Arlo: Serving Transformer-based Language Models with Dynamic Input Lengths

Xin Tan
The Chinese University of Hong Kong
Hong Kong SAR, China
xtan22@cse.cuhk.edu.hk

Jiamin Li
Microsoft
Vancouver, Canada
jiaminli@microsoft.com

Yitao Yang
The Chinese University of Hong Kong
Hong Kong SAR, China
ytyang@cse.cuhk.edu.hk

Jingzong Li
The Hang Seng University of Hong Kong
Hong Kong SAR, China
jingzongli@cuhk.edu.hk

Hong Xu
The Chinese University of Hong Kong
Hong Kong SAR, China
hongxu@cuhk.edu.hk

ABSTRACT

A prominent challenge in serving requests for NLP tasks is handling the varying length of input texts. Existing solutions, such as uniform zero-padding and compiler support, suffer from either computational inefficiency or suboptimal latency. To address these practical issues, we propose an approach called *polymorphing*. Polymorphing involves creating and utilizing multiple runtimes of the model, each statically compiled with a different input length, to serve requests accordingly. This fine-grained use of statically-compiled runtimes reduces the overheads of zero-padding while improving latency performance compared to dynamic compilation. To practically realize polymorphing, we have developed an inference scheduling system, Arlo, which leverages the observed input length distribution to periodically allocate compute resources across multiple runtimes by solving an integer linear program. Upon request arrival, Arlo uses a multi-level queue-based heuristic to dispatch requests to the most suitable runtime instances, efficiently adapting to the dynamics of request length and instance load. Extensive testbed evaluations and large-scale simulations using production traces demonstrate Arlo's promising potential. It achieves 23.7%–98.1% mean latency reductions compared to existing schemes while significantly reducing tail latency.

CCS CONCEPTS

• **Computing methodologies** → *Distributed computing methodologies.*

KEYWORDS

ML inference, resource scheduling, language model

ACM Reference Format:
Xin Tan, Jiamin Li, Yitao Yang, Jingzong Li, and Hong Xu. 2024. Arlo: Serving Transformer-based Language Models with Dynamic Input Lengths. In *The 53rd International Conference on Parallel Processing (ICPP '24), August 12–15, 2024, Gotland, Sweden.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3673038.3673124

ICPP '24, August 12–15, 2024, Gotland, Sweden

ACM ISBN 979-8-4007-1793-2/24/08
https://doi.org/10.1145/3673038.3673124

1 INTRODUCTION

Transformer-based language models (LMs) have revolutionized natural language processing (NLP) with remarkable success in many applications [17, 24, 25, 29, 36], such as machine translation [36], question answering [39], etc. In the realm of Transformer-based LMs, two fundamental types have emerged: generative and discriminative models. Generative models like GPT-4 [28] auto-regressively predict the next word's probability distribution given the sentence context. Their popularity, exemplified by ChatGPT [2], has spurred systems [4, 23, 40, 43] into enhancing these models' efficiency and scalability. Despite the spotlight on generative models, discriminative models, which consider the entire context to predict labels or classifications, continue to occupy a substantial portion of open-source NLP libraries[6] and real-world deployment.

In this work, we focus on discriminative models [17, 52]. These models have shown exceptional performance in various NLP tasks like text classification and sentiment analysis, making them valuable assets in many applications. Besides, it is crucial not to overlook the significance of discriminative models as middleware components in application pipelines. For example, these models are extensively used to detect and flag misleading or fake news articles in social media platforms [19, 34]. Moreover, search engines and vector databases leverage their context-aware embeddings to retrieve results that align with user intent [25, 42], enhancing the relevance of search results. The widespread adoption of discriminative models in both academia and industry underscores their relevance and practicality today.

Deploying these models in practice, however, faces a particular challenge of working with intrinsically dynamic lengths of input sequences (§2.1). This variability has salient implications for the efficiency of the inference process. Current inference systems [15, 21, 27, 30, 46] mostly use uniform zero-padding with static-shape[1] runtime compilation, where models are configured and compiled with a particular sequence length. Shorter requests are

[1]Input sequence length is an important factor to determine the shape of the tensors for which DL compilers strive to optimize. It is thus customary to use the term "shape" with compilers, while using "length" with requests. We follow this convention throughout the paper.

then padded with zeros before processing. Zero-padding clearly results in suboptimal resource utilization and increased latency for shorter sequences, as a significant portion of computation is wasted on the padded data. Recently, some deep learning (DL) compilers including TensorRT [8] and TVM Unity [10] have provided support for dynamic-shape inputs at runtime. Yet, our experiments in §2.2 and some existing studies [33, 51] reveal that it often comes with worse latency compared to using static compilation for the same shape. Additionally, it also requires time-consuming kernel tuning [10, 13], making it cumbersome for practical use.

We propose a different approach called *polymorphing* to address dynamic input lengths. Polymorphing leverages *multiple* runtimes, each statically compiled with a different maximum input length, effectively creating different forms of the same model (hence the term). By directing requests to the best runtime that can handle them with the least amount of padding, polymorphing optimizes the trade-off between latency and zero-padding size. This also avoids the need of extensive kernel tuning or compromising on computational efficiency.

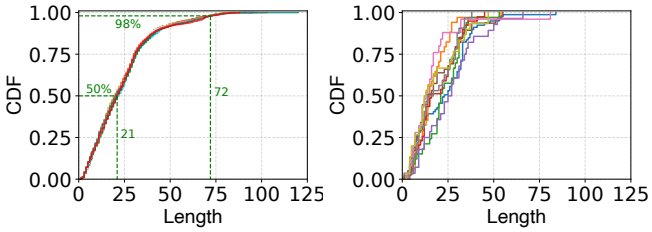
To realize polymorphing effectively, we build Arlo, an inference scheduling system that works with existing serving systems [9, 15, 21, 22, 27, 46, 47]. Arlo has two key components, Runtime Scheduler and Request Scheduler, each addressing an imminent technical question brought by polymorphing.

First, Arlo needs to determine the resource allocation across runtimes (§3.3). Based on the principle that each request should ideally be processed by the runtime with the least amount of padding, this problem can be formulated as an optimization that minimizes the overall latency given resource capacity and request demand constraints. This optimization is based on the assumption that request length distribution can be obtained over a coarse time scale (e.g. every 10 minutes), which we establish empirically.

