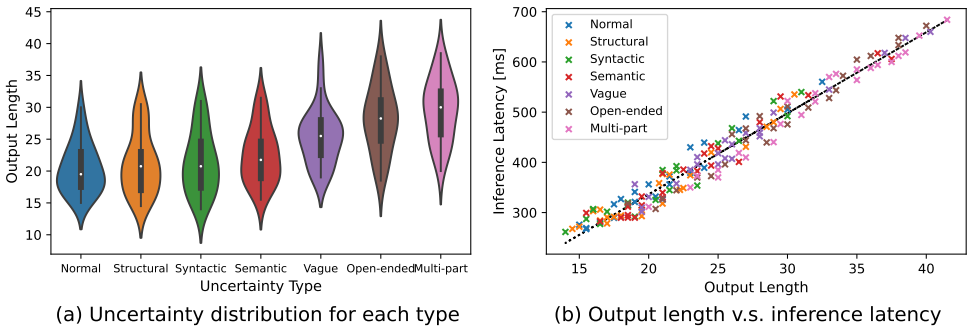
Table 2.1: Types of linguistic uncertainty, their definitions and example statements or questions.

Type	**Definition**	**Statement/Question**
Structural ambiguity	Uncertainty related to multiple possible parse structures, leading to outputs with varying lengths.	"John saw a boy in the park with a telescope."
Syntactic ambiguity	Uncertainty arising from multiple part-of-speech tags of a word, resulting in different interpretations.	"Rice flies like sand."
Semantic ambiguity	Uncertainty stemming from words with multiple meanings, leading to varying interpretations.	"What's the best way to deal with bats?"
Vague expressions	Uncertainty arising from broad concepts or highly-generalized topics that demand specific analysis.	"Tell me about the history of art."
Open-endedness	Questions or statements that lack a single definitive answer and require providing relevant context, background, and explanations.	"What are the causes and consequences of poverty in developing countries?"
Multi-partness	Questions or statements containing multiple subquestions or topics, which demand detailed answers.	"How do cats and dogs differ in behavior, diet, and social interaction?"

response, whereas an ambiguous or broad query may demand a considerably longer output. This variability, often called *linguistic uncertainty* [136] by the NLP community, in output length and the subsequent impact on latency, can pose significant challenges when deploying LMs on resource-constrained devices, as the performance requirements and computational constraints must accommodate a wide range of potential latencies.

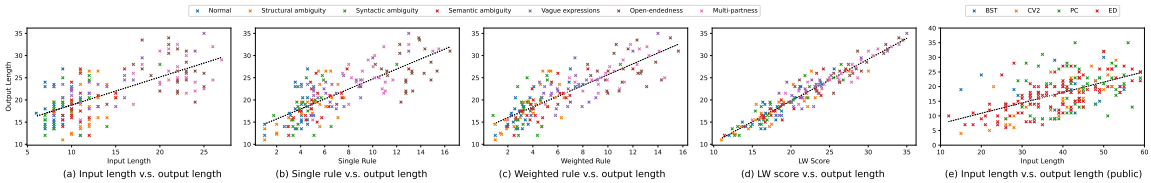
2.2.2 Sources and Impacts of Linguistic Uncertainty

Linguistic uncertainty is a challenging and diverse sub-domain in NLP, which often leads to multiple interpretations of inputs and potentially varied outputs in dialogue systems. The language and linguistics community has well-defined a categorization of linguistic uncertainty that encompasses the majority of uncertainty sources, including three types of lexical ambiguity (structural ambiguity [49, 158], syntactic ambiguity [78, 104], semantic



(a) Uncertainty distribution for each type (b) Output length v.s. inference latency

Figure 2.1: Observations of (a) distribution of LM output lengths for inputs with different uncertainty types, and (b) the correlation between LM output lengths and inference latency.



(a) Input length v.s. output length (b) Single rule v.s. output length (c) Weighted rule v.s. output length (d) LW score v.s. output length (e) Input length v.s. output length (public)

Figure 2.2: Correlation between average output length across the five LMs and (a) input length, (b) single rule-based score, (c) weighted rule-based score, (d) LW model scores for self-generated sentences that contain different types of uncertainties, as well as (e) input length for sentences from the four benchmark datasets.

ambiguity [58, 77]), vague expressions [51], open-ended questions [36, 105], and multi-part questions [177] that demand comprehensive answers and additional explanations. Their definitions and example statements or questions are listed in Table 2.1.

2.3 Key Observations and Ideas

2.3.1 Uncertainty-Induced Negative Impact on Latency and Root Cause

We conducted a comprehensive set of studies investigating the correlation between inputs' uncertainty characteristics and the resulting inference latency of several LMs. Specifically, we create 1,000 utterances for each of the six uncertainty types (defined in Sec. 2.2.2) and record the averaged output length as well as inference latency across Di-

aloGPT, GODEL, BlenderBot, BART, and T5, as shown in Fig. 2.1a. We observe that all types of linguistic uncertainties lead to longer outputs and non-trivially larger latencies to varying degrees. Specifically, vague expressions, open-endedness, and multi-partness are generally more deterministic compared to the three types of lexical ambiguities. This can be attributed to that modern neural networks (NNs) lack uncertainty awareness and are prone to overconfidence when making decisions [52], which results in LMs understanding one potential interpretation and respond accordingly without seeking further clarifications. Furthermore, semantic ambiguity has a more significant impact on output lengths than structural and syntactic ambiguities. We speculate that this is because some words with multiple meanings such as *"trunk"* or *"monitor"* are more likely to cause confusions for a LM and thereby triggering longer responses, e.g., by enumerating all potential interpretations of a word sense and asking for explanations.

Fig. 2.1b plots the correlation between inference latency and output length for sentences that contain different types of uncertainties. We observe that inference latency is proportional to the output length, with longer outputs generally requiring larger inference latencies. Some sentences with uncertainties such as open-endedness and multi-partness may even take over 700ms for a LM to generate corresponding responses, which is 2$\sim$4 times the latency of normal sentences. This presents a substantial opportunity for system-level optimization, as resource manager can leverage this uncertainty impact as an estimation of task execution times to enhance system efficiency and resource utilization.

Listing 2.1: Code for measuring vague expression scores.

```
def vague_expressions_score(sentence, weight=1):
    vague_count = 0
    words = word_tokenize(sentence)
    stem_words = [lemmatizer.lemmatize(word) for word in words]
    # VAGUE_WORDS are pre-defined in the literature
    for phrase in VAGUE_WORDS:
        matches = stem_words.count(lemmatizer.lemmatize(phrase))
        vague_count += matches
    return weight * vague_count
```

2.3.2 Predicting the Output Length for a Given Input

Upon observing that inference latency is determined by output length, we develop methods that can accurately yet efficiently predict such length for a given input at runtime. As discussed earlier, uncertainty of an input text may increase the output length and thus negatively impact inference latency. Our methods shall take uncertainty into account when making such predictions. ***Uncertainty score****: In this work, we define uncertainty score for an input text as the estimated number of tokens (output length) required to formulate a comprehensive and unambiguous response that sufficiently addresses the posed inquiry.*

Input length. Intuitively, longer inputs may lead to LMs generating longer outputs, even without considering uncertainty. We demonstrate the impact of this naive heuristic on output lengths in Fig. 2.2a. We observe that although the correlation is not deterministic and noisy, longer input lengths generally induce longer generated outputs. This inspires us to further improve it by considering uncertainty.

Single rule. We measure the intensity of each uncertainty using hand-crafted rules introduced in the literature. Specially, we use the spaCy[2] language tool to tokenize input text and obtain the Part-of-Speech (PoS) tag for each token in the original text. Then, we quantify uncertainty scores by searching for pre-defined patterns inherently existing in

[2] https://spacy.io/

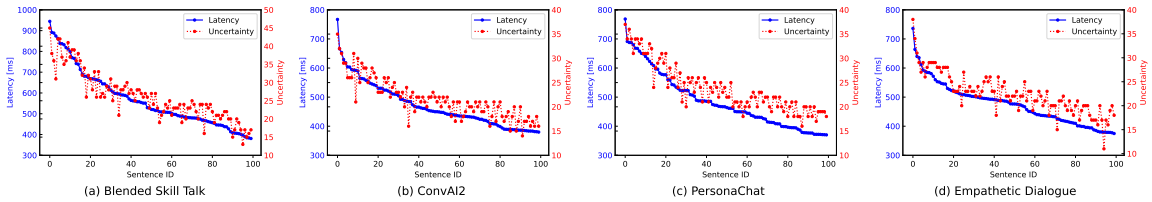


Figure 2.3: Distribution of latency and corresponding uncertainty on four benchmark DG datasets, (a) *Blended Skill Talk*, (b) *ConvAI2*, (c) *PersonaChat*, (d) *Empathetic Dialogue*. The data points are ranked by descending order of latency.

each uncertainty source using regular expressions. Listing 2.1 shows an example code for quantifying vague expression uncertainty. Note that for input sentences that do not contain the defined six uncertainty sources, we use input lengths as their single rule scores. We evaluate the correlation between single rule scores and the output lengths for inputs containing the corresponding type of uncertainty in Fig. 2.2b. We observe that the correlation is slightly more apparent and less noisy, which demonstrates the impact of uncertainty on LM generation process.

Weighted rule. The previous method assumes a primary uncertainty source for each sentence, which is not generic for real-world test cases that may contain multiple uncertainty sources. Instead, we measure the six defined uncertainty scores for a given text and assign a weight to each category by learning a linear regression to the previously fitted line. We evaluate the correlation between weighted rule scores and output lengths for inputs with the corresponding type of uncertainty in Fig. 2.2c. We observe that the dependency between uncertainty scores and output lengths noticeably increases, without more data points getting close to the trend line.

Lightweight model. While hand-crafted rules can capture certain uncertainty for sentences, they are heuristic methods and not comprehensive enough since the data distribution

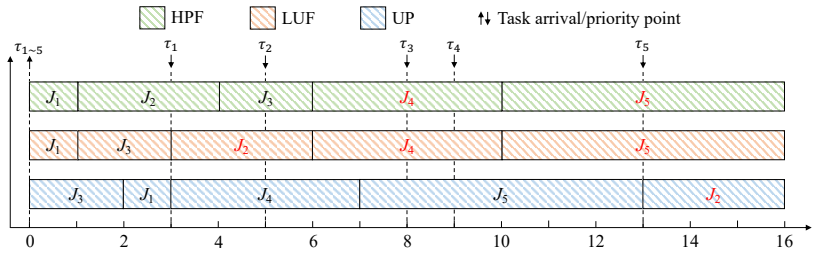


Figure 2.4: Prioritization example for HPF, LUF, and UP. J_i denotes the i-th task, τ_i denotes its arrival/priority point. Tasks depicted in red color denote those missing their priority points.

is not learned. To make such estimation more reliable, we introduce a data-driven black-box lightweight (LW) multi-layer perceptron (MLP) [127] that takes the six rule-based scores as features and predicts the output length for any given query. Specifically, we train a LW model on the training sets of four benchmark datasets and evaluate the correlation between its predictions and output lengths for unseen queries in the test sets in Fig. 2.2d. We observe the output lengths are almost linearly dependent on our predicted scores, with only few noisy samples. We further evaluate the correlation between the predicted uncertainty scores and averaged inference latency across different LMs on the four benchmark datasets in Fig. 2.3. The predicted scores are highly consistent with the inference latencies across all datasets, i.e., sentences with smaller uncertainties generally require larger inference latencies. This suggests that our method can precisely estimate LM execution times for any unseen query in real-world dialogue scenarios.

2.3.3 System-level Optimization Opportunities

We now illustrate several precious system-level optimization ideas enabled by leveraging uncertainty score metric.

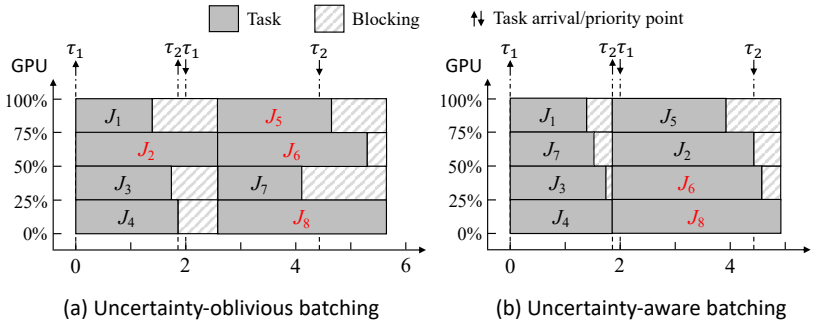


Figure 2.5: Comparison of (a) random batching and (b) consolidation using uncertainty on eight tasks with a batch size of four. τ_i denotes the arrival/priority point for the i-th batch. Tasks with red notations miss the priority points.

Prioritization. Online queries, though without intrinsic deadlines, have *priorities* (e.g., urgency of the task) that can be specified by `RT-LM` using the priority point parameter according to their estimated workloads. Leveraging the uncertainty score of each task (i.e., the estimated number of output tokens of each input), the scheduler shall make better prioritization decisions. Intuitively, prioritizing tasks that require shorter execution times and earlier priority points would improve throughput and timing correctness (often due to reduced head-of-line blocking), as illustrated in Fig. 2.4. In this example, five tasks that arrive at the same time (the length of each block presents its execution time) are scheduled by three strategies, namely Highest Priority Point First (HPF), Least Uncertainty First (LUF), and `RT-LM` utilizing Uncertainty-aware Prioritization (UP). As a result, HPF and LUF respectively miss two (J_4 and J_5) and three (J_2, J_4, and J_5) priority points, whereas UP misses only one priority point (J_2).

Consolidation. In any heavily-loaded systems requiring machine learning workload multi-tasking, batch execution is a commonly-used method to enhance response time and timing correctness. Our estimated uncertainty scores can assist deciding which tasks shall be batched and executed together to better utilize hardware resources. Fig. 2.5 describes this

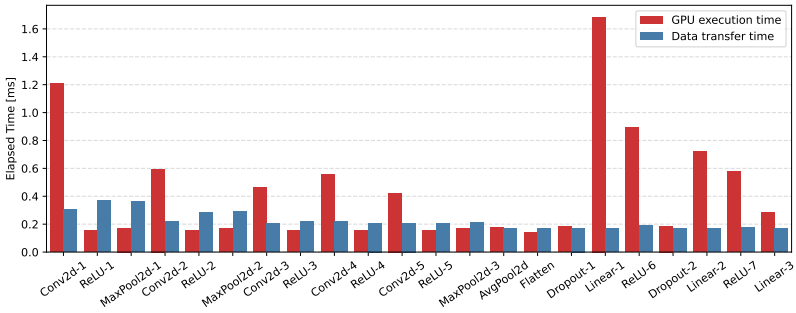


Figure 2.6: Data transfer time (offloading) compared to GPU execution time for AlexNet.

idea using an intuitive example comparing two batch executions for eight tasks with a batch size of four. Fig. 2.5a presents a schedule under uncertainty-oblivious batching, e.g., HPF where tasks in each batch have similar priority points. Four tasks (J_2, J_5, J_6, J_8) miss priority points with a fairly low GPU utilization. Fig. 2.5b describes uncertainty-aware batching, where tasks in each batch have similar uncertainty scores. Only two tasks (J_6, J_8) miss priority points with an improved GPU utilization and shorter response time.

Strategic offloading to CPU. Previous works [63, 67] and our experiments indicate that offloading machine learning workloads to CPU cores often introduces non-negligible communication and synchronization overhead, negating the benefits of parallel utilization of both CPUs and GPUs. Fig. 2.6 depicts an illustrative example, where we compare the layer-wise data transfer cost with layer-wise GPU execution times for running AlexNet [73]. As seen, data transfer takes nearly the same amount of time as GPU execution for the majority of layers. Nonetheless, under overloaded situations or scenarios containing computation-demanding workloads, `RT-LM` could identify such tasks by checking whether the estimated uncertainty scores exceeds a pre-defined threshold. The scheduler can then decide whether offloading such demanding tasks to CPUs can improve the overall efficiency of the system.

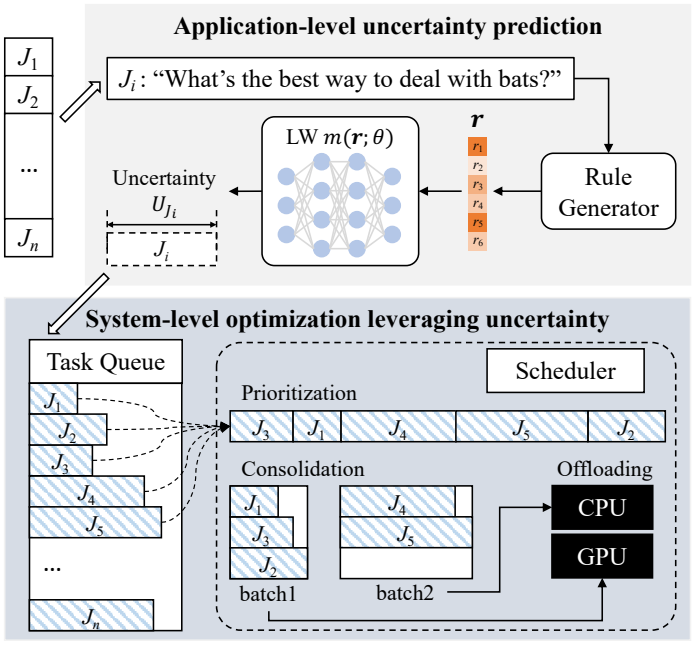


Figure 2.7: Design overview of RT-LM.

While it is likely that the negative impact due to offloading and communication can be totally negated by freeing up the precious GPU resource for executing other normal tasks, our intuition is to leverage uncertainty scores to reflect different levels of task demand and offload demanding tasks to CPU. This strategic offloading balances the workload between CPU and GPU, enabling efficient use of system resources and ensuring that the overall system remains responsive and productive.

2.4 Design of RT-LM

2.4.1 Design Overview

In this section, we illustrate the overall design of RT-LM, as shown in Fig. 2.7. RT-LM comprises two major components: an application-level framework that quantifies

task uncertainty, and a system-level framework that leverages this information for optimized scheduling (prioritization, dynamic consolidation) and resource allocation (strategic offloading).

Defined in Sec. 2.3.2, the uncertainty score of an input text reflects the required output length and thus, its execution times. Leveraging this critical uncertainty information of input texts, `RT-LM` develops an uncertainty-aware system-level resource manager that makes better scheduling decisions. To ensure timing correctness, `RT-LM` introduces an uncertainty-aware priority scheduler that takes into account both uncertainty scores and priority points of tasks to reflect how critical a task is. By smartly considering both factors, `RT-LM` is capable of improving the system's throughput. Moreover, `RT-LM` includes a runtime consolidation mechanism to enhance the system's latency performance through uncertainty-aware batching. Our uncertainty estimation aids in deciding which tasks should be batched together to better utilize hardware resources. The system dynamically forms batches of tasks with similar execution times by reordering the tasks in two adjacent batches according to their uncertainty scores. In this way, tasks within each batch have both similar criticality and latency, leading to improved GPU utilization and less response time. Lastly, `RT-LM` integrates strategic CPU offloading to handle highly-demanding or malicious workloads. By leveraging uncertainty scores to indicate the demand of a task, `RT-LM` strategically offload tasks that may potentially lead to overloaded situations on GPUs to the CPU core to maintain a balanced workload distribution across the system.

2.4.2 Uncertainty-aware Prioritization

For any given input J, our rule generator RuleGen$(\cdot)$ first yields a feature vector containing the intensity of the six linguistic uncertainties. Then our LW model m_θ takes the feature vector and predicts the final uncertainty score:

$$u_J = m_\theta(\textsc{RuleGen}(J)) \tag{2.1}$$

In some scenarios such as conversational AI in healthcare [13], if an LM request has a user-specified deadline t_J, `RT-LM` can specify the priority point parameter using that deadline (d_J in Eq. 2.3 is replaced by t_J); whereas most LM-assisted dialogue systems do not have such user-specified deadlines. Based on our observations in Fig. 2.2e where longer inputs generally induce longer outputs, we empirically define a priority point for each task according to its input length $d_J = \varphi_f|J|$, where φ_f is a coefficient that projects input length to the latency of an LM f.

A straightforward way of factoring in both uncertainty and priority point into a system is to use the concept of "slack" (ζ), which measures the remaining time until the priority point:

$$p_J = \frac{1}{\zeta_J} = \frac{1}{d_J - r_J - \eta_f \cdot u_J} \tag{2.2}$$

Here r_J, d_J denote the arrival time and priority point of the task, respectively. The term u_J presents the uncertainty score of the task, reflecting the estimated output length, while η_f is a coefficient that projects output lengths to latencies, regarding the LM

f. This slack-based approach prioritizes urgent tasks that are close to their priority points, which is suitable for systems with stringent priority point constraints and relatively stable task execution times.

However, for on-device LM systems facing workloads with high variability in uncertainties, such as input texts with a large uncertainty range causing LMs to generate outputs with varied lengths, a more flexible approach that can prioritize tasks with shorter execution times when needed ensures more predictable and consistent system performance. In `RT-LM`, we design Uncertainty-aware Prioritization (UP) where each task is assigned a priority p_J that reflects its weighted criticality:

$$p_J = \frac{1 - \alpha \cdot u_J}{d_J - r_J - \eta_f \cdot u_J} \tag{2.3}$$

Here α is a system-level hyper-parameter that provides a control over the impact of uncertainty on the priority. Specifically, $d_J - r_J - \eta_f \cdot u_J$ represents the estimated slack for the execution of the task, and $\alpha \cdot u_J$ is a scaled uncertainty score. The fraction computes the estimated execution time after considering the scaled uncertainty, normalized by how much time is left, to represent the criticality of a task. The intuition behind this priority assignment is that a task with a shorter slack window or smaller uncertainty score should have a higher priority. This ensures that tasks with imminent priority points or short execution times are attended to promptly, enhancing the likelihood of meeting their priority points. The factor α provides a level of adaptability to the system. A larger value of α implies that the system is placing a higher emphasis on tasks with lower uncertainties, regardless of how soon their priority points are, while a smaller α value reduces the impact

of uncertainty on the priority calculation, placing a higher emphasis on the remaining time until the priority point. We search an optimal α value from 0 to 2.0 with a increment of 0.1 by testing the corresponding response time (see Fig. 2.13a).

2.4.3 Dynamic Consolidation

In the dynamic consolidation process, we aim to enhance the overall system efficiency by executing batches of tasks with similar estimated uncertainties, as they are more likely to have comparable processing requirements. The intuition is that executing tasks with similar workload characteristics as a batch can potentially lead to better resource utilization and reduced overheads, as illustrated in Fig. 2.5. Specifically, we maintain a queue of tasks sorted by the priority based on our UP algorithm (Eq. 2.3). We then group tasks with similar uncertainty scores together by introducing two hyper-parameters, λ and b. Among them, b determines the number of tasks to consider for a batch. Given a pre-defined batch size C, once the current batch accumulates $b \times C$ tasks from the task queue, we reorder these tasks according to their uncertainty scores. We then select the top-C tasks from this reordered list for execution. This mechanism ensures that tasks are executed in an order that prioritizes higher urgency as well as shorter execution times. Additionally, parameter λ controls the maximum allowable ratio in uncertainty scores between tasks within a batch. As we traverse the sorted list of tasks within the current batch, if we encounter a situation where the uncertainty score of the current task is more than λ times that of the previous one, we segment the list at this point. The tasks preceding this point are executed as a batch, while the remaining tasks are returned to the queue for future processing. The whole consolidation process unfolds as follows: (1) Maintain a queue of tasks ordered by

descending priority, based on the UP algorithm. (2) Once accumulating $b \times C$ tasks in the current batch, reorder them in accordance with their uncertainty scores. (3) Traverse the reordered batch of tasks. If the uncertainty of a task exceeds λ times the uncertainty of the previous task, or if the batch size C is met, segment the list at this point. (4) Execute the tasks before the segmentation point as a batch, while returning the remaining tasks to the queue.

Dynamic consolidation provides flexibility in adjusting to varied workload characteristics and system conditions through the adjustment of the parameters b and λ. For instance, in scenarios where tasks exhibit diverse uncertainty scores, a smaller b or larger λ can be utilized to ensure that only tasks with similar uncertainties are grouped together. Conversely, if tasks have similar uncertainty scores, a larger b or smaller λ will form larger batches, potentially achieving higher system throughput. Moreover, dynamic consolidation can help balance the trade-off between throughput and predictability. By executing tasks with similar uncertainties as a batch, the system may exhibit more predictable behaviors, as estimating the execution time of a batch is often simpler than predicting individual task execution times. Meanwhile, by executing tasks in batches, the system can potentially achieve higher throughput compared to executing tasks individually.

2.4.4 Strategic Offloading to CPU

In the dynamic consolidation process described above, tasks are assigned to batches and then executed based on uncertainty scores. However, such a process can lead to the situation where some tasks with high uncertainty scores (e.g., malicious, adversarial tasks) may potentially delay the execution of the whole batch, negatively affecting the overall

system performance. To address this, we propose a protective mechanism, termed 'strategic offloading', to offload potentially malicious tasks and execute them separately on CPU cores.

In our implementation, we define a parameter k ($0 < k < 1$) which denotes the top-k percentage of uncertainty scores in the training set to control the malicious threshold τ:

$$\tau = \text{quantile}_k\left(\{m_\theta(\textsc{RuleGen}(J))|J \in \mathcal{D}_{train}\}\right) \tag{2.4}$$

In essence, τ corresponds to the boundary of the highest k-percentile of uncertainty scores. If the uncertainty score of a task is larger than τ, it is offloaded to a CPU batch for separate execution. Otherwise, it is assigned to a GPU batch for grouped execution. Furthermore, we ensure that there is always a batch of tasks ready for execution. If the task queue is empty and there are remaining tasks in the GPU batch, these tasks are offloaded for execution. Similarly, if there are no tasks in the GPU and CPU batches, the remaining tasks from the task queue are offloaded to the appropriate execution batch based on their uncertainty scores. This strategic offloading mechanism provides a layer of protection against extreme execution times, ensuring malicious tasks do not excessively delay the execution of a batch and promising a more predictable and reliable system performance, particularly under workloads with high variability. By carefully controlling the offloading parameter k, this mechanism can be tuned to balance the benefits of grouping tasks for efficient execution against the potential delays caused by malicious tasks.

Table 2.2: Hardware platforms used in our experiments.

	Edge Server	**NVIDIA AGX Xavier**
CPU	96-core AMD EPYC 7352 24-Core Processor	8-core NVIDIA Carmel Armv8.2 64-bit CPU
GPU	NVIDIA RTX A4500	NVIDIA Volta GPU
Memory	512GB	16GB LPDDR4x
Storage	8TB SSD	32GB eMMC

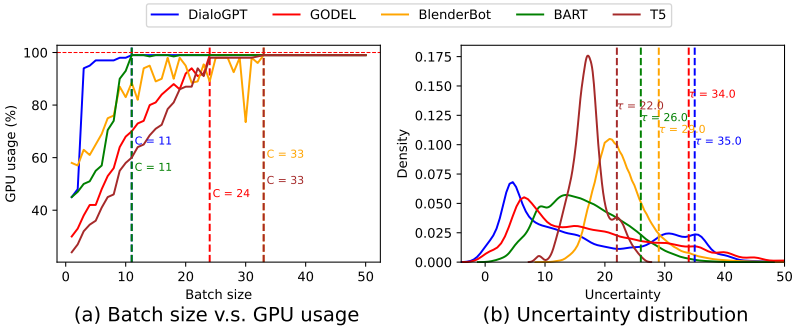


Figure 2.8: Offline decisions on (a) optimal batch size C and (b) malicious threshold τ ($k = 0.9$) for the five LMs.

2.4.5 Pseudo Code and Illustration

The whole framework of `RT-LM`, known as UaSched, takes several aforementioned control parameters, α, λ, k, and b, and operates in two main phases: offline profiling and online scheduling.

