CoRT: Supporting Hard and Soft Real-Time Tasks in Batched LLM Serving

Anonymous Author(s)

***Abstract*—Recent advances in reasoning capabilities have made large language models (LLMs) promising for safety-critical systems, where a single model serves both hard and soft real-time tasks. However, jointly supporting them is difficult because batched GPU execution can break temporal isolation among tasks, and the autoregressive nature of LLMs makes execution time hard to predict. In this paper, we propose CoRT, a real-time LLM framework that guarantees hard-task deadlines while improving soft-task responsiveness. To satisfy these heterogeneous timing requirements, CoRT regulates LLM execution time by reducing autoregressive computation through three control knobs: early exit, output-length control, and model selection. We implement CoRT in PyTorch and demonstrate that it achieves up to 4.9× higher schedulability for hard tasks and 7× higher responsiveness for soft tasks than prior approaches. To the best of our knowledge, this is the first work that jointly supports hard and soft tasks on batched LLM serving platforms.**

I. Introduction

Recent advances in reasoning capabilities have made LLMs increasingly attractive for safety-critical systems. In these domains, LLMs are emerging as high-level decision-support modules for perception and control beyond natural language processing. For example, they can support action planning in autonomous driving [1]–[3] and robotics [4]–[8], as well as real-time anomaly detection from time-series data (e.g., sensor streams or system logs) [9]–[11]. These trends highlight the growing role of LLMs in safety-critical domains.

In such systems, the broad generality of LLMs allows a single model to concurrently serve diverse workloads, including both *hard* and *soft* real-time tasks. Hard tasks correspond to safety-critical functions, such as action planning or anomaly monitoring, and must meet strict deadlines. In contrast, soft tasks arise from on-demand user interactions, such as question answering, and require high responsiveness while tolerating occasional deadline misses. Jointly supporting these tasks is challenging because their timing requirements fundamentally conflict: soft-task requests arrive unexpectedly and delay hard tasks, whereas conservative provisioning for hard tasks can significantly reduce soft-task responsiveness.

The real-time systems community has long studied the joint scheduling of hard and soft tasks. A conventional approach is *server-based scheduling*, which encapsulates asynchronous soft tasks within a dedicated server to provide temporal isolation from hard tasks while allowing safe interleaving. However, this approach is insufficient for LLM serving systems for two reasons. First, LLM serving platforms typically execute multiple tasks in GPU batches, where temporal isolation can break even under server-based scheduling. In particular, hard tasks can be delayed by co-batched tasks because tasks within a batch progress synchronously, potentially causing deadline violations. Second, the autoregressive nature of LLMs makes task execution time inherently unpredictable. This is especially severe for soft tasks with open-ended responses, whose output lengths can vary substantially across queries. As a result, even when runtime resources are available, soft tasks may fail to achieve satisfactory quality of service (QoS), because their responses can become unexpectedly long.

To address these problems, we propose CoRT (**Co**-batching **R**eal-**T**ime LLM tasks), a real-time LLM framework that guarantees hard-task deadlines while improving soft-task responsiveness. CoRT meets heterogeneous timing requirements by reducing LLM execution time via autoregressive computation control. Since LLM execution time is primarily determined by the number of decoding iterations, decoder layers executed per iteration, and model scale (i.e., layer dimensions), CoRT exposes three control knobs to regulate these factors: *i) early exit* to skip decoder layers, *ii) output-length control* to bound generated tokens, and *iii) model selection* to choose a smaller model variant. Specifically, CoRT first selects a model variant and an early-exit point that guarantee hard-task deadlines, and then performs output-length control at runtime to improve soft-task responsiveness.

However, realizing this approach raises three main challenges. *C1)* Jointly supporting early exit and output-length control can amplify accuracy loss. To mitigate this, CoRT adopts selective fine-tuning by restricting early-exit learning to robust later layers and enabling output-length control via special tokens. *C2)* Selecting model variants and early-exit points to satisfy timing constraints is difficult under batched GPU execution, where worst-case interference depends on runtime co-batching. To address this challenge, CoRT introduces a batch-aware response-time analysis (RTA) and uses it to search for feasible configurations. *C3)* Choosing an appropriate output length for soft tasks is non-trivial because the appropriate response length depends on query type, while soft-task queries are often open-ended. CoRT addresses this by classifying query types, estimating suitable response length ranges, and determining runtime output length based on both the estimated range and residual execution capacity.

We implemented CoRT in PyTorch [12] on NVIDIA Jetson AGX Orin. Our simulation and real-system experiments show that CoRT achieves up to 4.9× higher hard-task schedulability and 7× higher soft-task responsiveness than prior approaches. The main contributions of this paper are as follows:

- To the best of our knowledge, the first framework that jointly supports hard and soft tasks on LLM serving platforms.
- A selective fine-tuning method that enables early exit and output-length control with little accuracy loss.
- A batch-aware response-time analysis that upper-bounds worst-case response times under batched GPU execution and guides feasible configuration search.
- An adaptive output-length control method that determines appropriate response lengths based on query type and runtime-available execution capacity.

II. Related Work

Hard/soft real-time task scheduling. The real-time systems community has long studied the co-scheduling of hard and soft real-time tasks, primarily through server-based scheduling. In these approaches, soft tasks are encapsulated within a dedicated scheduling entity, or *server*, with a reserved execution budget to avoid violating hard-task deadlines. Prior work [13]–[15] proposed a variety of server mechanisms for this purpose, and subsequent studies [16], [17] developed co-scheduling frameworks built on top of them. However, these approaches assume that temporal isolation can be enforced at the scheduler level. This assumption does not directly hold in real-time LLM serving systems, where multiple requests are synchronously executed in GPU batches. Under such batched execution, co-batched tasks can interfere with one another and increase response

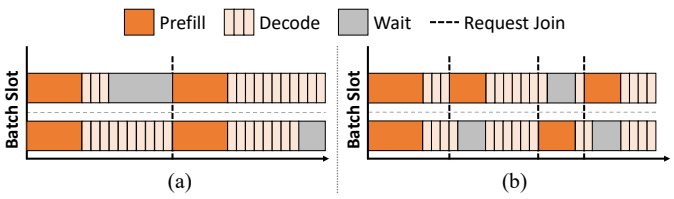

Fig. 1. (a) Static batching and (b) continuous batching

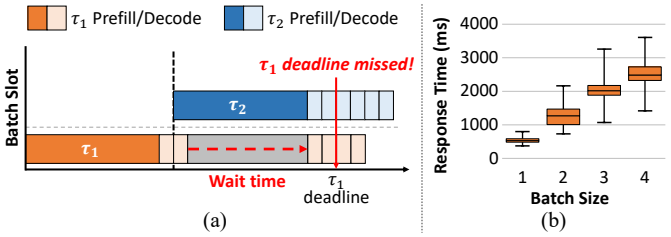

Fig. 2. (a) Example for temporal-isolation breakdown, and (b) the response-time distributions of a hard task under different batch sizes

times, potentially undermining the isolation intended by server-based scheduling. We discuss this limitation in detail in Sec. III-B.

LLM serving systems. System-level optimizations for LLM serving have received significant attention, largely targeting best-effort metrics such as latency and throughput. Orca [18] introduced continuous batching to improve GPU utilization. vLLM [19] used paged attention to address memory fragmentation and support higher request concurrency, while Sarathi-Serve [20] proposed chunked prefill to reduce interference with decoding requests. Prior work [21]–[23] has also explored scheduling strategies to reduce latency by optimizing execution order and mitigating queuing delays. Other studies [24], [25] used output-length prediction to improve task scheduling, thereby reducing average response times. In contrast, our goal is not average-case efficiency, but guaranteeing hard-task deadlines while improving soft-task responsiveness under batched GPU execution.

Controlling LLM inference execution. Prior work has explored reducing LLM inference cost through several mechanisms. Static methods [26]–[28], such as quantization, pruning, and distillation, reduce model size offline and are orthogonal to our framework. Other approaches instead control inference behavior through early exit [29], model selection [30], and output-length control [31]. While these mechanisms have been studied individually, to the best of our knowledge, our work is the first to integrate them toward jointly supporting hard and soft real-time tasks in batched LLM serving. More importantly, this integration creates new challenges in fine-tuning, timing analysis, and runtime output-length selection, which are central to our framework.

III. Background and Motivation

A. Background: LLM Serving Systems

Autoregressive inference. Modern LLMs, such as Qwen [32] and Llama [33], are based on a decoder-only architecture, which consists of three components: a *token embedding layer* that converts input tokens into embeddings, a stack of *decoder layers* that compute contextualized representations through self-attention and feed-forward networks, and a final *language model (LM) head* that projects hidden states to vocabulary logits for next-token prediction.