Second, despite our best effort, the instantaneous request length dynamics seen by the serving system deviates from the average. Many idiosyncratic factors such as failures and bugs also leads to imbalanced load even across instances of the same runtime [44]. These dynamics cannot be handled by resource scheduling due to the inherent difficulty of short-term time-series prediction and the overheads of adjusting deployment at scale. Thus, Arlo relies on a Request Scheduler (§3.4) to dynamically select a runtime instance for each request. Using a multi-level queue based heuristic, Request Scheduler can opportunistically re-direct requests to runtimes for longer inputs when ideal runtime instances are overloaded, achieving a better tradeoff between queuing delay and running time.

Putting everything together, we design, implement and evaluate Arlo, that exploits multiple runtimes. In the offline stage, multiple runtimes are prepared and their computation time are profiled to facilitate Arlo's decision. During online serving, Runtime Scheduler collects the long-term request length distribution pattern and identifies the ideal runtime for requests. It periodically computes the runtime resource allocation to match with such pattern and ensure no violation on the service level objective (SLO). Request Scheduler actively dispatches queuing requests to the most suitable runtime instances by considering the trade-offs of all candiate runtime instances to minimize the average latency.

The results of Arlo are promising. We use Twitter's production trace [11] to evaluate Arlo. In a 10-GPU testbed and large-scale simulation scenarios, Arlo surpasses uniform zero-padding and dynamic compilation schemes, reducing mean latency by up to 98.1% and 30.7%, and 98%ile tail latency by up to 98.4% and 26.0%, respectively. Besides, we compare Arlo with a state-of-the-art system INFaaS[30] and Arlo consistently outperforms it, achieving reductions in mean and tail latency of up to 41.7% and 40.1%.



(a) Sequence length distribution over 10 consecutive one-minute traces. **(b) Sequence length distribution over 10 one-second traces.**

Figure 1: Sequence length distribution of real-world Twitter traces [11] at different time scales. We randomly select a one-second trace from each one-minute trace (left) to illustrate instantaneous request length dynamics (right).

We summarize our contributions as follows.

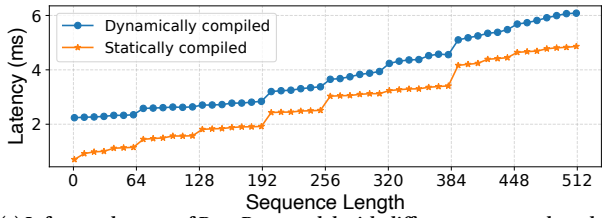
- We report problems of the existing solutions in handling requests with varying lengths, i.e. inflated latency with zero-padding and low computational efficiency using compilers with dynamic-shape input support.
- We introduce Arlo, an inference scheduler that minimizes the computation resource waste caused by zero-padding, without the support of dynamic-shape compiled runtime.
- We design Runtime Scheduler to dynamically allocate runtime resources based on the long-term request length distribution pattern, and Request Scheduler to actively dispatch requests to proper instances to address short-term fluctuations and minimize the average latency.
- We implement Arlo and conduct comprehensive evaluations, demonstrating the effectiveness of Arlo in reducing latency and optimizing resource utilization for discriminative language models with real-world traces.

2 BACKGROUND AND MOTIVATION

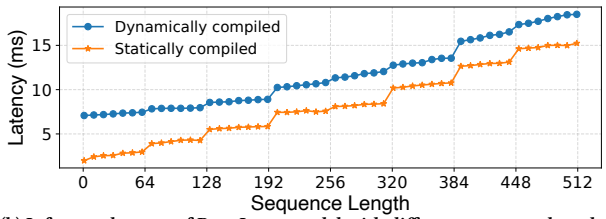
2.1 Transformer-based language models

In this work, we focus on serving discriminative Transformer-based LMs that have shown remarkable success across many NLP applications in the wild. The serving process involves setting up appropriate configurations for models and compiling them to run on specific hardware. In this stage, one key configuration is the shape of the data, which put constraints on the requests. This shape includes two essential components: (a) the batch size, which determines the number of data samples processed by the runtime simultaneously (usually set to 1 or 2 to maintain low latency in latency-sensitive scenarios[30, 37, 38]), and (b) the sequence length, representing the length of the input text. It is applied in most existing serving systems[9, 15, 21, 30, 38, 46, 47].

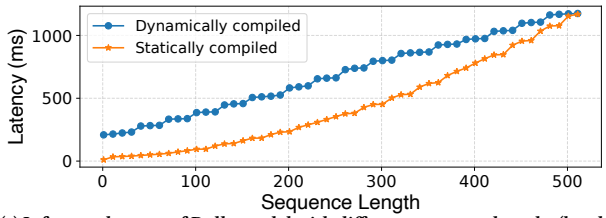
One unique property of serving Transformer-based models is the variability of the request sequence length. Empirical evidence indicates that input texts can vary widely in length, from short sentences to long documents, significantly impacting computation time. Fig. 1a depicts the cumulative distribution function (CDF) of



(a) Inference latency of Bert-Base model with different sequence lengths (batch size 1) compiled via TensorRT.



(b) Inference latency of Bert-Large model with different sequence lengths (batch size 1) compiled via TensorRT.



(c) Inference latency of Dolly model with different sequence lengths (batch size 1) compiled via TVM Unity.

Figure 2: Inference latency comparison of three representative Transformer-based LMs compiled with static and dynamic length dimension on NVIDIA GeForce RTX 3090. The Bert-Base and Bert-Large are compiled with FP32 via TensorRT, while the Dolly is compiled with FP16 via TVM Unity.

input length over randomly selected 10-minute traces from Twitter's production [11]. We observe that the 50%ile of sequence length is 21 tokens, whereas the 98%ile significantly rises to 72 tokens. The variability in sequence length drastically influences computation time. To demonstrate it, we measure the computation time of varying input lengths with different Transformer-based models, as depicted by the orange lines in Fig. 2. The computation time for a sequence of length 512 is 4.22x and 5.25x longer than for a sequence of length 64 in Bert-Base and Bert-Large models, respectively.

2.2 Handling varying lengths of input sequences

Existing solutions to address the issue of varying input lengths encompass the two main approaches:

Uniform zero-padding. Padding is the most common approach when deploying models for serving[9, 22, 27, 46, 47]. The idea is to treat each input sequence uniformly by zero-padding shorter sequences to match a fixed maximum length. Specifically, the input length of these models is configured to the *max_length*, a parameter representing the upper limit of lengths that can be handled by the runtime. This strict configuration enables uniformity in processing, but the inherent limitations are also pronounced. First, shorter sequences suffer from inflated latency when served by runtimes configured for longer sequences. Fig. 2a shows that a sequence of length 20 would end up with a latency of 4.86ms when served by a runtime with *max_length* as 512, which is 4.28x longer than its actual computation time. Second, computation resources are wasted. Computation over the zero-padding components are redundant and the corresponding output would be truncated eventually. For example, one trace clip in Fig. 1a results in 80.6% of the FLOPs wasted when served by a runtime with *max_length* as 125.