Offline profiling. The algorithm starts by initializing a LW regressor m_θ. For each task in the training set, RuleGen$(\cdot)$ generates rule scores $\mathbf{r}_J$, which is taken by m_θ as features and calculates the output length from the LM. The algorithm then minimizes the Mean Squared Error (MSE) between the estimated output lengths and the LM output lengths, thereby updating the LW model. It also records GPU utilization to determine the minimum batch size C_f for the LM $f(\cdot)$ that can better utilize hardware resources, e.g., when GPU usage

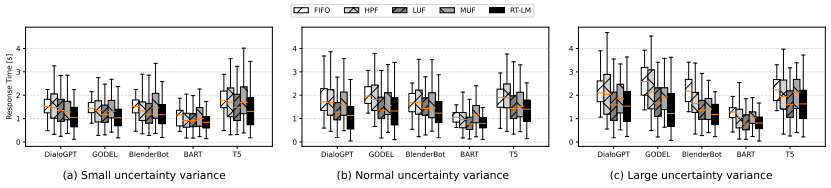


Figure 2.9: The distribution of response time across five LMs for sentences with (a) small, (b) normal, and (c) large uncertainty variance on the edge server.

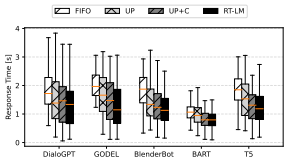


Figure 2.10: Ablation study of response time on the edge server.

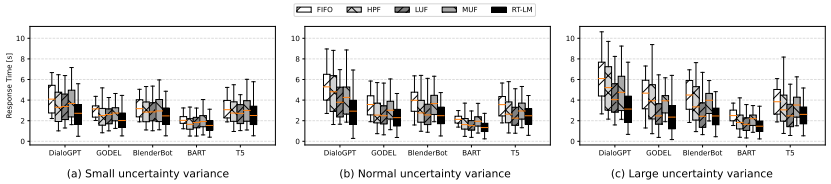


Figure 2.11: The distribution of response time across five LMs for sentences with (a) small, (b) normal, and (c) large uncertainty variance on the AGX Xavier.

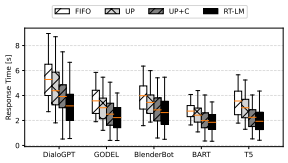


Figure 2.12: Ablation study of response time on the AGX Xavier.

reaches 100%. Finally, it determines the malicious threshold τ according to the uncertainty score distribution.

Online scheduling. The algorithm iterates over tasks in the test set, calculating uncertainty scores using the pre-trained LW model m_θ, and then placing them into a task queue. The tasks are then popped and processed in a descending order of priority scores. If a task's uncertainty score is greater than the threshold, it is offloaded to a CPU batch; otherwise, it is placed in a temporary batch. If the temporary batch reaches a size of $b \cdot C_f$, the scheduler sorts tasks in the batch in ascending order of uncertainty scores. It then segments the batch at a point where the current uncertainty score is larger than λ times that of the previous one or if the pre-defined batch size C_f has been reached. The segmented tasks are offloaded to a GPU batch, while the remaining ones are put back into the queue.

2.5 Implementation and Evaluation

2.5.1 Experiment Setup

Testbeds. We implement `RT-LM` and conduct an extensive set of experiments on an edge server, as shown in Table 2.2, simulating the single-device multitasking scenarios of online chatbots or services, and live-translation services.

Benchmark. We evaluate `RT-LM` across five state-of-the-art LMs that are widely used in dialogue systems—DialoGPT [167], GODEL [120], BlenderBot [130], BART [80], and T5 [126]—on four benchmark datasets: *Blended Skill Talk* [143], *PersonaChat* [166], *ConvAI2* [33], and *Empathetic Dialogues* [128]. We use the pre-trained versions of these models—*DialoGPT-medium*, *GODEL-v1_1-base-seq2seq*, *blenderbot-400M-distill*, *bart-base*, *t5-base* and annotated datasets released by Hugging Face[3].

Metrics. We evaluate `RT-LM`'s performance w.r.t. the average response time, throughput, and runtime overhead. We also delve deeper into the effect of different components of `RT-LM` on the system-level performance, the robustness of `RT-LM` against different parameter settings, and its effectiveness under different proportions of malicious tasks.

Hyper-parameters. For the offline profiling, we initialize a lightweight MLP which has four layers of hidden size [100, 200, 200, 100], and train the model with a learning rate of 1e-4. We record the average GPU usage for the five LMs with different batch sizes in Fig. 2.8a. Specifically, we choose an optimal batch size (i.e., minimum batch size that a LM can reach 100% GPU usage) of 11, 24, 33, 11, 33, for DialoGPT, GODEL, BlenderBot, BART, T5, respectively. We further record the distribution of uncertainty scores for each

[3] `https://huggingface.co/`

Table 2.3: Maximum response time (s) and percentage of improvement for sentences with small, normal, and large uncertainty variance on the edge server. The evaluated methods consist of uncertainty-oblivious (former) and uncertainty-aware (latter) ones. **Bold** numbers denote the best metric values among them.

Method	DialoGPT			GODEL			BlenderBot			BART			T5		
	Small	Normal	Large	Small	Normal	Large	Small	Normal	Large	Small	Normal	Large	Small	Normal	Large
FIFO	2.25	3.75	3.90	**2.15**	3.06	3.93	**2.24**	2.52	3.41	**1.87**	1.93	1.95	**2.90**	2.95	3.30
HPF	3.25	3.92	4.68	2.75	3.79	4.53	2.90	3.54	3.37	2.34	2.13	2.63	3.56	4.13	3.97
LUF	2.85	**2.77**	3.55	2.47	3.06	3.41	2.86	2.79	2.93	1.98	1.82	2.19	3.24	3.43	**3.17**
MUF	3.03	3.68	3.93	3.52	3.74	4.21	3.36	3.52	3.10	2.97	3.00	2.38	4.01	3.30	3.98
RT-LM	**2.24**	2.96	**3.18**	2.52	**2.80**	**3.17**	2.92	**2.26**	**2.38**	1.93	**1.66**	**1.86**	3.45	**2.64**	3.25
	-0.4%	-21.1%	-18.5%	+17.2%	-8.5%	-19.3%	+30.4%	-10.3%	-30.2%	+3.2%	-14.0%	-4.6%	+19.0%	-23.4%	-1.5%

LM in Fig. 2.8b, and select a malicious threshold of 35, 34, 29, 26, 22 for DialoGPT, GODEL, BlenderBot, BART, T5, respectively. We set the uncertainty-weight α as 1.0, the output-latency coefficients η as 0.05, 0.04, 0.1, 0.05, 0.04, and the input-latency coefficients φ as 0.08, 0.10, 0.13, 0.08, 0.07 across the five LMs for priority assignment; λ, b as 1.5, 1.8, respectively for dynamic consolidation; and k as 0.9 for protective mechanism. To gather necessary statistics, we employ the tegrastats utility for recording GPU and CPU memory usage. Additionally, we use Python's time library to track the arrival and end time of each task, as well as the latency incurred by `RT-LM`.

Workload setup. Real-world human-generated processes, such as phone calls to a call center, can often be represented as a Poisson process, where the number of arrivals within a specific time interval is governed by a Poisson distribution [29, 132]. Given the independent nature of user queries in our context, we adopt a similar model to simulate task arrivals. This model is principally defined by its average arrival rate, denoted as β (representing queries per minute). We generated synthetic traces by sampling inter-arrival times from an exponential distribution with differing mean $\mu = \frac{1}{\beta}$ to modulate the arrival rate. To create time-varying synthetic workloads, we continuously evolve the workload generator across different exponential distributions throughout the process. This involves iterating through integer

values of β ranging from 10 to 150. For each minute, we sample from the corresponding exponential distribution, ensuring a comprehensive representation of workload scenarios, from light-load phases to high-traffic peaks. Following the generation of these traces, we shuffle the test dataset and map them to the created arrival patterns. To enhance realism, acknowledging that users may require some time to complete a query, we introduced a wait time interval $\xi = 2$ seconds so that tasks arriving within this span are processed as either a single batch or multiple batches[4].

2.5.2 Latency Performance

We evaluate the latency performance of various strategies by calculating their response time – the time elapsed between a task's end time and its arrival time across the five LMs. Naturally, a lower average response time indicates a more efficient system. We compare `RT-LM` to the following baselines:

- First-In-First-Out (FIFO): Tasks are queued based on their arrival times, creating uncertainty-oblivious random batches with a fixed size for execution.
- Highest Priority-Point First (HPF) [102]: Tasks with higher priority points are prioritized. This approach batches tasks with similar priority points together, maintaining a fixed batch size, yet remains uncertainty-oblivious.
- LUF: Tasks with lower uncertainty scores are given precedence. Those with comparable uncertainty scores (or execution times) are batched together using a fixed size.

[4]We conducted supplementary experiments using diverse sets of μ and ξ values. The findings consistently align with the trends observed in Fig.2.9∼2.11.

- Maximum Uncertainty First (MUF): This strategy prioritizes tasks with higher uncertainty scores. Those with analogous scores are batched together with a set size.

To gauge the impact of uncertainty on system-level performance, we evaluate all methods across three subsets of tasks featuring small, medium, and large variance of uncertainty scores on the edge server. Fig. 2.9 demonstrates the distribution of response time values, while Table 2.3 records the worst-case response time for each method across task subsets. From our observations: 1) Uncertainty-aware strategies tend to surpass uncertainty-oblivious ones, especially when input data exhibits varied uncertainty scores. For the small-variance subset, all methods display similar response times in Fig. 2.9a, with the maximum values of LUF, MUF, `RT-LM` even larger than FIFO, HPF in some cases in Table 2.3, but on the large-variance subset, LUF, MUF, `RT-LM` consistently outperform FIFO and HPF. This is because when tasks exhibit similar workloads, all strategies essentially mimic FIFO. However, when there's significant variance in task uncertainty, grouping tasks with analogous uncertainty scores reduces the likelihood of computation-intensive tasks holding up the entire batch. 2) Generally, LUF produces a better performance than MUF. By prioritizing tasks with high uncertainty, MUF can inadvertently cause the entire system to lag, thus compromising average response times. 3) `RT-LM` consistently exhibits superior performance, achieving the most efficient response times across all LMs. The average response time of `RT-LM` is roughly 0.8s less than FIFO for BART in Fig. 2.9c; and its maximum response time is up to 30% smaller than FIFO for BlenderBot in Table 2.3. This suggests that considering both execution times and priority points in task prioritization can further optimize latency performance. This dual consideration ensures `RT-LM` is versatile across varied workload dis-

Table 2.4: Average throughput for sentences with small, normal, and large uncertainty variance on the edge server.

Method	DialoGPT			GODEL			BlenderBot			BART			T5		
	Small	Normal	Large	Small	Normal	Large	Small	Normal	Large	Small	Normal	Large	Small	Normal	Large
FIFO	21.68	18.00	15.68	17.89	18.28	13.36	21.15	17.90	17.63	32.30	30.62	25.92	17.86	16.57	16.15
HPF	20.26	19.27	16.41	19.55	18.69	13.99	21.26	18.28	17.32	**33.59**	30.22	26.56	18.10	16.75	17.18
LUF	23.19	21.09	19.97	19.71	19.17	17.68	**21.34**	19.48	19.81	32.86	31.02	28.52	19.75	18.94	18.84
MUF	22.40	20.06	19.44	19.14	18.34	16.76	21.20	18.92	20.08	32.03	31.28	27.97	19.58	17.78	17.52
RT-LM	**24.61**	**23.89**	**22.34**	**23.73**	**21.54**	**19.78**	21.12	**20.66**	**20.80**	32.14	**31.71**	**28.64**	**22.28**	**21.94**	**20.03**

tributions. 4) Larger LMs are more sensitive to variations in task uncertainty, requiring even more execution times for tasks with high uncertainty scores, thereby benefiting more from uncertainty-aware strategies, e.g., RT-LM improves the maximum response time over FIFO to a larger extent for GODEL and BlenderBot (20% and 30%) than other LMs.

2.5.3 Throughput Performance

We further evaluate the throughput of various strategies as the average completed tasks per minute, across the five LMs, on the edge server. As expected, a higher throughput implies a more efficient system. Table 2.4 summarizes the results on the three subsets. We observe the throughput profiles of all methods are highly consistent with their latency performance metrics. Specifically, uncertainty-aware strategies notably exhibit larger advantages over uncertainty-oblivious ones when the uncertainty variance of test inputs grows, e.g., RT-LM can process over 6 more tasks per minute than FIFO, with DialoGPT in the large-variance subset. Among these, LUF is generally superior to MUF. RT-LM, however, stands out by consistently outperforming all other strategies. Moreover, uncertainty-aware strategies, particularly on larger LMs, can significantly boost system efficiency, e.g., RT-LM boosts the average throughput by 10% to 30% for BART and GODEL.

2.5.4 Ablation Study

To elucidate the superiority of `RT-LM`, we conduct an ablation study investigating the individual contributions of each component of our method to the response time and throughput performance:

- Uncertainty-aware prioritization (UP): We compare uncertainty-oblivious prioritization strategies, namely FIFO and HPF, with UP for response time and throughput evaluation, respectively.
- Dynamic consolidation: We contrast UP (using static batching) with its dynamic consolidation counterpart (UP+C).
- Strategic offloading: We compared UP+C with `RT-LM`, which facilitates execution of malicious tasks on the CPU.

Fig. 2.10 illustrates the subtle improvements of each component-enabled method over its component-oblivious counterpart in terms of reduced response times on the edge server, with `RT-LM` consistently outperforming the rest. For example, UP achieves an average response time of 0.2$\sim$0.7s less than FIFO. This indicates all three components of `RT-LM` are integral to its superior performance. Notably, the performance boost derived from prioritization and consolidation is typically larger than offloading, e.g., the average response time gap between UP+C and `RT-LM` is smaller than other pairs in most cases. This suggests that our prioritization and consolidation are more consequential in improving efficiency. Interestingly, strategic offloading has slightly more significant impact on larger LMs, e.g., `RT-LM` reduces the average response time over UP+C to 0.4s for GODEL, while

their performance are nearly the same for BART. This is because computational demanding tasks have larger impact on sophisticated LMs, causing even more severely overloaded systems.

2.5.5 On-Device Evaluation

Emerging embedded devices, augmented with powerful computing capabilities and LM intelligence [137], have the potential to serve as the local central service in future smart homes. These devices may support hundreds of IoT devices, facilitating concurrent multi-user or multi-device (e.g., refrigerator, air conditioner) communications with a single LM, a concept known as connected intelligence. In this context, we delve into the performance evaluation of various methods on an NVIDIA AGX Xavier (see Table 2.2), which is widely used in various applications such as autonomous driving [68,69] and robotics [112–114,123], to reflect the feasibility of `RT-LM` in on-device multitasking scenarios.

Fig. 2.11 showcases the response time of all evaluated methods across three subsets on the AGX Xavier. The observed patterns largely mirror those seen on the edge server. For instance, uncertainty-aware strategies excel, particularly in subsets with diverse uncertainty characteristics. LUF is generally more efficient than MUF, `RT-LM` consistently outperforms other baselines across all LMs, and uncertainty-aware strategies derive greater efficiency benefits from larger LMs, such as GODEL. Furthermore, a comparative analysis between the two platforms reveals an interesting insight: high-performance devices, being quicker in execution, tend to display a smaller disparity in performance across different methods compared to embedded devices. This subtly hints at a diminished relative advantage for `RT-LM` on more powerful devices.

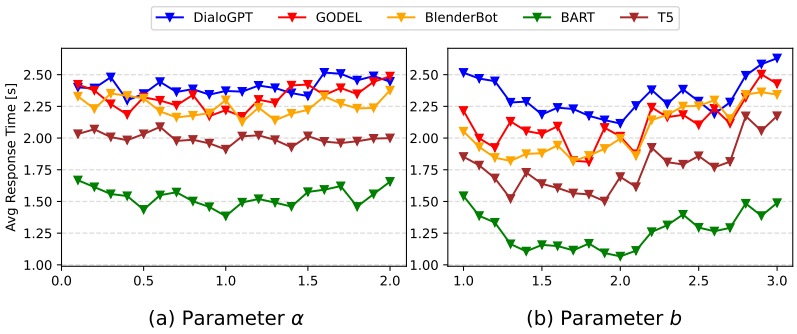


Figure 2.13: Study of average response time with different values of (a) α and (b) b across five LMs on the edge server.

Fig. 2.12 depicts the individual contributions of each `RT-LM` component, in terms of reduced response time on the embedded device. The findings align with on the edge server: all three components collectively boost its performance, prioritization and consolidation emerge as more influential factors in enhancing efficiency than offloading, and larger LMs generally derive more pronounced benefits from offloading.

2.5.6 Parameter Study

We explore the impact of two key hyperparameters, α and b, which control the influence of uncertainty in priority computation and the batch size determined by the number of tasks, on `RT-LM`. We vary α from 0.1 to 2.0 (with a fixed $b = 2.0$) and b from 1.0 to 3.0 (with a fixed $\alpha = 1.0$), incrementing by 0.1 in both cases, and assess the resulting average response time of `RT-LM` across different LMs.

Fig. 2.13a shows that `RT-LM` is robust to changes in α, with a maximum divergence in response time not exceeding 0.35s for each LM. This resilience indicates that UP functions as a well-balanced, uncertainty-aware priority, aptly mediating between priority points and execution times for tasks. An optimal α value of 1.0 is indicated by our performance

Table 2.5: An example of crafted sentence that causes DialoGPT to generate much longer outputs. *Italics* and ~~strike through~~ denote added and removed tokens, respectively.

Q: Not really. Let's ~~talk~~ *think* about food. What do you like to eat? I ~~love~~ *like* fish. **A**: I love fish too! What is your favorite kind? $\hat{\mathbf{A}}$: I like to eat fish too. What is your favorite kind? I like pasta, filipino, steak, etc. I talk a lot on IRC and it is fun to learn about it with some other guys.

metrics. Placing a higher emphasis on either uncertainty (larger α) or remaining time until the priority point (smaller α) results in a slight increase of response time.

Fig. 2.13b reveals that b has a more significant impact on latency performance than α, with the maximum deviation in response time reaching about 0.75s for T5. This indicates a considerable dependence of dynamic consolidation on the number of tasks considered for a batch. Optimal performance is achieved at $b = 1.8$. Values below or above this introduce inefficiencies, either mimicking static batching or causing delays in task completion due to longer wait time.

2.5.7 Evaluating Malicious Scenarios

To evaluate the robustness of `RT-LM` against malicious inputs, we apply a state-of-the-art adversarial attack method [87] that crafts provided input texts to elongate LM outputs. Table 2.5 presents an example of a malicious sentence designed to prompt an LM to generate longer output $\hat{\mathbf{A}}$ than the original one $\mathbf{A}$, leading to a computational burst and degraded system performance. Tasks are deemed malicious if their uncertainty scores exceed a predefined threshold (see Eq. 2.4). To assess the response, we control the proportion of

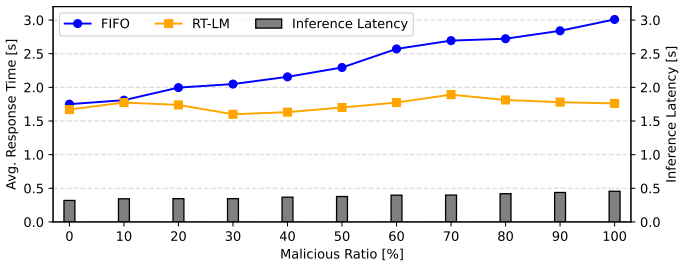


Figure 2.14: Average response time and LM inference latency on the edge server, under varying ratios of malicious tasks.

deliberately crafted malicious tasks within a range of 0% to 100%, increasing in increments of 10%, and evaluate subsequent system latency performance.

Fig. 2.14 shows the effects of varying ratios of malicious tasks on the average response time of both FIFO and `RT-LM`, as well as the associated average inference latency across different LMs. As seen, `RT-LM` is proficient in managing extreme conditions wherein a large proportion of malicious tasks need to be processed, outperforming the uncertainty-oblivious FIFO. When the malicious task ratio exceeds 30%, FIFO exhibits high sensitivity, with the average response time increases from around 2.0s to 3.0s. Whereas `RT-LM` is resilient against malicious tasks, maintaining a steady average response time of around 1.5$\sim$1.9s. Our results confirm that `RT-LM` effectively prevents malicious tasks from hindering the execution of other critical tasks. This resilience enhances `RT-LM`'s suitability for applications like chatbots [176], personal assistants [153], and conversational AI in healthcare [13] where defense against adversarial attacks is crucial.

2.5.8 Overhead Analysis

Analyzing overhead is crucial in practical real-time systems which are more complicated and variant. A solution with high overhead may undermine response time and

Table 2.6: Latency and memory of offline profiling.

LM	Total LW latency (s)		Memory
	Train	Ratio	Train
DialoGPT	351	3.01%	14,607 MB
GODEL	490	3.96%	14,768 MB
BlenderBot	448	3.71%	14,723 MB
BART	392	3.25%	14,631 MB
T5	369	3.06%	14,639 MB

Table 2.7: Latency, memory, and CPU/GPU utilization of online scheduling. Prior., consol., and off. denote prioritization, consolidation, and offloading.

LM	Avg. per-task latency (ms)				Memory	CPU / GPU util.
	Prior.	Consol.	Off.	Ratio	Test	Ratio
DialoGPT	8.04	0.42	0.37	2.10%	11,293 MB	97% / 92%
GODEL	7.78	0.43	0.49	2.04%	12,795 MB	93% / 97%
BlenderBot	9.24	0.53	0.40	2.39%	12,136 MB	99% / 95%
BART	7.84	0.35	0.10	2.06%	11,979 MB	97% / 91%
T5	8.39	0.33	0.18	2.27%	11,653 MB	95% / 90%

throughput, as the scheduling process may severely block task execution. We present an analysis of both latency and memory usage introduced by `RT-LM` on the edge server, offering insights into the practical efficiency of our design.

Offline Profiling. We initialize an LW model and train it for 100 epochs, using the LM outputs as ground truths. We report both the average training time per epoch and its proportion relative to the LM inference time. Memory usage during this phase is also recorded. As shown in Table 2.6, our training consumes merely around 3$\sim$4% of the LM inference latency, and less than 3% of the total available memory (512 GB), demonstrating the overhead efficiency of `RT-LM`.

Online Scheduling. We evaluate the average per-task latency of each component of `RT-LM` and compare the combined latency to the LM inference time. We also record the average memory usage as well as CPU/GPU utilization during online scheduling. Table 2.7

reveals that RT-LM introduces less than 3% additional latency overhead relative to the LM inference time (around 415 milliseconds per task). Such small overheads are unlikely to affect real-time dialogue systems noticeably. Notably, prioritization accounts for the majority of scheduling time, as uncertainty is computed and queued at this stage. For all LMs, CPU/GPU utilization reach over 90%, which suggests effective resource allocation under RT-LM.

2.6 Related Work and Discussion

2.6.1 Real-time DNN Inference.

Recent research has improved real-time Deep Neural Network (DNN) performance with strategies optimizing performance-accuracy trade-offs [9,11,175], and exploring system design for DNN execution [10, 62, 63, 65, 67, 96, 109, 159]. Despite these advancements, previous works neither consider the dynamics of DNNs for different execution times of inputs. In contrast, our proposed method, RT-LM, builds upon these existing scheduling algorithms by incorporating uncertainty estimation to further enhance performance and resource allocation.

2.6.2 Uncertainty Estimation.

Uncertainty estimation has been a topic of interest in the machine learning and NLP community, particularly in the context of deep learning [118]. Methods like Monte Carlo dropout [43] and Bayesian neural networks [70] have been proposed to quantify the uncertainty in model predictions. Previous works [36,92,105] also show that uncertainty may

cause an LM to generate outputs with varied lengths. Our method employs a lightweight regressor to estimate the uncertainty in terms of the output length of an LM inference, which can be used to inform the scheduling process, improving resource utilization and response time.

2.6.3 Intelligent Edge Server Systems.

In cloud-edge-client hierarchical systems, AI models are co-deployed on the cloud and edge servers [137, 161], where multiple requests from diverse users via edge devices can be processed concurrently by the DNNs. Notable examples of such applications include online chatbots and live translation services. Additionally, cloud servers frequently grapple with load balancing across multiple workers [50]. `RT-LM` could prioritize critical requests and redirect malicious tasks to CPU cores, thereby enhancing overall system performance and reducing the threat of performance attacks against DNNs [16, 18, 19, 22, 54, 87].

2.6.4 Limitations of `RT-LM`.

`RT-LM` mainly targets system-level optimization in heavy-workload scenarios, emphasizing concurrent task processing by taking into account the uncertainty characteristics of each task. In real-world on-device LM-embedded systems, where queries typically arrive sequentially, there's room for further improvement, e.g., optimizing performance for each individual task by leveraging the correlation between uncertainty and layer-level LM inference/training efficiency could be pursued. Additionally, our current approach is designed for single-machine scenarios. Expanding to hybrid deployment setups, such as server-edge combinations, is an avenue worth exploring. Moreover, `RT-LM` doesn't account for mem-

ory and power constraints, which could cause potential out-of-memory (OOM) issues on edge environments and pose challenges when deploying on low-power devices. Although deep learning compilers [20, 21] may mitigate the challenges posed by limited resources in such scenarios, adapting `RT-LM` to work efficiently in memory-constrained edge settings and optimizing LM inference from a power-efficiency standpoint is an area yet to be addressed.

2.7 Conclusion

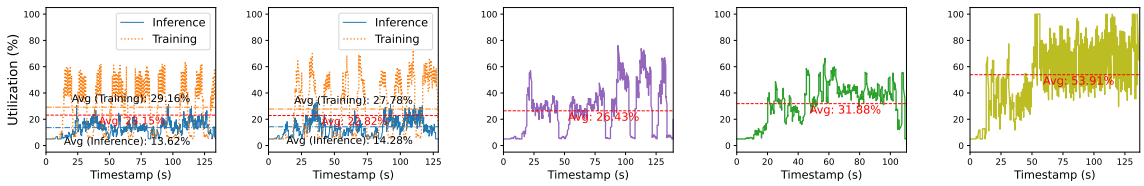
In this paper, we introduced `RT-LM`, a novel uncertainty-aware resource management for real-time on-device LMs. Our extensive evaluations demonstrated the superior performance of `RT-LM` in terms of response time, system throughput, and robustness to various system settings, while maintaining low overhead and excellent memory efficiency. In the future, we will focus on further optimizing the uncertainty estimation mechanism and expanding the applicability of `RT-LM` to more diverse and dynamic real-world workloads.

Chapter 3

LeMix: Unified Scheduling for LLM Training and Inference on Multi-GPU Systems

3.1 Introduction

Recent advances in large language models (LLMs) have underscored the importance of *continuous* adaptation to maintain alignment with evolving data and user feedback [154, 163, 172]. Techniques like test-time training [3, 146] exemplify the potential for LLMs to improve dynamically during deployment. However, achieving this adaptability requires *concurrent* training and inference workloads, which are traditionally partitioned and seperated across dedicated infrastructures [26, 76, 152]. For instance, platforms such as Amazon SageMaker provide isolated environments for training and serving, where models



(a) SEPARATE (2-2) (b) SEPARATE (1-3) (c) SEPARATE (dynamic) (d) NAIVEMIX (e) LEMIX

Figure 3.1: GPU utilization of three SEPARATE setups, NAIVEMIX, and LEMIX when deploying Llama-8B on eight A100 GPUs under LMSYS workloads. SEPARATE (2-2) and SEPARATE (1-3) dedicate 2 (1) nodes to inference and 2 (3) nodes to training, while SEPARATE (dynamic) alternates between these configurations based on request rates. NAIVEMIX and LEMIX co-locate both workloads across all four nodes.

are periodically retrained on new data and redeployed to inference endpoints [4]. While this SEPARATE setup simplifies management, it inherently results in resource inefficiency.