LLM inference is an autoregressive process in which the model performs one forward pass per iteration and generates exactly one output token. This process operates in two phases: *prefill* and *decoding*. In the prefill phase (i.e., the first iteration), the model processes the entire input prompt to compute the initial hidden states and generates the first output token, making its latency proportional to the input length. The model then enters the decoding phase, where each iteration generates one new token conditioned on the prompt and previously generated tokens. To avoid redundant computation, the model uses a *KV cache* to store and reuse intermediate activations of prior tokens. Although each decoding iteration is typically much shorter than prefill, decoding repeats until an end-of-sequence (EOS) token is produced. Thus, for long outputs, the total decoding latency can exceed the prefill latency.

Batch execution model. To improve throughput, LLM serving systems execute multiple requests in batches on the GPU. Instead of processing each request independently, the system groups several requests into a single batch to better exploit hardware parallelism. Traditional static batching, however, is inefficient because the next batch can start only after all requests in the current batch complete, as illustrated in Fig. 1(a). To address this limitation, modern serving systems commonly adopt *continuous batching* [18], as shown in Fig. 1(b). Unlike static batching, continuous batching allows requests to be scheduled at every iteration. At each scheduling point, completed requests can leave the batch, newly arrived requests can join it, and existing requests may be preempted by higher-priority ones. This flexibility improves GPU utilization and overall serving efficiency.

B. Motivation

In this paper, we consider LLM serving systems in safety-critical environments, where a single LLM instance serves both hard and soft real-time tasks. Hard tasks correspond to safety-critical functions that are released periodically and must complete before their deadlines. Because their objectives are well defined, predefined templates are often used for both prompts and outputs. In contrast, soft tasks involve user-facing functions such as open-ended question answering. They are released aperiodically by user queries and require high responsiveness, but can tolerate occasional deadline misses. Their prompts and outputs are typically less structured and more diverse than those of hard tasks. As discussed in Sec. II, a conventional approach to jointly supporting such heterogeneous tasks is *server-based scheduling*, which encapsulates soft tasks within a dedicated scheduling entity to enforce temporal isolation from hard tasks. However, this approach alone is insufficient in LLM serving systems for two reasons: batched execution can break temporal isolation, and autoregressive generation makes execution time difficult to predict.

P1. Breakdown of temporal isolation. Batch execution can undermine temporal isolation across tasks. As shown in Fig. 2(a), continuous batching allows a new task τ_2 to join an active batch while τ_1 is executing. If τ_1 is in decoding and τ_2 enters the batch in prefill, τ_1 must wait until τ_2 completes prefill. Because a prefill iteration is typically much longer than a decoding iteration, this delay can cause τ_1 to miss its deadline. Such interference persists even when τ_2 is a soft task encapsulated in a server. One possible mitigation is *chunked prefill* [20], which splits prefill into smaller chunks. However, each chunk still processes hundreds to thousands of tokens (e.g., 512–8,192), which can remain longer than a decoding iteration and thus still introduce blocking. Although using smaller chunks can further reduce this blocking, it also significantly degrades GPU utilization.

To empirically demonstrate this breakdown of temporal isolation, we run one hard task while continuously issuing additional user queries (i.e., soft tasks) to induce co-batching. The hard task uses Loghub queries [34] for anomaly detection and is released with a period of 2,000 ms. The co-batched user queries are drawn from the No Robots dataset [35], representing instruction-following workloads. The hard task is assigned a higher priority than these queries. Fig. 2(b) shows the response-time distributions of the hard task under different batch sizes. At batch size 1, the hard task experiences

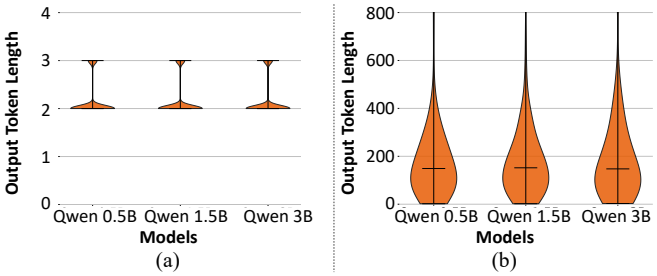

Fig. 3. Output-length distributions for (a) hard tasks and (b) soft tasks

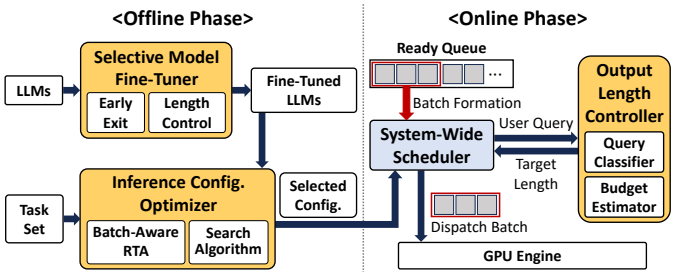

Fig. 4. Overview of CoRT system design

little interference from user queries, effectively preserving temporal isolation from soft tasks. As the batch size increases, however, response times become longer and more variable due to frequent co-batching and prefill delays caused by newly arriving queries.

P2. Output-length variability. The second challenge stems from the autoregressive nature of LLMs, which makes task execution times difficult to predict. To examine this issue, we measure output token lengths produced by Qwen-2.5 models of three sizes (0.5B, 1.5B, and 3B) for both hard- and soft-task queries. We use Loghub for hard tasks and No Robots for soft tasks, and fine-tune each model on Loghub so that it can generate correct responses for this domain-specific workload. Fig. 3 shows the resulting output-length distributions. Soft tasks exhibit much larger variation because they involve diverse queries and lack fixed output templates. This makes it difficult to provide satisfactory QoS, since user-query responses can become unexpectedly long and lead to highly variable response times. In contrast, hard tasks produce more consistent output lengths due to their predefined templates. Nevertheless, they may still occasionally exceed the expected length [36], which can be handled by truncating excess output tokens when needed to preserve timing predictability.

IV. System Design

A. System Overview

In this paper, we present CoRT (**Co**-batching **R**eal-**T**ime LLM tasks), a real-time LLM framework that guarantees hard-task deadlines while improving soft-task responsiveness. The challenges discussed in Sec. III-B lead to deadline misses for hard tasks and degraded QoS for soft tasks. CoRT addresses these challenges by reducing autoregressive computation. Since this computation is primarily determined by the number of decoder layers, decoding iterations, and model scale (i.e., layer dimensions), CoRT exposes three control knobs to regulate these factors: *early exit* to skip decoder layers, *output-length control* to bound generated tokens, and *model selection* to choose a smaller model variant. Through these knobs, CoRT can adapt LLM execution to meet the heterogeneous timing requirements.

However, realizing this approach requires addressing three challenges. *C1)* Jointly training a model to support both early exit and output-length control can significantly degrade model accuracy. Output-length control already imposes a constraint that may reduce model accuracy, and early exit further amplifies this effect by requiring the model to generate accurate outputs at earlier layers with limited semantic information. *C2)* Guaranteeing hard-task deadlines requires carefully selecting each task's inference configuration, such as its model variant and early-exit point, while accounting for worst-case interference from other tasks. This is particularly challenging under batched execution, where interference depends on runtime co-batching. *C3)* Determining appropriate output lengths for soft tasks is difficult, because suitable response lengths vary across user queries; overly short outputs can significantly degrade quality.

To address these challenges, CoRT consists of several components operating in two phases, as illustrated in Fig. 4.

Offline phase. Given base LLMs, *i) the selective model fine-tuner* jointly enables early exit and output-length control while restricting early-exit training to later layers and using special tokens for output-length control to mitigate accuracy loss. Given a task set, *ii) the inference-config optimizer* uses our batch-aware RTA to estimate each hard task's worst-case response time (WCRT) under batched execution, and based on this, determines a feasible inference configuration—i.e., a pair of a model variant and an early-exit point—and a batch-size limit that satisfy all hard-task deadlines.

Online phase. As tasks arrive, their requests are inserted into a ready queue. *iii) the system-wide scheduler* orders ready tasks by priority, forms a batch group of the highest-priority tasks without exceeding the batch-size limit, and dispatches the batch to the GPU using the selected inference configuration. Meanwhile, when a soft task arrives, *iv) the output-length controller* identifies its query type, estimates an appropriate response-length range, and selects an output length based on both this range and the residual execution capacity available to the server. The server then issues the request with the selected output length when it is admitted to the batch group.

The rest of this section details CoRT's key components. Sec. IV-B describes selective fine-tuning, Sec. IV-C presents batch-aware configuration optimization, and Sec. IV-D introduces adaptive output-length control for soft tasks.