Compiler support. A more sophisticated approach involves compiler-level design. Engineers provided enabled dynamic shape support to existing DL compilers, such as TensorRT[8], DISC[51], TVM Nimble[33] and TVM Unity[10]. By configuring specific axes in the data shape as dynamic, runtimes can be compiled only once and accept a diverse range of input sequences during inference. While this support brings more flexibility to the serving system, we find that they are not as efficient as those compiled with a static shape. We profile the TensorRT runtime latency of two representative Bert models [17] in Fig. 2a and 2b, and compare results when dynamic shape support is enabled. The minimum latency inflation is 1.22x and the maximum can be up to 3.56x. As for another compiler TVM, it needs time-intensive tuning to well support dynamic-shape compilation. Here we directly use Dolly [3], an official well-tuned model release compiled with TVM Unity. The result is shown in Fig. 2c. Even with kernel-tuning, the latency is still, on average, 2.86x worse than untuned statically-compiled runtime, highlighting a large performance gap. The performance gap may be attributed to the kernel dispatching overhead from dynamic shape and missed aggressive fusion optimization opportunity without shape information [33, 51], which is still an ongoing area to improve its efficiency.

It is worth mentioning that researchers have also introduced other inference optimizations tailored for Transformer-based models, with removing zero-padding being one of the key optimizations [4, 20, 45]. However, these optimizations require intricate handcrafted kernel modifications to achieve optimal performance and can not be easily transferred to models with different architectures, which further complicates the deployment process.

2.3 Our Solution: Arlo

We present Arlo, an inference scheduler that efficiently handles varying-length requests in Transformer-based LMs using *polymorphing*. Our primary goal is to achieve a comparable performance to static-shape compiled runtimes while simultaneously minimizing zero-padding. We exploit multiple static-shape compiled runtimes instead of using a single unified runtime for each model. These runtimes possess different *max_length* values and are distributed within the largest request length range. Arlo involves determining an *ideal runtime* for each request to minimize zero-padding. It dynamically deploys runtimes and routes requests to the appropriate runtime based on the request length distribution.

Note that a state-of-the-art inference system INFaaS [30] also studies a problem about model selection across multiple variants with varying efficiency and accuracy. While INFaaS could potentially be adapted for transformer-based models, its vertical autoscaling strategy is designed for request load changes and does not take into account the distribution of input lengths for overall resource allocation, ultimately failing to minimize the overall latency. Furthermore, its bin-packing-like dispatching does not consider the dynamic nature of request length variability and fluctuating instance loads. We compare INFaaS with Arlo experimentally in §5.

3 SYSTEM DESIGN

3.1 Design Overview

System architecture. Arlo is an inference scheduler designed to handle the variability of input lengths in LMs requests. It strategically allocates resources to different runtimes based on the request length pattern, deploys them on GPU instances within the cluster, and efficiently schedules incoming requests.

Fig. 3 presents Arlo's architecture. In an inference system, Arlo works on one request stream (requests with the same SLO and target model) [32, 47] and we can have a dedicated Arlo for each request stream. Arlo comprises two key components, Runtime Scheduler and Request Scheduler. Runtime Scheduler determines which runtimes should be deployed and how many GPU instances are allocated to each runtime. When new requests arrive, Request Scheduler decides which runtime instance will handle each request. We also incorporate an offline profiler to obtain each runtime's computation time. Arlo exploits this knowledge to make an optimal decision.

Workflow. Initially, Arlo fragments the request length span into sub-spans ¬, referred to as length bins later. Next the model is compiled into multiple runtimes -, each with a different *max_length* value that defines the upper limit for the request length it can handle. The offline profiler measures the computation times for all the runtimes ®. During serving, Runtime Scheduler assesses the available cluster resources and the history request distribution pattern (a) to compute the resource allocation for each runtime, with runtime profiles from profiler (b). It then deploys the runtimes onto the GPU instances accordingly (c). Request Scheduler actively monitors the request buffer (e) and schedules each incoming request to the most suitable runtime instance (f) under the resource allocation decision of Runtime Scheduler (d). For each request, it takes into account all runtime candidates and the load of each instance.

Key questions. Arlo's design is centered on three questions:

- How to decide the appropriate number of runtimes required and the corresponding *max_length*?
- Given a request length distribution, how to efficiently allocate resources to each runtime to minimize the overall latency?
- How to design an efficient Request Scheduler to schedule requests to the appropriate instances?

3.2 Challenges

Arlo introduces two challenges that are not present in existing solutions.

Runtimes have different length ranges. Arlo introduces a new variable to the problem, where each runtime has a different length range it can handle. This is in contrast to existing approaches like uniform zero-padding and compiler-level support, where the same runtime is deployed across all instances, allowing requests to be executed on any instance in the cluster. In Arlo, a request with a length of *len* can only be executed by runtimes with a *max_length* greater than or equal to *len*. Therefore, it is crucial to determine a proper *max_length* for each runtime, as it affects the number of runtimes required for each model. Compiling a runtime for every possible length value is neither scalable nor efficient since it also increases Runtime Scheduler's search space when performing resource allocation. Additionally, the request length distribution must be considered during runtime scheduling, as the computation time of each runtime differs. Merely favoring runtimes with longer length ranges may lead to increased zero-padding size and higher latency, offsetting the advantages of flexibility.

Short-term request length distribution. While the distribution of request lengths tends to be stable over a longer term, it may not hold true in short term. Fig.1 depicts the length distribution for 10-minute and 10-second clips. The median length for both is 21.0, while the 98%ile is 71 for the 10-minute clips and 58 for the 10-second clips. Ideally, each request should have minimal padding. However, ensuring optimal performance for near-future requests is challenging due to the fluctuation of request lengths in the short term. Therefore, we have to carefully design the scheduling algorithm so that Request Scheduler can make a sensible decision independent of the runtime resource allocation. Common strategies may not be sufficient to achieve this objective. For example, in Fig. 4, a 4-GPU cluster is deployed with two runtime instances with a length of 128, one with length 256, and one with length 512. The length of the initial eight requests is less than 128. Later, 14 requests arrive, with lengths ranging from 257 to 512. If the ideal policy of scheduling requests to the runtime with the least padding is followed, five initial requests cannot meet the SLO as instances with length 128 can only handle three more requests. For a greedy algorithm which selects the instance with the least load, it ends up with scheduling all eight initial requests to GPU3. It makes eight latecomers fail to meet the SLO. Interestingly, by scheduling 5 out of the first eight requests to GPU2, no SLO violations would occur.