Potential resource inefficiencies. In distributed systems, resource under-utilization arises from both independent training and inference workflows. On the inference side, advanced serving systems [2, 27, 40, 74, 145, 157, 174] employ continuous batching [164] to handle dynamic request arrivals. Although these approaches improve throughput and response times under high traffic, they often assume sustained demand, leading to *serving idleness* during off-peak periods when resources remain frequently inactive. On the training side, parallelism techniques [7,25,45,75,97,107,141] are widely used to accommodate the resource demands of LLMs by vertically partitioning computations across distributed systems. While these approaches improve utilization, they introduce sequential dependencies in forward and backward passes, causing *pipeline idleness* (*a.k.a.* "bubbles") [60] during execution.

Figure 3.1a empirically demonstrates inefficiencies of the SEPARATE strategy running a Llama-8B model [149] across four nodes (servers) with eight A100 GPUs. Each

model instance is divided into two stages on a node, with two nodes allocated to serving inference requests based on real-world traces from the LMSYS Chatbot Arena [170,171], and the remaining two nodes dedicated to training. We observe that inference suffers from low average GPU utilization at just 13.62% due to inactive request periods. Despite a higher utilization of 29.16%, distributed training exhibits interleaving patterns caused by pipeline idle intervals.

Joint inference and training. To address these inefficiencies, we first propose NaiveMix, a strategy that *co-locates* training and inference workloads on shared nodes via a simple round-robin (RR) policy. As shown in Figure 3.1d, NaiveMix increases the average GPU utilization by around 8% over Separate through dynamically interweaving inference and training tasks. This increased utilization arises from the overlap of inference and training workloads, enabling otherwise idle GPU resources to be utilized more efficiently. Moreover, NaiveMix enables *on-the-fly* adaptation for concurrent workloads, where training updates immediately enhance the accuracy of subsequent co-located inference tasks, eliminating the delay associated with periodic inter-node model synchronization [40] required in Separate setups.

However, naïve co-location alone offers limited gains, with NaiveMix achieving an average of 31.88% GPU utilization. One reason is that NaiveMix is not sufficiently *fine-grained*, leading to co-execution interference between asynchronous workloads. For example, unfinished training backward passes can race and delay subsequent inference requests, especially under high traffic. Furthermore, real-world workloads exhibit *dynamic* request arrival patterns [145], with *heterogeneous* request lengths [2,174] that exacerbate execution

delays and additional GPU idle periods. During high traffic, these inefficiencies compound as memory contention induces prolonged overruns, causing degraded responsiveness and potential service level objective (SLO) violations [139].

A comprehensive solution. Recognizing these challenges, we propose LeMix, a fine-grained framework for co-operating LLM training and inference workloads in distributed systems. LeMix aims to: (1) maximize resource utilization, (2) improve inference accuracy via continuous retraining, and (3) meet response time SLO. Unlike NaiveMix, LeMix integrates task-specific execution awareness and runtime adaptability to dynamically balance these objectives.

Specifically, LeMix leverages an offline profiler to gather latency and memory coefficients on each hardware (§3.4.1). These profiled results enable the system to speculate the execution behaviors of online tasks and their system-wide impact (§3.4.2), while also providing actionable insights for co-optimizing multiple objectives during resource allocation (§3.4.3) and runtime execution scheduling (§3.4.4).

As a result, LeMix can *consolidate* workloads onto fewer nodes during periods of light demand—such as low request rates or reduced training intensity—without compromising SLO compliance. As shown in Figure 3.1e, LeMix significantly improves the average GPU utilization by around 22% over NaiveMix, through fine-grained node partitioning and runtime scheduling based on real-time workload conditions (e.g., request rate, retraining burden). This adaptability ensures LeMix remains robust in real-world scenarios with dynamic arrival patterns and heterogeneous computational demands.

Contributions. We evaluate LeMix on diverse models and hardware, including three GPT models on A6000 GPUs and three Llama models on A100 GPUs, leveraging model parallelism under both synthetic and real workload traces. LeMix delivers up to 3.53× higher throughput, 0.61× lower inference loss, and 2.12× higher SLO attainment over Separate under various conditions. Our contributions are:

- To our knowledge, this is the first study of concurrent LLM training and inference in distributed systems, unveiling inefficiencies in both Separate (§3.2) and NaiveMix (§3.3) setups and identifying optimization opportunities.
- We propose LeMix, a dynamic task-specific framework that coordinates mixed workloads, optimizing resource utilization, inference accuracy, and response SLO compliance under diverse real-time conditions (§3.4).
- We extend the scope of co-location techniques to support a wide range of LLM serving (e.g., autoregressive generation) and parallel training (e.g., data parallelism) scenarios, demonstrating the generality of LeMix (§3.5).

3.2 Background and Analysis of Separate

Recent surge in deploying LLMs for continuous user interaction [138] has necessitated effective strategies for handling concurrent training and inference tasks. These workloads span responding to user queries and retraining aimed at aligning LLM outputs with human preferences [28,90] and ensuring factuality [30,42]. A widely adopted approach to manage such workloads in distributed systems is the Separate strategy, which dedicates servers in different phases to either inference or training tasks [26]. While this setup sim-

plifies management and ensures inference accuracy, it exhibits resource inefficiencies under realistic workload conditions.

3.2.1 System Model and Terminology

Table 3.1: List of key terms used in the paper.

Stage	Sharded sub-model (e.g., layers 0-8) on a specific GPU.
Node	A single server containing multiple GPUs (e.g., 8×A100).
Task	An inference or training operation across stages within a node, where an inference task is forward-only, while a training task includes a forward + backward.
Batch	Internal mini-/micro-batching strategies within each task. They do not affect how LeMix views or schedules tasks across nodes.

Table 3.1 includes a system model, defining key terms such as task, node, and stage of execution in this paper. We assume tasks are independent (no inter-task communication), and their arrivals are externally triggered (e.g., by user queries).

3.2.2 Empirical Analysis of Separate

Figure 3.1b highlights the limitation of Separate under another static "1-3" partition, allocating one node for inference and three nodes for training. While this setup slightly increases the average inference utilization to 14.28%, training utilization drops to 27.78% compared to the "2-2" partition, leading to negligible overall gains. These observations suggest that altering the fixed partition of nodes alone provides limited benefits.

To address workload variability, we further evaluate a more adaptive *dynamic* Separate strategy that alternates between "1-3" and "2-2" setups based on request rates ("1-3" for less than 50 rps and "2-2" otherwise), as shown in Figure 3.1c. However, even with these adjustments, the average utilization only improves marginally to 26.43%, indicating

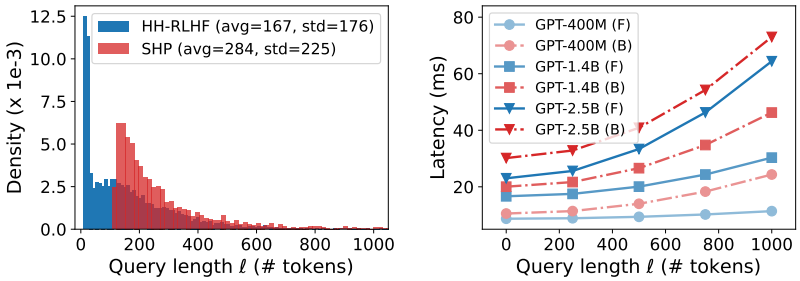


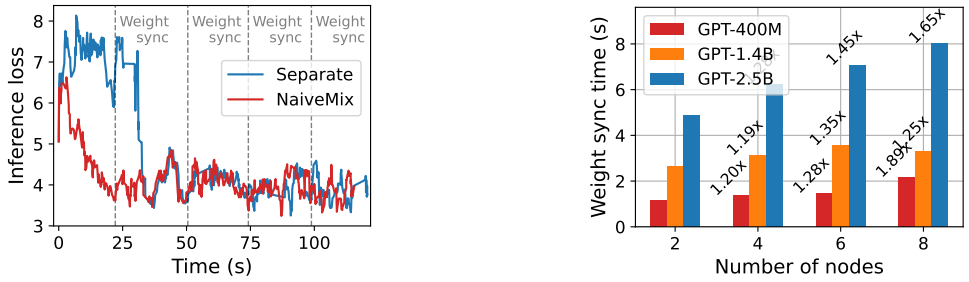
Figure 3.2: *Left:* The length distribution (w/ standard deviation) of two datasets and *Right:* the forward (F) and backward (B) latency running GPT models on a single RTX A6000 GPU.

the persistent inefficiencies of partitioning regardless of node allocation. These inefficiencies stem from two key challenges: serving idleness due to dynamic request arrivals and training pipeline idleness caused by workload heterogeneity.

3.2.3 Two Main Sources of Inefficiencies in Separate

Serving idleness in dynamic environments. Modern LLM deployments operate in a *multitenant* environment with highly *dynamic* request patterns across multiple node instances [145]. To address the significant memory demands of LLM inference, model parallelism (MP) [64] and continuous batching [164] are widely employed, wherein each GPU hosts a sharded model slice (*a.k.a.* stage) and sequentially executes forward passes of a mini-batch to reduce latency through statistical multiplexing [97]. While MP enables serving large models on memory-constrained devices, it exacerbates GPU idle periods during light traffic. For example, under low request rates in Figure 3.4a, idle GPU intervals, such as between tasks 2 and 3 on Node 1, are frequent, leaving significant portions of the pipeline underutilized. This inefficiency becomes more pronounced in large-scale distributed systems, where per-node request rate is even lower during off-peak hours [48].

Pipeline idleness from workload heterogeneity. To meet the high computational costs of LLM training, pipeline parallelism (PP) [7, 60] extends MP to divide mini-batches into smaller micro-batches that are processed in parallel to maximize utilization. While asynchronous PP (A-PP) [45] theoretically minimizes idleness by continuously processing micro-batches without flushing, LLM tasks with context-specific inputs introduce a unique challenge: *workload heterogeneity*. As shown in Figure 3.2, real-world user preference alignment datasets: HH-RLHF [8] and SHP [35], exhibit significant variability in query lengths, leading to diverse forward and backward latencies due to attention computation [151]. This variability disrupts the seamless execution of A-PP, as overlapping forward and backward passes contend for GPU resources, leading to execution delays and increased idle periods. For example, on Node 2 of Figure 3.4a, task $2'$ ($\bigtriangledown$) begins execution on GPU2 (S2) much later due to interference from the backward pass of task $1'$. Similarly, task $5'$ ($\diamondsuit$) cannot leverage the pipeline idle periods left by task $3'$ or task $4'$. This postponed inter-stage execution is termed as *far dependency* [25], as opposed to *immediate dependency* when tasks have the same lengths and execute without delays. These inefficiencies accumulate over time, resulting in prolonged E2E latency and fragmented memory. Although increasing concurrency, such as splitting mini-batches into more micro-batches, partially mitigates idleness, the associated communication overhead [7] and far dependencies impose fundamental limits on system efficiency.



(a) Continuous vs. periodic retrain

(b) Inter-node sync latency

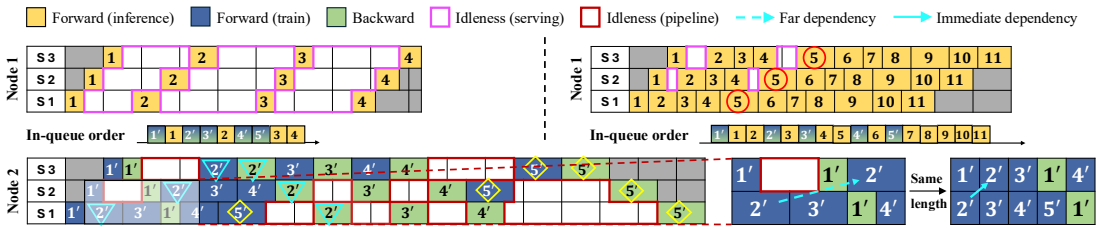
Figure 3.3: Real data showcasing (a) inference loss over time and (b) weight synchronization latency across nodes for SEPARATE.

3.3 Motivation of Workload Co-Location

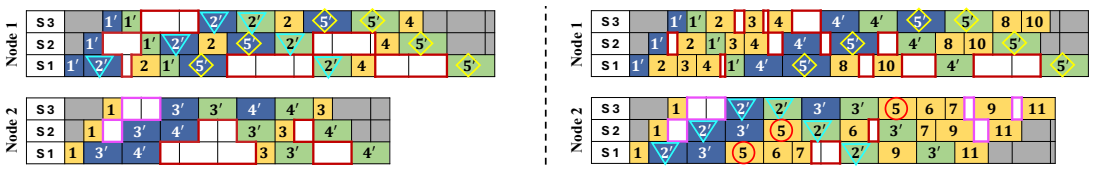
The evolving in-context learning capabilities of LLMs [154] present an opportunity to optimize resource utilization by learning while serving inference requests. We propose that *co-locating* training and inference workloads on shared resources mitigate the inefficiencies inherent in SEPARATE. A baseline strategy, NAIVEMIX, assigns tasks to nodes using a fair Round-Robin (RR) policy based on their in-queue order, as shown in Figure 3.4b. Inference tasks are queued according to their online arrival times, while training tasks are queued according to the first-stage (S1) forward end times of previous training tasks. For example, in the right part of Figure 3.4, the first four tasks enqueued are $1'$, 1, 2, $2'$. Consequently, NAIVEMIX co-locates tasks $1'$ and 2 on Node 1, while allocating tasks 1 and $2'$ to Node 2 for execution.

3.3.1 Advantages of NaiveMix

De-fragmentation. NAIVEMIX effectively enhances resource utilization compared to SEPARATE by reducing idle periods across various real-time conditions:



(a) SEPARATE operates inference workloads under low (left) and high (right) request rates on Node 1 and training workloads on Node 2.



(b) NAIVEMIX co-locates both workloads by assigning tasks in a Round-Robin (RR) manner based on their in-queue order.

Figure 3.4: Comparison of (a) SEPARATE and (b) NAIVEMIX under *Left*: low and *Right*: high request rates on a two-node cluster. For simplicity, we present one micro-batch for each mini-batch.

- At low request rates, NAIVEMIX alleviates serving idleness by running training workloads within the arrival gaps. For example, majority of training tasks $3'$ and $4'$ on Node 2 are executed in the interval between inference tasks 1 and 3.
- At high request rates, NAIVEMIX reduces pipeline idleness by inserting inference workloads into underutilized stages of training pipelines. For example, task 8 on Node 1 utilizes the idle pipeline period left by tasks $4'$ and $5'$.

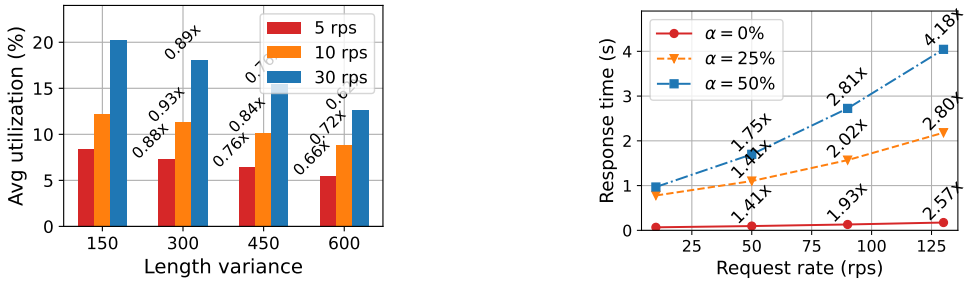
Latency reduction. By distributing training tasks across potentially more nodes, NAIVEMIX naturally relaxes the far dependencies caused by forward-backward interference in A-PP. For example, task $5'$ completes much earlier than in SEPARATE training. These optimizations in utilization consequently reduces E2E latency—the cumulative execution time across all nodes for executing a given workload, enabling more GPU time to be devoted to actual computations.

Enhanced serving quality. NaiveMix continuously improves inference accuracy by co-locating both workloads, which allows model instances to be updated in near real-time, benefiting serving quality on shared nodes. For example, under low traffic, the performance of task 4 on Node 1 is enhanced due to updates after the backward pass of training task $1'$. Conversely, as shown in Figure 3.3a, Separate requires periodically synchronizing (e.g., checkpointing) updated weights [40] from training to inference nodes, sacrificing serving quality before each communication. The synchronization overhead is also non-trivial and grows with model size and cluster scale (e.g., the number of nodes), as illustrated in Figure 3.3b.

3.3.2 Optimization Opportunities

While NaiveMix alleviates multi-source inefficiencies of Separate, its effectiveness is limited due to coarse-grained task-agnostic execution awareness.

Suboptimal resource utilization. NaiveMix still leaves spaces for utilization optimization. For example, low *training rate*—the proportion of training tasks in the queue—limits the reduction in serving idleness when insufficient training tasks are available to reduce the serving idleness if distributed across more nodes, such as between tasks 2 and 4 on Node 1 in Figure 3.4b under low request rates. Moreover, workload heterogeneity remains a challenge. In Figure 3.4b, task 1 is assigned to Node 2 under low request rates. However, allocating task 1 to Node 1 would instead utilize the idle periods left by task $1'$. Figure 3.5a validates this impact on NaiveMix's utilization across subsets with different length variances sampled from HH-RLHF [8]. We observe a consistently decreasing GPU utilization under highly-variable query lengths, particularly under heavy traffic. These inefficiencies



(a) Workload heterogeneity (b) Traffic intensity

Figure 3.5: Impact of (a) workload heterogeneity on utilization under different request rates and (b) request rate on serving responsiveness under different training rates α.

highlight the need for a *fine-grained* policy that accounts for task-specific contributions to GPU idle periods during resource allocation.

Prolonged serving response time. Naïve co-location inevitably enlarges inference response times due to resource contention between training and serving, especially during high-traffic periods. In Figure 3.4b, task 5 (◯) on Node 2 takes longer to complete compared to its execution on Node 1 under SEPARATE. Our empirical results of deploying GPT-2.5B under NAIVEMIX in Figure 3.5b show a prolonged inference response time as request and training rates grow, due to more frequent co-execution interference. A fine-grained approach is needed to estimate task-specific response times and *prioritize* inference tasks that risk violating SLO targets.

Memory contention in overloaded systems. NAIVEMIX lacks runtime awareness of resource availability, leading to potential memory contention [155] when concurrent workloads interfere in their pipelines. Under heavy traffic, autoregressive inference workloads may overlap with computation-intensive backward passes, especially when longer sequences and KV cache [122] are involved. These simultaneous workloads can trigger memory overruns and further extend latencies, which is compounded by the unpredictable behaviors of

accelerators under high load, such as non-deterministic scheduling delays and memory fragmentation [129]. Therefore, a runtime profiler is needed to monitor resource usage, defer or offload memory-blocking workloads to ensure SLO compliance.

3.4 Design of LeMix

From §3.3, we discuss that LLM systems with concurrent workloads benefit from joint training-inference and can be further optimized by task-specific scheduling. We propose LeMix, a fine-grained framework for co-operating LLM training and inference workloads in distributed systems, addressing inefficiencies such as serving idleness and pipeline delays. As shown in Figure 3.6, LeMix first conducts **offline profiling** to derive latency- and memory-sensitive coefficients for estimating hardware-dependent execution latencies (§3.4.1). Using these profiled results, LeMix speculates the idle impact and serving latency of incoming tasks on each node through **task-specific execution planning** (§3.4.2). These predictions aid in determining task preferences for each node, which are integrated with global task prioritization for **hierarchical resource (node) allocation** to balance utilization and SLO compliance (§3.4.3). After assigning tasks, LeMix executes them while mitigating memory contention through **runtime scheduling**, selectively offloading caches, and balancing the memory demands of concurrent workloads (§3.4.4).

3.4.1 Offline Profiling

To support efficient scheduling and memory-aware execution, LeMix starts with an *offline profiler* that captures the latency and resource characteristics of LLM workloads.

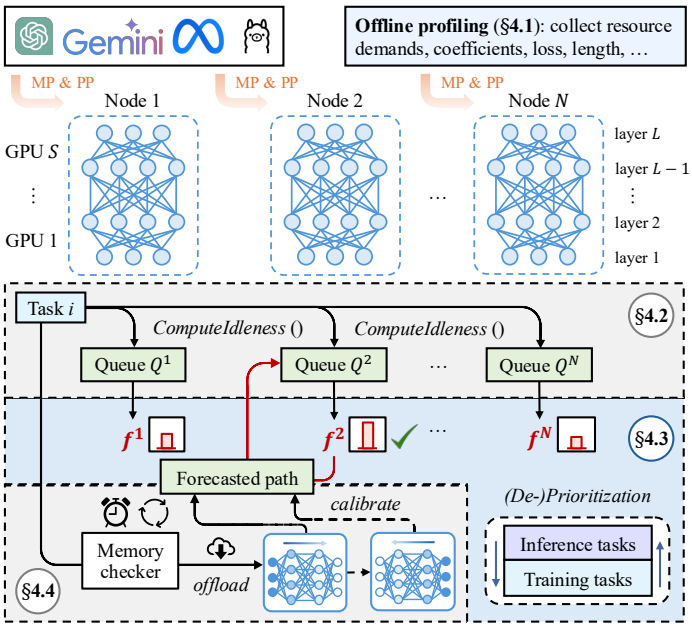


Figure 3.6: LeMix's components and their interactions.

Latency estimation. Forward and backward execution latencies scale quadratically with *query length* ℓ (Figure 3.2) and linearly with *batch size* C. We model Stage-level execution time as $\Delta_F = \eta_F^n \cdot C \cdot \ell^2$, $\Delta_B = \eta_B^n \cdot C \cdot \ell^2$, where η_F^n and η_B^n are hardware-dependent coefficients. These are obtained by measuring average stage runtimes across various batch sizes[1] and query lengths from the HH-RLHF [8] and SHP [35] datasets, normalized by $C \cdot \ell^2$.

Memory profiling. To preempt memory saturation, we profile peak GPU memory usage under synthetic workloads with increasing ℓ and C. To balance throughput and stability, LeMix defines a memory utilization threshold, $M_{\text{threshold}} = \kappa \cdot M_{\text{peak}}$, where κ is a safety factor. We select κ empirically to avoid memory overflows while preserving throughput and responsiveness under diverse real-time conditions. When memory constraints prevent immediate execution, tasks are delayed in the queue. However, excessive delay impacts responsiveness. Offline profiler empirically varies the maximum wait time $T_{\max}$ and identify

[1]For serving requests, we apply iteration-level FCFS continuous batching [164] to form inference mini-batches.

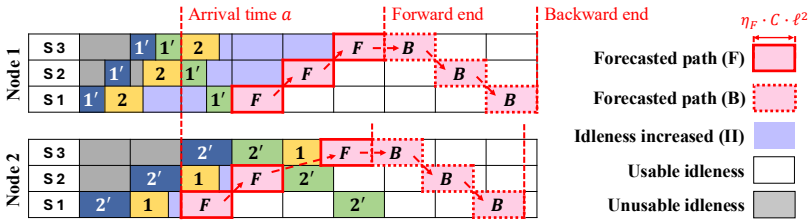


Figure 3.7: An illustration of how LeMix estimates task-specific II ( ) and response time R based on arrival time a, length ℓ, and batch size C through execution planning. Dashed lines indicate that backward operations are omitted for inference tasks.

a cutoff beyond which task latency degrades sharply. This value bounds deferrals before offloading.

Insights for online scheduling. The results of offline profiling are integrated into LeMix for online scheduling and resource management. Latency coefficients (η_F^n, η_B^n) enable fine-grained planning of forward and backward executions (§3.4.2), while memory thresholds ($M_{\text{threshold}}$) and waiting tolerance limits ($T_{\max}$) guide adaptive queueing and runtime scheduling decisions (§3.4.4). Additionally, offline profiler offers practical insights in *continuous retraining strategies*. By correlating training rates with model accuracy and SLO requirements, practitioners can adaptively adjust system priorities:

- If performance metrics (e.g., loss) are acceptable offline or SLO deadline is stringent, the system reduces the training rate during online to prioritize inference throughput.
- Conversely, resources can be prioritized for training to pursue accuracy gains while attaining SLO (§3.4.3).

3.4.2 Task-Specific Execution Planning

When a task arrives in a distributed LLM system, the first challenge is to assess its impact if executed on a specific node. This involves anticipating both resource utilization

and response latency introduced by the task. To address NaiveMix's limitations (§3.3.2), LeMix adopts fine-grained execution planning of the task by considering its arrival (enqueued) time, execution latencies, and inter-stage dependencies, as shown in Figure 3.7. Specifically:

- *Idleness increased* (II): LeMix distinguishes between "usable" and "unusable" idle periods on each node. Task-specific II measures the resource utilization gap created by the new task's execution, which transforms previously "usable" idle periods into "unusable" ones.
- *Response time* (R): the estimated latency from the task's arrival time to the completion of its forward pass.

Their calculation is detailed in Algorithm 1.

Initialization. For each node n, LeMix maintains two trace queues Q^n_{train} and $Q^n_{\text{inference}}$, which track the execution paths of ongoing tasks. The forward path of a new task is initialized based on the end time of the preceding task $task_{\text{prev}}$ and the immediate dependency. For each stage s, its forward start time is the later of two events: its forward end time in the preceding stage $s-1$ or $task_{\text{prev}}$'s forward end time in the current stage (line 5). The forward end time is incremented by an estimated forward execution latency (line 6).

Far dependency rescheduling. To efficiently account for far dependencies caused by co-execution interference, LeMix introduces a temporary queue, Q_{temp}, which tracks pending training tasks in Q^n_{train}. For each training task in Q_{temp}:

Algorithm 1 COMPUTEIDLENESS

1: **Input:** number of GPUs per node S, trace queues $Q^n := Q^n_{\text{train}} \cup Q^n_{\text{inference}}$ for each node n, new *task* batch: $\{a, \ell, C\}$
2: Parameters: forward and backward coefficient η^n_F, η^n_B
3: Initialize: II $\leftarrow 0$, $task_{\text{prev}} \leftarrow Q^n[-1]$, $Q_{\text{temp}} \leftarrow Q^n_{\text{train}}$
4: **for** each stage $s \in \{1...S\}$ **do**
5: $\quad task.\text{start}^s_f \leftarrow \max(task.\text{end}^{s-1}_f, task_{\text{prev}}.\text{end}^s_f)$
6: $\quad task.\text{end}^s_f \leftarrow task.\text{start}^s_f + \eta^n_F \cdot C \cdot \ell^2$
7: $\quad offset \leftarrow 0$
8: $\quad$ **while** $Q_{\text{temp}} \neq \emptyset$ **do**
9: $\quad\quad task_{\text{train}} : \{a_{\text{train}}, \ell_{\text{train}}, C_{\text{train}}\} \leftarrow Q_{\text{temp}}.\text{dequeue}()$
10: $\quad\quad$ **if** $task.\text{end}^s_f \leq task_{\text{train}}.\text{start}^s_b$ **then**
11: $\quad\quad\quad Q_{\text{temp}} \leftarrow Q_{\text{temp}} \cup \{task_{\text{train}}\}$
12: $\quad\quad\quad$ **break**
13: $\quad\quad task.\text{start}^s_f \leftarrow \max(task.\text{start}^s_f, task_{\text{train}}.\text{end}^s_b)$
14: $\quad\quad task.\text{end}^s_f \leftarrow task.\text{start}^s_f + \eta^n_F \cdot C \cdot \ell^2$
15: $\quad\quad$ **if** $task_{\text{prev}}.\text{end}^s_f \leq task_{\text{train}}.\text{start}^s_b$ **then**
16: $\quad\quad\quad offset \leftarrow offset + \eta^n_B \cdot C_{\text{train}} \cdot \ell^2_{\text{train}}$
17: $\quad\quad$ **if** $s = 1 \wedge$ CHECKEXECUTED$(task_{\text{train}}, s)$ **then**
18: $\quad\quad\quad$ Remove $task_{\text{train}}$ from Q^n_{train}
19: $\quad$ II $\leftarrow$ II $+ task.\text{start}^s_f - task_{\text{prev}}.\text{end}^s_f - offset$
20: R $\leftarrow task.\text{end}^S_f - a$
21: **return** II, R

- If the current forward can finish before the backward of the training task begins, the stage executes the forward pass during this idle pipeline period. The task is then reinstated in Q_{temp} for subsequent stages (lines 10-12).
- Otherwise, the forward start time is postponed to the backward end time of this interfered task, and the forward end time is recalculated accordingly (lines 13-14). Particularly, if this training task completes its backward pass, it is removed from the node queue Q^n_{train} to avoid recomputation in subsequent stages (lines 17-18).