B. Selective Model Fine-Tuning

To control both the number of decoder layers and decoding iterations, the base LLM must be trained to support early exit and output-length control. However, naively training these capabilities together causes substantial accuracy loss due to *objective interference*. Early exit requires intermediate layers to produce final predictions; when coupled with output-length control, this enforces each layer to generate answers under length constraints. Because early layers in LLMs lack sufficient contextual information for both objectives [37], joint training significantly degrades accuracy. To mitigate this interference, CoRT restricts early-exit training to later decoder layers while enabling output-length control via special tokens.

1) Selective early-exit training: For early-exit learning, we adopt a shared LM head [29], which projects the hidden states of all decoder layers into the same logit space for token prediction. Rather than training all layers, we freeze the first half of the decoder layers and train only the remaining layers, since earlier layers generally provide less refined representations. This restricts early exit to later layers and mitigates objective interference. The loss function for early-exit training is defined as:

$$\mathcal{L} = -\sum_{(p_i,r_i)\in\mathcal{D}} \sum_{l=\lceil L/2\rceil}^{L} \alpha_l \sum_{t=1}^{|r_i|} \log q_l(y_t \mid y_{<t}, p_i) \tag{1}$$

where $\mathcal{D}$ is the training dataset of prompt-response pairs (p_i, r_i), including datasets such as OpenHermes2.5 [38] and LMSYS_Chat [39]; L is the total number of decoder layers; y_t is the t-th token; q_l is

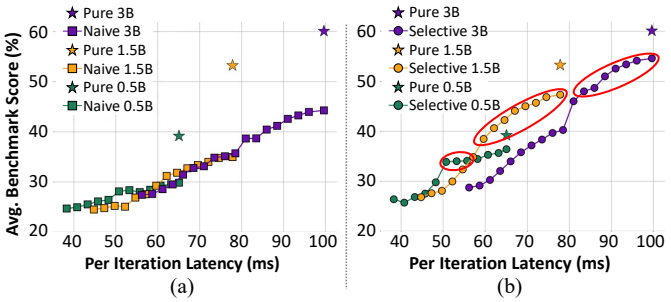

Fig. 5. Benchmark score comparison of (a) *naive* and (b) *selective*. For each line, the rightmost point denotes using all layers, each point to the left represents removing one additional layer.

the shared-head output distribution at layer l; and α_l is a layer-wise weight used for curriculum learning.

2) Special tokens for output-length control: To precisely control the number of decoding iterations, we adopt a special token-based learning strategy inspired by prior work [31], but tailor it to token-level length control. Prior work regulates output length by word count, which is insufficient for fine-grained execution-time control because a single word may correspond to multiple tokens. We therefore define a set of special tokens $S = \{s_i\}_{i=1}^{n}$, where each $s_i \in \{8, 15, 45, 75, 120, 225, 450, 750\}$ specifies the maximum response length in tokens. These values cover a broad range of response lengths, with finer granularity for short responses and coarser granularity for long responses.

During training, each prompt is augmented with a length-control token s_i, and the model learns to generate responses within the specified token budget. We extend Eq. (1) as follows:

$$\mathcal{L} = -\sum_{(p_i,s_i,r_i)\in\mathcal{D}'} \sum_{l=\lceil L/2\rceil}^{L} \alpha_l \sum_{t=1}^{|r_i|} \log q_l(y_t \mid y_{<t}, p_i, s_i) \qquad (2)$$

where $\mathcal{D}'$ is the training set formed by augmenting $\mathcal{D}$ with length-control tokens, and s_i specifies the token budget for r_i. This enables joint learning of early exit and output-length control.

3) Model validation: To validate our method, we evaluate five base models—Qwen-2.5 (0.5B, 1.5B, and 3B) and Llama-3.2 (1B and 3B)—on standard benchmarks[1]. For each model, we compare three variants: *i) pure* without additional fine-tuning, *ii) naive* fine-tuned across all layers for early exit and output-length control, and *iii) selective* trained using our method.

Fig. 5(a) and (b) show the average benchmark score and decoder-layer latency across early-exit points for Qwen-2.5 models using *naive* and *selective*, respectively. With full layers, *selective* reduces the benchmark-score drop from *pure* by up to 70.2% compared with *naive*. This indicates that *selective* effectively mitigates the performance loss caused by jointly enabling early exit and output-length control. Similar trends were observed for Llama-3.2 models and are omitted for brevity. In addition, as the early-exit point moves earlier and model size decreases, decoder-layer latency decreases nearly linearly, accompanied by a corresponding drop in benchmark score (highlighted in the red oval). Based on this trade-off, we construct a Pareto-optimal set Γ of configurations, where each configuration is defined by a pair of a model variant and an early-exit point. We obtain Γ by empirically sampling configurations whose adjacent decoder-layer latencies differ by roughly 7 ms and retaining only those with benchmark scores of at least 30. This set serves as the candidate configuration space of CoRT.

[1]We use six benchmarks: ARC-C [40], MMLU [41], MMLU-Pro [42], HellaSwag [43], IFEval [44], and Winogrande [45].

We also evaluate how accurately the configurations in Γ follow output-length control on held-out test subsets of Hermes2 and LM-SYS_Chat that were not used during training. On average, these configurations achieve 95% length-control accuracy.

C. Batch-Aware Configuration Optimization

To guarantee hard-task deadlines, CoRT controls execution latency by selecting an inference configuration from the Pareto frontier Γ. A straightforward approach is to always use the lowest-latency configuration. However, as shown in Fig. 5(a), this overly conservative choice incurs accuracy loss. CoRT therefore seeks the *maximum feasible configuration*, i.e., the highest-accuracy configuration that still meets all hard-task deadlines. This requires estimating each task's WCRT, which is challenging under continuous batching because task interference depends on runtime co-batching. To address this challenge, we develop a batch-aware RTA that upper-bounds each task's WCRT under arbitrary runtime co-batching, and use it to determine the maximum feasible configuration and batch-size limit.

1) System Model: We model an LLM platform serving hard and soft tasks under continuous batching as $\pi = (\gamma, \kappa, C^p, C^d)$, where γ is an inference configuration selected from the Pareto-optimal set $\Gamma = \{\gamma_j\}_{j=1}^{m}$, and each γ_j is indexed so that smaller j corresponds to a faster but less accurate configuration. $K = \{\kappa_l\}_{l=1}^{q}$ denotes the candidate batch-size limits, and let $\kappa \in K$ be the selected one. C^p and C^d are the execution-time functions of a single batch iteration in the prefill and decoding phases, respectively, obtained through measurement-based profiling. A runtime batch Λ contains up to κ ready tasks. If at least one task in Λ is in the prefill phase, Λ is a *prefill batch* with execution time $C^p(\gamma, L^{in}_{sum}(\Lambda))$, where $L^{in}_{sum}(\Lambda)$ is the sum of the input-token lengths of tasks in Λ. Otherwise, Λ is a *decoding batch* with execution time $C^d(\gamma, |\Lambda|)$, where $|\Lambda|$ is the runtime batch size, i.e., the number of active tasks in Λ that each generate one token in that iteration.

We model the schedulable task set $\tau = \{\tau_i\}_{i=1}^{n} \cup \{\tau_s\}$, where $\{\tau_i\}_{i=1}^{n}$ denotes the n hard tasks and τ_s is a server for soft tasks. Each hard task τ_i is modeled as a sporadic task that issues one inference request per job, specified as $\tau_i = (T_i, L_i^{in}, L_i^{out})$, where T_i is the minimum inter-arrival time, which is equal to its relative deadline, and L_i^{in} and L_i^{out} are fixed input and output token lengths determined by predefined templates. To preserve timing predictability, the system truncates any output exceeding L_i^{out}, trading off potential accuracy loss. We assume task-level fixed-priority scheduling for hard tasks.

Soft tasks are modeled as aperiodic tasks corresponding to user queries in an interactive session, where each query is issued only after the previous response returns. To isolate them from hard tasks outside a batch, they are served by a server, $\tau_s = (T_s, L_s^{in}, L_s^{out})$, where T_s is the replenishment period, L_s^{in} is the maximum query length, and L_s^{out} is the output budget, i.e., the maximum number of tokens generated within T_s. The server has the lowest fixed priority. At each release, it executes a pending soft task within budget. If it is preempted by a higher-priority hard task at the next iteration boundary, it preserves the remaining budget and resumes when scheduled again.