3.3 Runtime Scheduler

We first discuss how to address the first two questions in this section and introduce the design of Arlo's Runtime Scheduler.

Determine the *max_length* of each runtime. We have discussed that determining the value of *max_length* is non-trivial. Even though compilation is performed offline, creating runtimes for every possible value becomes impractical due to the trend of an increasing request length. To simplify this problem, we exploit one empirical observation: staircase pattern in latency. As shown in Fig. 2a and 2b, when using static-shape compilation, the increase of latency is significant for every 64 length step. Within each 64 length step, the latency change is tiny, usually less than 5%. This observation aligns with previous literature [18, 33] indicating that GPUs are most efficient at matrix multiplication when the sequence length is a multiple of the tile size. Therefore, it is evident that having runtimes within the 64-length step range would offer little improvement in latency (discussed in §5.2.3). Arlo linearly increases the *max_length* values for each runtime with a coefficient corresponding to the step size observed in the staircase pattern. For instance, in the case of the Bert model in Fig. 2a and 2b, the original model with a *max_length* of 512 would have eight runtimes (512/64=8). This approach ensures that the runtimes are adequately spaced out to cover the likely range of request lengths while avoiding unnecessary granularity within each 64-length step. Arlo can thus strike a balance between efficiently handling varying request lengths and avoiding excessive runtime compilation. It is noted that the 64-length step here is specific to TensorRT runtimes of Bert.

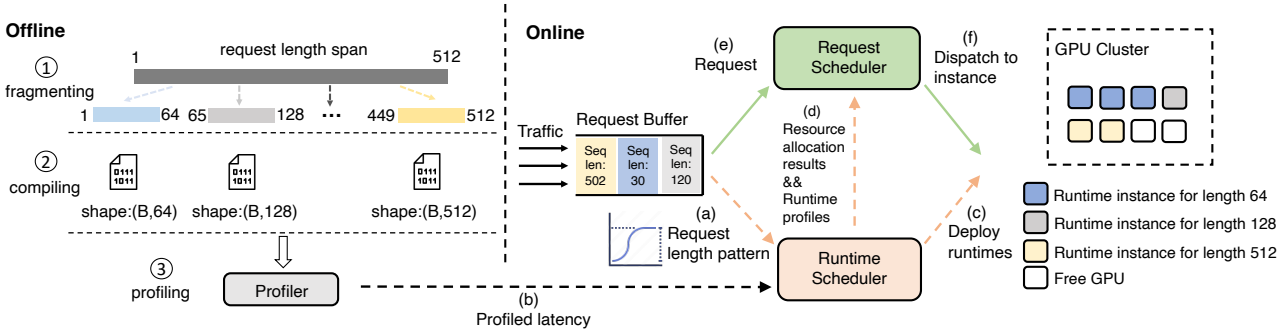


Figure 3: The system overview of Arlo.

timestamp	#request	length	SLO
0	8	<128	480ms
3	14	257~512	480ms

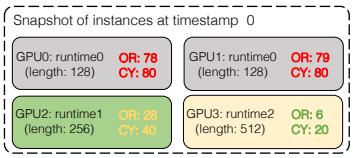


Figure 4: An example illustrating dispatching challenges with short-term variability. The outstanding requests (OR) represent the quantity of requests in the queue and in execution. Each instance's capacity (CY) is determined by per-request latency and the predefined SLO.

For other models or compilers, the step sizes may vary and not necessarily uniform.

Runtime resource allocation. With the runtimes well prepared, Runtime Scheduler now focuses on resource allocation for the online serving stage. It exploits the request length distribution and profiled runtime performance to make the decision. The problem then becomes: given G available GPUs, I different runtimes sorted by their max_length, the average number of requests Q_i during a period of the SLO in each length bin of the i-th runtime, and the runtime performance (M_i representing the maximum capacity within SLO and $\mathcal{L}_i$ representing the mapping from batch size to mean latency for the i-th runtime, which are obtained by profiling), our goal is to calculate the resource allocation N_i (GPU instances) for each runtime in a way that minimizes overall latency. We can formulate the problem with integer linear program (ILP) as follows:

$$\min \quad \sum_{i=1}^{I} \mathcal{L}_i(B_i) \cdot C_i \tag{1}$$

$$\text{s.t.} \quad \sum_{i=1}^{I} N_i = G, \tag{2}$$

$$N_i \geq \lfloor \frac{Q_i}{M_i} \rfloor, \quad \forall i, \tag{3}$$

$$R_i = \begin{cases} 0, & i = 0, \\ \max(R_{i-1} + Q_i - N_i \cdot M_i,\ 0), & i \geq 1, \end{cases} \tag{4}$$

$$C_i = \begin{cases} \min(R_{i-1} + Q_i,\ N_i \cdot M_i), & i < I, \\ R_{i-1} + Q_i, & i = I, \end{cases} \tag{5}$$

$$B_i = \frac{C_i}{N_i}, \quad \forall i, \tag{6}$$

$$N_I \geq 1. \tag{7}$$

R_i represents unprocessed requests for the i-th runtime, while C_i is the actual requests to be processed. B_i denotes the workload for each individual instance of the i-th runtime. We provide an explanation for each constraint below.

- Eq. 2: the number of instances deployed with runtimes should be the same as the number of available GPUs.
- Eq. 3: the deployed instances for each runtime should be able to handle their requests within the SLO constraints. In other words, the number of requests handled by each instance should not exceed its maximum capacity.
- We also consider the non-ideal scenario. Eqs. 4 and 5 is used for demoting requests to a runtime with a larger max_length. It happens when the current instances of the ideal runtime cannot handle the requests without violating the SLO constraints.
- Eq. 6: requests assigned to the i-th runtime are evenly distributed among its instances to balance the load.
- Eq. 7: the runtime with the largest max_length should be deployed on at least one instance. This ensures prompt handling of all requests without waiting in the buffer.

Although this formulation represents a non-linear and non-convex problem, the constraints in Eq. 3 effectively narrow down the search space. By leveraging optimization solvers such as GUROBI [5], Arlo can solve this formulation efficiently even in large-scale clusters, taking less than one second in most cases. We present more analysis on overhead in §5.1.4.