All backwards executed between the previous and current forward (e.g., task $1'$ on Node 1 in Figure 3.7) are recorded, and their cumulative execution times are subtracted from the serving intervals to yield II (line 15-19). The response time R is estimated after forward path planning (line 20).

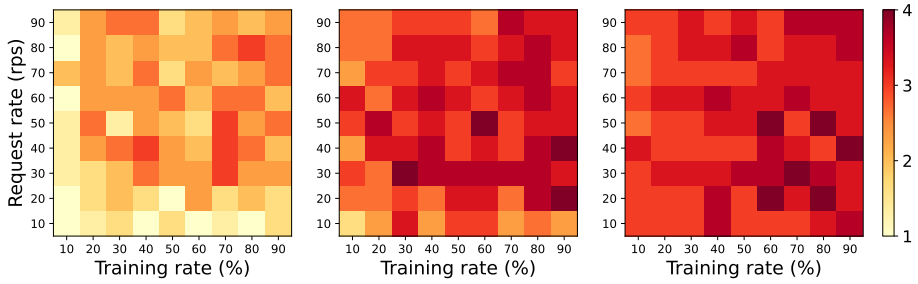


Figure 3.8: Number of allocated nodes (out of 4) by LeMix for GPT-400M (left), GPT-1.4B (middle), and GPT-2.5B (right), when processing concurrent workloads under various setups.

Backward planning. For tasks requiring training, the backward paths are initialized immediately after the forward paths. The backward start and end times are sequentially computed for each stage in reverse order, propagating back through the pipeline and maintaining a dependency chain between stages until the traversal ends. The planned backward paths aid in forecasting subsequent tasks before the execution.

3.4.3 Hierarchical Resource (Node) Allocation

Based on anticipated idle periods and response times for each node, LeMix optimizes resource allocation to address three objectives—maximizing resource utilization, minimizing serving response time, and improving serving quality under dynamic workload conditions. To achieve this, LeMix employs a two-level hierarchical policy:

- *Task-level* (local): New tasks are assigned to nodes with the highest priority scores, calculated using heuristic balancing the three objectives. For example, the new task in Figure 3.7, if assigned to Node 2, would produce lower II and R.
- *Queue-level* (global): LeMix dynamically adjusts task priorities in the global task queue, deprioritizing training tasks that risk delaying subsequent inference tasks (and violating SLO goals) at high request rates.

Task-level multi-objective node allocation. LeMix defines *idleness profit* (IP) to capture the utilization benefit of accommodating the incoming task to a specific node

$$\mathrm{IP} = -\max\left\{\frac{\mathrm{II}}{S} - (a - a_{[-1]}), \tau\right\}, \tag{3.1}$$

where $a - a_{[-1]}$ represents the *inter-arrival interval* between the preceding and incoming task, S is the number of GPUs per node, and τ is a threshold. IP penalizes nodes with large increased idle periods relative to arrival intervals, enabling LeMix to adapt to dynamic request rates—a key characteristic of modern LLM-serving systems [145].

LeMix incorporates *length consistency* (LC), which quantifies how well a task with query length ℓ aligns with a node's historical workload, as the serving quality heuristic

$$\mathrm{LC} = \frac{1}{\sigma_{<a}\sqrt{2\pi}} \exp\left\{-\frac{(\ell - \mu_{<a})^2}{2\sigma^2_{<a)}}\right\}, \tag{3.2}$$

where $\mu_{<a}$ and $\sigma_{<a}$ are the mean and standard deviation for previously executed tasks on the node. Higher LC scores align tasks with a node's workload profile, which has been shown benefiting training convergence [89, 95] and minimizing GPU idle periods caused by workload heterogeneity (§3.3.2). For inference tasks assigned to different nodes, LeMix minimizes cache miss rate by prioritizing *prefix reuse* [173] for length-bucketed queries, which keeps most requests on warm caches.

The node priority score integrates the three objectives

$$f = \frac{\mathrm{IP} + \lambda_2 \cdot \mathrm{LC}}{\lambda_1 \cdot \mathrm{R}}, \tag{3.3}$$

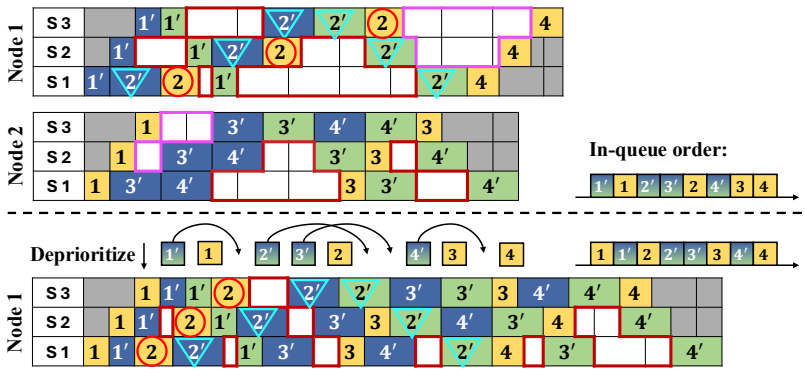


Figure 3.9: An illustration of how *Bottom:* LeMix consolidates workloads from *Top:* NaiveMix into one node to optimize utilization while maintaining SLOs via deprioritization.

where λ_1 and λ_2 balance the trade-off between response latency R and serving quality heuristic. Tasks are assigned to nodes with the highest priority scores, dynamically updating the node queues. This priority-based mechanism enables *workload consolidation*, dynamically adjusting the number of active nodes based on real-time conditions, as shown in Figure 3.8. Key observations include:

- Higher training rates lead to an increased II, as more unfinished backward passes create contention for resources, reducing IP and f for allocated nodes. This shift tasks to new nodes to alleviate resource contention.
- As request rates rise, arrival intervals shorten, amplifying II due to the added interference caused by workload heterogeneity, again lowering f and favoring new nodes.
- Larger model sizes exacerbate workload heterogeneity, as they magnify η_F^n and η_B^n (Figure 3.2). This increase in II reduces the profitability (IP) of assigning tasks to allocated nodes, instead favoring new nodes.

Queue-level task prioritization. While local-level selection optimizes individual assignment, it neglects the inter-task impact that arises under high-traffic scenarios. In such cases,

both bursty inference requests and training tasks compete for shared resources, preempting subsequent inference tasks and risking SLO violations. To maintain serving throughput, LeMix *deprioritizes* training tasks if the minimum estimated response time of its subsequent enqueued inference task $task_{\text{next}}$ across all nodes violates the SLO target

$$\min_{1\le n\le N}\{\max_{task\in Q^n}\{task_{\text{next}}.\text{end}_f^S\}+\eta_F^n\cdot C'\cdot {\ell'}^2\}-a'>\tau_{\text{R}}, \tag{3.4}$$

where C', ℓ', a' are the batch size, query length, and arrival time of $task_{\text{next}}$. τ_{R} is a response SLO goal, e.g., 5× inference latency. In Figure 3.9, LeMix defers training tasks $2'$ and $3'$ on Node 1, as either would significantly delay inference task 2. These severe delays, if arise from far dependencies in preceding tasks (e.g., task $2'$), also indicate under-utilization and could be optimized through this deprioritization process. Practitioners can set different SLO goals to balance serving responsiveness and training efficiency. A larger τ_{R} increases delay tolerance, preserving training priority for improving serving quality, while a smaller τ_{R} penalizes SLO violations more strictly, prioritizing serving responsiveness.

3.4.4 Runtime Memory-Aware Scheduling

After dispatching tasks to specific nodes, LeMix coordinates all ongoing tasks with the new one on each node, ensuring reliable service under dynamic traffic. To handle the risk of memory overruns—especially critical when co-locating computation-intensive batched operations—LeMix incorporates a runtime scheduler that maintains SLOs without sacrificing resource efficiency. Memory overruns occur when overlapping pipelines exceed the system's available memory, potentially delaying inference generation or disrupting training

Algorithm 2 ExecuteTaskMemoryAware

1: **Input:** number of GPUs per node S, task queues $Q^n := Q^n_{\text{train}} \cup Q^n_{\text{inference}}$ for each node n.
2: Parameters: Memory utilization threshold $M_{\text{threshold}}$, check interval Δ_t, and maximum wait time $T_{\max}$
3: **while** $Q^n \neq \emptyset$ **do**
4: $task \leftarrow Q^n$.dequeue()
5: **for** each stage $s \in \{1...S\}$ **do**
6: $wait \leftarrow 0$
7: **while not** MemoryAvailable(n, s, $M_{\text{threshold}}$) **do**
8: $wait \leftarrow wait + \Delta_t$
9: **if** $wait \geq T_{\max}$ **then**
10: Offload KV cache of *task* from GPU s
11: **break**
12: **if** MemoryAvailable(n, s, $M_{\text{threshold}}$) **then**
13: Forward(*task*, s)
14: Calibrate $< task.\text{start}^s_f, task.\text{end}^s_f > \in Q^n$
15: **if** $s = S \wedge task$.require_backward **then**
16: Backward(*task*)
17: Calibrate $< task.\text{start}^*_b, task.\text{end}^*_b > \in Q^n_{\text{train}}$

operations. To preemptively manage memory and mitigate these issues, LeMix employs a stage-level *wait-or-drop* policy, as shown in Algorithm 2.

Wait-or-drop with execution calibration. Runtime monitoring is conducted before each forward on a stage. When an incoming task's memory demands exceeds the device's capacity due to concurrency, it is placed in a temporary wait state (lines 6-8). By deferring execution until enough memory is available, this phase prevents immediate conflicts without disrupting ongoing tasks. It is particularly beneficial for autoregressive generation tasks, where low latency is critical. If the memory remains constrained beyond a profiled threshold $T_{\max}$, LeMix *offloads* intermediate activations and KV cache [122] to CPU [34] to scale down memory demands and free resources for inference to attain SLOs (lines 9-10). After execution, forecasted paths (§3.4.2) in the node queues are *calibrated* by the execution traces (lines 13–17), facilitating future task planning and scheduling optimization.

3.5 Implementation and Discussion

3.5.1 Implementation

LeMix is built on top of vLLM [74] and DeepSpeed [5] for MP and PP, and FlashAttention [31] for memory optimization during inference generation. We realize asynchronous execution of stages using Python *concurrent* library. Specifically, LeMix employs a global scheduler that makes scheduling decisions oriented to the node instances, according to the priority and memory load of them. It invokes each node as a *multi-threaded* execution environment, where each thread is assigned to manage a stage of the instance pipeline. For each node, tasks are scheduled across these threads following the forward and backward dependencies. Each thread is responsible for coordinating and executing the forward and backward (if applicable) jobs for its assigned stage. Once the memory availability for a task is confirmed, the appropriate thread invokes the execution process.

3.5.2 Model Update Synchronization

In Separate, model on the final training node is periodically checkpointed (e.g., every 100 training tasks) and loaded onto inference nodes to ensure consistent serving quality.

Co-location methods, such as NaiveMix and LeMix, adopt a *decentralized* strategy similar to federated learning where each node independently updates its local model instance based on its local training tasks. This setup resolves privacy concerns where sharing weight can be highly risky.

3.5.3 Scalability and Distributed Architecture

We assume an implicit client-server architecture. Real-world LLM systems (like Azure, AWS) do not rely on a centralized scheduler for every request. Instead, they adopt a distributed, hierarchical architecture that scales efficiently to millions of requests per minute. LeMix operates on both *cluster-level* and *GPU-level*, with its global queue stored in a cluster, and each local queue stored in a server (node):

- **Front-end load balancers**: Distribute client requests geographically across server clusters (e.g., by regions, time zones) using stateless heuristics like round-robin or SLO-aware routing to handle high traffic volumes (e.g., 10k+ rps) without per-request scheduling.
- **Cluster-level scheduler**: LeMix precisely allocate requests to specific servers within each cluster to coordinate training and inference tasks, which typically handles < 150 rps. This aligns with our experimental setups and reflects the practical scale of resource allocation at the cluster level. As shown in analysis (§3.6.6), LeMix incurs negligible overhead under this workload.
- **GPU-level runtime scheduler**: Once tasks are assigned to a server, LeMix manages execution using fine-grained techniques such as memory-aware batching, KV cache offloading, to optimize throughput and avoid OOM.

The efficiency of LeMix is rooted in its lightweight design, enabling scalability in large-scale distributed systems. Task-specific execution planning involves forecasting idle periods and response times for incoming tasks. This step incurs an $O(S)$ complexity per node, where S is the number of stages (GPUs) in the node. Resource allocation computes priority scores for all N nodes and selects the optimal node for each task, resulting in

Algorithm 3 ContinuousBatching

Input: Maximum batch size C, maximum waiting time T_w, task queue Q^n for a specific node n.
Initialize: batch $B^n \leftarrow \emptyset$, batch start $T_{start} \leftarrow$ current time
while True **do**
    if $B^n = \emptyset$ **then**
        $T_{start} \leftarrow$ current time
    while $Q^n \neq \emptyset$ **and** $|B^n| < C$ **do**
        $r \leftarrow$ get_next_request(Q^n)
        if not r.require_backward $\wedge$ $T_{start} + T_w >$ current time **then**
            Add r to B^n with padding (if necessary)
        else
            break
    if $B^n \neq \emptyset$ **then**
        Execute batch B^n on node n
        $B^n \leftarrow$ filter_finished_requests(B^n)

an $O(N \cdot S)$ complexity. Memory-aware scheduling adjusts task execution based on run-time conditions such as memory availability and task inter-dependencies, maintaining an $O(S)$ overhead per node by only considering active queues. Overall, LeMix 's end-to-end scheduling operates with $O(N \cdot S)$ complexity across N nodes and S stages, ensuring efficient coordination even in large clusters.

3.5.4 Autoregressive Generation

Serving requests are handled using continuous batching into mini-batches up to a maximum size C on a FCFS basis, as shown in Algorithm 3. To prevent excessive delays during low-traffic periods, a maximum waiting time T_w (defaulting to 0.5× inference latency) ensures timely execution even when the batch size C is not reached. In particular, for autoregressive generation, we adopt hybrid iteration-level batching [164]. Each incoming request represents a prefilling workload, which is integrated with ongoing decoding workloads into mini-batches to improve throughput and responsiveness. In LeMix, prefilling

Table 3.2: Model configuration and latency requirements.

Name	Size	# Layers	Hidden size	Forward	Backward
GPT-400M	1.0GB	12	768	0.03s	0.04s
GPT-1.4B	2.3GB	24	1024	0.08s	0.09s
GPT-2.5B	4.5GB	36	1280	0.12s	0.14s
Llama-8B	13GB	32	4096	0.11s	0.15s
Llama-13B	26GB	40	5120	0.24s	0.36s
Llama-70B	132GB	80	8192	0.73s	1.05s

requests, which initialize new sequences, are allocatable across nodes based on our task assignment strategy, informed by execution predictions of continuously batched task on each node. Decoding requests, which extend prefilled contexts, operate continuously on the same node as their corresponding prefilling tasks, maintaining data locality and supporting dynamic scheduling, such as offloading, when memory bottlenecks arise.

3.6 Evaluation

We evaluate LeMix across different LLM sizes and workload conditions, including diverse request rates and training rates. Experimental results show that LeMix consistently outperforms baseline systems in all scenarios, achieving up to 3.53× higher throughput, 0.61× lower inference loss, and 2.12× higher SLO attainment compared to Separate (§3.6.2). We further delve into node-level latency and response time breakdown under real workloads, showcasing how LeMix's dynamic resource allocation enhances utilization and operational efficiency (§3.6.3). The benefits of LeMix become even more pronounced under high workload heterogeneity, where task-specific scheduling proves critical to performance gains (§3.6.4). We also analyze fine-grained inference latency in different phases (§3.6.5). Finally, we conduct ablation studies to provide deeper insights into its techniques (§3.6.6).

3.6.1 Experimental Setup

Datasets and models. We use two popular preference datasets targeting different alignment domains: HH-RLHF [8] (harmlessness) and SHP [35] (helpfulness), to mimic the serving and alignment fine-tuning of LLM deployment. Specifically, we sample 1,000 query-reference pairs in each online test to evaluate the model and system performance. In this work, we consider GPT [168] and Llama [149] family models of various sizes. Table 3.2 shows their detailed configurations.

Testbed setups. We use two testbed configurations. For GPT models, we use 4 servers, each equipped with 2 NVIDIA RTX 6000 Ada GPUs (48 GB), 64 AMD EPYC-7543 CPU cores, and 2 TB memory. For larger Llama models, we use 4 AWS EC2 servers, each equipped with 2 NVIDIA A100 Tensor Core GPUs (80 GB), connected with pairwise NVLINK.

Synthetic workloads. We mimic human requests as a Poisson process [89, 97, 139] and synthesize traces by sampling inter-arrival times from an exponential distribution with mean of the reciprocal of the various request rates. Tasks are sampled from the same dataset as training (e.g., SHP) to simulate online serving and retraining scenarios.

Real workloads. We further explore the effectiveness of LeMix in more complex inference workloads constructed from the trace of LMSYS Chatbot Arena [170]—a real-world LLM serving platform for clients.

Metrics. We measure throughput as the average number of completed tasks per second and analyze the E2E latency across all nodes running the same amount of workloads, for utilization analysis. For serving quality, we evaluate the average inference decoding loss.

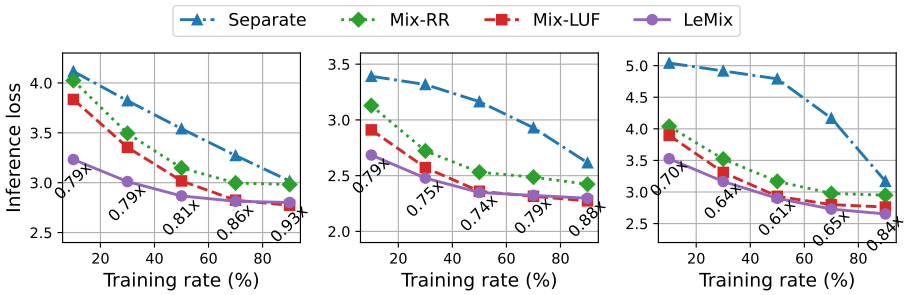


Figure 3.10: Average inference loss at various training rates for Llama-8B (left), Llama-13B (middle), and Llama-70B (right).

For serving efficiency, we evaluate prefilling latency, i.e., time-to-first-token (TTFT), decoding latency, i.e., time-between-tokens (TBT) [174], and SLO attainment—proportion of inference tasks whose TTFT meet a SLO deadline of 5× forward latency [97].

Baselines. We compare LeMix with three baselines as below:

- Separate [25, 97]: Nodes are partitioned into training and inference nodes based on training rates α. $N_{\text{train}} = \lfloor N \cdot \alpha + 0.5 \rfloor$ nodes are dedicated to training and the remaining to inference to balance workloads. Each task is assigned to a training or inference node in a RR fashion.
- Mix-RR (NaiveMix): Tasks are assigned to nodes in a sequential RR order. This method ensures a uniform distribution of tasks across all nodes.
- Mix-LUF: Lowest utilization first [38], serving as a strong baseline with more fine-grained execution awareness over RR, measures the GPU utilization and allocates tasks to nodes with the lowest average scores. Mix-LUF balances the active duration of all nodes over time, potentially preventing any single node from becoming a bottleneck.

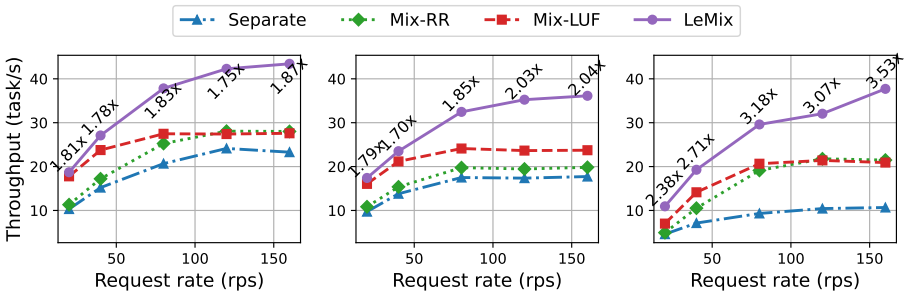


Figure 3.11: Throughput (task/s) across various request rates for Llama-8B (left), Llama-13B (middle), and Llama-70B (right).

3.6.2 Results on Synthetic Workloads

We assess LeMix and the baselines in three dimensions—*inference loss* of decoding human references to measure service quality under continuous retraining, *throughput* as a resource utilization metric when running mixed workloads, and *SLO attainment* to ensure prompt service responsiveness, under a spectrum of training rates for multiple models at different request rates on a four-node cluster.

Improved serving quality under retraining. Figure 3.10 shows Llama models' average inference loss when running concurrent serving and training workloads. We observe a general decreasing loss under larger training rates, as the model are retrained on more samples. All mixed approaches achieve a lower loss compared to Separate due to continuous updates from retraining (§3.3.1), with LeMix consistently outperforming the others, reducing the average loss by up to 0.61× over Separate. This superiority indicates that our idea of workload co-location and consideration of LC (§3.4.3) in LeMix benefits the training procedure and convergence. The discrepancy between Separate and co-location methods gets larger as model sizes grow, due to prolonged inter-node weight synchronization latencies (§3.3.1).

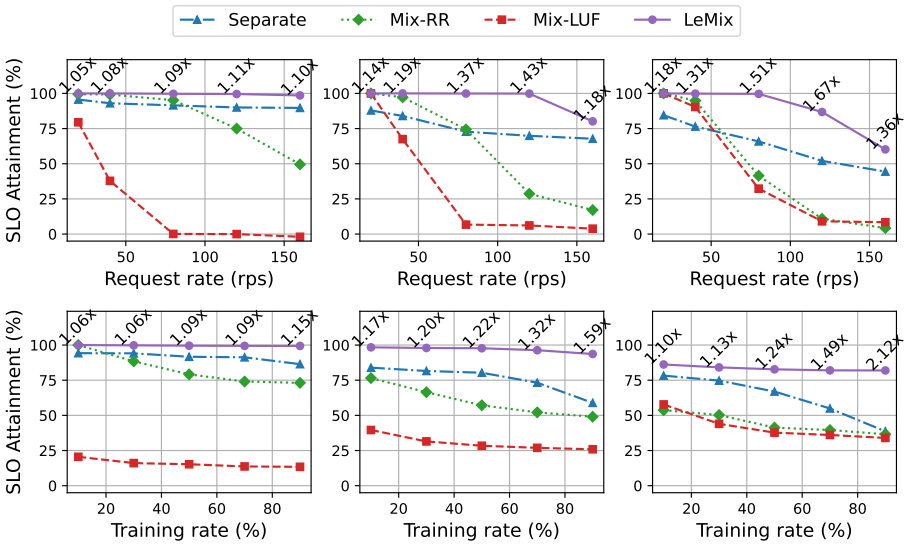


Figure 3.12: SLO attainment under various request and training rates for Llama-8B (left), Llama-13B (middle), and Llama-70B (right).

Throughput under varying traffic intensity. Figure 3.11 shows Llama models' throughput at various request rates when processing concurrent mixed workloads. We observe a notable enhance in resource efficiency for all methods at higher request rates, where LeMix achieves the overall highest scores, improving throughput by up to 3.53× over Separate. Particularly, Mix-LUF also excels under light traffic conditions (e.g., less than 50 rps), which demonstrates the benefit of execution awareness in optimizing resource efficiency (§3.3.2), though such improvement over NaiveMix diminishes in rapid task arrival due to the time-intensive GPU utilization querying procedure (Table 3.3) that significantly delays task execution. Moreover, the advantage of mixed methods over Separate is pronounced for larger models at high request rates, as the execution delays caused by workload heterogeneity are amplified under these circumstances (§3.2).

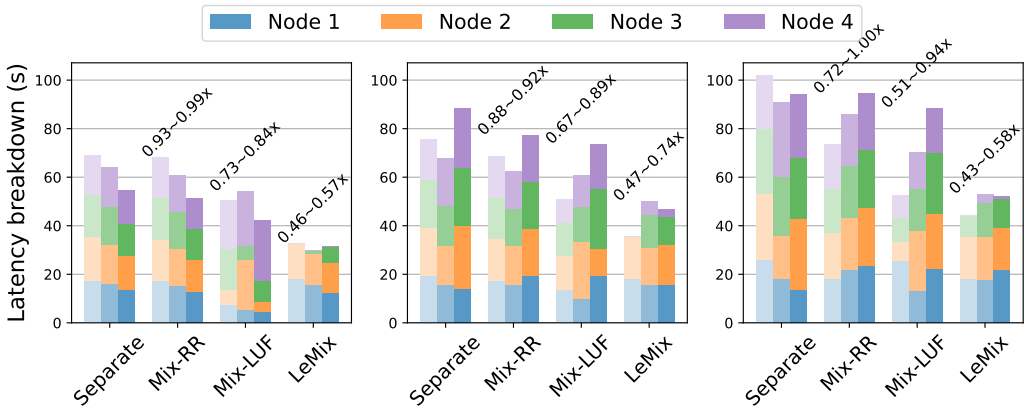
SLO attainment in varying conditions. Figure 3.12 demonstrates a decreasing SLO attainment for Llama models as request rates (top row) and training rates (bottom row)

increase. LeMix consistently outperforms alternative methods, maintaining high SLO compliance across workload spectrum. This resilience stems form its dynamic latency-aware scheduling which reduces response times for inference tasks. In contrast, other mixed approaches suffer significant drop in SLO attainment, particularly at higher request rates, as co-locating workloads exacerbates contention for shared resources, delaying inference execution (§3.3.2). While Separate achieves comparable performance to LeMix under low workloads, its static property leads to declines in SLO compliance under higher training rates or request intensities. The performance gap widens with larger model sizes (e.g., Llama-70B), where LeMix achieves up to 2.12× better SLO attainment over Separate at high training rates, indicating its scalability in managing heterogeneous workloads.

3.6.3 Breakdown Analysis on Real Workloads

We constructed real workload traces from LMSYS platforms following a similar process in [139], where we treat each LLM as a client and sample requests from the trace and re-scale the real-time stamps to a time window. During this serving window, training samples from the datasets are mixed to form in total 1,000 samples with varying training rates. We evaluate node-level breakdown of *E2E latencies* and *response time* for LeMix and the baselines.

E2E latencies across nodes. Figure 3.13 shows GPT models' E2E latencies breakdown on each node when processing real workloads. We observe a clear increased latency when running larger models due to more computational demands. All mixed approaches require less time than Separate to operate concurrent workloads, especially for larger models



(a) GPT-400M (b) GPT-1.4B (c) GPT-2.5B

Figure 3.13: Breakdown E2E latencies under training rates of 10% (light), 50% (medium), and 90% (dark color).