We set T_s to the user-specified soft deadline, allowing each server release to serve a user query on a best-effort basis. At each server release, we determine L_s^{out} based on the type of the arriving query and the residual execution capacity available within T_s (see Sec. IV-D). Unlike conventional servers with time-based budgets, we express L_s^{out} in tokens, because server execution under continuous batching is controlled at the granularity of iterations, with each iteration consuming one token of budget.

2) Batch-aware RTA: Conventional RTA [46] bounds τ_i's WCRT using its execution, interference from higher-priority tasks, and blocking from lower-priority tasks. Under batch execution, however,

these terms cannot be directly bounded in the time domain because they depend on how tasks are co-batched at runtime. A conservative approach is to assume that every batch is a prefill batch composed of tasks with the largest input lengths. However, this assumption is overly pessimistic and leads to a loose WCRT bound. We therefore derive a tighter WCRT bound by first bounding the number of iterations that can appear in τ_i's response-time window regardless of the runtime batch composition, and then classifying those iterations as either prefill or decoding.

We assume that tasks in τ are indexed in non-increasing priority order, with $\tau_s = \tau_{n+1}$; thus, if $i < j$, then τ_i has higher priority than τ_j. Given a configuration γ and a batch-size limit $\kappa \geq 2$, the WCRT of a hard task τ_i is computed iteratively as

$$w_i^{a+1}(\gamma,\kappa) = C_i(w_i^a,\gamma,\kappa) + I_i(w_i^a,\gamma,\kappa) + B_i(w_i^a,\gamma,\kappa) \quad (3)$$

where $C_i(\cdot)$ is the cumulative worst-case execution time of batches executing τ_i, $I_i(\cdot)$ is the cumulative interference from batches containing only higher-priority tasks, and $B_i(\cdot)$ is the blocking from a batch containing at least one lower-priority task. Starting from $w_i^0 = 0$, the recurrence stops when $w_i^{a+1} = w_i^a$ or $w_i^{a+1} > T_i$. Since these terms cannot be bounded in the time domain under batch execution, we derive the WCRT through the following four steps.

Step 1. We first bound the worst-case number of iterations within the response-time window w_i^a as

$$N_i(w_i^a,\kappa) = N_i^C + N_i^I(w_i^a,\kappa) + N_i^B \quad (4)$$

where N_i^C, $N_i^I(\cdot)$, and N_i^B denote the numbers of iterations for executing τ_i, interference from higher-priority batches, and blocking from a lower-priority batch, respectively. Since one output token of τ_i is generated per iteration, including prefill, we have $N_i^C = L_i^{out}$. Also, $N_i^B = 1$ because under continuous batching, scheduling decisions are made only at iteration boundaries. Thus, τ_i can be blocked by at most one lower-priority batch that has already started in the current iteration. $N_i^I(\cdot)$ is upper-bounded by

$$N_i^I(w_i^a,\kappa) = \begin{cases} 0, \text{ if } i \leq \kappa, \\ \left\lfloor \frac{\sum_{\tau_h \in hp(\tau_i)} \left(\lceil \frac{w_i^a}{T_h} \rceil \cdot N_h^C \right)}{\kappa} \right\rfloor, \text{ otherwise,} \end{cases} \quad (5)$$

where $hp(\tau_i)$ is the set of tasks with higher priority than τ_i. If $i \leq \kappa$, τ_i suffers no interference because fewer than κ higher-priority tasks exist, so it can always join the next batch. Otherwise, $\sum_{\tau_h \in hp(\tau_i)} \left(\lceil \frac{w_i^a}{T_h} \rceil \cdot N_h^C \right)$ bounds the total number of higher-priority iterations in w_i^a without co-batching. Dividing by κ captures the worst case under co-batching, because only fully occupied higher-priority batches can exclude τ_i. If higher-priority iterations are unevenly clustered, some batches leave empty slots, allowing τ_i to join earlier and thereby reducing interference.

Step 2. Next, we bound the worst-case number of prefill batches within w_i^a. Since each released job can trigger at most one prefill batch, and τ_i itself necessarily incurs one prefill iteration before decoding, this quantity is upper-bounded by

$$P_i(w_i^a) = \Big(\sum_{\tau_j \in \tau \setminus \{\tau_i\}} \lceil \frac{w_i^a}{T_j} \rceil \Big) + 1 \quad (6)$$

Step 3. We then distribute $P_i(w_i^a)$ over $N_i(w_i^a)$ iterations. To maximize the WCRT, we pessimistically assign the limited number of prefill batches to the iteration categories that incur the largest prefill cost first. We first define Λ_i^C, Λ_i^I, and Λ_i^B as the worst-case prefill batch groups for execution, interference, and blocking iterations, respectively, each chosen to maximize the prefill execution time C^p. Since C^p increases with $L_{sum}^{in}(\Lambda)$, each batch is constructed to maximize the total input length. Let $M_k(\tau)$ denote the subset of up to k tasks in τ with the largest input lengths. We then define $\Lambda_i^C = \{\tau_i\} \cup M_{\kappa-1}(\tau \setminus \{\tau_i\})$, $\Lambda_i^I = M_\kappa(hp(\tau_i))$, and $\Lambda_i^B = \{\tau_i^L\} \cup M_{\kappa-1}(\tau \setminus \{\tau_i, \tau_i^L\})$, where τ_i^L is the lower-priority task of τ_i with the largest input length.

Next, we assign $P_i(w_i^a)$ to the three iteration categories in descending order of C^p. Let ord be a permutation of $\{C, I, B\}$ such that $C^p(\gamma, L_{sum}^{in}(\Lambda_i^{\mathrm{ord}(1)})) \geq C^p(\gamma, L_{sum}^{in}(\Lambda_i^{\mathrm{ord}(2)})) \geq C^p(\gamma, L_{sum}^{in}(\Lambda_i^{\mathrm{ord}(3)}))$. We then greedily assign prefill batches in this order, up to the number of available iterations in each category. Let $N_i^{X,p}$ and $N_i^{X,d}$ denote the numbers of prefill and decoding iterations in category $X \in \{C, I, B\}$, respectively. Omitting the dependence on w_i^a for brevity,

$$N_i^{\mathrm{ord}(1),p} = \min(P_i, N_i^{\mathrm{ord}(1)}) \quad (7)$$

$$N_i^{\mathrm{ord}(2),p} = \min(P_i - N_i^{\mathrm{ord}(1),p}, N_i^{\mathrm{ord}(2)}) \quad (8)$$

$$N_i^{\mathrm{ord}(3),p} = \min(P_i - N_i^{\mathrm{ord}(1),p} - N_i^{\mathrm{ord}(2),p}, N_i^{\mathrm{ord}(3)}) \quad (9)$$

The remaining iterations are treated as decoding batches, i.e., $N_i^{X,d} = N_i^X - N_i^{X,p}$. This assignment is worst-case because it gives priority to categories with larger prefill cost.

Step 4. Finally, we convert the iteration counts into time by mapping prefill and decoding iterations to C^p and C^d, respectively. Based on the assignment above, Eq. (3) is instantiated as

$$\begin{aligned} C_i(w_i^a,\gamma,\kappa) = {} & N_i^{C,p}(w_i^a) \cdot C^p(\gamma, L_{sum}^{in}(\Lambda_i^C)) \\ & + N_i^{C,d}(w_i^a) \cdot C^d(\gamma,\kappa) \end{aligned} \quad (10)$$

$$\begin{aligned} I_i(w_i^a,\gamma,\kappa) = {} & N_i^{I,p}(w_i^a) \cdot C^p(\gamma, L_{sum}^{in}(\Lambda_i^I)) \\ & + N_i^{I,d}(w_i^a) \cdot C^d(\gamma,\kappa) \end{aligned} \quad (11)$$

$$\begin{aligned} B_i(w_i^a,\gamma,\kappa) = {} & N_i^{B,p}(w_i^a) \cdot C^p(\gamma, L_{sum}^{in}(\Lambda_i^B)) \\ & + N_i^{B,d}(w_i^a) \cdot C^d(\gamma,\kappa) \end{aligned} \quad (12)$$

The WCRT is computed iteratively by applying the above four steps with the current response-time window w_i^a to obtain w_i^{a+1}.

The above formulation assumes batched execution with $\kappa \geq 2$. If the system does not batch tasks, i.e., $\kappa = 1$, the WCRT can be computed directly as

$$C_i(\gamma) = C^p(\gamma, L_i^{in}) + (L_i^{out} - 1) \cdot C^d(\gamma, 1) \quad (13)$$

$$I_i(w_i^a,\gamma) = \sum_{\tau_h \in hp(\tau_i)} \Big(\lceil \frac{w_i^a}{T_h} \rceil \cdot \big(C^p(\gamma, L_h^{in}) + (L_h^{out} - 1) \cdot C^d(\gamma, 1) \big) \Big) \quad (14)$$

$$B_i(\gamma) = C^p(\gamma, L_{sum}^{in}(\{\tau_i^L\})) \quad (15)$$

3) Search algorithm: We state the *inference configuration and batch-size selection* problem as follows.