It is crucial to note that Runtime Scheduler does not consider scenarios in which multiple runtime instances of the same model are co-located on a single GPU. Although there are studies investigating concurrent execution on GPU devices [14, 35, 41], these methods often result in suboptimal performance, due to the unavoidable interference. To prevent performance degradation and maintain optimal runtime behavior, Arlo deliberately avoids such co-location. To enhance system utilization, particularly during periods of low request load, Arlo can be combined with resource time-multiplexing in a multi-request stream serving scenario, and the implications of such scheme are discussed in §6.

Resource auto-scaling. Periodic resource allocation is crucial for optimizing overall latency based on length distribution when the load level is stable under given resources. However, to address load fluctuations, an auto-scaling method is necessary. In fact, many existing scaling methods in current systems[1, 15, 16, 30, 46], such as threshold-based heuristics, could be integrated. In Arlo, an auto-scaling mechanism is implemented as described in §4. For scaling-out, a runtime instance compiled with the maximum input length is added, while for scaling-in, an instance with the least load is removed. After each auto-scaling action, the Runtime Scheduler would still make an optimal resource allocation for different runtimes with scaled resources upon a decision period and automatically adapt to the length distribution.

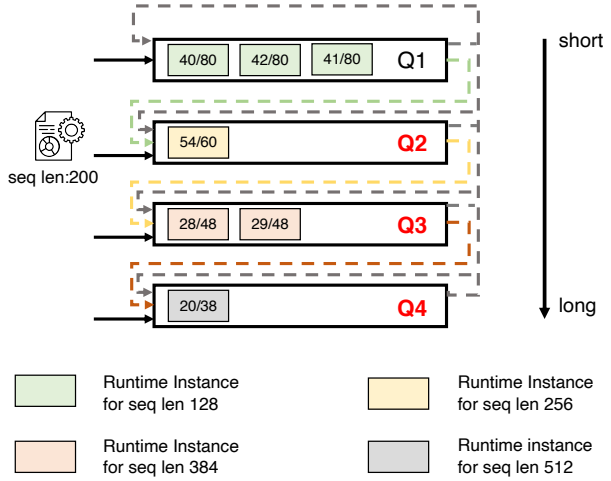


Figure 5: A multi-level queue maintained by Request Scheduler, with arrows representing the potential dispatch decision sequence. Grey arrows indicate traversing back. Each instance's outstanding requests and maximum capacity within SLO are labeled.

3.4 Request Scheduler

We then present the design of Request Scheduler to address the third design question.

Short-term requests. Recall that we have shown an inconsistency between the short-term and long-term request pattern. Runtime Scheduler can dynamically allocates resources for each runtime to match the long-term request pattern as it is performed in a lower frequency. To address the short-term pattern fluctuation, Arlo relies on Request Scheduler to dispatch request properly under the current runtime resource allocation. The design is guided by two key intuitions.

Requests could be demoted to another runtime when the ideal runtime is overloaded.

When a burst of similar-length requests arrives and overwhelms the ideal runtime instances, Request Scheduler considers the option of demoting queuing requests to less busy but less ideal runtime instances with a larger *max_length* value. This demotion is performed to alleviate the load on the ideal runtime and expedite the execution of arriving requests. Request Scheduler's decision involves measuring the trade-off between the queuing time in the ideal runtime instances and the increased latency incurred due to additional zero-paddings in the non-ideal candidates.

Demotion should be conservative to account for potential longer requests.

In Arlo, enabling demotion allows shorter requests a broader range of runtime options than longer ones. However, excessively demoting requests to non-ideal candidates may inadvertently impact the processing of longer requests. Therefore, it is critical to adopt a conservative approach to demotion. Specifically, we need to select the non-ideal candidate that is closest in *max_length* to the length of the request being considered for demotion. By doing so, Request Scheduler minimizes interference with longer requests while still optimizing the processing of shorter requests.

Multi-level queue. Based on the two intuitions mentioned earlier, Request Scheduler implements a multi-level queue, illustrated in Fig. 5. Each level of the queue corresponds to one runtime, ordered by the increasing value of their *max_length*. Within each level, Request Scheduler maintains a priority queue of instances

Algorithm 1 Request Scheduler Algorithm

Inputs: Q: The multi-level queue with K runtime queues
λ: The initial threshold for picking a runtime instance.
α: The threshold decay coefficient.
L: The maximum peeking level.
R: The arriving request

1: Initialize Q_e to $\emptyset$, $flag$ to $false$
2: $Q_e \leftarrow$ GET_SORTED_CANDIDATE_RUNTIME_INDEXES(R.length)
3: **if** $Q_e.size() > L$ **then**
4: $Q_e \leftarrow$ TopK (Q_e, L) ▷ Only peek the first L candidate runtimes.
5: **end if**
6: **for** $i \in Q_e$ **do**
7: $N \leftarrow Q[i].front().outstanding\_requests$
8: $M \leftarrow Q[i].front().max\_capacity$
9: $P \leftarrow \frac{N}{M}$ ▷ Measure the congestion level of the head instance
10: **if** $P < \lambda$ **then**
11: $q \leftarrow i$
12: $flag \leftarrow true$
13: break
14: **else**
15: $\lambda \leftarrow \lambda * \alpha$ ▷ Decrease the threshold for a lower runtime
16: **end if**
17: **end for**
18: **if** $flag == false$ **then**
19: $q \leftarrow Q_e[0]$ ▷ Pick the top candidate runtime when all candidates fail to meet the requirement
20: **end if**
21: DISPATCH(R, Q[q]) ▷ Dispatch the request to the head instance
22: UPDATE(Q[q]) ▷ Update the priority queue of selected runtime

deployed with the corresponding runtime. The instance with the least ongoing load is always positioned at the head of a queue.

As shown in Algorithm 1, when a request arrives, Request Scheduler identifies all candidate runtimes based on the request length (line 2). It iterates through the candidate runtimes in the increasing order of their *max_length*, looking up for a suitable runtime for the request. During this process, Request Scheduler computes the congestion level P for the head instance of each candidate runtime (line 7-9) and utilizes a threshold λ with a decay rate of α to adaptively determine the appropriate runtime instance. If the value of P for a candidate runtime instance is less than λ, it is considered suitable for dispatching the request and the lookup process terminates (line 10-13). Otherwise, Request Scheduler decreases the threshold with α (line 15) and continues iterating through the remaining candidate runtimes. The algorithm is constrained by parameter L (lines 3-4), limiting the number of lookup attempts to control overhead. If no candidate runtimes meet the conditions, Request Scheduler returns to the top candidate runtime queue (lines 18-19) and dispatches the request to the head instance (line 21). After dispatching, the selected runtime queue is updated accordingly (line 22).