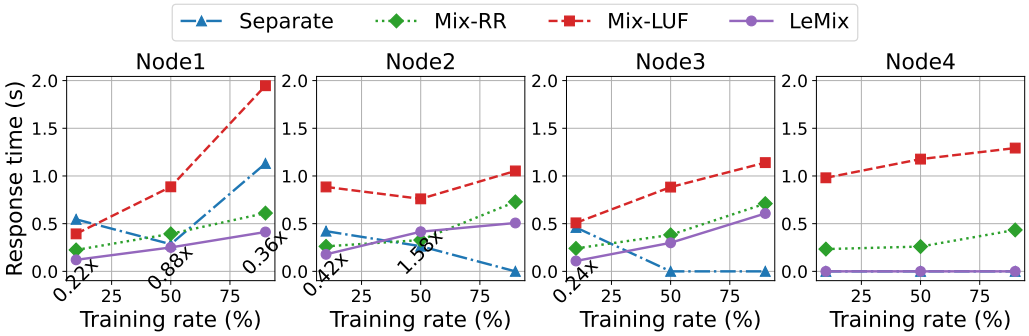


Figure 3.14: Breakdown average response time on each node. Zero values mean no inference workloads are allocated on that node.

(e.g., GPT-2.5B) where workload heterogeneity amplifies the pipeline idleness of SEPARATE (§3.2). LEMIX achieves the lowest accumulative latencies, reducing the scores by up to 0.43× over SEPARATE. One notable reason behind such gain is its ability to "dynamically" adjust resource allocation (§3.4.3) based on real-time conditions. For examples, LEMIX allocates only 2 nodes when running two smaller models under lower training rates (e.g., 10%), effectively mitigating the sparsity issue in low workload demands.

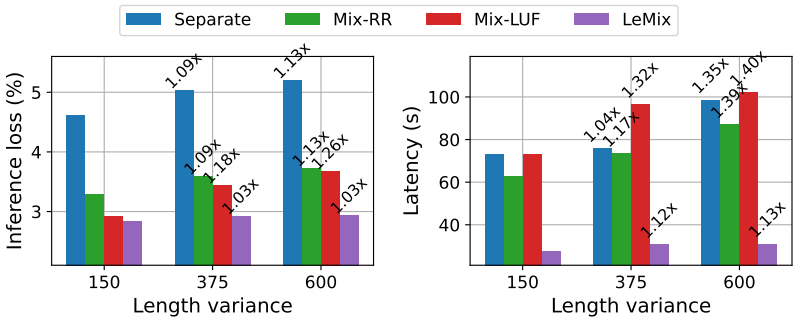


Figure 3.15: Impact of length heterogeneity on *Left:* inference loss and *Right:* E2E latencies under heavy traffic (150 rps).

Response time across nodes. Figure 3.14 details Llama-8B's average response time (TTFT) on four nodes across various training rates. By default, Node 1 is dedicated to inference workloads in SEPARATE, where LEMIX achieves lower response times, reducing the average scores by up to 0.22×. Such efficiency gain is more obvious under larger training rates where inference execution delays on Node 1 in SEPARATE are amplified due to overloaded requests. For the remaining three nodes, LEMIX still achieves the lowest response latencies under lower training rates (e.g., less than 50%), while SEPARATE excels under larger training rates since few or even no inference workloads are allocated to those nodes (e.g., Node 3). MIX-LUF exhibits significant response delays due to the latency of querying GPU utilization, indicating its limited practical applicability when SLO attainment is pursued.

3.6.4 Impact of Workload Heterogeneity

Rapid task arrival combined with larger workload heterogeneity could result in substantial idleness (§3.2) and hampered training convergence (§3.4.3) when operating synthetic workloads. To explore how LEMIX handles this property inherent in real-world

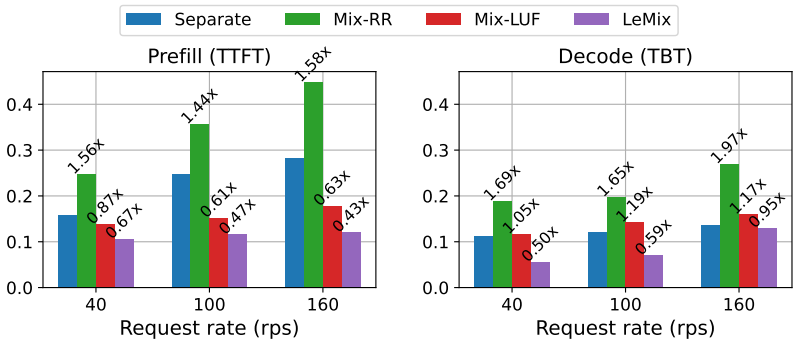


Figure 3.16: *Left:* prefill and *Right:* decode latency of inference serving under various request rates for GPT-2.5B.

workloads, we simulate a range of length heterogeneities by sampling subsets from the two datasets. We run GPT-2.5B model on each subset under a request rate of 150 rps across various training rates. Figure 3.15 demonstrates a general increase in loss and E2E latency for LeMix and the baselines as length heterogeneity grows. LeMix is more robust against workload heterogeneity compared to the baseline methods: with up to 1.03× increase in inference loss and the maximum latency gain is 1.13×. This suggests that task-specific resource allocation can effectively mitigate the idleness incurred by co-execution interference and training inconsistency caused by workload heterogeneity.

3.6.5 Inference (Generation) Efficiency Study

To independently study inference latency under mixed training workloads, we run GPT-2.5B with a memory pool size of 1000 tokens allocated for KV cache [122] across various request rates with 50% training rate. Figure 3.16 highlights the performance variations across methods in both prefilling and decoding phases. LeMix consistently achieves the lowest latency in both TTFT and TBT across all request rates, achieving up to 0.43× lower TTFT and 0.50× lower TBT compared to Separate. Both Mix-LUF and NaiveMix

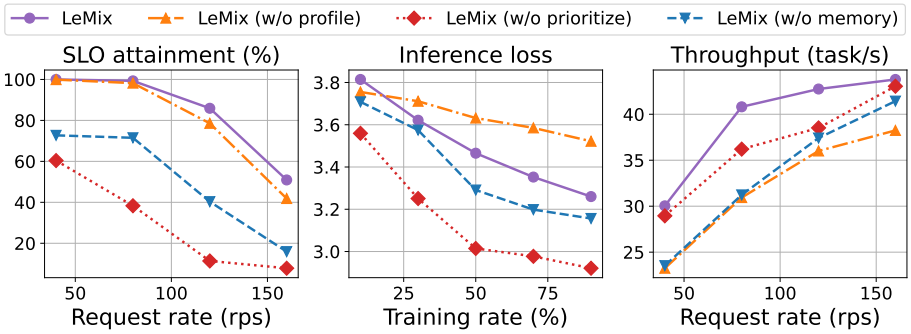


Figure 3.17: Ablation analysis of LeMix's components.

incurs additional serving latency when co-locating serving and training workloads due to resource contention, and their TBT discrepancies are caused by the hybrid batches of both prefilling and decoding requests where node allocation (§3.4.3) makes difference. Mix-LUF exhibits relatively better TTFT performance as the computation-intensive prefilling workloads are allocated to low-utilization nodes, but its TBT latency becomes a bottleneck due to frequent GPU utilization tracking during decoding. LeMix dynamically allocates prefilling workloads across nodes (§3.4.3) to prevent resource bottlenecks and cut TTFT latency, and leverages runtime scheduling (§3.4.4) to fill idle GPU slots and use batching opportunities to minimize TBT during decoding. This adaptability ensures balanced resource utilization across computation- (prefill) and memory-bound (decode) phases, allowing LeMix to excel in diverse conditions.

3.6.6 Understanding LeMix's Improvements

We run Llama-70B on synthetic workloads across varying request and training rates to evaluate the contributions of individual components in LeMix and their overheads.

Robustness to offline priors. LeMix employs offline profiling (§3.4.1) to improve the planning precision of execution latencies and memory demands, which enhances resource utilization. As shown in Figure 3.17, removing offline profiling (w/o profile) causes notable decrease in throughput, dropping by up to 0.82× compared to the full LeMix design, particularly under high request rates when misjudgments tend to be accumulated. Inference loss also increases from 3.3 to 3.6 at 75% training rate due to potential update delays for inference tasks. Despite these setbacks, LeMix (w/o profile) still outperforms baseline methods in higher throughput and SLO attainment due to fine-grained scheduling. This robustness to profiling accuracy highlights LeMix's applicability even when offline resources are limited.

Prioritization in responsiveness. Queue-level task prioritization (§3.4.3) in LeMix is essential for meeting SLOs under high request rates. As shown in Figure 3.17, removing prioritization (w/o prioritize) causes SLO attainment to collapse drastically from 95% to below 30% at 100 rps and further plummeting to 0% at 150 rps. This is caused by the uncontrolled interference from training tasks which delay inference requests. LeMix, in contrast, dynamically deprioritizes training tasks that risk SLO breaches, maintaining SLO attainment above 50% and improving throughput across the workload spectrum. Additionally, its slight drop in loss suggests the strength of dynamic scheduling where deprioritization sacrifices short-term accuracy improvements to ensure responsiveness.

Memory awareness in efficiency. Memory-aware scheduling (§3.4.4) enables LeMix to adaptively manage GPU resources and prevent memory overruns, ensuring system responsiveness. When memory awareness is disabled (w/o memory), SLO attainment degrades

Table 3.3: Average time overhead (ms) and memory (MB) of schedulers. EP, RA, MS represents execution planning, resource allocation, and memory-aware scheduling.

	Separate	RR	LUF	LeMix		
				EP	RA	MS
Latency (ms)	1.4e-2	1.5e-2	76	1.4e-1	1.4e-2	5.6e-2
Memory (MB)	1.6	1.8	727.5	9.6	1.9	1.8

significantly, dropping from 95% to 55% at 100 rps, as shown in Figure 3.17. Consequently, throughput drops from 8 to 6 at 80 rps due to memory-induced latency. By proactively offloading activations and enforcing memory thresholds, the full LeMix ensures stable performance across diverse workloads while balancing resource demands.

Overhead analysis. Table 3.3 shows the average latency and memory usage under a request rate of 50 rps and training rate of 50%. LeMix achieves its scheduling improvements with minimal computational cost. Specifically, execution planning incurs an average latency of 0.14 ms, slightly higher than simpler policies like RR, but still negligible compared to the model forward latency (Table 3.2). Meanwhile, resource allocation and scheduling achieve latency comparable to RR, ensuring real-time responsiveness. Memory usage remains similarly efficient. While methods like LUF incur high overhead (727.5 MB) due to system-level GPU utilization tracking, LeMix remains lightweight, consuming only 9.6 MB for its execution prediction and less than 4 MB for other modules. This efficient nature of LeMix ensures scalability under heavy traffic conditions and supports real-time decision-making without introducing performance bottlenecks.

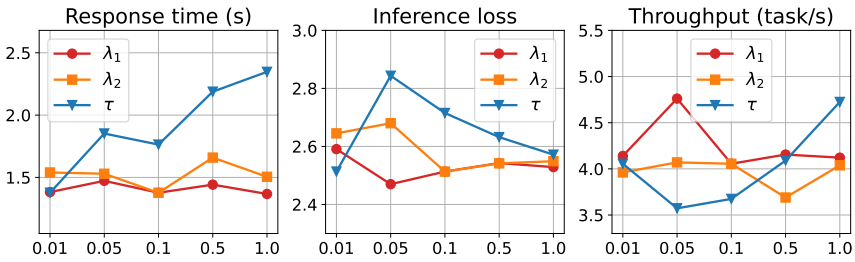


Figure 3.18: Trade-offs in LeMix's multi-objective allocation.

3.6.7 Parameter Study

Figure 3.18 illustrates the trade-offs between response time, serving quality (inference loss), and throughput under varying priority weights (λ_1, λ_2) and idleness tolerance threshold (τ) in LeMix's resource allocation. Increasing λ_1 (response time weight) reduces latency by prioritizing faster execution but limits resource utilization, while higher λ_2 (length consistency weight) improves serving quality by reducing workload heterogeneity at the cost of slight delays. Meanwhile, larger τ allows better idle-period utilization and higher throughput by consolidating workloads to fewer nodes, but risks prolonged response times as tasks wait for resources. Overall, LeMix achieves a robust balance with "*sweet spots*" of λ_1, λ_2, and τ: it maintains low response times (e.g., 150 ms), high throughput (e.g., 4.5 task/s), and stable inference quality (e.g., 2.5), showcasing its adaptability to dynamic real-time conditions.

3.7 Related Work

3.7.1 Distributed training.

Distributed training of large models has been extensively explored to address the computational demands of increasing complexity. Techniques such as data parallelism [32,

66,135] where the data is split across multiple processors, model parallelism [141] where the model is horizontally sharded into multiple computational units, and pipeline parallelism [7, 60, 82, 116] which divides each input mini-batch into small micro-batches to further reduce idleness, are well-established. While these approaches optimize utilization, they leave inter-pipeline idle periods unaddressed.

3.7.2 Inference serving.

LLM serving systems have evolved to tackle the dynamic requirements of LLM workloads. General-purpose systems [17, 20, 88, 110, 115] are widely used for production environments, while LLM-optimized systems [2, 84, 108] have emerged to handle KV cache management and chunked prefill processing. Techniques like Orca [164] and FastServe [156] introduce continuous batching and preemptive scheduling [145], addressing delays caused by long jobs, but often exacerbate interference when workloads are co-located. Furthermore, disaggregation-based approaches [119,144,174] reduce contention by isolating specific workload stages. Unlike these methods, LeMix focuses on node allocation, idleness utilization, and retraining alignment, demonstrating compatibility with autoregressive generation.

3.7.3 Data drift and continual learning.

Adapting to data drift [85,86] has been widely studied in continuous learning scenarios, where models evolve with new incoming data. Traditional methods include transfer learning [57, 90, 146] and catastrophic forgetting mitigation [47, 91, 93], while edge deployment systems [12, 81, 142] demonstrate scheduling techniques for model retraining. Inference serving for LLMs also faces challenges similar to data drift, as models require frequent

alignment to updated user interactions and factual knowledge. LeMix builds upon continuous learning principles by co-locating training and inference workloads, enabling real-time adaptation with reduced inter-node synchronization delays, which is critical for emerging "learning while serving" paradigms.

3.8 Conclusion

In this work, we introduced LeMix, a framework that schedules concurrent LLM training and inference workloads in distributed systems. The key innovation is exploring optimizations when co-locating both computational workloads under dynamic traffic and execution heterogeneity, and integrating these insights to optimize resource utilization while attaining SLOs. By real-time quantifying task-specific contributions to idle periods, accuracy, and response time, LeMix achieves up to 3.53× higher throughput, 0.61× lower loss, and 2.12× higher SLO attainment over traditional Separate setups.

Chapter 4

MACE: A Hybrid LLM Serving System with Colocated SLO-aware Continuous Retraining Alignment

4.1 Introduction

Large language models (LLMs) have demonstrated impressive capabilities across diverse real-world applications, including open-domain question answering, code generation, and mathematical reasoning. As a result, LLMs have been widely adopted in interactive systems such as chat assistants [1, 124], search engines [121], multimodal agents [101, 134], and real-time code assistants [46]. These systems process user queries in real-time and have increasingly shifted toward edge deployment, where models are hosted on local edge servers closer to the user [37, 142].

Edge servers offer low-latency responses and improved data privacy, complying with regional policies like GDPR that prohibit the transmission of sensitive data to distant cloud regions [12]. However, edge servers often have limited GPU capacity [106], making it challenging to concurrently serve inference requests and support continual adaptation of models to user feedback or real-time data shifts [12]. For instance, user responses may signal changes in preference (e.g., political leaning, toxicity tolerance) [99], requiring the system to fine-tune the model to reflect these updated objectives—departing from generalized alignment toward personalized alignment.

While compression methods (e.g., quantization, distillation) can reduce inference costs, their limited generalization capacity makes them highly sensitive to data distribution shifts [106, 142]. Therefore, continual retraining (e.g., every 30–60 seconds) becomes necessary to maintain alignment [12, 142], using user-specific preferences and temporal contexts extracted from live inference interactions. Unlike conventional supervised training pipelines, labels for LLM alignment data are not manually annotated. Instead, they are derived from implicit user feedback signals. A common approach is to use preference-based supervision, where users are presented with multiple generated responses and asked to select the most preferred one. User feedback is converted into supervision through preference signals (e.g., chosen, rejected), enabling ranking-based pairwise losses like Direct Preference Optimization (DPO) [125]. Additional implicit signals—e.g., thumbs-up/down, skip rate, or dwell time—can be mapped to preferences or reward scores. In multi-turn dialogs, user rephrasing or corrections also serve as supervision. Over time, this generates a personalized feedback

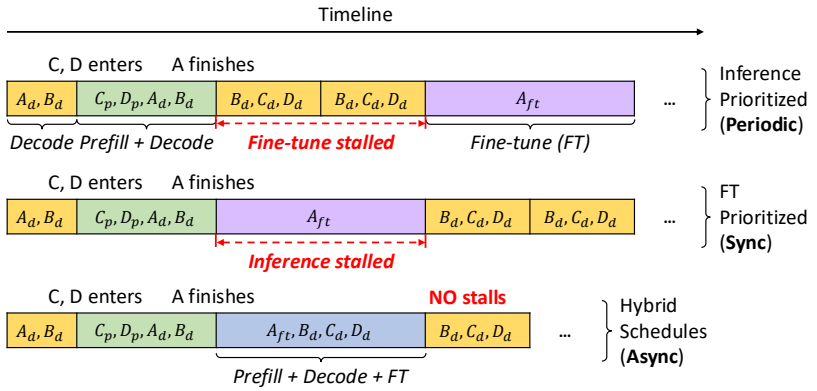


Figure 4.1: Requests A, B, C, D arrive over time. Subscripts p and d indicate prefill and decode iterations, while ft marks fine-tuning. **Periodic** retraining delays model updates for A due to inference priority. **Sync** retraining preempts decodes for B, C, D. **Async** (hybrid) schedule *colocates* A_{ft}, B_d, C_d, D_d into the same iteration, reducing latency and ensuring B, C, D benefit (in subsequent iterations) from the updated model—without stalling either workload.

corpus for fine-tuning models toward user-specific goals, without centralized fine-tuning or privacy violations.

However, co-running fine-tuning and inference on a resource-constrained GPU introduces a fundamental scheduling dilemma. Figure 4.1 illustrates the trade-offs among three commonly used scheduling strategies, building upon iteration-based continuous batching [164], where time is divided into discrete iterations, and each iteration forms a batch of tasks for the model to process concurrently:

- **Periodic** retraining defers model updates to fixed intervals, preserving inference responsiveness. Yet, this can delay alignment—decoding requests (B, C, D) are served using outdated models, reducing accuracy.
- **Continuous** (sync) retraining immediately preempts inference to perform fine-tuning, ensuring alignment but often stalling inference requests and causing SLO violations.

- **Hybrid** (async) retraining aims to interleave both workloads within the same GPU iteration, packing complementary workloads to reduce idle fragments and maximize both alignment and inference throughput.

The core challenge arises from their conflicting resource usage patterns under tight GPU budgets. Inference—especially LLM autoregressive decoding—requires sequential generation and sustained attention memory, which makes batching difficult and memory-intensive [2, 74]. Fine-tuning, on the other hand, demands high GPU memory bandwidth and model parameter updates, which can stall ongoing inference [25].

To mitigate this tension, recent approaches employ *parameter-efficient fine-tuning* (PEFT) methods such as low-rank adaptation (LoRA) [59], which injects trainable low-rank matrices into the frozen weights of the base model. PEFT significantly reduces the number of trainable parameters and memory overhead during training, making it a natural fit for resource-constrained edge environments. It allows fine-tuning to be performed with minimal disruption to inference serving. However, even with PEFT, co-executing fine-tuning and inference remains non-trivial. For example, LoRA introduces additional memory (for adapters and optimizer state), and its backward pass can conflict with autoregressive decoding, particularly when multiple user requests are queued. Moreover, *resource fragmentation*—caused by variable-length inputs and dynamic arrival rates—can lead to suboptimal utilization and SLO violations (for inference requests) if not carefully managed [145].

To this end, we propose MACE, a hybrid execution framework that co-optimizes fine-tuning and inference on shared edge GPU resources. MACE introduces a fine-grained iteration-level hybrid scheduler that selects and packs both workloads together based on

alignment potential, memory feasibility, and latency constraints. It adopts a *unified scheduling* strategy that handles inference and continual learning under a shared memory and execution budget. Compared to traditional periodic or sync baselines, MACE adapts dynamically to the queue state and application demand, maximizing GPU utilization while minimizing latency for heterogeneous workloads. MACE is built on two key components:

- *Memory-aware hybrid scheduler*, which jointly allocates prefill, decode, and fine-tune workloads per iteration using memory and alignment heuristics (§4.4.2).
- *Cache manager*, including prefix sharing at prefilling, enabling reuse of previously computed KV cache across requests to reduce response time and free memory, and KV cache pruning at decoding, which identifies and removes redundant attention heads or layers with negligible contribution to final output (§4.4.3).

We conduct extensive experiments on real-world traces from personalized chat datasets using Mistral-7B and LLaMA3-8B. As shown in Figures 4.9, 4.10, MACE consistently achieves the best trade-off between alignment accuracy and inference throughput, outperforming both periodic and synchronous baselines across retraining frequencies.

4.2 Background

Recent surge in deploying LLMs for real-time user interaction [138] has necessitated effective strategies for handling concurrent training and inference tasks. These workloads span responding to user queries and retraining aimed at aligning LLMs with human preferences [6, 28] to ensure helpfulness [35] and harmlessness [8, 30].

Alignment fine-tuning and personalization. Modern LLMs often rely on post-training alignment techniques, such as *reinforcement learning from human feedback* (RLHF) [117] or DPO [125] to better align generated responses with human preferences. To reduce the cost of full fine-tuning, PEFT like LoRA [59] and surgical fine-tuning [79] have been widely adopted. LoRA introduces learnable low-rank matrices into attention layers, enabling efficient adaptation to new data without updating the backbone model. Surgical fine-tuning, on the other hand, identifies and updates only specific components of the model that are most impactful for alignment [140]. Such resource-efficient techniques are particularly advantageous for user personalization or continual alignment during deployment, where rapid adaptation to new preferences is essential.

Advanced LLM serving systems. Deploying LLMs in real-time and high-throughput environments requires serving systems that can simultaneously satisfy low latency, high efficiency, and hardware resource constraints. Continuous batching has become the de facto design in modern LLM systems to handle dynamic request arrival, which group them on-the-fly to improve GPU utilization without introducing excessive queuing delays [89], as shown in Algorithm 3. Systems like Orca [164], vLLM [74], Llumnix [145], and DistServe [174] have proposed various optimizations in this space. For example, Orca enables mixed prefill/decode execution via iteration (token)-level scheduling; vLLM introduces PagedAttention for flexible KV cache reuse; Llumnix proposes chunked prefill to pipeline long-context inputs; and DistServe decouples scheduling stages to support distributed execution. Despite implementation differences, these systems share a core challenge: *How to efficiently pack heterogeneous requests with diverse memory and latency profiles?* We argue that continuous

batching can be viewed through the lens of bin-scheduling—treating each GPU or sub-batch as a bin and scheduling tasks based on joint memory-latency fit. This abstraction forms the basis for our unified scheduling approach in later sections.

4.3 Motivation Study

4.3.1 Accuracy Benefits of Continuous Retraining

Traditional approaches to adapting LLMs to evolving user preferences often involve periodic retraining at fixed intervals, potentially leading to stale models that lag behind rapidly changing user needs. In contrast, continuous retraining allows models to adapt in near real-time, incorporating new information as it becomes available.

Alignment metrics. To evaluate the effectiveness of continuous retraining in aligning LLMs with user preferences, we define the following setting. Let $\mathcal{D} = (x, y^+, y^-)$ denote a dataset of user preference annotations, where x represents the input prompt, y^+ is the *preferred* response, and y^- is the *dispreferred* response. We assess model π_θ's performance using the following alignment metrics:

- **Preference accuracy (win rate)** [165] ↑: Win rate measures the fraction of preference pairs where the model assigns a higher probability to the preferred response:

$$\text{Win Rate} = \mathbb{E}_{(x,y^+,y^-)\sim\mathcal{D}}\left[\mathbf{1}\left[\pi_\theta(y^+ \mid x) > \pi_\theta(y^- \mid x)\right]\right],$$

  where $\mathbf{1}[\cdot]$ is the indicator function. A higher win rate signifies better alignment with user preferences.

- **Contrastive log-probability difference (CLPD)** [15] ↑: CLPD quantifies the average difference in log-probabilities between the preferred and less preferred responses:

$$\text{CLPD} = \mathbb{E}_{(x,y^+,y^-)\sim\mathcal{D}}\left[\log \pi_\theta(y^+ \mid x) - \log \pi_\theta(y^- \mid x)\right].$$

Higher CLPD means stronger preference for preferred responses over less preferred ones.

Retraining setup. To avoid the complexity of reinforcement learning, DPO [125] provides a lightweight alternative by leveraging contrastive learning over user preferences. Instead of optimizing a reward model, DPO directly fine-tunes LLMs by comparing the preferred and dispreferred outputs, while regularizing against a reference policy π_{ref} (e.g., the initial model). The goal is to maximize CLPD between y^+ and y^-, balanced by a regularization term that discourages excessive divergence from π_{ref}:

$$\mathcal{L}_{\text{DPO}} = \mathbb{E}_{(x,y^+,y^-)\sim\mathcal{D}}\left[\log \sigma\left(\beta \cdot \left(\Delta_\theta^+ - \Delta_\theta^-\right)\right)\right],$$
$$s.t. \quad \Delta_\theta^+ = \log \frac{\pi_\theta(y^+ \mid x)}{\pi_{\text{ref}}(y^+ \mid x)}, \Delta_\theta^- = \log \frac{\pi_\theta(y^- \mid x)}{\pi_{\text{ref}}(y^- \mid x)}.$$

Here, π_θ is the current model policy, β is a temperature parameter controlling the sharpness of preference, and $\sigma(\cdot)$ denotes the sigmoid function.

Empirical results. We study the impact of retraining strategies under a shared LLM serving system using two alignment fine-tuning benchmarks: Anthropic HH-RLHF [8] and StanfordNLP SHP [35]. The former focuses on harmlessness, where preferred responses tend to reject unsafe or illegal requests, while the latter focuses on helpfulness, where preferred responses are more factually correct.

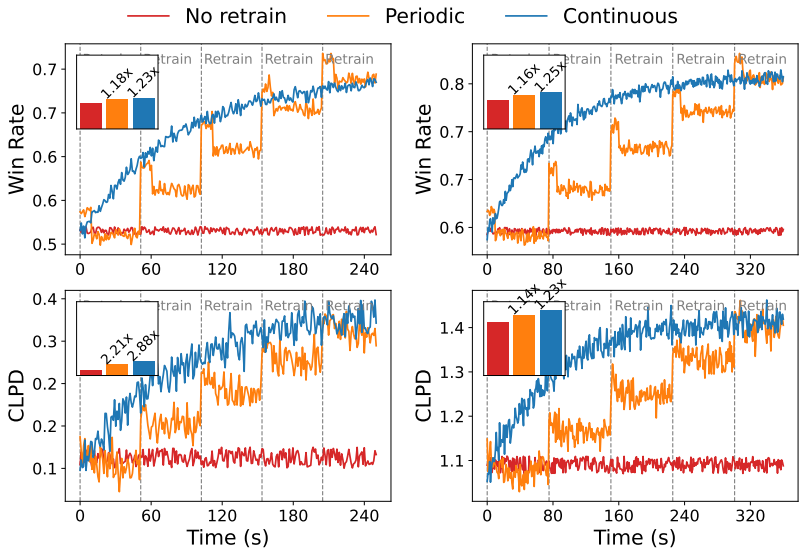


Figure 4.2: Win rate and CLPD over time when serving Mistral-7B on *Left*: RLHF and *Right*: SHP dataset. The inset bar plots show the average metric value of each method, highlighting the overall performance difference beyond temporal variations.