Problem 1 (Inference Configuration and Batch-Size Selection): Given an LLM serving platform π and a task set τ, determine the feasible pair (γ^*, κ^*) that maximizes inference accuracy while satisfying all hard-task deadlines.

To address Problem 1, we use a brute-force search guided by the batch-aware RTA. We search $\Gamma = \{\gamma_j\}_{j=1}^m$ from larger j to smaller j, since larger j corresponds to higher accuracy. For each γ_j, we then search $K = \{\kappa_l\}_{l=1}^q$ from larger κ_l to smaller κ_l, since a larger batch-size limit can improve GPU throughput and create more opportunities for the server τ_s to execute concurrently. For each candidate pair (γ_j, κ_l), we apply the batch-aware RTA to every hard task τ_i and regard the pair as feasible if $w_i(\gamma_j, \kappa_l) \leq T_i$ for all $\tau_i \in \tau \setminus \{\tau_s\}$. The search stops at the first feasible pair and returns it as (γ^*, κ^*). This search is simple yet efficient in our setting. Let n be the number of hard tasks and $A_{\max}$ the maximum

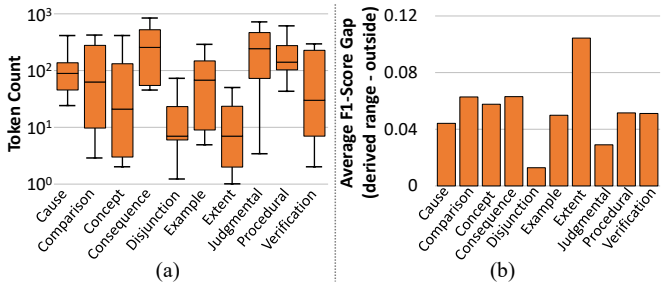

Fig. 6. (a) Output-length distributions by question type and (b) BERTScore F1-score gaps by question type (between responses generated within and outside the derived range)

iteration required for fixed-point convergence. Since evaluating one hard task's WCRT costs $O(n \cdot A_{\max})$, the overall search complexity is $O(|\Gamma| \cdot |K| \cdot n^2 \cdot A_{\max})$, which reduces to $O(n^2 \cdot A_{\max})$ because $|\Gamma|$ and $|K|$ are fixed and small.

D. Adaptive Output-Length Control

Under (γ^*, κ^*), CoRT schedules the server τ_s at the lowest priority to serve user queries using residual execution capacity. To improve soft-task responsiveness, CoRT adaptively controls output length on a best-effort basis with respect to the user-specified soft deadline. This requires two estimates: *i) a query-specific target response-length range*, since overly short responses may be uninformative while excessively long ones increase latency, and *ii) the server budget available at runtime*, since (γ^*, κ^*) is derived from a pessimistic full-batch analysis whereas actual runtime batches are often smaller, leaving additional execution capacity. Based on these estimates, CoRT selects an appropriate output length for each query.

1) Output-length range prediction: CoRT first predicts a suitable response-length range for each query based on its query type. For this, we train a lightweight query classifier on an open-ended question ontology [47] with ten question categories, using DistilBERT [48] to keep classification overhead low[2]. We then apply the classifier to the queries in Dolly-15k [49] and obtain the output-length distribution of each query type from the corresponding answers (Fig. 6(a)). From each distribution, CoRT identifies densely populated length regions and selects, from the predefined special-token set S, three representative output lengths that best match them, corresponding to *short*, *medium*, and *long* responses. For each query type q, these lengths are denoted by ℓ_q^{short}, ℓ_q^{med}, and ℓ_q^{long} and used at runtime for output-length selection.

We validate this design in two dimensions. First, the DistilBERT-based query classifier achieves a 74.2% F1 score on the ontology test split, indicating acceptable classification accuracy. Second, we evaluate the quality of the derived query-specific output lengths on No Robots using fine-tuned Qwen-2.5 models (3B, 1.5B, and 0.5B) from Sec. IV-B. For each length-control token in S, we generate responses at the corresponding length and measure quality using BERTScore [50]. Fig. 6(b) shows, for each query type, the average F1-score gap between responses generated within the derived range and those generated outside it. The positive gap supports the usefulness of the query-specific output-length ranges.

2) Server budget estimation: To estimate the runtime-available server budget, we introduce an online schedule-simulation algorithm (Algorithm 1). Under (γ^*, κ^*), it unfolds the batch schedule after the query arrival time and estimates how many iterations can be allocated to the server τ_s. The algorithm is triggered whenever a user query arrives and τ_s is released. Here, t_c denotes the effective current time after accounting for ϵ, which upper-bounds the latency of

[2]We empirically observed that, with the maximum supported input length of 512 tokens, DistilBERT performs query classification in 8 ms.

Algorithm 1 Server Budget Estimation

Input: γ^*, κ^*, $\tau = \{\tau_i\}_{i=1}^n \cup \{\tau_s\}$, ℓ_q^{short}
Output: L_s^{out}

1: $L_s^{out} \leftarrow 0$; $t_c \leftarrow$ query arrival time $+ \epsilon$;
   $t_d \leftarrow$ query arrival time $+ T_s$; $N_s^{rem} \leftarrow \infty$
2: **for each** $\tau_i \in \tau \setminus \{\tau_s\}$ **do**
3: $\quad N_i^{rem} \leftarrow$ the number of remaining iterations of τ_i at t_c
4: $\quad a_i^{next} \leftarrow$ next arrival time of τ_i after t_c
5: **end for**
6: **while** $(t_c < t_d) \vee (L_s^{out} < \ell_q^{short})$ **do**
7: $\quad \Lambda \leftarrow \varnothing$
8: $\quad$ **for each** $\tau_i \in \tau$ in descending order of priority **do**
9: $\quad\quad$ **if** $(N_i^{rem} > 0) \wedge (|\Lambda| < \kappa^*)$ **then**
10: $\quad\quad\quad \Lambda \leftarrow \Lambda \cup \{\tau_i\}$
11: $\quad\quad\quad N_i^{rem} \leftarrow N_i^{rem} - 1$
12: $\quad\quad$ **end if**
13: $\quad$ **end for**
14: $\quad$ **if** $\tau_s \in \Lambda$ **then**
15: $\quad\quad L_s^{out} \leftarrow L_s^{out} + 1$
16: $\quad$ **end if**
17: $\quad$ **if** Λ is a prefill batch **then**
18: $\quad\quad t_c \leftarrow t_c + C^p(\gamma^*, L_{sum}^{in}(\Lambda))$
19: $\quad$ **else**
20: $\quad\quad t_c \leftarrow t_c + C^d(\gamma^*, |\Lambda|)$
21: $\quad$ **end if**
22: $\quad$ **for each** $\tau_i \in \tau \setminus \{\tau_s\}$ **do**
23: $\quad\quad$ **if** $a_i^{next} \le t_c$ **then**
24: $\quad\quad\quad N_i^{rem} \leftarrow L_i^{out}$
25: $\quad\quad\quad a_i^{next} \leftarrow a_i^{next} + T_i$
26: $\quad\quad$ **end if**
27: $\quad$ **end for**
28: **end while**
29: **return** L_s^{out}

query classification and budget estimation, and t_d denotes the query's soft deadline. We first initialize, for each hard task τ_i, the remaining iterations N_i^{rem} of its active job and the next release time a_i^{next} after t_c (lines 1–5). If τ_i has not yet been released at t_c, N_i^{rem} is set to zero. In contrast, τ_s is assigned an infinite remaining-iteration count, since its output length is not yet determined.

We then repeatedly construct a runtime batch Λ by scanning tasks in descending priority order and admitting those with remaining work until the batch reaches κ^* (lines 7–13). Because τ_s has the lowest priority, it joins Λ only when residual capacity remains after accommodating all hard tasks. Whenever τ_s is included, we increment L_s^{out} by one (lines 14–16), since one output token is generated per iteration. We then advance the simulated time by the batch execution time: $C^p(\gamma^*, L_{sum}^{in}(\Lambda))$ for a prefill batch, and $C^d(\gamma^*, |\Lambda|)$ otherwise (lines 17–21). Next, we update each hard-task state by checking whether a new job is released by t_c; if so, N_i^{rem} is reset to L_i^{out} and a_i^{next} is advanced by one period (lines 22–26). This process continues until t_d is reached. If the budget available by t_d is insufficient to support even ℓ_q^{short}, CoRT continues the simulation beyond t_d until at least ℓ_q^{short} tokens become achievable. This reflects our best-effort policy: CoRT strives to meet the soft deadline, but tolerates a deadline miss when necessary to generate a minimally useful response. We finally return L_s^{out}, the estimated output budget for τ_s. This algorithm runs in $O(n \cdot N^{sim})$ time, where N^{sim} is the number of simulated batch iterations until termination.