Request scheduling example. Fig. 5 shows a simple example with four runtimes. Each maintain several running instances. Assume that L is set to 3, λ is 0.85, and α is 0.9. When a request with a length of 200 arrives, Request Scheduler begins by identifying runtime candidates ($Q2$, $Q3$, and $Q4$). It looks up at the top runtime candidate ($Q2$) first and finds out that the congestion level of its head instance is 54/60, which is greater than λ. Request Scheduler then moves on to the next candidate ($Q3$) and repeats the process with a smaller λ (0.85*0.9=0.765). Lastly, its head instance, with a congestion level of 28/48 and below 0.765, is selected for dispatching.

Time complexity. Given K deployed runtimes and N running instances, with a maximum loop iteration limit of L, the time complexity for dispatching is $O(L) + O(\log(N/K))$. The overhead remains low, as empirically tested and discussed in §5.1.4.

4 IMPLEMENTATION

We have implemented the prototype of Arlo on top of Triton Server [9] with additional ~2200 lines of code (LoC) in C++. It could support runtimes compiled by different mainstream compilers, such as XLA [12], TensorRT [8] and TVM [10]. Moreover, we develop a discrete event simulator with ~2000 LoC in Python. It accurately models the process of periodic resource allocation, instance replacement, request dispatching and batch execution with great care.

Resource scaling. As mentioned in §3.3, Arlo could integrate existing auto-scaling methods to address the load fluctuation. Therefore, we adopt a target tracking scaling mechanism as per [1]. A GPU worker is added when the 98%ile latency of recently executed requests reaches 95% of the SLO. By default, the new worker will load a runtime instance compiled for the maximum sequence length. For scaling in, we employ a conservative approach where we release the least busy instance if the 98%ile latency of recently completed requests falls below 50% of the SLO in every given time period (60 seconds here). More advanced scaling methods, like using prediction models [16, 22, 46], could also be incorporated.

Instance replacement. Each time Runtime Scheduler resolves a new allocation, it makes a replacement plan that replaces the minimum number of current runtime instances to adjust the deployment. The replacement process is carried out in small batches to prevent excessive traffic pressure on uninvolved instances. A replacement is low-overhead and usually lasts approximately 1 second. Moreover, if feasible, hot instances can be pre-loaded or cached in GPU RAM to avoid the need for swapping in and out [21, 47].

5 EVALUATION

We conduct a comprehensive evaluation of Arlo through both testbed experiments and large-scale simulations. We summarize the key takeaways as follows:

- Arlo could greatly reduce the mean latency compared with existing schemes, while also achieving much lower tail latency in both testbed and simulation scenarios.
- The key components in Arlo distinctly contribute to the performance uplift and work in concert with others.
- The overhead introduced by components in Arlo is minimal, ensuring the high efficiency and responsiveness.

Setup. For our testbed experiments, we utilize five GPU servers, each outfitted with 52 vCPUs, 125GB RAM, and two Nvidia GTX 3090 GPUs. An additional CPU server, equipped with 32 vCPUs and 32GB of RAM, is employed to deploy Arlo. The network bandwidth between servers is 25Gb/s. All servers operate under Ubuntu 20.04 and CUDA 11.8. For simulations, we use a CPU server with 52 vCPUs and 125GB RAM to emulate the operations of GPU workers.

Models. We evaluate Bert-Base and Bert-Large [17], which are two representative discriminate LMs of different scales. Their input is two-dimensional: the first dimension is the batch size, and the second is the sequence length. Statically-compiled runtimes only accept inputs with a fixed shape, while dynamically-compiled ones can handle inputs with variable sequence lengths without padding. Notably, we set the batch size for all runtimes to 1, a common practice in live data analytics [30, 37, 38].

Metrics. Our primary focus is on the mean latency and tail (98%ile) latency of the served requests.

Features	ST	DT	INFaaS	Arlo
Best Runtime Compilation Performance	✓	✗	✓	✓
Reduced Padding Size	✗	✓	✓	✓
Multiple Runtime Variant	✗	✗	✓	✓
Input length-aware Resource Allocation	✗	✗	✗	✓
Dynamics-aware Request Dispatching	✗	✗	✗	✓

Table 1: Feature comparison among different schemes

Workloads. We synthesize workload traces based on the Twitter production trace [11], the only publicly available production trace that includes text data to our knowledge. As the Twitter trace has a maximum sentence length of approximately 125, we recalibrate the sentence length distribution to span up to a maximum length of 512. We do not consider the tokenization pre-processing overhead, as modern tokenizers can efficiently tokenize gigabytes of text within seconds [7]. Since the Twitter trace only provides per-second time information, we generate the request arrival pattern within each second using a stable pattern (Poisson process) and a bursty pattern (Markov-modulated Poisson process), as in [46, 47]. We denote them as Twitter-Stable and Twitter-Bursty. The SLO is set to 150ms for Bert-Base and 450ms for Bert-Large.

Parameter settings. In Arlo, the period of Runtime Scheduler is empirically set to 120 seconds, and 8 runtimes are compiled, as explained in §3.3. For Request Scheduler, λ is set to 0.85, α to 0.9, and L to 6.

Compared schemes. We compare Arlo with other three schemes using the state-of-the-art (SOTA) DL compiler TensorRT v8.6.1 [8], which is specially optimized for dynamic shapes.

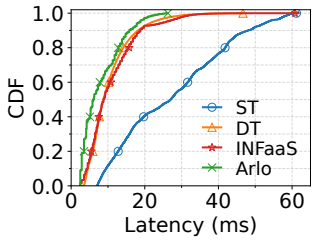
- **ST**: Deploy **s**tatically-compiled **T**ensorRT runtimes with a unified maximum length.
- **DT**: Deploy **d**ynamically-compiled **T**ensorRT runtimes.
- **INFaaS** [30]: A SOTA inference system which also utilizes the multi-variant model runtimes.

ST and DT employ the headroom-based auto-scaling heuristics from INFaaS for dynamic resource adjustment and use load balancing for request dispatching due to their uniform runtimes. INFaaS retains its bin-packing dispatching scheme, allocating requests among instances that satisfy the specified input length requirements. Table 1 compares the features of different schemes.