Figure 4.2 compares win rate and CLPD over time under different retraining strategies. We observe that: *Both periodic and continuous retraining consistently outperform the no-retrain baseline.* For instance, in the RLHF dataset (top-left), periodic retraining improves win rate by up to 1.19×, while continuous retraining yields a 1.24× gain. Similar trends hold for CLPD, where the confidence gap grows even more substantially (bottom-left, 2.47× and 3.17×, respectively). *Continuous retraining provides a larger performance boost.* Since retraining is triggered more frequently, continuous updates adapt the model faster to distributional shifts, leading to higher overall metrics compared to periodic retraining. This advantage is consistently seen across both datasets and both metrics. *CLPD reveals larger fluctuations and improvements than win rate.* While win rate rises more smoothly, CLPD exhibits stronger variance and larger relative gains. This is expected, as CLPD directly measures the preference confidence difference, which is more sensitive and interpretable in reflecting alignment strength. *Harmlessness alignment is harder than helpfulness align-*

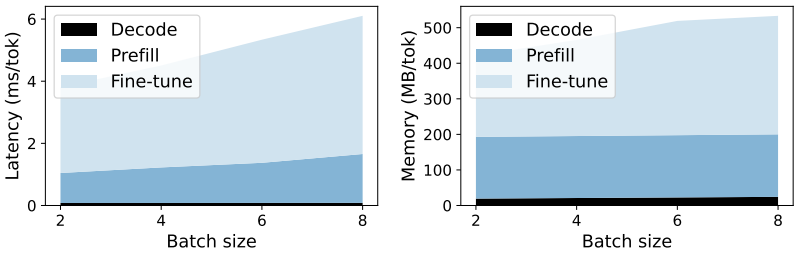


Figure 4.3: *Left*: latency per-token and *Right*: memory per-token for the three workloads across varying batch sizes.

ment. Comparing the two datasets, performance improvement is smaller for RLHF (left column) than for SHP (right column). This indicates that aligning models to avoid harmful behaviors (harmlessness) is inherently more challenging than aligning them to be helpful.

4.3.2 Optimization Opportunities for Latency Induced by Continuous Retraining

Profiling LLM workload heterogeneity. To effectively co-serve inference and continual fine-tuning for LLMs, it is essential to understand the latency and memory footprints of the constituent workloads. We profile the latency and memory usage of three major LLM operations—*prefill*, *decode*, and *fine-tune*—across varying batch sizes, shown in Figure 4.3. We observe that:

- *Prefill* latency and memory scale with input sequence length and batch size (linearly under FlashAttention [31]), since each token attends to the full prefix.
- *Decode* operations are much cheaper, as only a single token is appended per iteration, but memory gradually increases due to KV cache accumulation [74].

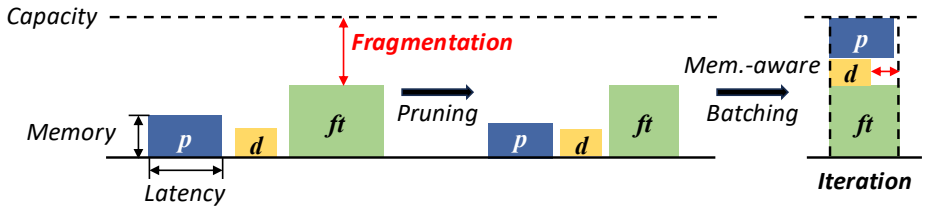
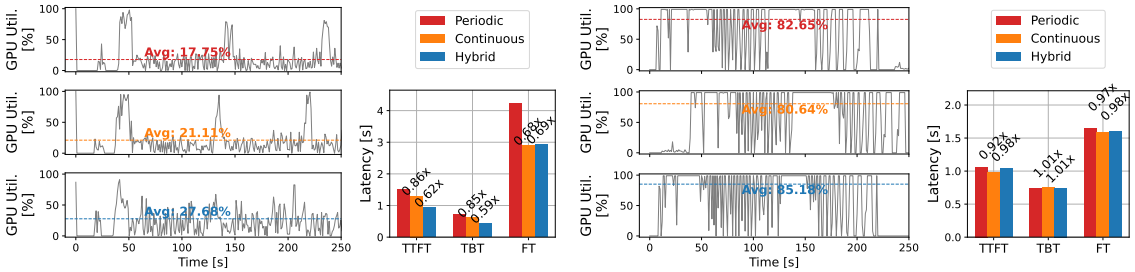


Figure 4.4: Abstracted memory–latency footprint for three workloads. Hybrid scheduling together with pruning techniques mitigates memory fragmentation and increases concurrency.



(a) Mistral-7B on A6000 Ada.

(b) Mistral-3B on NVIDIA AGX Orin.

Figure 4.5: *Left*: GPU utilization, and *Right*: Latency breakdown on two (a) A6000 Ada server and (b) AGX Orin edge device.

- *Fine-tune*, even with efficient PEFT methods like LoRA [59], exhibits significantly higher latency and memory consumption—making it the dominant bottleneck in edge inference-tuning pipelines.

Memory fragmentation and hybrid scheduling. Figure 4.4 (left) visualizes the resource usage of each workload as a memory-latency box. When scheduled independently, these misaligned workloads leave significant *holes* in GPU memory and timeline—leading to fragmentation and low utilization. For instance, a retraining task may consume only a fraction of GPU memory but blocks the entire iteration window. Conversely, prioritizing inference may delay critical fine-tunes that personalize the model. Inspired by continuous batching [164], we extend iteration-level scheduling to jointly colocate fine-tuning with prefill and decode. By dynamically adjusting batch sizes to opportunistically fill leftover

memory within each iteration, the system mitigates fragmentation and exploits latency complementarity while respecting alignment freshness. Figure 4.4 (right) illustrates this idea.

A6000 server (Figure 4.5a)—fragmentation-dominated and hybrid gains. On the A6000 server, average utilization is low: Periodic 17.8% / Continuous 21.1% / Hybrid 27.7%. Here, the bottleneck is not compute saturation but fragmentation: alternating between training and inference leaves idle gaps in both time and memory. Continuous retraining partially fills these gaps, but Hybrid scheduling is far more effective by colocating small-batch fine-tuning with inference workloads and resizing batches per iteration. This directly translates into significant latency reduction: $TTFT$: 0.62×-0.86×, TBT: 0.59×-0.85×, and FT: 0.68×-0.86×.

Orin (Figure 4.5b)—compute-bound, consistently saturated. On the AGX Orin edge device, the average utilization remains consistently high across all strategies: Periodic 82.7% / Continuous 80.6% / Hybrid 85.2%. With tighter compute and memory budgets, any workload—particularly fine-tuning—pushes the GPU close to saturation. As a result, the opportunity to further increase parallelism is limited. The latency breakdown confirms this: relative to Periodic, Continuous and Hybrid only provide marginal improvements (TTFT 0.92×-0.98×, TBT∼1.01×, FT 0.97×-0.98×). In short, on compute-constrained edge platforms, scheduling refinements mainly ensure that continuous training does not worsen inference latency, while the overall improvement remains bounded by hardware ceilings.

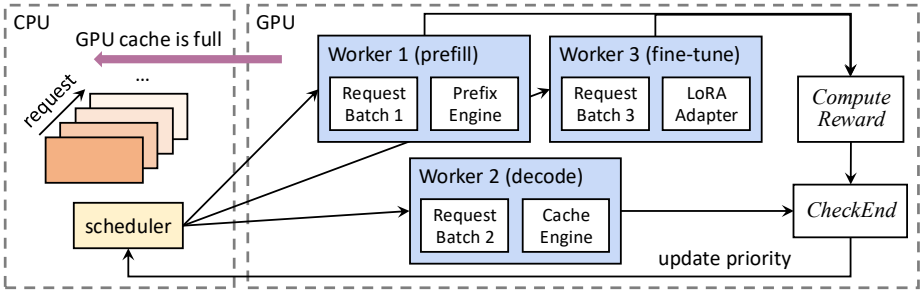


Figure 4.6: Design overview of MACE. The scheduler dispatches requests to prefill (prefix sharing), decode (KV cache pruning), and fine-tune (LoRA) workers, while feedback from reward computation and queuing time enables real-time priority updates under CPU–GPU cache coordination.

4.4 System Design of MACE

We present MACE, a fine-grained GPU-local scheduling system designed to support colocated LLM inference serving and continuous retraining. Figure 4.6 illustrates the design: user requests arrive asynchronously into a shared priority queue. The scheduler dynamically assigns priorities and batches them into iterations using a best-fit bin packing strategy, accounting for memory fragmentation and alignment reward. These batches are dispatched to GPU workers, where each worker concurrently handles prefilling, decoding, or fine-tuning workloads. After execution, tasks are re-prioritized if not finished and pushed back into the queue. MACE is built on three key components:

- **Priority computation:** Assign dynamic priority to each request based on workload type, enqueued time, and accuracy-driven rewards (e.g., DPO loss).
- **Iteration-level memory-aware batching:** A GPU-local scheduler that allocates tasks to minimize memory fragmentation and latency interference.
- **Cache management:** Optimizes memory reuse through prefix sharing in prefilling and cache pruning in decoding, and evicts unused cache to prevent memory bloat.

4.4.1 Alignment-aware Prioritization

Each incoming request is tagged with its arrival time t_{arr} and workload type $\mathsf{w} \in \{\mathsf{prefill}, \mathsf{decode}, \mathsf{ft}\}$. We define its base priority π_{w} and temporal growth rate δ_{w}, allowing a dynamic priority $P(t)$ at current time t to grow over time:

$$P_{\mathsf{w}}(t) = \pi_{\mathsf{w}} + \delta_{\mathsf{w}} \cdot (t - t_{\text{arr}}),$$

where we set $\pi_{\mathsf{decode}} > \pi_{\mathsf{prefill}} > \pi_{\mathsf{ft}}$ to prioritize inference serving and avoid SLO violation caused by frequent preemption, and set $\delta_{\mathsf{ft}} > \delta_{\mathsf{prefill}} > \delta_{\mathsf{decode}}$ to gradually escalate the criticality of fine-tuning requests over time to avoid *retraining starvation*.

Additionally, to promote retraining samples that are particularly valuable for model alignment, MACE integrates their alignment reward (i.e., DPO loss) to the priority:

$$P_{\mathsf{ft}}^{\text{total}}(t) = P_{\mathsf{ft}}(t) + \gamma \cdot \mathcal{L}_{\text{DPO}}(x, y^+, y^-),$$

where γ is a tunable weight. This mechanism prioritizes retraining samples with higher misalignment. As shown in Figure 4.7, after prioritization at iteration 9, the tasks are ordered by priority as A_{ft}, C_d, F_d, E_d, H_p, G_d, and D_{ft} (from highest to lowest).

4.4.2 Iteration-level Resource-aware Batching

Best-fit heuristic motivation. Scheduling a mix of LLM workloads on a single GPU can be viewed as a bin-packing problem in which each iteration's "bin" is the available GPU memory and each request (prefill, decode, ft) is an execution unit with particular

Algorithm 4 GPU-Local Request Scheduling

Require: Task queue $\mathcal{Q}$, memory capacity $\mathcal{P}$, thresholds $(\tau_{\text{mem}}, \tau_{\text{task}})$, memory-latency weights (λ_1, λ_2)
Ensure: Scheduled bin $\mathcal{B}_1$ for execution
Initialize bin list $\mathcal{B} \leftarrow []$, task counter $c \leftarrow 0$
while $\mathcal{Q}$ not empty **do**
    if $\mathcal{B} \neq \emptyset \wedge (\mathcal{B}_1.\text{memory} \geq \tau_{\text{mem}} \cdot \mathcal{P} \vee c \geq \tau_{\text{task}})$ **then**
        break
    Retrieve task $\leftarrow \mathcal{Q}$.dequeue(), increment $c \leftarrow c + 1$
    Estimate workload: $(m, \ell) \leftarrow$ GetWorkload(task)
    Initialize best_bin $\leftarrow$ `Null`, best_score $\leftarrow \infty$
    for each bin $\mathcal{B}_i$ in $\mathcal{B}$ **do**
        if $\mathcal{B}_i$.free_memory $\geq m$ **then**
            Compute fragmentation score f
            if score $<$ best_score **then**
                Update: best_score $\leftarrow$ score, best_bin $\leftarrow \mathcal{B}_i$
    if best_bin $\neq$ `None` **then**
        Assign task to best_bin, update memory and latency
    else
        Create new bin $\mathcal{B}_{\text{new}}$, insert task, append to $\mathcal{B}$
if $\mathcal{B} \neq \emptyset$ **then**
    for $i = 2$ to $|\mathcal{B}|$ **do**
        for each task in $\mathcal{B}_i$ **do**
            if CheckEnd(task) == `False` **then**
                Push task back into $\mathcal{Q}$ with updated priority
Output: $\mathcal{B}_1$

sizes (memory footprint) and duration (latency). Unlike general static bin packing, MACE involves an online, heterogeneous stream of tasks with different memory/latency profiles (Figure 4.3). A naive greedy scheduler that runs tasks sequentially or without regard for workload size can lead to low GPU utilization—e.g. large training jobs leaving gaps of unused memory that smaller inference jobs could have filled, or short decoding tasks waiting behind long-running jobs.

MACE adopts a best-fit heuristic as a pragmatic solution to this dynamic packing problem. Best-fit scheduling greedily fills the GPU with the combination of tasks that most tightly fits the available memory, while also accounting for execution time compatibility. It minimizes memory fragmentation (unutilized memory, which appears as the red gap

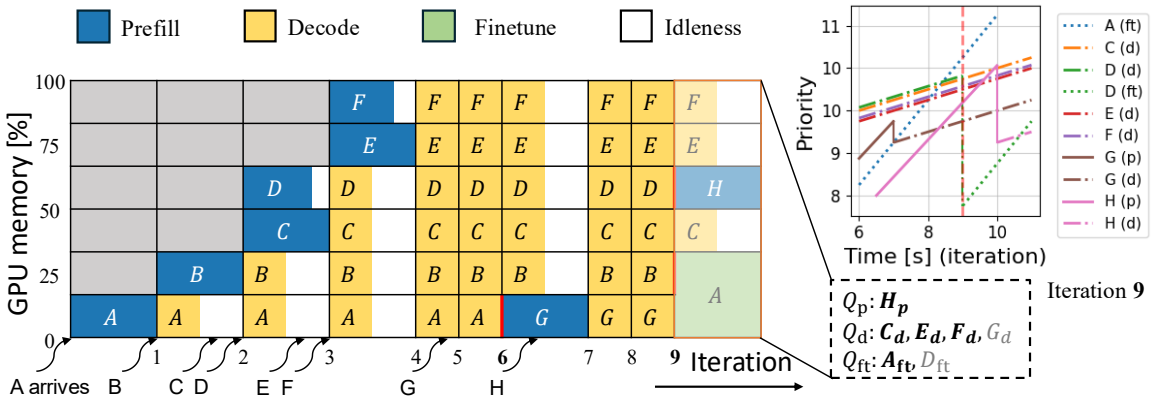


Figure 4.7: GPU-local scheduling of inference and training workloads using best-fit bin packing method. **Bold** texts denote scheduled tasks for the next bin execution.

in Figure 4.4) and helps balance latency so that no single task unduly delays the others. Formally, the scheduler assigns a composite score f to each candidate set of tasks using a two-dimensional objective:

$$f = \lambda_1 \cdot |\mathcal{B}_i.\text{free\_memory} - M| + \lambda_2 \cdot |\mathcal{B}_i.\text{max\_latency} - \ell|,$$

where λ_1, λ_2 trade off vertical and horizontal memory fragmentation. In practice, this best-fit heuristic is fast, adaptive, and well-suited to colocated workloads, even though globally optimal packing is NP-hard. By prioritizing a tight memory fit, we ensure that the GPU's capacity is used as much as possible each iteration, while the latency-aware term prevents pathological cases (e.g. packing a very slow task with many fast tasks that would all finish and leave the GPU underutilized waiting for the slow one).

Bin allocation. As illustrated in Algorithm 4, the scheduler dequeues tasks sorted by dynamic priority (§4.4.1) and attempts to assign them to the current iteration bin using the best-fit score (lines 1-16). It considers memory fit to reduce fragmentation and latency

divergence to avoid stalls. This enables effective co-scheduling of short decode jobs and large fine-tune jobs within a single GPU iteration. After execution, the system decides whether to reschedule unfinished tasks (lines 17–21) according to the `CheckEnd` output:

- For inference, it determines if the next decoded token is [EOS] or if the decoding iteration reaches the predefined maximum steps.
- For retraining, it checks whether DPO loss is still above a pre-defined threshold.

Tasks needing further iterations are requeued with updated priorities; others are retired. Figure 4.7 demonstrates the effectiveness of MACE. In iteration 6, a bin is formed by packing several decode jobs $B_d \sim F_d$ and one prefill G_p, with a large fine-tune A_{ft} temporarily delayed to prioritize throughput. While in iteration 9, the scheduler flexibly prioritizes the fine-tune A_{ft} which now has a higher priority, and delays the decode G_d to ensure timely model updates. This memory-aware batching balances the idleness-dependent throughput and continuous alignment learning process for heterogeneous workloads.

4.4.3 Prefix Sharing and Cache Management

To further mitigate the retraining–inference tradeoff, MACE leverages two key optimization techniques: *prefix sharing* [100, 173] and *KV cache pruning* [41, 94]. These techniques aim to reduce memory and latency overhead for inference, thereby creating more opportunities for retraining without compromising serving throughput.

Reduced prefill via prefix tree. Prefix sharing exploits input redundancy across user queries to avoid redundant prefill computation. As illustrated in Figure 4.8, multiple user queries often share common token prefixes, e.g., "*What is the best way to...*". We construct

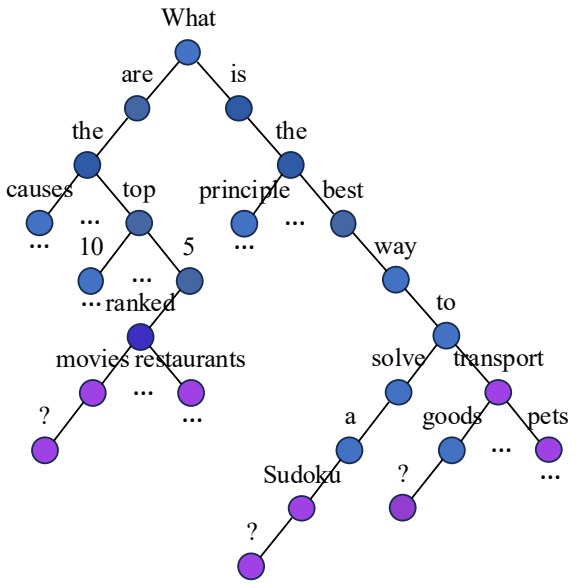


Figure 4.8: Prefix (Trie) tree from overlapping prompts. Leaf nodes are requests while others denote common prefix in prompts. Shared prefixes enable reduced prefill cost through reuse.

a prefix tree over active inference requests, where each leaf represents a complete request and each internal node represents a shared prefix segment. This design follows a Trie structure [173], where a path from the root to any leaf captures the longest common prefix of a request. To maximize reuse, we traverse the tree in a DFS manner to co-execute shared prefixes. At each internal node of the prefix tree, we cache the KV pairs from the prefill outputs. This cache is efficiently reused by descendant nodes without recomputation. To manage memory usage, we offload least recently used (LRU) cache when inference concurrency is high to release more memory budget. By avoiding repeated computation for shared prefix tokens, prefix sharing reduces both latency and memory consumption in prefill. It also provides more headroom for opportunistic retraining, as the scheduler can pack training tasks without disrupting inference throughput. Given that aggressive resource overlapping

(e.g., colocation) may interfere with prefix alignment, reducing reuse opportunities [169], we enforce prefix sharing only within inference requests.

Norm-guided decode cache pruning. Decoding presents another challenge: KV cache for each output token accumulates linearly and quickly saturates GPU memory. To mitigate this, we introduce a norm-based cache pruning mechanism, inspired by the revealed contextual sparsity of LLMs [103]. At each decoding iteration, we compute the L2-norm of the output attention vector across heads. Let $\mathbf{a}_{t,h} \in \mathbb{R}^d$ denote the output at iteration t and attention head h, and define its norm as $\|\mathbf{a}_{t,h}\|_2$. We maintain a rolling average $\bar{\mathbf{a}}_h$ for each head and reallocate per-head KV storage capacity proportionally:

$$\bar{\mathbf{a}}_h = \frac{1}{T}\sum_{t=1}^{T} \|\mathbf{a}_{t,h}\|_2, \; w_h = \frac{\bar{\mathbf{a}}_h}{\sum_j \bar{\mathbf{a}}_j}, \; C_h = \lfloor w_h \cdot C_{\text{total}} \rfloor .$$

Here, $\bar{\mathbf{a}}_h$ is the average ℓ_2 norm of the attention output at head h, C_{total} is the total available KV cache capacity, and C_h is the per-head allocation. The remaining capacity is greedily assigned to heads with the largest weight w_h:

$$\text{Prune if} \quad t - t_{\text{last\_used}} > W \quad \vee \quad \|\mathbf{a}_{t,h}\|_2 < \tau,$$

where W is a fixed sliding window size and τ is a norm threshold below which attention heads are considered uninformative [41]. This dynamic reallocation selectively retains high-signal attention heads while pruning less informative ones. If a head's norm is consistently small, it receives lower cache budget and may prune earlier attention positions. Practically, this reduces decoding memory while retaining output quality—especially helpful under long

outputs or memory pressure. By combining prefix sharing for prefill and norm-guided cache pruning for decoding, we allow more training jobs to execute without sacrificing inference throughput.

4.5 Discussion and Implementation

We build our system by integrating efficient LLM inference engine, lightweight retraining modules, and a scheduler with fine-grained concurrency and batching support. Below, we elaborate on each component.

Inference engine. Our system adopts vLLM [74], an inference backend optimized for decoding throughput using PagedAttention and FlashAttention [31] mechanisms. This enables our system to leverage token-level parallelism and key-value memory optimization for high-throughput generation.

Personalized adapter. To enable low-latency retraining without degrading inference throughput, we leverage PEFT techniques by assigning each user or domain an independent LoRA adapter ϕ_u [59]. This allows user-personalized updates while sharing the frozen base model θ across all requests. Inspired by recent findings on layer-wise alignment sensitivity [79, 140], , we adopt surgical fine-tuning that selectively updates only the top-k alignment-sensitive layers (typically the final few transformer blocks), drastically reducing compute and memory overhead. Concretely, for a transformer model with L layers, we fine-tune only the last $k \ll L$ layers' LoRA weights, which retains the learning capability of alignment fine-tuning with much less cost. By combining per-user adapters with top-layer selective tuning, MACE supports rapid retraining even on large models within a shared

GPU environment. This improves retraining responsiveness and allows tighter integration with online serving, as fine-tuning can occur opportunistically during memory gaps between inference batches.

Concurrency support. Our system is implemented with multithreaded concurrency using a thread pool design: A *producer* thread ingests inference/retraining requests. A *scheduler* thread computes dynamic priorities and forms hybrid iteration-level batches, where a bin allocator maps scheduled jobs to physical GPU sessions. Multiple *executor* threads run on-device workloads asynchronously.

To balance latency and throughput, the scheduler maintains a priority queue that supports: *Dynamic priority refresh*: periodically updates request priorities based on utility functions (e.g., model freshness, deadline slack). *Preemption-aware iteration batching*: schedules batches with overlapping prefill tokens and similar generation lengths to maximize GPU reuse. This pipeline ensures high concurrency and responsiveness while preserving fine-grained control for hybrid scheduling decisions.

4.6 Evaluation

4.6.1 Experimental Setup

Hardware platforms. We test both *server* with NVIDIA A6000 Ada GPU (48 GB VRAM) and *edge* device with NVIDIA AGX Orin (32GB LPDDR5), ensuring a controlled environment to isolate the scheduler's effects.

Workloads and datasets. We simulate a mixed workload of inference requests and retraining jobs representing single and multiple user domains with varying alignment needs.

The inference requests are generated as a Poisson arrival process [89], and additional fraction of them (i.e., retrain rate, which is less than 50% [12]) are fine-tuning jobs. We consider two datasets to emulate different alignment domains or user preferences:

- **RLHF** [8] (harmlessness): Human feedback dataset focusing on safety alignment—preferred responses tend to avoid unsafe or illegal content. This represents users requiring the model to refuse or modify harmful outputs.
- **SHP** [35] (helpfulness): A broad human preference dataset with 385k comparisons across 18 topics, emphasizing helpfulness of responses over others. This represents general user satisfaction alignment.

By using these distinct datasets, we create multi-tenant scenarios where different "tenants" (user groups) issue requests with different alignment objectives. This diversity lets us evaluate LeMix 's overall alignment capability.

Baselines. We compare LeMix against several baseline scheduling strategies:

- **Ekya** [12]: A policy that periodically preempts inference to run retraining at fixed intervals. Inference requests queue up during retraining epochs (synchronous blocking). This baseline represents traditional FT where serving and training are separated in phases.
- **AdaInf** [142]: A continuous scheduling that gives retraining jobs highest priority in a FIFO queue, where incoming FT tasks can preempt or delay inference until they complete. It maximizes retraining frequency but with no special handling for interference, e.g., often hurting inference latency significantly.

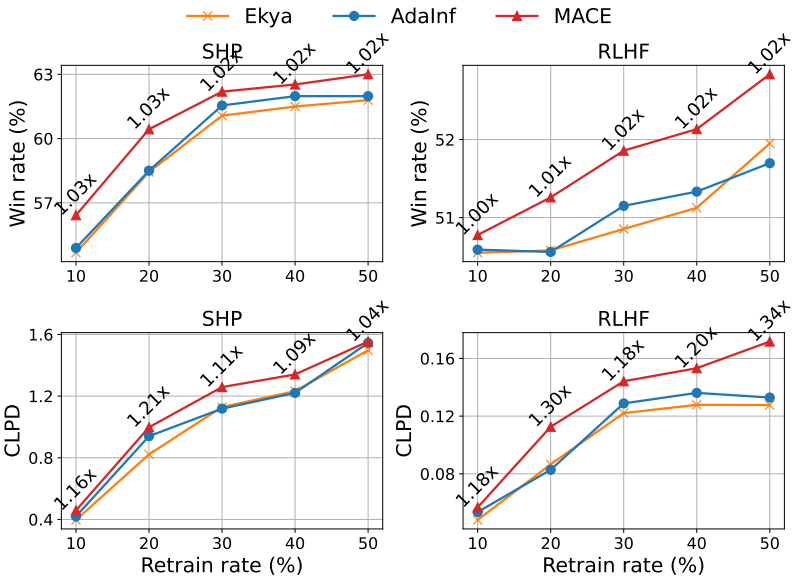
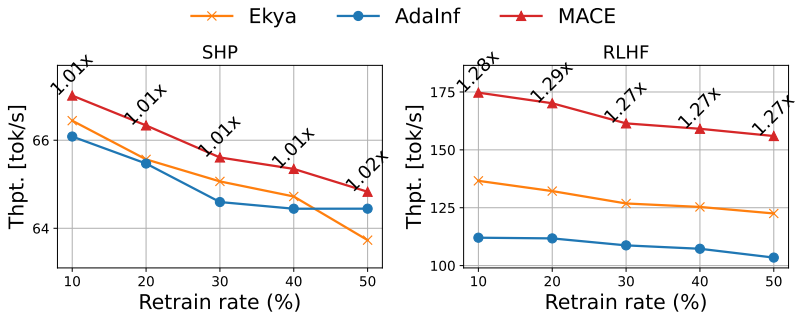


Figure 4.9: Average win rate and CLPD across various retrain rates of different methods on *Left*: SHP and *Right*: RLHF dataset.