Based on L_s^{out}, CoRT selects the final output length as the largest feasible candidate among ℓ_q^{short}, ℓ_q^{med}, and ℓ_q^{long}:

$$\ell_q^* = \max\{\ell \in \{\ell_q^{short}, \ell_q^{med}, \ell_q^{long}\} \mid \ell \le L_s^{out}\}. \tag{16}$$

CoRT serves the query using ℓ_q^*. If the response is generated before

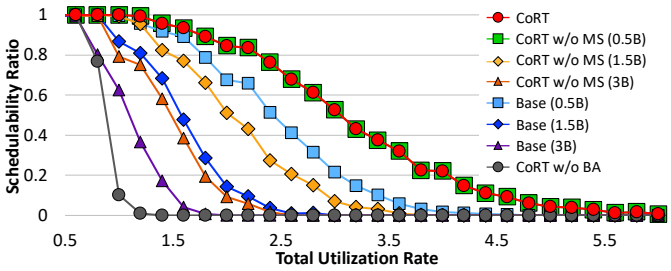

Fig. 7. Schedulability ratio with varying total utilization rate

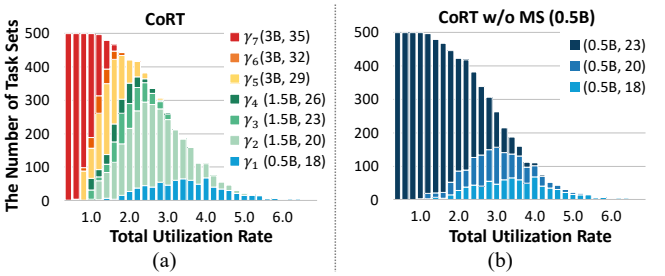

Fig. 8. Distribution of selected inference configurations under (a) CoRT and (b) CoRT w/o MS 0.5B. Each label in parentheses denotes the model variant and the layer index executed up to the selected early-exit point.

t_d, CoRT delays returning it until t_d. Although this waiting policy may appear conservative, returning the response earlier could cause the next query to arrive sooner than assumed, potentially creating additional prefill batches beyond those accounted for in the analysis.

V. Evaluation

A. Evaluation Setup

We implemented CoRT in PyTorch [12] and evaluated it on NVIDIA Jetson AGX Orin. We fine-tuned Qwen-2.5 [32] 0.5B, 1.5B, and 3B models as described in Sec. IV-B, and constructed a Pareto-optimal set Γ of seven inference configurations based on their accuracy-latency measurements. As hard real-time workloads, we used three anomaly-detection datasets from Loghub [34]: HDFS, BGL, and Thunderbird. As soft real-time workloads, we used No Robots [35]. We compared five approaches: *i)* CoRT: our full framework; *ii)* CoRT w/o BA: CoRT using an RTA that pessimistically treats all iterations as prefill batches; *iii)* CoRT w/o LC: CoRT without adaptive output-length control; *iv)* CoRT w/o MS: CoRT without model selection, which uses early-exit points drawn from a Pareto-optimal set constructed from a single model; and *v)* Base: the original Qwen-2.5 models. For fair comparison, all approaches determine the batch-size limit for each task set via brute-force search.

B. Simulation Experiments

To evaluate how well CoRT guarantees hard-task deadlines under diverse task-set configurations, we conducted extensive simulations using synthetic task sets. We used the *schedulability ratio* as the performance metric, defined as the fraction of schedulable task sets among all generated task sets. We randomly generated 16,500 task sets using the UUniFast algorithm [51], which is widely used for real-time task-set generation. Each task set contains eight tasks, where each task has an input-token length uniformly chosen from 27 to 1,000 and an output-token length uniformly chosen from 1 to 300. Each task's priority is determined by deadline-monotonic [52].

Fig. 7 shows the schedulability ratio as the total utilization rate U_τ of each task set varies from 0.5 to 6.0, where U_τ is defined as $\sum_{\tau_i \in \tau} \left(\sum_{\gamma_j \in \Gamma} C_i(\gamma_j)/|\Gamma| \right)/T_i$. Here, $C_i(\gamma_j)$ is given by Eq. (13). The results show that CoRT outperforms all compared approaches except CoRT w/o MS (0.5B), improving the schedulability ratio by 1.3× to 4.9×. This performance gap becomes larger as U_τ increases.

To better understand how each component of CoRT improves schedulability, we isolate their effects. We first compare CoRT w/o MS with Base for each model size. CoRT w/o MS finds 20%, 34%, and 47% more schedulable task sets than Base for the 0.5B, 1.5B, and 3B models, respectively. This confirms that early exit improves schedulability by reducing the decoding computation per iteration, and that its benefit becomes more pronounced as model size increases.

Next, we evaluate the contribution of model selection by comparing CoRT with CoRT w/o MS. CoRT achieves 49% and 119% higher schedulability than CoRT w/o MS with the 1.5B and 3B models, respectively. This indicates that reducing model size generally has a larger impact on execution time than early exit alone. In contrast, CoRT shows the same schedulability as CoRT w/o MS with the 0.5B model, because the two ultimately share the same lowest-latency configuration despite having different Pareto-optimal sets. However, Fig. 8 shows that CoRT selects substantially more high-accuracy configurations across utilization rates, indicating that it can preserve higher output quality while maintaining the same schedulability whenever timing slack permits.

Finally, we evaluate the impact of the RTA design by comparing CoRT with CoRT w/o BA. CoRT finds 192% more schedulable task sets than CoRT w/o BA, showing that configuration flexibility alone is insufficient under an overly pessimistic RTA. Unlike CoRT w/o BA, which treats all iterations as prefill batches, CoRT bounds only the worst-case number of prefill iterations and derives a much tighter WCRT bound.

TABLE I
Deadline satisfaction ratio (%) of hard and soft tasks

Configs	Hard Tasks			Soft Tasks		
	τ_1	τ_2	τ_3	$T_s = 5\,\text{s}$	$T_s = 10\,\text{s}$	$T_s = 15\,\text{s}$
Base 3B	0	0	0	0	0	0
Base 1.5B	0.11	0.93	8.5	0	0	0
Base 0.5B	99.89	100	100	12.50	12.50	37.50
CoRT w/o LC	100	100	100	0.67	20.0	40.0
CoRT	100	100	100	57.58	88.24	96.77

C. Case Study Experiments

To evaluate CoRT in a real setting, we deployed it on NVIDIA Jetson AGX Orin and ran three hard real-time tasks together with soft real-time question-answering tasks. The hard tasks were instantiated from anomaly-detection datasets: $\tau_1(T_1 = 650\,\text{ms}$, Thunderbird) for high-level system operation monitoring (e.g., resource usage), $\tau_2(T_2 = 1{,}400\,\text{ms}$, HDFS) for file-system monitoring (e.g., file I/O), and $\tau_3(T_3 = 3{,}000\,\text{ms}$, BGL) for low-level system event monitoring (e.g., hardware errors). Soft tasks were generated from the No Robots dataset by issuing user queries with inter-arrival times uniformly sampled from 10 to 20 seconds. Each query arrival released the server τ_s for question answering. We repeated the experiment with three soft-deadline settings, $T_s \in \{5\,\text{s}, 10\,\text{s}, 15\,\text{s}\}$.

Table I reports the deadline satisfaction ratio (DSR) of hard and soft tasks for each approach over a 10-minute run, where DSR is defined as the fraction of released jobs that meet their deadlines. The results show that Base fails to guarantee all hard-task deadlines for any model size; even the smallest 0.5B model misses a small fraction of τ_1 deadlines. This indicates that simply reducing the model size is insufficient to absorb the additional delay caused by co-batching. In

TABLE II
PERFORMANCE COMPARISON OF BASE AND CORT

Configs	Hard Tasks (Detection accuracy %)			Soft Tasks (BERTScore-F1)
	Thunderbird	HDFS	BGL	No-Robots
Base 3B	100	99.50	99.95	0.603
Base 1.5B	100	99.30	100	0.600
Base 0.5B	100	98.40	100	0.580
γ_7 (3B, 35)	100	99.15	99.95	0.596
γ_6 (3B, 32)	100	98.80	99.95	0.590
γ_5 (3B, 29)	100	97.90	100	0.587
γ_4 (1.5B, 26)	100	98.15	100	0.584
γ_3 (1.5B, 23)	100	98.35	100	0.569
γ_2 (1.5B, 20)	100	98.50	99.00	0.538
γ_1 (0.5B, 18)	100	98.40	99.10	0.519

Note: Parenthesized values in each γ_j label denote the model variant and the layer index executed up to the selected early-exit point.

contrast, both CoRT and CoRT w/o LC achieve a perfect DSR for all hard tasks. This is because model selection and early exit effectively reduce execution time, offsetting the additional delay introduced by co-batching and thereby preserving hard real-time guarantees.