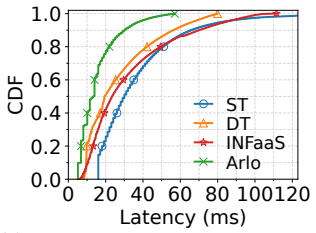
5.1 Testbed Results

5.1.1 Latency comparison with fixed GPU resource. We conduct experiments in a 10-GPU testbed for two request streams under Twitter-Stable traces with given request loads. Fig. 6 shows that ST exhibits the poorest performance due to padding each request to the maximum length (512 here). Conversely, DT significantly reduces overall latency without the need for padding. Despite this, DT still results in long tail latency due to the suboptimal performance introduced by dynamic compilation, while INFaaS suffers similarly due to suboptimal instance allocation and naive dispatching. However, Arlo successfully strikes a balance between padding size and runtime latency. Specifically, Arlo reduces the mean latency by 70.3% and 66.7% compared to ST, by 23.7% and 29.2% compared to DT, and by 24.9% and 39.3% compared to INFaaS, for two different streams. Moreover, Arlo significantly improves tail latency, reducing it by up to 89.4%, 25.9%, and 40.1% compared to ST, DT, and INFaaS.

5.1.2 Performance with varying load. The impact of request load is also evaluated by varying the load level using the Twitter-Stable



(a) The latency CDF of Bert-Base stream under Twitter-Stable (1.5k req/s) with 10 GPUs.



(b) The latency CDF of Bert-Large stream under Twitter-Stable (1k req/s) with 10 GPUs.

Figure 6: Testbed results. Latency for Bert-Base and Bert-Large streams under different loads and scales .

# GPU	# runtimes	Time (s)
50	8	0.156
200	12	0.623
1000	16	2.612

Table 2: The ILP solving time of Runtime Scheduler with varying GPU number, runtime number and request traffic, averaged over 20 runs.

trace for the Bert-Base stream in our testbed, with the results presented in Fig. 7. At a low arrival rate (e.g., less than 1k req/s), all systems exhibit good performance, and their metrics do not differ significantly. However, with increasing arrival rates, the burden on the systems and the resultant queue sizes escalate. This effect is particularly pronounced for ST, where the requirement to process all requests with full padding leads to elongated queuing times and deteriorated mean latency. In contrast, Arlo's efficient resource allocation and sensible request dispatching markedly diminish queue times compared with other schemes and enhance overall performance, particularly in high-load scenarios.

5.1.3 Consumed GPU number with auto-scaling. Apart from the fixed GPU scenario, we also experiment with auto-scaling enabled, using a highly varying-load Twitter-Bursty trace for Bert-Large. Initially, 5 GPUs are provisioned, and during inference, the GPU scaling process is activated through the target tracking mechanism mentioned in §4, allowing for GPUs to scale in and out. Fig. 8 shows that Arlo utilizes fewer GPUs than others, with a time-weighted GPU number of 5.49, in contrast to 6.38 for DT, 6.80 for INFaaS, and 8.13 for ST. Despite using fewer GPUs, Arlo achieves a better tail latency of 330.41, compared to 397.10 for DT, 404.12 for INFaaS and 430.54 for ST. This highlights the benefits of incorporating Arlo with resource scaling in multi-tenant data centers, as it can handle the same traffic load for a request stream with fewer compute resources.

5.1.4 The overhead analysis. We systematically evaluate the overheads of Runtime Scheduler and Request Scheduler to show neither of them impact the performance of Arlo.

Overheads in Runtime Scheduler. We measure the ILP solve time in Runtime Scheduler for various GPU workers, runtimes, and request traffic. Table 2 shows minimal overhead, with a solution time of about 2.612 seconds for 1000 GPUs and 16 runtimes. This duration is significantly shorter than the observed request fluctuation period of at least several minutes in the Twitter trace.

Overheads in Request Scheduler. §3.4 mentions that the theoretical overhead of Request Scheduler is minimal, with each dispatch operation completing within microseconds in the testbed. To measure overheads in large-scale deployments, we emulate runtime

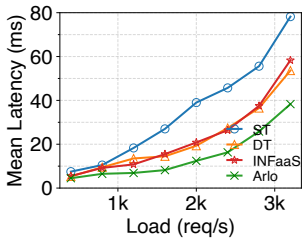


Figure 7: The mean latency with varying request load under Twitter-Stable with 10 GPUs.

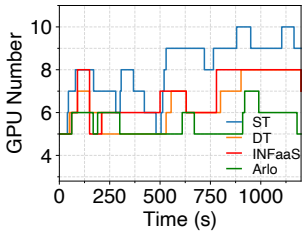


Figure 8: Consumed GPU number for four schemes under highly varying load.

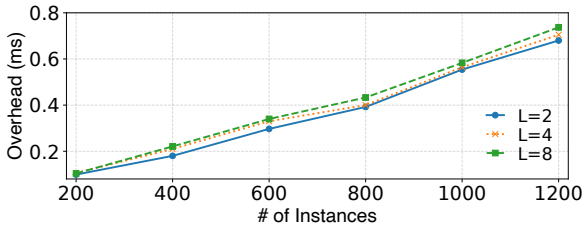


Figure 9: The overhead of Request Scheduler under different load and scales. Note that the concurrent request load scales with deployed instances.

instances using CPU cores. With 12 runtimes, we vary the number of instances from 200 to 1200 and generate between 400 and 2400 concurrent requests. Fig. 9 shows the overhead in processing these concurrent requests with different L (maximum peeking level in Algorithm 1) under different scales. Even with 1200 instances and a burst of 2400 requests, it takes only about 0.737 ms. Besides, a larger L could bring a slight increase in overhead, depending on the request pattern and instance load. We could see that Request Scheduler can easily handle 150k requests per second, the largest load setting among other works [30, 32, 46, 47], without being the bottleneck of Arlo.

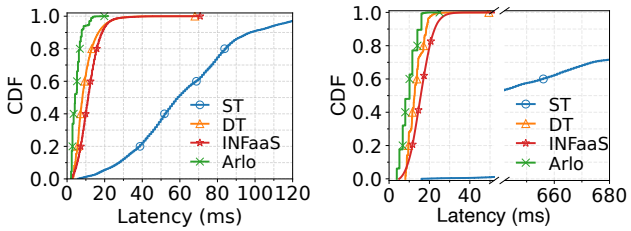
5.2 Large-scale Simulations

5.2.1 Simulator calibration and fidelity. To establish its fidelity, we evaluate our simulator against the prototype in a testbed using small traces (5-10 min). To account for overheads, such as data transmission in the network and from CPU to GPU RAM, we add a fixed overhead of 0.8ms per request in the simulator. Simulation results closely match the testbed results, with mean and 98%ile latency differing by only 4.3% and 2.6%, respectively.