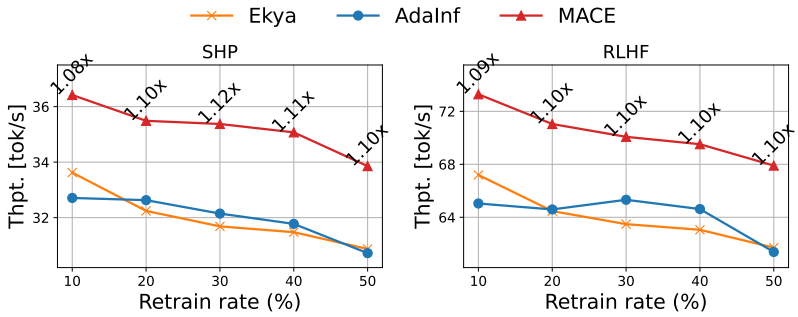
4.6.2 Performance under Varying Retraining Intensity

We evaluate LeMix against Ekya and AdaInf under varying retraining intensities, focusing on alignment accuracy, inference throughput, and latency breakdown.

Alignment accuracy. Figure 4.9 shows average win rate and CLPD across SHP and RLHF datasets. LeMix consistently outperforms both baselines, especially under higher retraining rates. On SHP, LeMix achieves up to 1.03× higher win rate and 1.21× higher CLPD compared to AdaInf. On RLHF, the advantage is more pronounced: LeMix improves win rate by up to 1.02× and CLPD by 1.34×, reflecting its ability to retain alignment accuracy even under intensive retraining. In contrast, Ekya and AdaInf show diminishing returns as retraining frequency increases, highlighting the importance of LeMix 's fine-grained scheduling and cache-aware optimizations.



(a) Throughput (token/sec) of Mistral-7B on RTX A6000 Ada.

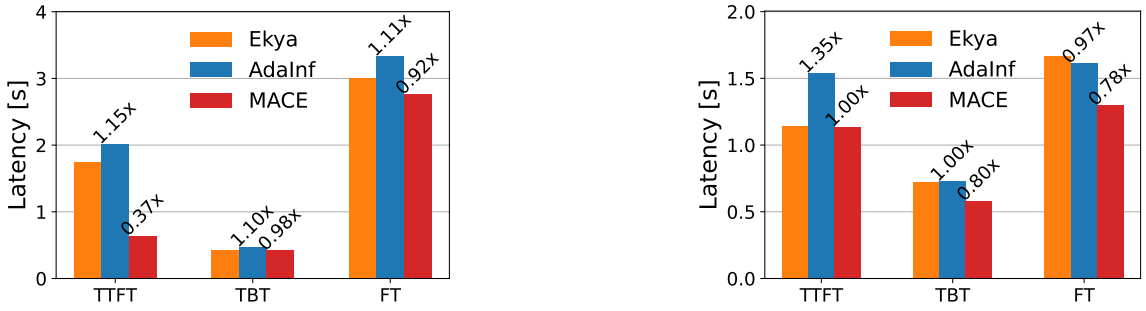


(b) Throughput (token/sec) of Mistral-3B on NVIDIA AGX Orin.

Figure 4.10: Inference throughput comparison of different methods.

Throughput. Figure 4.10 reports inference throughput on both server-scale and edge-scale platforms. On the A6000, LeMix delivers up to 1.29× higher throughput than AdaInf at 20% retraining rate, while maintaining superior accuracy. On the AGX Orin, which is more resource constrained, LeMix sustains 1.12× higher throughput on average, with benefits most evident at higher retraining intensities where baselines degrade rapidly. These results demonstrate that LeMix 's hybrid batching and pruning strategies effectively amortize retraining costs without sacrificing inference performance, a critical property for deployment on heterogeneous hardware.

Latency breakdown. Figure 4.11 decomposes end-to-end latency into prefill, decode, and fine-tuning components. LeMix achieves lower decode latency (TBT) via cache pruning,



(a) RTX A6000 Ada

(b) NVIDIA AGX Orin

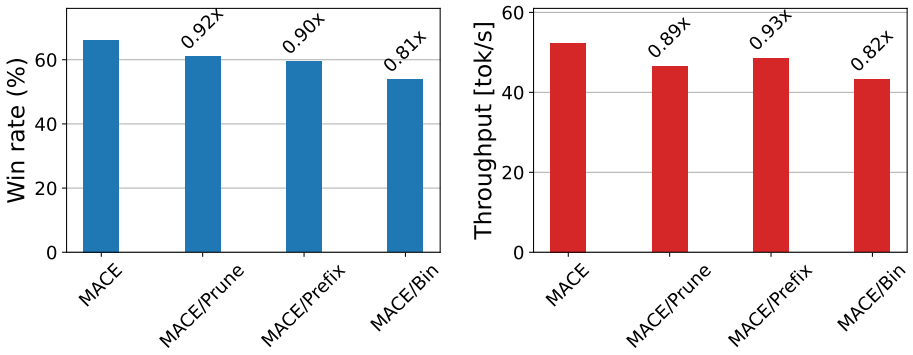
Figure 4.11: Latency breakdown on (a) A6000 and (b) AGX Orin.

while prefix sharing substantially reduces prefill cost (TTFT). Fine-tuning overhead remains bounded due to iteration-level batching, which overlaps retraining with inference execution. Compared to AdaInf, LeMix reduces prefill latency by 63% on A6000 server and decode latency by 30% on edge device platforms. This balanced reduction explains its ability to simultaneously improve alignment accuracy and throughput, highlighting the effectiveness of joint scheduling across retraining and inference phases.

4.6.3 Understanding LeMix

To better understand the contribution of each component in LeMix, we create the following variants by disabling specific mechanisms:

- *MACE/Bin*: This variant disables memory-aware batching by fixing maximum batch sizes and applying FIFO queuing without hybrid scheduling.
- *MACE/Prefix*: This variant disables prefix sharing during the prefill phase, resulting in redundant computation across similar requests.



(a) Accuracy (in SHP) (b) Throughput (on A6000)

Figure 4.12: Performance of different variants of LeMix.

- *MACE/Prune*: This variant disables KV cache pruning under memory constraints, forcing full cache retention during decoding.

Alignment accuracy. Figure 4.12a presents the average alignment accuracy across all applications and time periods. The results show the following order: LeMix > LeMix/Prune > LeMix/Prefix > LeMix/Bin. LeMix achieves up to 5.2% higher accuracy than LeMix/Prune, demonstrating the effectiveness of dynamic KV cache pruning in reducing latency-induced alignment degradation. Compared to LeMix/Prefix, LeMix achieves 6.7% higher accuracy, showing the benefit of prefix sharing in avoiding redundant computation and enabling faster response without sacrificing alignment. Notably, LeMix achieves 12.3% higher accuracy than LeMix/Bin, which validates the core idea that fixed batching and FIFO scheduling fail to adapt to the heterogeneous retraining/inference workloads, leading to less timely model updates and accuracy loss.

Throughput. Figure 4.12b reports the average inference throughput (in tokens per second). The trend is: LeMix > LeMix/Prefix > LeMix/Prune > LeMix/Bin. LeMix maintains throughput at 52.4 tok/s, outperforming all variants. LeMix/Prefix sees a 7.5%

Table 4.1: Average time overhead (ms) and memory (MB) of schedulers. ~~PA~~, ~~IB~~, ~~CM~~ mean without prioritization (§4.4.1), iteration-level batching (§4.4.2), and cache management (§4.4.3).

	Ekya	AdaInf	LeMix		
			~~PA~~	~~IB~~	~~CM~~
Latency (ms)	3e-2	8e-2	1.1e-1	1.2e-1	1.5e-1
Memory (MB)	0.6	2	3.5	4.5	4

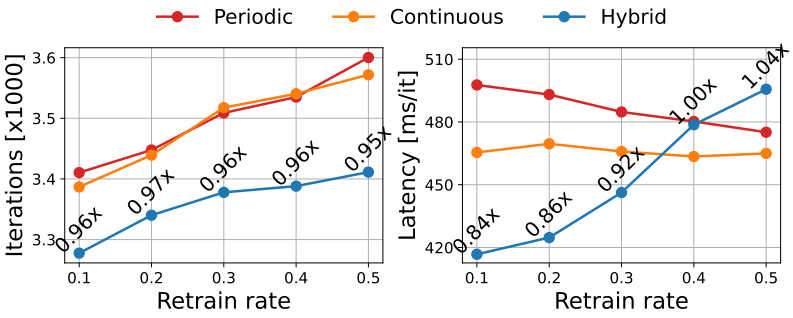


Figure 4.13: *Left*: total iterations and *Right*: average per-iteration latency for Mistral-7B on A6000 Ada.

drop due to redundant prefix tokenization and attention. LeMix/Prune shows an 11.2% drop due to KV cache pressure causing decoding slowdowns. The worst throughput is observed in LeMix/Bin (down by 17.6%) since it statically batches requests and cannot prioritize workloads that benefit from hybrid iteration-level handling. These results confirm that each component in LeMix —iteration-aware scheduling, prefix sharing, and cache pruning—contributes uniquely to achieving both high alignment accuracy and low-latency inference throughput.

Bin efficiency. Figure 4.13 shows that under the same concurrency constraints (e.g., inference batch size ≤ 50, train ≤ 2) and queuing policy (e.g., FIFO), hybrid requires fewer iterations to finish the same workload, especially when retraining is light. This suggests a better iteration efficiency by reducing idle memory slots. Despite mixing heterogeneous

workloads, its per-iteration latency remains close to baselines, indicating that hybrid effectively fills memory and latency gaps (i.e., fragmentation) without incurring extra runtime cost.

Overhead analysis. As shown in Table 4.1, the added overhead is computationally lightweight and memory efficient (under 5MB total), and the latency cost is amortized across each batch. In real workloads with high GPU compute cost, this overhead is a worthwhile tradeoff for improved throughput and accuracy. The scheduling decision in our full method introduces ∼15 ms/request overhead, about 5× higher than Ekya, which is expected given our scheduler tracks per-request priority, memory fragmentation estimates, and iteration-level packing. However, this is still negligible (<0.1%) compared to typical request execution time (which ranges from tens to hundreds of milliseconds). Compared to AdaInf, LeMix adds ∼7 ms/request additional overhead due to more fine-grained request-level decision-making instead of coarse DAG session-level scheduling. Our full scheduler uses up to 5MB, largely for per-iteration job queues, lightweight memory pressure estimation, and cache tracking structures (prefix/KV/retention flags). In contrast, Ekya and AdaInf maintain only coarse batch/session metadata (no fine-grained tracking), hence use <2MB.

4.7 Limitations and Discussion

Scalability beyond a single node. Our current design and evaluation focus on a single edge server equipped with limited GPU resources. While the method naturally generalizes to multiple concurrent models and tasks within a node, scaling across multiple edge nodes or hybrid cloud-edge architectures introduces challenges such as synchronization of

model versions, cross-node retraining coordination, and distributed cache consistency. Future extensions may explore hierarchical or federated variants of MACE that operate over distributed GPU clusters.

GPU heterogeneity and offline profiling. Our current system assumes access to a single GPU type within a node, following prior work [131, 142]. Extending MACE to support heterogeneous GPU types or multi-node edge clusters introduces new challenges in profiling transfer latency, maintaining cache coherency, and synchronizing retraining checkpoints across devices. Additionally, as with prior systems, we rely on offline profiling to characterize prefill/decode latencies under different batch sizes and context lengths. Reducing this profiling overhead or making the scheduler online-adaptive remains an important direction for future work.

CPU execution alternatives. Several inference systems consider fallbacks to CPU execution under low-load scenarios [39, 53], which can be more cost-efficient. While our work focuses on GPU-based deployment due to the high latency sensitivity and memory footprint of LLMs, lightweight variants or early-exit configurations could be executed on CPUs under certain regimes. Integrating such hybrid CPU-GPU execution into MACE is a promising extension to reduce energy and cost.

4.8 Related Work

4.8.1 LLM serving.

Serving systems for LLMs have rapidly evolved to meet the stringent demands of low-latency, high-throughput workloads. General-purpose platforms [110, 115] are widely

adopted in production, while LLM-specific frameworks [2, 108] introduce optimizations such as KV-cache management and segmented prefill. To mitigate straggler effects, systems like Orca [164] and FastServe [156] leverage continuous batching and preemptive scheduling, with recent advances like Llumnix [145] further improving request interleaving. Meanwhile, disaggregation-based methods [119,144,174] isolate execution stages across resources to reduce interference. Distinct from these approaches, LeMix targets GPU-local scheduling, emphasizing iteration-level batching, cache-aware execution, and hybrid retraining–inference co-location, ensuring efficiency without compromising autoregressive generation latency.

4.8.2 Online alignment fine-tuning.

Handling data drift remains a central challenge in machine learning systems. Classical strategies such as transfer learning [57, 146] and catastrophic forgetting mitigation [47] have been extended to real-time deployments. At the edge, systems like Ekya [12], AdaInf [142], and Lyra [81] schedule retraining alongside inference to maintain accuracy while respecting resource limits. LLM serving faces a similar problem: user-driven interactions and shifting knowledge bases require frequent model adaptation. LeMix extends this line of work by embedding continual retraining directly into the serving loop, co-scheduling inference and finetuning within a single GPU. This design eliminates cross-node synchronization overheads and directly supports the emerging paradigm of "learning while serving."

4.9 Conclusion

This work introduced MACE, a hybrid GPU-local scheduler that unifies inference and continual retraining for large language models. By combining priority-aware request handling, iteration-level batching, and cache management, MACE maximizes GPU utilization while maintaining low inference latency. Our evaluation demonstrates that MACE reduces fragmentation, accelerates workload completion, and sustains alignment accuracy under data drift—surpassing existing systems designed for either inference-only or periodic retraining.

Overall, MACE highlights the importance of fine-grained runtime scheduling as a key enabler for practical "learning while serving" LLM deployments. Future directions include extending hybrid scheduling to heterogeneous accelerators, incorporating multi-user adaptation policies, and generalizing the approach to multimodal foundation models.

Chapter 5

Conclusions

This dissertation addressed the central challenge of deploying LLMs under strict accuracy–latency constraints through a progression of three systems. RT-LM tackled input-level variability via uncertainty-aware resource management for real-time inference. LeMix extended scheduling to the cluster level, enabling efficient co-location of inference and retraining on multi-GPU systems. Finally, MACE focused on fine-grained runtime control within a single GPU, introducing iteration-level scheduling, batching, and cache management to balance retraining and inference.

Together, these works demonstrate that adaptive scheduling across different granularities—input, node, and GPU-runtime—is essential for sustainable and efficient LLM deployment. They highlight the need to co-optimize latency, throughput, and continual learning while respecting limited GPU resources.

Looking forward, several directions emerge: (1) extending scheduling to heterogeneous accelerators beyond GPUs, (2) enabling scalable multi-user continual adaptation

with robust rollback and fairness, (3) developing profiling-free, self-adaptive schedulers, and (4) generalizing beyond language to multimodal foundation models.

In summary, this dissertation shows that fine-grained, workload-aware scheduling is a principled approach to bridging real-time inference and continual retraining, paving the way for practical and resilient deployment of future foundation models.

Bibliography

[1] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023.

[2] Amey Agrawal, Nitin Kedia, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav Gulavani, Alexey Tumanov, and Ramachandran Ramjee. Taming {Throughput-Latency} tradeoff in {LLM} inference with {Sarathi-Serve}. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 117–134, 2024.

[3] Ekin Akyürek, Mehul Damani, Linlu Qiu, Han Guo, Yoon Kim, and Jacob Andreas. The surprising effectiveness of test-time training for abstract reasoning. *arXiv preprint arXiv:2411.07279*, 2024.

[4] Amazon Web Services, Inc. Train a model with amazon sagemaker. `https://docs.aws.amazon.com/sagemaker/latest/dg/how-it-works-training.html`, 2024. Accessed: 2024-05-12.

[5] Reza Yazdani Aminabadi, Samyam Rajbhandari, Ammar Ahmad Awan, Cheng Li, Du Li, Elton Zheng, Olatunji Ruwase, Shaden Smith, Minjia Zhang, Jeff Rasley, et al. Deepspeed-inference: enabling efficient inference of transformer models at unprecedented scale. In *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15. IEEE, 2022.

[6] Amanda Askell, Yuntao Bai, Anna Chen, Dawn Drain, Deep Ganguli, Tom Henighan, Andy Jones, Nicholas Joseph, Ben Mann, Nova DasSarma, et al. A general language assistant as a laboratory for alignment. *arXiv preprint arXiv:2112.00861*, 2021.

[7] Sanjith Athlur, Nitika Saran, Muthian Sivathanu, Ramachandran Ramjee, and Nipun Kwatra. Varuna: scalable, low-cost training of massive deep learning models. In *Proceedings of the Seventeenth European Conference on Computer Systems*, pages 472–487, 2022.

[8] Yuntao Bai, Andy Jones, Kamal Ndousse, Amanda Askell, Anna Chen, Nova DasSarma, Dawn Drain, Stanislav Fort, Deep Ganguli, Tom Henighan, et al. Training

a helpful and harmless assistant with reinforcement learning from human feedback. *arXiv preprint arXiv:2204.05862*, 2022.

[9] Soroush Bateni and Cong Liu. Apnet: Approximation-aware real-time neural network. In *2018 IEEE Real-Time Systems Symposium (RTSS)*, pages 67–79. IEEE, 2018.

[10] Soroush Bateni and Cong Liu. Neuos: A latency-predictable multi-dimensional optimization framework for dnn-driven autonomous systems. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 371–385, 2020.

[11] Soroush Bateni, Husheng Zhou, Yuankun Zhu, and Cong Liu. Predjoule: A timing-predictable energy optimization framework for deep neural networks. In *2018 IEEE Real-Time Systems Symposium (RTSS)*, pages 107–118. IEEE, 2018.

[12] Romil Bhardwaj, Zhengxu Xia, Ganesh Ananthanarayanan, Junchen Jiang, Yuanchao Shu, Nikolaos Karianakis, Kevin Hsieh, Paramvir Bahl, and Ion Stoica. Ekya: Continuous learning of video analytics models on edge compute servers. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 119–135, 2022.

[13] Urmil Bharti, Deepali Bajaj, Hunar Batra, Shreya Lalit, Shweta Lalit, and Aayushi Gangwani. Medbot: Conversational artificial intelligence powered chatbot for delivering tele-health after covid-19. In *2020 5th International Conference on Communication and Electronics Systems (ICCES)*, pages 870–875. IEEE, 2020.

[14] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. 2020.

[15] Huayu Chen, Guande He, Lifan Yuan, Ganqu Cui, Hang Su, and Jun Zhu. Noise contrastive alignment of language models with explicit rewards. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024.

[16] Simin Chen, Mirazul Haque, Cong Liu, and Wei Yang. Deepperform: An efficient approach for performance testing of resource-constrained neural networks. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, pages 1–13, 2022.

[17] Simin Chen, Hamed Khanpour, Cong Liu, and Wei Yang. Learning to reverse dnns from ai programs automatically. *arXiv preprint arXiv:2205.10364*, 2022.

[18] Simin Chen, Cong Liu, Mirazul Haque, Zihe Song, and Wei Yang. Nmtsloth: understanding and testing efficiency degradation of neural machine translation systems. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1148–1160, 2022.

[19] Simin Chen, Zihe Song, Mirazul Haque, Cong Liu, and Wei Yang. Nicgslowdown: Evaluating the efficiency robustness of neural image caption generation models. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 15365–15374, 2022.

[20] Simin Chen, Shiyi Wei, Cong Liu, and Wei Yang. Dycl: Dynamic neural network compilation via program rewriting and graph optimization. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 614–626, 2023.

[21] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. {TVM}: An automated {End-to-End} optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 578–594, 2018.

[22] Yiming Chen, Simin Chen, Zexin Li, Wei Yang, Cong Liu, Robby Tan, and Haizhou Li. Dynamic transformers provide a false sense of efficiency. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 7164–7180, Toronto, Canada, July 2023. Association for Computational Linguistics.

[23] Yiming Chen, Yan Zhang, Bin Wang, Zuozhu Liu, and Haizhou Li. Generate, discriminate and contrast: A semi-supervised sentence representation learning framework. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pages 8150–8161, Abu Dhabi, United Arab Emirates, December 2022. Association for Computational Linguistics.

[24] Yiming Chen, Yan Zhang, Chen Zhang, Grandee Lee, Ran Cheng, and Haizhou Li. Revisiting self-training for few-shot learning of language model. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 9125–9135, Online and Punta Cana, Dominican Republic, November 2021. Association for Computational Linguistics.

[25] Sangjin Choi, Inhoe Koo, Jeongseob Ahn, Myeongjae Jeon, and Youngjin Kwon. {EnvPipe}: Performance-preserving {DNN} training framework for saving energy. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, pages 851–864, 2023.

[26] Seungbeom Choi, Sunho Lee, Yeonjae Kim, Jongse Park, Youngjin Kwon, and Jaehyuk Huh. Multi-model machine learning inference serving with gpu spatial partitioning. *arXiv preprint arXiv:2109.01611*, 2021.

[27] Seungbeom Choi, Sunho Lee, Yeonjae Kim, Jongse Park, Youngjin Kwon, and Jaehyuk Huh. Serving heterogeneous machine learning models on Multi-GPU servers with Spatio-Temporal sharing. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 199–216, Carlsbad, CA, July 2022. USENIX Association.

[28] Paul F Christiano, Jan Leike, Tom Brown, Miljan Martic, Shane Legg, and Dario Amodei. Deep reinforcement learning from human preferences. *Advances in neural information processing systems*, 30, 2017.

[29] Erhan Cinlar. *Introduction to stochastic processes*. Courier Corporation, 2013.

[30] Josef Dai, Xuehai Pan, Ruiyang Sun, Jiaming Ji, Xinbo Xu, Mickel Liu, Yizhou Wang, and Yaodong Yang. Safe rlhf: Safe reinforcement learning from human feedback. In *The Twelfth International Conference on Learning Representations*, 2023.

[31] Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness. *Advances in neural information processing systems*, 35:16344–16359, 2022.

[32] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Marc'aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, et al. Large scale distributed deep networks. *Advances in neural information processing systems*, 25, 2012.

[33] Emily Dinan, Varvara Logacheva, Valentin Malykh, Alexander Miller, Kurt Shuster, Jack Urbanek, Douwe Kiela, Arthur Szlam, Iulian Serban, Ryan Lowe, et al. The second conversational intelligence challenge (convai2). In *The NeurIPS'18 Competition*, pages 187–208. Springer, 2020.

[34] Assaf Eisenman, Kiran Kumar Matam, Steven Ingram, Dheevatsa Mudigere, Raghuraman Krishnamoorthi, Krishnakumar Nair, Misha Smelyanskiy, and Murali Annavaram. {Check-N-Run}: A checkpointing system for training deep learning recommendation models. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 929–943, 2022.

[35] Kawin Ethayarajh, Yejin Choi, and Swabha Swayamdipta. Understanding dataset difficulty with $\mathcal{V}$-usable information. In *Proceedings of the 39th International Conference on Machine Learning*, volume 162 of *Proceedings of Machine Learning Research*, pages 5988–6008. PMLR, 17–23 Jul 2022.

[36] Anthony Fader, Luke Zettlemoyer, and Oren Etzioni. Open question answering over curated and extracted knowledge bases. In *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '14, page 1156–1165, New York, NY, USA, 2014. Association for Computing Machinery.

[37] Darek Fanton. Edge server. `https://www.onlogic.com/company/io-hub/whatare-edge-servers/`, 2021. Accessed: 2025-06-16.

[38] Shahrooz Feizabadi, William Beebee Jr, Binoy Ravindran, Peng Li, and Martin Rinard. Utilitiy accrual scheduling with real-time java. In *OTM Confederated International Conferences" On the Move to Meaningful Internet Systems"*, pages 550–563. Springer, 2003.

[39] Yao Fu, Leyang Xue, Yeqi Huang, Andrei-Octavian Brabete, Dmitrii Ustiugov, Yuvraj Patel, and Luo Mai. ServerlessLLM: Low-Latency serverless inference for large language models. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 135–153, Santa Clara, CA, July 2024. USENIX Association.

[40] Yao Fu, Leyang Xue, Yeqi Huang, Andrei-Octavian Brabete, Dmitrii Ustiugov, Yuvraj Patel, and Luo Mai. {ServerlessLLM}:{Low-Latency} serverless inference for large language models. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 135–153, 2024.

[41] Yu Fu, Zefan Cai, Abedelkadir Asi, Wayne Xiong, Yue Dong, and Wen Xiao. Not all heads matter: A head-level KV cache compression method with integrated retrieval and reasoning. In *The Thirteenth International Conference on Learning Representations*, 2025.

[42] Yu Fu, Yufei Li, Wen Xiao, Cong Liu, and Yue Dong. Safety alignment in nlp tasks: Weakly aligned summarization as an in-context attack. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 8483–8502, 2024.

[43] Yarin Gal and Zoubin Ghahramani. Dropout as a bayesian approximation: Representing model uncertainty in deep learning. In Maria-Florina Balcan and Kilian Q. Weinberger, editors, *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016*, volume 48 of *JMLR Workshop and Conference Proceedings*, pages 1050–1059. JMLR.org, 2016.

[44] Dawei Gao, Xiaoxi He, Zimu Zhou, Yongxin Tong, Ke Xu, and Lothar Thiele. Rethinking pruning for accelerating deep inference at the edge. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, KDD '20, page 155–164, New York, NY, USA, 2020. Association for Computing Machinery.

[45] Alexander L Gaunt, Matthew A Johnson, Maik Riechert, Daniel Tarlow, Ryota Tomioka, Dimitrios Vytiniotis, and Sam Webster. Ampnet: Asynchronous model-parallel training for dynamic neural networks. *arXiv preprint arXiv:1705.09786*, 2017.

[46] GitHub Copilot. GitHub Copilot. `https://github.com/features/copilot`. Accessed: 2025-06-16.

[47] Ian J Goodfellow, Mehdi Mirza, Da Xiao, Aaron Courville, and Yoshua Bengio. An empirical investigation of catastrophic forgetting in gradient-based neural networks. *arXiv preprint arXiv:1312.6211*, 2013.

[48] GPT AI Team. How many servers are needed to run chatgpt?, August 2024. Accessed: 2024-12-05.

[49] Barbara J. Grosz, Aravind K. Joshi, and Scott Weinstein. Centering: A framework for modeling the local coherence of discourse. *Computational Linguistics*, 21(2):203–225, 1995.

[50] Arpan Gujarati, Reza Karimi, Safya Alzayat, Wei Hao, Antoine Kaufmann, Ymir Vigfusson, and Jonathan Mace. Serving dnns like clockwork: Performance predictability from the bottom up. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020*, pages 443–462. USENIX Association, 2020.

[51] Chulaka Gunasekara, Guy Feigenblat, Benjamin Sznajder, Ranit Aharonov, and Sachindra Joshi. Using question answering rewards to improve abstractive summarization. In *Findings of the Association for Computational Linguistics: EMNLP 2021*, pages 518–526, Punta Cana, Dominican Republic, November 2021. Association for Computational Linguistics.

[52] Chuan Guo, Geoff Pleiss, Yu Sun, and Kilian Q. Weinberger. On calibration of modern neural networks. In Doina Precup and Yee Whye Teh, editors, *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017*, volume 70 of *Proceedings of Machine Learning Research*, pages 1321–1330. PMLR, 2017.

[53] Peizhen Guo, Bo Hu, and Wenjun Hu. Mistify: Automating DNN model porting for On-Device inference at the edge. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 705–719. USENIX Association, April 2021.

[54] Mirazul Haque, Rutvij Shah, Simin Chen, Berrak Sisman, Cong Liu, and Wei Yang. Slothspeech: Denial-of-service attack against speech recognition models. *CoRR*, abs/2306.00794, 2023.

[55] Jianfeng He, Xuchao Zhang, Shuo Lei, Zhiqian Chen, Fanglan Chen, Abdulaziz Alhamadani, Bei Xiao, and ChangTien Lu. Towards more accurate uncertainty estimation in text classification. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 8362–8372, Online, November 2020. Association for Computational Linguistics.

[56] Xiaoxi He, Zimu Zhou, and Lothar Thiele. Multi-task zipping via layer-wise neuron sharing. *Advances in Neural Information Processing Systems*, 31, 2018.