For soft tasks, Base with the 1.5B and 3B models is too large to serve any queries within the evaluated soft deadlines, while Base with the 0.5B model can serve only a subset of them. Notably, under the 5-second soft deadline, Base 0.5B achieves a higher DSR than CoRT w/o LC. We attribute this to the fine-tuned Qwen-2.5 models used in our system, which tend to generate more verbose responses than the original Qwen-2.5 model due to the fine-tuning data. However, when the soft deadline is relaxed to 10 seconds or longer, CoRT w/o LC outperforms Base 0.5B by serving more user queries within the deadline, as early exit reduces execution time and creates additional slack for more frequent server execution. Finally, CoRT achieves up to 7× and 86× higher DSR than Base 0.5B and CoRT w/o LC, respectively. By adaptively controlling response length according to the currently available execution capacity, CoRT mitigates the increased verbosity introduced by fine-tuning and maximizes soft-task deadline satisfaction.

D. Output Quality Evaluation

As discussed in Sec. IV-B3, lower inference configurations can improve schedulability, but may also reduce output quality. To quantify this impact on output quality, we compare the Base models and the seven candidate inference configurations $\{\gamma_j\}_{j=1}^{7}$ used by CoRT on the same datasets used in our case study. For anomaly-detection datasets (i.e., Thunderbird, HDFS, and BGL), we use *detection accuracy* as the evaluation metric, defined as the fraction of dataset samples for which the model correctly identifies the anomaly status. For No Robots, we use BERTScore to measure the semantic similarity between the generated response and the reference answer. Specifically, for each query, CoRT generates responses using the candidate output lengths identified in Sec. IV-D, namely ℓ_q^{short}, ℓ_q^{med}, and ℓ_q^{long}, and uses the average of their BERTScore F1 values as its output-quality score. For each Base model, which does not apply output-length control, we use the BERTScore F1 value of its generated response as the output-quality score.

Table II compares the output quality of the Base models and the seven inference configurations $\{\gamma_j\}_{j=1}^{7}$ on each dataset. For the hard-task datasets, the configurations used by CoRT incur only marginal accuracy degradation compared with the Base models. This suggests that these anomaly-detection workloads are relatively insensitive to lower inference configurations. In contrast, for the soft-task dataset, output quality decreases almost linearly as the configuration becomes smaller. Nevertheless, in safety-critical systems, guaranteeing hard-task deadlines takes precedence over providing higher-quality responses to user queries, making this trade-off acceptable in our setting.

VI. DISCUSSION AND FUTURE WORKS

Multi-Session Support for User Queries. Our current CoRT prototype considers only a single interactive session. In practice, however, multiple users may issue concurrent queries, requiring CoRT to support multiple soft real-time sessions. One possible extension is to associate a separate server with each session, so that multiple sessions can share the residual execution capacity while preserving hard real-time guarantees. A key challenge is how to allocate residual service opportunities across session servers. For example, CoRT may divide them equally among active sessions or prioritize certain sessions based on their QoS requirements. Exploring such multi-session scheduling policies is an important direction for future work.

Balancing Accuracy and Soft-Task Responsiveness. In Sec. IV-C3, our search algorithm is designed to find the maximum feasible inference configuration that satisfies all hard-task deadlines while maximizing model accuracy. As a result, soft-task responsiveness is not considered during configuration selection and is handled only by output-length control on a best-effort basis. A promising direction is to design a more flexible objective function that jointly considers accuracy and soft-task responsiveness, for example through a simple weighting parameter. This extension would allow CoRT to account for soft-task QoS during configuration selection, potentially improving responsiveness without sacrificing hard real-time guarantees.

Extension to Vision-Language Models. Although CoRT currently targets LLMs, extending it to VLMs is an important direction for future work. Because a VLM typically consists of a vision encoder followed by an LLM decoder, such an extension would require augmenting our batch-aware RTA to explicitly account for the execution time of the vision encoder. In addition, VLM-related hard real-time tasks are often more challenging and accuracy-sensitive than the anomaly-detection tasks used in our experiments, making aggressive configuration reduction less acceptable. This suggests the need for more advanced fine-tuning methods that can further mitigate accuracy loss in VLMs. One possible direction is to additionally inject vision features into early-exit layers during fine-tuning, helping earlier exits retain stronger image-grounding capability despite reduced computation. We leave this extension for future work.

VII. CONCLUSION

We presented CoRT, a real-time LLM framework for jointly supporting hard and soft real-time tasks in safety-critical systems. CoRT combines model selection, early exit, and adaptive output-length control to guarantee hard-task deadlines while improving soft-task responsiveness under batched GPU execution. To enable this design, we introduced selective fine-tuning to mitigate accuracy loss, a batch-aware RTA to bound worst-case co-batching interference, and adaptive output-length control based on query type and residual execution capacity. Results from simulation and real-system experiments on NVIDIA Jetson AGX Orin showed that CoRT achieves up to 4.9× higher hard-task schedulability and 7× higher soft-task responsiveness than prior approaches. Overall, our results highlight the potential of autoregressive computation control as a practical foundation for mixed-criticality real-time LLM serving.

REFERENCES

[1] D. Xu, E. Qiao, T. Gu, H. Fu, C. Sun, H. Guo, and Y. Liu, "Lead: Llm-enhanced deep reinforcement learning for stable decision-making in critical autonomous driving scenarios," *Neurocomputing*, 2025.

[2] Z. Zhou, H. Huang, B. Li, S. Zhao, Y. Mu, and J. Wang, "Safedrive: Knowledge-and data-driven risk-sensitive decision-making for autonomous vehicles with large language models," *Accident Analysis & Prevention*, vol. 224, 2026.

[3] K. Agyei, P. Sarhadi, and W. Naeem, "Large language model-based decision-making for colregs and the control of autonomous surface vehicles," in *ECC*, 2025.

[4] J. Wang, E. Shi, H. Hu, C. Ma, Y. Liu, X. Wang, Y. Yao, X. Liu, B. Ge, and S. Zhang, "Large language models for robotics: Opportunities, challenges, and perspectives," *Journal of Automation and Intelligence*, vol. 4, no. 1, 2025.

[5] F. Joublin, A. Ceravola, P. Smirnov, F. Ocker, J. Deigmoeller, A. Belardinelli, C. Wang, S. Hasler, D. Tanneberg, and M. Gienger, "Copal: corrective planning of robot actions with large language models," in *IEEE ICRA*, 2024.

[6] A. Tagliabue, K. Kondo, T. Zhao, M. Peterson, C. T. Tewari, and J. P. How, "Real: Resilience and adaptation using large language models on autonomous aerial robots," in *IEEE CDC*, 2024.

[7] Y. Liu, L. Palmieri, S. Koch, I. Georgievski, and M. Aiello, "Delta: Decomposed efficient long-term robot task planning using large language models," in *IEEE ICRA*, 2025.

[8] Z. Zhou, J. Song, K. Yao, Z. Shu, and L. Ma, "Isr-llm: Iterative self-refined large language model for long-horizon sequential task planning," in *IEEE ICRA*, 2024.

[9] M. Dong, H. Huang, and L. Cao, "Can llms serve as time series anomaly detectors?" *arXiv preprint arXiv:2408.03475*, 2024.

[10] J. Liu, C. Zhang, J. Qian, M. Ma, S. Qin, C. Bansal, Q. Lin, S. Rajmohan, and D. Zhang, "Large language models can deliver accurate and interpretable time series anomaly detection," in *ACM KDD*, 2025.

[11] Y. Liu, Y. Luo, X. Li, X. Dong, B. Gu, and Z. Jin, "Evaluating large language models for time series anomaly detection in aerospace software," in *IEEE/ACM ASE*, 2025.

[12] "Pytorch," 2026. [Online]. Available: https://pytorch.org/

[13] B. Sprunt, L. Sha, and J. Lehoczky, "Aperiodic task scheduling for hard-real-time systems," *Real-Time Systems*, vol. 1, no. 1, 1989.

[14] Spuri and Buttazzo, "Efficient aperiodic service under earliest deadline scheduling," in *IEEE RTSS*, 1994.

[15] L. Abeni and G. Buttazzo, "Integrating multimedia applications in hard real-time systems," in *IEEE RTSS*, 1998.

[16] Z. Deng and J.-S. Liu, "Scheduling real-time applications in an open environment," in *IEEE RTSS*, 1997.

[17] S. A. Brandt, S. Banachowski, C. Lin, and T. Bisson, "Dynamic integrated scheduling of hard real-time, soft real-time, and non-real-time processes," in *IEEE RTSS*, 2003.

[18] G.-I. Yu, J. S. Jeong, G.-W. Kim, S. Kim, and B.-G. Chun, "Orca: A distributed serving system for transformer-based generative models," in *USENIX OSDI*, 2022.

[19] W. Kwon, Z. Li, S. Zhuang, Y. Sheng, L. Zheng, C. H. Yu, J. Gonzalez, H. Zhang, and I. Stoica, "Efficient memory management for large language model serving with pagedattention," in *ACM SOSP*, 2023.

[20] A. Agrawal, N. Kedia, A. Panwar, J. Mohan, N. Kwatra, B. Gulavani, A. Tumanov, and R. Ramjee, "Taming throughput-latency tradeoff in llm inference with sarathi-serve," in *USENIX OSDI*, 2024.

[21] B. Wu, Y. Zhong, Z. Zhang, S. Liu, F. Liu, Y. Sun, G. Huang, X. Liu, and X. Jin, "Fast distributed inference serving for large language models," *arXiv preprint arXiv:2305.05920*, 2023.

[22] B. Sun, Z. Huang, H. Zhao, W. Xiao, X. Zhang, Y. Li, and W. Lin, "Llumnix: Dynamic scheduling for large language model serving," in *USENIX OSDI*, 2024.

[23] K. Hong, X. Li, L. Chen, Q. Mao, G. Dai, X. Ning, S. Yan, Y. Liang, and Y. Wang, "Sola: Optimizing slo attainment for large language model serving with state-aware scheduling," *MLSys*, vol. 7, 2025.

[24] Y. Li, Z. Li, W. Yang, and C. Liu, "Rt-lm: Uncertainty-aware resource management for real-time inference of language models," in *IEEE RTSS*, 2023.

[25] H. Qiu, W. Mao, A. Patke, S. Cui, S. Jha, C. Wang, H. Franke, Z. T. Kalbarczyk, T. Başar, and R. K. Iyer, "Efficient interactive llm serving with proxy model-based sequence length prediction," in *AIOps Workshop at ASPLOS*, 2024.

[26] E. Frantar, S. Ashkboos, T. Hoefler, and D. Alistarh, "Gptq: Accurate post-training quantization for generative pre-trained transformers," *arXiv preprint arXiv:2210.17323*, 2022.

[27] E. Frantar and D. Alistarh, "Sparsegpt: Massive language models can be accurately pruned in one-shot," in *ICML*, 2023.

[28] R. Agarwal, N. Vieillard, Y. Zhou, P. Stanczyk, S. R. Garea, M. Geist, and O. Bachem, "On-policy distillation of language models: Learning from self-generated mistakes," in *ICLR*, 2024.

[29] N. Varshney, A. Chatterjee, M. Parmar, and C. Baral, "Investigating acceleration of LLaMA inference by enabling intermediate layer decoding via instruction tuning with 'LITE'," in *Findings of NAACL*, 2024.

[30] S. Ahmad, H. Guan, B. D. Friedman, T. Williams, R. K. Sitaraman, and T. Woo, "Proteus: A high-throughput inference-serving system with accuracy scaling," in *ACM ASPLOS*, 2024.

[31] J. Li, L. Zhang, Y. Li, Z. Liu, Y. Bai, R. Luo, L. Chen, and M. Yang, "Ruler: A model-agnostic method to control generated length for large language models," in *Findings of EMNLP*, 2024.

[32] Q. Team, "Qwen2.5 technical report," *arXiv preprint arXiv:2412.15115*, 2024.

[33] AI@Meta, "Llama 3 model card," 2024. [Online]. Available: https://github.com/meta-llama/llama3/blob/main/MODEL_CARD.md

[34] Z. Jiang, J. Liu, J. Huang, Y. Li, Y. Huo, J. Gu, Z. Chen, J. Zhu, and M. R. Lyu, "A large-scale evaluation for log parsing techniques: How far are we?" in *ACM SIGSOFT ISSTA*, 2024.

[35] N. Rajani, L. Tunstall, E. Beeching, N. Lambert, A. M. Rush, and T. Wolf, "No robots," 2023.

[36] H. Xie, Y. Chen, L. Wang, L. Hu, and D. Wang, "Predicting llm output length via entropy-guided representations," *arXiv preprint arXiv:2602.11812*, 2026.

[37] E. Hernandez, A. S. Sharma, T. Haklay, K. Meng, M. Wattenberg, J. Andreas, Y. Belinkov, and D. Bau, "Linearity of relation decoding in transformer language models," in *ICLR*, 2024.

[38] Teknium, "Openhermes 2.5: An open dataset of synthetic data for generalist llm assistants," 2023. [Online]. Available: https://huggingface.co/datasets/teknium/OpenHermes-2.5

[39] L. Zheng, W.-L. Chiang, Y. Sheng, T. Li, S. Zhuang, Z. Wu, Y. Zhuang, Z. Li, Z. Lin, E. P. Xing, J. E. Gonzalez, I. Stoica, and H. Zhang, "Lmsys-chat-1m: A large-scale real-world llm conversation dataset," *arXiv preprint arXiv:2309.11998*, 2023.

[40] P. Clark, I. Cowhey, O. Etzioni, T. Khot, A. Sabharwal, C. Schoenick, and O. Tafjord, "Think you have solved question answering? try arc, the ai2 reasoning challenge," *arXiv:1803.05457*, 2018.

[41] D. Hendrycks, C. Burns, S. Basart, A. Zou, M. Mazeika, D. Song, and J. Steinhardt, "Measuring massive multitask language understanding," in *ICLR*, 2021.

[42] Y. Wang, X. Ma, G. Zhang, Y. Ni, A. Chandra, S. Guo, W. Ren, A. Arulraj, X. He, Z. Jiang, T. Li, M. Ku, K. Wang, A. Zhuang, R. Fan, X. Yue, and W. Chen, "Mmlu-pro: A more robust and challenging multi-task language understanding benchmark," in *NeurIPS*, 2024.

[43] R. Zellers, A. Holtzman, Y. Bisk, A. Farhadi, and Y. Choi, "Hellaswag: Can a machine really finish your sentence?" in *ACL*, 2019.

[44] J. Zhou, T. Lu, S. Mishra, S. Brahma, S. Basu, Y. Luan, D. Zhou, and L. Hou, "Instruction-following evaluation for large language models," *arXiv preprint arXiv:2311.07911*, 2023.

[45] K. Sakaguchi, R. L. Bras, C. Bhagavatula, and Y. Choi, "Winogrande: An adversarial winograd schema challenge at scale," *Communications of the ACM*, vol. 64, no. 9, 2021.

[46] M. Joseph and P. Pandya, "Finding response times in a real-time system," *The Computer Journal*, vol. 29, no. 5, 1986.

[47] S. Cao and L. Wang, "Controllable open-ended question generation with a new question type ontology," in *IJCNLP*, 2021.

[48] V. Sanh, L. Debut, J. Chaumond, and T. Wolf, "Distilbert, a distilled version of bert: smaller, faster, cheaper and lighter," *arXiv preprint arXiv:1910.01108*, 2019.

[49] M. Conover, M. Hayes, A. Mathur, J. Xie, J. Wan, S. Shah, A. Ghodsi, P. Wendell, M. Zaharia, and R. Xin. (2023) Free dolly: Introducing the world's first truly open instruction-tuned llm. [Online]. Available: https://www.databricks.com/blog/2023/04/12/dolly-first-open-commercially-viable-instruction-tuned-llm

[50] T. Zhang, V. Kishore, F. Wu, K. Q. Weinberger, and Y. Artzi, "Bertscore: Evaluating text generation with bert," *arXiv preprint arXiv:1904.09675*, 2019.

[51] E. Bini and G. C. Buttazzo, "Measuring the performance of schedulability tests," *Real-Time Systems*, vol. 30, 2005.

[52] N. C. Audsley, A. Burns, M. F. Richardson, and A. J. Wellings, "Hard real-time scheduling: The deadline-monotonic approach," *IFAC Proceedings Volumes*, vol. 24, no. 2, 1991.