[57] G. Hinton, O. Vinyals, and J. Dean. Distilling the knowledge in a neural network. In *NeurIPS Deep Learning and Representation Learning Workshop*, 2015.

[58] Julia Hirschberg and Diane J. Litman. Empirical studies on the disambiguation of cue phrases. *Comput. Linguistics*, 19(3):501–530, 1993.

[59] Edward J Hu, yelong shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. LoRA: Low-rank adaptation of large language models. In *International Conference on Learning Representations*, 2022.

[60] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, et al. Gpipe: Efficient training of giant neural networks using pipeline parallelism. *Advances in neural information processing systems*, 32, 2019.

[61] Ruben Janssens, Pieter Wolfert, Thomas Demeester, and Tony Belpaeme. Cool glasses, where did you get them?: Generating visually grounded conversation starters for human-robot dialogue. In Daisuke Sakamoto, Astrid Weiss, Laura M. Hiatt, and Masahiro Shiomi, editors, *ACM/IEEE International Conference on Human-Robot Interaction, HRI 2022, Sapporo, Hokkaido, Japan, March 7 - 10, 2022*, pages 821–825. IEEE / ACM, 2022.

[62] Joo Seong Jeong, Jingyu Lee, Donghyun Kim, Changmin Jeon, Changjin Jeong, Youngki Lee, and Byung-Gon Chun. Band: coordinated multi-dnn inference on heterogeneous mobile processors. In *Proceedings of the 20th Annual International Conference on Mobile Systems, Applications and Services*, pages 235–247, 2022.

[63] Mingoo Ji, Saehanseul Yi, Changjin Koo, Sol Ahn, Dongjoo Seo, Nikil D. Dutt, and Jong-Chan Kim. Demand layering for real-time DNN inference with minimized memory usage. In *IEEE Real-Time Systems Symposium, RTSS 2022, Houston, TX, USA, December 5-8, 2022*, pages 291–304. IEEE, 2022.

[64] Zhihao Jia, Matei Zaharia, and Alex Aiken. Beyond data and model parallelism for deep neural networks. *Proceedings of Machine Learning and Systems*, 1:1–13, 2019.

[65] Jingyan Jiang, Ziyue Luo, Chenghao Hu, Zhaoliang He, Zhi Wang, Shutao Xia, and Chuan Wu. Joint model and data adaptation for cloud inference serving. In *42nd IEEE Real-Time Systems Symposium, RTSS 2021, Dortmund, Germany, December 7-10, 2021*, pages 279–289. IEEE, 2021.

[66] Yimin Jiang, Yibo Zhu, Chang Lan, Bairen Yi, Yong Cui, and Chuanxiong Guo. A unified architecture for accelerating distributed {DNN} training in heterogeneous {GPU/CPU} clusters. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 463–479, 2020.

[67] Woo-Sung Kang, Kilho Lee, Jinkyu Lee, Insik Shin, and Hoon Sung Chwa. Lalarand: Flexible layer-by-layer CPU/GPU scheduling for real-time DNN tasks. In *42nd IEEE Real-Time Systems Symposium, RTSS 2021, Dortmund, Germany, December 7-10, 2021*, pages 329–341. IEEE, 2021.

[68] Shinpei Kato, Shota Tokunaga, Yuya Maruyama, Seiya Maeda, Manato Hirabayashi, Yuki Kitsukawa, Abraham Monrroy, Tomohito Ando, Yusuke Fujii, and Takuya Azumi. Autoware on board: Enabling autonomous vehicles with embedded systems. In *2018 ACM/IEEE 9th International Conference on Cyber-Physical Systems (ICCPS)*, pages 287–296. IEEE, 2018.

[69] Branislav Kisačanin. Deep learning for autonomous vehicles. In *2017 IEEE 47th International Symposium on Multiple-Valued Logic (ISMVL)*, pages 142–142. IEEE, 2017.

[70] Aaron Klein, Stefan Falkner, Jost Tobias Springenberg, and Frank Hutter. Learning curve prediction with bayesian neural networks. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017.

[71] Lingkai Kong, Haoming Jiang, Yuchen Zhuang, Jie Lyu, Tuo Zhao, and Chao Zhang. Calibrated language model fine-tuning for in- and out-of-distribution data. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1326–1340, Online, November 2020. Association for Computational Linguistics.

[72] Matthias Kraus, Nicolas Wagner, Wolfgang Minker, Ankita Agrawal, Artur Schmidt, Pranav Krishna Prasad, and Wolfgang Ertel. KURT: A household assistance robot capable of proactive dialogue. In Daisuke Sakamoto, Astrid Weiss, Laura M. Hiatt, and Masahiro Shiomi, editors, *ACM/IEEE International Conference on Human-Robot Interaction, HRI 2022, Sapporo, Hokkaido, Japan, March 7 - 10, 2022*, pages 855–859. IEEE / ACM, 2022.

[73] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In Peter L. Bartlett, Fernando C. N. Pereira, Christopher J. C. Burges, Léon Bottou, and Kilian Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25: 26th Annual Conference on Neural Information Processing Systems 2012. Proceedings of a meeting held December 3-6, 2012, Lake Tahoe, Nevada, United States*, pages 1106–1114, 2012.

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

[75] Zhiquan Lai, Shengwei Li, Xudong Tang, Keshi Ge, Weijie Liu, Yabo Duan, Linbo Qiao, and Dongsheng Li. Merak: An efficient distributed dnn training framework with automated 3d parallelism for giant foundation models. *IEEE Transactions on Parallel and Distributed Systems*, 34(5):1466–1478, 2023.

[76] Langton. Machine learning model training over time, May 2018.

[77] Logan Lebanoff and Fei Liu. Automatic detection of vague words and sentences in privacy policies. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 3508–3517, Brussels, Belgium, October-November 2018. Association for Computational Linguistics.

[78] Kenton Lee, Luheng He, Mike Lewis, and Luke Zettlemoyer. End-to-end neural coreference resolution. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, pages 188–197, Copenhagen, Denmark, September 2017. Association for Computational Linguistics.

[79] Yoonho Lee, Annie S Chen, Fahim Tajwar, Ananya Kumar, Huaxiu Yao, Percy Liang, and Chelsea Finn. Surgical fine-tuning improves adaptation to distribution shifts. In *The Eleventh International Conference on Learning Representations*, 2023.

[80] Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Veselin Stoyanov, and Luke Zettlemoyer. BART: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 7871–7880, Online, July 2020. Association for Computational Linguistics.

[81] Jiamin Li, Hong Xu, Yibo Zhu, Zherui Liu, Chuanxiong Guo, and Cong Wang. Lyra: Elastic scheduling for deep learning clusters. In *Proceedings of the Eighteenth European Conference on Computer Systems*, pages 835–850, 2023.

[82] Shigang Li and Torsten Hoefler. Chimera: efficiently training large-scale neural networks with bidirectional pipelines. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–14, 2021.

[83] Shuyang Li, Yufei Li, Jianmo Ni, and Julian McAuley. SHARE: a system for hierarchical assistive recipe editing. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pages 11077–11090, Abu Dhabi, United Arab Emirates, December 2022. Association for Computational Linguistics.

[84] Xinjin Li, Yu Ma, Yangchen Huang, Xingqi Wang, Yuzhen Lin, and Chenxi Zhang. Integrated optimization of large language models: Synergizing data utilization and compression techniques. 2024.

[85] Yufei Li, Simin Chen, Yanghong Guo, Wei Yang, Yue Dong, and Cong Liu. Uncertainty awareness of large language models under code distribution shifts: A benchmark study. *arXiv preprint arXiv:2402.05939*, 2024.

[86] Yufei Li, Simin Chen, and Wei Yang. Estimating predictive uncertainty under program data distribution shift. *arXiv preprint arXiv:2107.10989*, 2021.

[87] Yufei Li, Zexin Li, Yingfan Gao, and Cong Liu. White-box multi-objective adversarial attack on dialogue generation. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1778–1792, Toronto, Canada, July 2023. Association for Computational Linguistics.

[88] Yufei Li, Zexin Li, Yingfan Gao, and Cong Liu. White-box multi-objective adversarial attack on dialogue generation. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1778–1792, 2023.

[89] Yufei Li, Zexin Li, Wei Yang, and Cong Liu. Rt-lm: Uncertainty-aware resource management for real-time inference of language models. In *2023 IEEE Real-Time Systems Symposium (RTSS)*, pages 158–171. IEEE, 2023.

[90] Yufei Li, John Nham, Ganesh Jawahar, Lei Shu, David Uthus, Yun-Hsuan Sung, Chengrun Yang, Itai Rolnick, Yi Qiao, and Cong Liu. Dr genre: Reinforcement learning from decoupled llm feedback for generic text rewriting. *arXiv preprint arXiv:2503.06781*, 2025.

[91] Yufei Li, Xiao Yu, Yanghong Guo, Yanchi Liu, Haifeng Chen, and Cong Liu. Distantly-supervised joint extraction with noise-robust learning. In *Findings of the Association for Computational Linguistics ACL 2024*, pages 10202–10217, 2024.

[92] Yufei Li, Xiao Yu, Yanchi Liu, Haifeng Chen, and Cong Liu. Uncertainty-aware bootstrap learning for joint extraction on distantly-supervised data. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pages 1349–1358, Toronto, Canada, July 2023. Association for Computational Linguistics.

[93] Yufei Li, Xiao Yu, Yanchi Liu, Haifeng Chen, and Cong Liu. Uncertainty-aware bootstrap learning for joint extraction on distantly-supervised data. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pages 1349–1358, 2023.

[94] Yuhong Li, Yingbing Huang, Bowen Yang, Bharat Venkitesh, Acyr Locatelli, Hanchen Ye, Tianle Cai, Patrick Lewis, and Deming Chen. Snapkv: Llm knows what you are looking for before generation. *Advances in Neural Information Processing Systems*, 37:22947–22970, 2024.

[95] Zexin Li, Jiancheng Zhang, Yufei Li, Yinglun Zhu, and Cong Liu. Mixtraining: A better trade-off between compute and performance. *arXiv preprint arXiv:2502.19513*, 2025.

[96] Zexin Li, Yuqun Zhang, Ao Ding, Husheng Zhou, and Cong Liu. Efficient algorithms for task mapping on heterogeneous cpu/gpu platforms for fast completion time. *Journal of Systems Architecture*, 114:101936, 2021.

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

[98] Zuchao Li, Jiaxun Cai, Shexia He, and Hai Zhao. Seq2seq dependency parsing. In *Proceedings of the 27th International Conference on Computational Linguistics*, pages 3203–3214, Santa Fe, New Mexico, USA, August 2018. Association for Computational Linguistics.

[99] Jian Liang, Ran He, and Tieniu Tan. A comprehensive survey on test-time adaptation under distribution shifts. *International Journal of Computer Vision*, 133(1):31–64, 2025.

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

[101] Yi-Cheng Lin, Kang-Chieh Chen, Zhe-Yan Li, Tzu-Heng Wu, Tzu-Hsuan Wu, Kuan-Yu Chen, Hung-yi Lee, and Yun-Nung Chen. Creativity in llm-based multi-agent systems: A survey. *arXiv preprint arXiv:2505.21116*, 2025.

[102] Jane WSW Liu. Real-time systems. 2000.

[103] Zichang Liu, Jue Wang, Tri Dao, Tianyi Zhou, Binhang Yuan, Zhao Song, Anshumali Shrivastava, Ce Zhang, Yuandong Tian, Christopher Re, et al. Deja vu: Contextual sparsity for efficient llms at inference time. In *International Conference on Machine Learning*, pages 22137–22176. PMLR, 2023.

[104] Daniel Loureiro and Alípio Jorge. Language modelling makes sense: Propagating representations through WordNet for full-coverage word sense disambiguation. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 5682–5691, Florence, Italy, July 2019. Association for Computational Linguistics.

[105] Andrew Mao, Naveen Raman, Matthew Shu, Eric Li, Franklin Yang, and Jordan Boyd-Graber. Eliciting bias in question answering models through ambiguity. In *Proceedings of the 3rd Workshop on Machine Reading for Question Answering*, pages 92–99, Punta Cana, Dominican Republic, November 2021. Association for Computational Linguistics.

[106] Viyom Mittal, Shixiong Qi, Ratnadeep Bhattacharya, Xiaosu Lyu, Junfeng Li, Sameer G Kulkarni, Dan Li, Jinho Hwang, KK Ramakrishnan, and Timothy Wood. Mu: An efficient, fair and responsive serverless framework for resource-constrained edge clouds. In *Proceedings of the ACM symposium on cloud computing*, pages 168–181, 2021.

[107] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R Devanur, Gregory R Ganger, Phillip B Gibbons, and Matei Zaharia. Pipedream: generalized pipeline parallelism for dnn training. In *Proceedings of the 27th ACM symposium on operating systems principles*, pages 1–15, 2019.

[108] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, et al. Efficient large-scale language model training on gpu clusters using megatron-lm. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15, 2021.

[109] Vinod Nigade, Pablo Bauszat, Henri E. Bal, and Lin Wang. Jellyfish: Timely inference serving for dynamic edge networks. In *IEEE Real-Time Systems Symposium, RTSS 2022, Houston, TX, USA, December 5-8, 2022*, pages 277–290. IEEE, 2022.

[110] Li Ning, Hamid Shojanazeri, Ke Wen, and the PyTorch Foundation. Torchserve: Serve, optimize and scale pytorch models in production. PyTorch Foundation, 2023. https://pytorch.org/serve/.

[111] Wei Niu, Zhenglun Kong, Geng Yuan, Weiwen Jiang, Jiexiong Guan, Caiwen Ding, Pu Zhao, Sijia Liu, Bin Ren, and Yanzhi Wang. Real-time execution of large-scale language models on mobile. *arXiv preprint arXiv:2009.06823*, 2020.

[112] NVIDIA. Duckiebot (db-j). https://get.duckietown.com/products/duckiebot-db21, 2022.

[113] NVIDIA. Sparkfun jetbot ai kit. https://www.sparkfun.com/products/18486, 2022.

[114] NVIDIA. Waveshare jetbot ai kit. https://www.amazon.com/Waveshare-JetBot-AI-Kit-Accessories/dp/B07V8JL4TF/, 2022.

[115] NVIDIA Corporation. Triton inference server: An optimized cloud and edge inferencing solution, 2019. https://developer.nvidia.com/nvidia-triton-inference-server.

[116] Kazuki Osawa, Shigang Li, and Torsten Hoefler. Pipefisher: Efficient training of large language models using pipelining and fisher information matrices. *Proceedings of Machine Learning and Systems*, 5, 2023.

[117] Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul F Christiano, Jan Leike, and Ryan Lowe. Training language models to follow instructions with human feedback. In S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh, editors, *Advances in Neural Information Processing Systems*, volume 35, pages 27730–27744. Curran Associates, Inc., 2022.

[118] Yaniv Ovadia, Emily Fertig, Jie Ren, Zachary Nado, David Sculley, Sebastian Nowozin, Joshua Dillon, Balaji Lakshminarayanan, and Jasper Snoek. Can you trust your model's uncertainty? evaluating predictive uncertainty under dataset shift. *Advances in neural information processing systems*, 32, 2019.

[119] Pratyush Patel, Esha Choukse, Chaojie Zhang, Aashaka Shah, Íñigo Goiri, Saeed Maleki, and Ricardo Bianchini. Splitwise: Efficient generative llm inference using phase splitting. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*, pages 118–132. IEEE, 2024.

[120] Baolin Peng, Michel Galley, Pengcheng He, Chris Brockett, Lars Liden, Elnaz Nouri, Zhou Yu, Bill Dolan, and Jianfeng Gao. GODEL: large-scale pre-training for goal-directed dialog. *CoRR*, abs/2206.11309, 2022.

[121] Perplexity AI. Perplexity AI. https://www.perplexity.ai/. Accessed: 2025-06-16.

[122] Reiner Pope, Sholto Douglas, Aakanksha Chowdhery, Jacob Devlin, James Bradbury, Jonathan Heek, Kefan Xiao, Shivani Agrawal, and Jeff Dean. Efficiently scaling transformer inference. *Proceedings of Machine Learning and Systems*, 5:606–624, 2023.

[123] Alexander Popov, Patrik Gebhardt, Ke Chen, Ryan Oldja, Heeseok Lee, Shane Murray, Ruchi Bhargava, and Nikolai Smolyanskiy. Nvradarnet: Real-time radar obstacle and free space detection for autonomous driving. *arXiv preprint arXiv:2209.14499*, 2022.

[124] Shuofei Qiao, Ningyu Zhang, Runnan Fang, Yujie Luo, Wangchunshu Zhou, Yuchen Jiang, Chengfei Lv, and Huajun Chen. Autoact: Automatic agent learning from scratch for qa via self-planning. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 3003–3021, 2024.

[125] Rafael Rafailov, Archit Sharma, Eric Mitchell, Christopher D Manning, Stefano Ermon, and Chelsea Finn. Direct preference optimization: Your language model is secretly a reward model. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023.

[126] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *J. Mach. Learn. Res.*, 21:140:1–140:67, 2020.

[127] Hassan Ramchoun, Youssef Ghanou, Mohamed Ettaouil, and Mohammed Amine Janati Idrissi. Multilayer perceptron: Architecture optimization and training. 2016.

[128] Hannah Rashkin, Eric Michael Smith, Margaret Li, and Y-Lan Boureau. Towards empathetic open-domain conversation models: A new benchmark and dataset. In Anna Korhonen, David R. Traum, and Lluís Màrquez, editors, *Proceedings of the 57th Conference of the Association for Computational Linguistics, ACL 2019, Florence, Italy, July 28- August 2, 2019, Volume 1: Long Papers*, pages 5370–5381. Association for Computational Linguistics, 2019.

[129] Brandon Reagen, Yakun Sophia Shao, Sam Likun Xi, Gu-Yeon Wei, and David Brooks. Methods and infrastructure in the era of accelerator-centric architectures. In *2017 IEEE 60th International Midwest Symposium on Circuits and Systems (MWSCAS)*, pages 902–905. IEEE, 2017.

[130] Stephen Roller, Emily Dinan, Naman Goyal, Da Ju, Mary Williamson, Yinhan Liu, Jing Xu, Myle Ott, Eric Michael Smith, Y-Lan Boureau, and Jason Weston. Recipes for building an open-domain chatbot. In *Proceedings of the 16th Conference of the European Chapter of the Association for Computational Linguistics: Main Volume*, pages 300–325, Online, April 2021. Association for Computational Linguistics.

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

[132] Sheldon M Ross. *Introduction to probability models*. Academic press, 2014.

[133] Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. Distilbert, a distilled version of bert: smaller, faster, cheaper and lighter. *arXiv preprint arXiv:1910.01108*, 2019.

[134] Vishnu Sarukkai, Zhiqiang Xie, and Kayvon Fatahalian. Self-generated in-context examples improve llm agents for sequential decision-making tasks. *arXiv preprint arXiv:2505.00234*, 2025.

[135] Alexander Sergeev and Mike Del Balso. Horovod: fast and easy distributed deep learning in tensorflow. *arXiv preprint arXiv:1802.05799*, 2018.

[136] Artem Shelmanov, Evgenii Tsymbalov, Dmitri Puzyrev, Kirill Fedyanin, Alexander Panchenko, and Maxim Panov. How certain is your Transformer? In *Proceedings of the 16th Conference of the European Chapter of the Association for Computational Linguistics: Main Volume*, pages 1833–1840, Online, April 2021. Association for Computational Linguistics.

[137] Yifei Shen, Jiawei Shao, Xinjie Zhang, Zehong Lin, Hao Pan, Dongsheng Li, Jun Zhang, and Khaled B. Letaief. Large language models empowered autonomous edge AI for connected intelligence. *CoRR*, abs/2307.02779, 2023.

[138] Yifei Shen, Jiawei Shao, Xinjie Zhang, Zehong Lin, Hao Pan, Dongsheng Li, Jun Zhang, and Khaled B Letaief. Large language models empowered autonomous edge ai for connected intelligence. *IEEE Communications Magazine*, 2024.

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

[140] Guangyuan Shi, Zexin Lu, Xiaoyu Dong, Wenlong Zhang, Xuanyu Zhang, Yujie Feng, and Xiao-Ming Wu. Understanding layer significance in llm alignment. *arXiv preprint arXiv:2410.17875*, 2024.

[141] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053*, 2019.

[142] Sudipta Saha Shubha and Haiying Shen. Adainf: Data drift adaptive scheduling for accurate and slo-guaranteed multiple-model inference serving at edge servers. In *Proceedings of the ACM SIGCOMM 2023 Conference*, pages 473–485, 2023.

[143] Eric Michael Smith, Mary Williamson, Kurt Shuster, Jason Weston, and Y-Lan Boureau. Can you put it all together: Evaluating conversational agents' ability to blend skills. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 2021–2030, Online, July 2020. Association for Computational Linguistics.

[144] Foteini Strati, Sara Mcallister, Amar Phanishayee, Jakub Tarnawski, and Ana Klimovic. Déjàvu: KV-cache streaming for fast, fault-tolerant generative LLM serving. In Ruslan Salakhutdinov, Zico Kolter, Katherine Heller, Adrian Weller, Nuria Oliver, Jonathan Scarlett, and Felix Berkenkamp, editors, *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pages 46745–46771. PMLR, 21–27 Jul 2024.

[145] Biao Sun, Ziming Huang, Hanyu Zhao, Wencong Xiao, Xinyi Zhang, Yong Li, and Wei Lin. Llumnix: Dynamic scheduling for large language model serving. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, 2024.

[146] Yu Sun, Xiaolong Wang, Zhuang Liu, John Miller, Alexei Efros, and Moritz Hardt. Test-time training with self-supervision for generalization under distribution shifts. In *International conference on machine learning*, pages 9229–9248. PMLR, 2020.

[147] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. Sequence to sequence learning with neural networks. In Zoubin Ghahramani, Max Welling, Corinna Cortes, Neil D. Lawrence, and Kilian Q. Weinberger, editors, *Advances in Neural Information Processing Systems 27: Annual Conference on Neural Information Processing Systems 2014, December 8-13 2014, Montreal, Quebec, Canada*, pages 3104–3112, 2014.

[148] Yi Tay, Mostafa Dehghani, Dara Bahri, and Donald Metzler. Efficient transformers: A survey. *ACM Computing Surveys*, 55(6):1–28, 2022.

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

[150] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, pages 5998–6008, 2017.

[151] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[152] Alexandre Vilcek. Optimized training and inference of hugging face models on azure, September 2022.

[153] Oriol Vinyals, Alexander Toshev, Samy Bengio, and Dumitru Erhan. Show and tell: A neural image caption generator. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2015, Boston, MA, USA, June 7-12, 2015*, pages 3156–3164. IEEE Computer Society, 2015.

[154] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837, 2022.

[155] Bingyang Wu, Zili Zhang, Zhihao Bai, Xuanzhe Liu, and Xin Jin. Transparent {GPU} sharing in container clouds for deep learning workloads. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 69–85, 2023.

[156] Bingyang Wu, Yinmin Zhong, Zili Zhang, Shengyu Liu, Fangyue Liu, Yuanhang Sun, Gang Huang, Xuanzhe Liu, and Xin Jin. Fast distributed inference serving for large language models. *arXiv preprint arXiv:2305.05920*, 2023.

[157] Bingyang Wu, Ruidong Zhu, Zili Zhang, Peng Sun, Xuanzhe Liu, and Xin Jin. {dLoRA}: Dynamically orchestrating requests and adapters for {LoRA}{LLM} serving. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 911–927, 2024.

[158] Mingzhu Wu, Nafise Sadat Moosavi, Dan Roth, and Iryna Gurevych. Coreference reasoning in machine reading comprehension. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 5768–5781, Online, August 2021. Association for Computational Linguistics.

[159] Yecheng Xiang and Hyoseung Kim. Pipelined data-parallel cpu/gpu scheduling for multi-dnn real-time inference. In *2019 IEEE Real-Time Systems Symposium (RTSS)*, pages 392–405. IEEE, 2019.

[160] Canwen Xu and Julian McAuley. A survey on model compression for natural language processing. *arXiv preprint arXiv:2202.07105*, 2022.

[161] Minrui Xu, Hongyang Du, Dusit Niyato, Jiawen Kang, Zehui Xiong, Shiwen Mao, Zhu Han, Abbas Jamalipour, Dong In Kim, Xuemin Shen, Victor C. M. Leung, and H. Vincent Poor. Unleashing the power of edge-cloud generative AI in mobile networks: A survey of AIGC services. *CoRR*, abs/2303.16129, 2023.

[162] Jiudong Yang, Peiying Wang, Yi Zhu, Mingchao Feng, Meng Chen, and Xiaodong He. Gated multimodal fusion with contrastive learning for turn-taking prediction in human-robot dialogue. In *IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP 2022, Virtual and Singapore, 23-27 May 2022*, pages 7747–7751. IEEE, 2022.

[163] Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Tom Griffiths, Yuan Cao, and Karthik Narasimhan. Tree of thoughts: Deliberate problem solving with large language models. *Advances in Neural Information Processing Systems*, 36, 2024.

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

[165] Lily H Zhang and Rajesh Ranganath. Preference learning made easy: Everything should be understood through win rate. *arXiv preprint arXiv:2502.10505*, 2025.

[166] Saizheng Zhang, Emily Dinan, Jack Urbanek, Arthur Szlam, Douwe Kiela, and Jason Weston. Personalizing dialogue agents: I have a dog, do you have pets too? In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 2204–2213, Melbourne, Australia, July 2018. Association for Computational Linguistics.

[167] Yizhe Zhang, Siqi Sun, Michel Galley, Yen-Chun Chen, Chris Brockett, Xiang Gao, Jianfeng Gao, Jingjing Liu, and Bill Dolan. DIALOGPT : Large-scale generative pre-training for conversational response generation. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics: System Demonstrations*, pages 270–278, Online, July 2020. Association for Computational Linguistics.

[168] Yizhe Zhang, Siqi Sun, Michel Galley, Yen-Chun Chen, Chris Brockett, Xiang Gao, Jianfeng Gao, Jingjing Liu, and William B Dolan. Dialogpt: Large-scale generative pre-training for conversational response generation. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics: System Demonstrations*, pages 270–278, 2020.

[169] Yilong Zhao, Shuo Yang, Kan Zhu, Lianmin Zheng, Baris Kasikci, Yang Zhou, Jiarong Xing, and Ion Stoica. Blendserve: Optimizing offline inference for auto-regressive large models with resource-aware batching. *arXiv preprint arXiv:2411.16102*, 2024.

[170] Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Tianle Li, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zhuohan Li, Zi Lin, Eric Xing, et al. Lmsys-chat-1m: A large-scale real-world llm conversation dataset. In *The Twelfth International Conference on Learning Representations*, 2023.

[171] Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric Xing, et al. Judging llm-as-a-judge with mt-bench and chatbot arena. *Advances in Neural Information Processing Systems*, 36:46595–46623, 2023.

[172] Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric Xing, et al. Judging llm-as-a-judge with mt-bench and chatbot arena. *Advances in Neural Information Processing Systems*, 36, 2024.

[173] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Livia Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E Gonzalez, et al. Sglang: Efficient execution of structured language model programs. *Advances in neural information processing systems*, 37:62557–62583, 2024.

[174] Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. {DistServe}: Disaggregating prefill and decoding for goodput-optimized large language model serving. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 193–210, 2024.

[175] Husheng Zhou, Soroush Bateni, and Cong Liu. Sˆ 3dnn: Supervised streaming and scheduling for gpu-accelerated real-time dnn workloads. In *2018 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 190–201. IEEE, 2018.

[176] Li Zhou, Jianfeng Gao, Di Li, and Heung-Yeung Shum. The design and implementation of XiaoIce, an empathetic social chatbot. *Computational Linguistics*, 46(1):53–93, 2020.

[177] Ming Zhu, Aman Ahuja, Da-Cheng Juan, Wei Wei, and Chandan K. Reddy. Question answering with long multiple-span answers. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 3840–3849, Online, November 2020. Association for Computational Linguistics.