5.2.2 Latency comparison with fixed GPU resource. We conduct large-scale simulations with different GPUs for two request streams. As depicted in Fig. 10, ST's performance suffers due to constant zero-paddings, further exacerbated by the bursty workload. DT outperforms ST in both mean and tail latency, benefiting from its flexible, padding-free processing. Despite employing multi-variant models, INFaaS still underperforms compared to DT, consistent with testbed results. Arlo surpasses all of them, reducing the mean latency by 70.3% and 98.1% compared to ST, by 24.1% and 30.7% compared to DT and by 31.3% and 41.7% compared to INFaaS. The tail latency is also reduced up to 98.4%, 26.0% and 29.3% respectively.

5.2.3 Deep-Dive. We now dig deeper into how each key component in Arlo contributes to overall performance improvements.

Benefit of Runtime Scheduler. Firstly, We vary the number of runtime types to validate Runtime Scheduler's choice. Fig. 11



(a) The latency CDF of Bert-Base stream under Twitter-Bursty (8k req/s) with 90 GPUs.

(b) The latency CDF of Bert-Large stream under Twitter-Bursty (25k req/s) with 300 GPUs.

Figure 10: Simulation results. Latency for Bert-Base and Bert-Large streams under different loads and scales. We truncate x axis to better display the data.

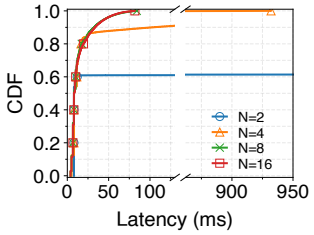


Figure 11: The latency CDF for N available runtimes under a trace with 40 GPUs for Bert-Large stream.

Scheme	Mean(ms)	Median(ms)	P98(ms)
Even	455.19	21.13	3407.02
Global	80.58	8.21	1514.53
Arlo	**11.51**	**8.02**	**75.04**

Table 3: Latency comparison of different runtime resource allocation methods under the same trace with 40 GPUs.

shows the latency CDF with N runtimes under a trace with 40 GPUs, where the max_length of each runtime has a step of $512/N$. With only 2 runtimes, Arlo fails to serve the stream with excessively long queuing times for many requests, while with 4 runtimes, it can roughly handle the load, albeit with a 2.5% SLO violation. However, with Runtime Scheduler's chosen configuration of 8 runtimes, Arlo causes no SLO violation and achieves a performance comparable to that with 16 runtimes, with a mean latency of 14.16 (98%ile of 84.04) as opposed to 14.45 (98%ile of 81.74) for 16 runtimes. This suggests that compiling too few runtimes can lead to suboptimal performance, as the impact of padding on computation efficiency remains significant. Additionally, creating an excessive number of runtimes does not necessarily bring remarkable benefits, as the performance difference between runtimes compiled with two close lengths is minimal. Since a larger amount of runtimes also intensify overhead of solving ILP in Runtime Scheduler (§5.1.4), it is sensible to decide the number of runtimes depending on the specific latency pattern of the model as discussed in §3.3. Next, we compare Runtime Scheduler's periodic resource allocation to two offline schemes: even GPU allocation per runtime and allocation based on global trace length distribution. Table 3 shows both offline schemes fail to achieve optimal performance with dynamic workloads, highlighting the need for periodic allocation. Fig. 12 illustrates the GPU numbers Runtime Scheduler allocates to eight runtimes throughout the trace.

Benefit of Request Scheduler. We also compare Request Scheduler (RS) with two conventional dispatching strategies. The first, Intra-group Load Balance (ILB), dispatches a request to the runtime requiring the least padding and maintains the load balance among instances of each runtime. The second, Inter-groups Greedy (IG), dispatches each request to the least busy instance among all candidate runtime queues. We replace RS with ILB and IG in Arlo and compare the overall latency with three different Twitter-Bursty traces for Bert-Large. The results in Table 4 demonstrate that RS significantly lowers tail latency, with reductions of up to 95.6% relative to ILB and up to 58.7% compared to IG. This marked improvement

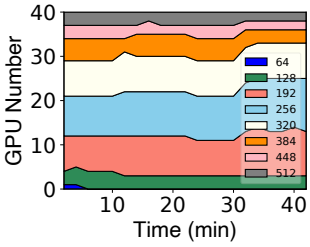


Figure 12: The GPU number allocated to eight runtimes by Runtime Scheduler.

Trace	Strategy	Mean(ms)	P98(ms)
6k req/s 25 GPUs	ILB	1035.90	9555.3
	IG	150.21	422.03
	RS	**77.8**	**414.58**
20k req/s 80 GPUs	ILB	77.83	1024.78
	IG	157.09	1183.85
	RS	**73.61**	**810.41**
250k req/s 1000 GPUs	ILB	76.23	419.04
	IG	172.00	990.29
	RS	**76.09**	**408.79**

Table 4: Latency comparison of different dispatching strategies for Bert-Large streams at different scales under Twitter-Bursty traces.

largely stems from RS's request demotion strategy and its anticipation of longer impending requests, which collectively alleviate potential congestion across different runtime instances. Besides, RS substantially reduces mean latency by up to 92.5% and 55.8% compared to ILB and IG, respectively. In the first two traces, RS consistently outperforms both ILB and IG, which alternate in performance. For the third trace with weak short-term length pattern fluctuation, RS slightly outperforms ILB but significantly dominates IG. Here, RS approximates ILB, while IG's greedy seizing of less busy runtime instances with larger max_length overloads them.

6 DISCUSSION

Multiple request streams. As discussed in §3.3, Arlo is designed for resource scheduling tailored to single request stream but can be extended to handle multiple streams. This can be achieved by deploying a dedicated Arlo for each stream and employing resource sharing among them through time-multiplexing scheduling. In this multi-stream setup, runtime instances for different models could be co-located on a same GPU, aligning with several extant systems [22, 47]. Concurrently serving multiple streams can improve system utilization compared to single-stream processing. While Arlo is adaptable to multi-stream scenarios, it also presents unique opportunities and challenges in resource provisioning and multiplexing efficiency, offering potential avenues for further development.

Dynamic batch execution. Due to the even worse performance of compilers for inference involving multiple dynamic axes (e.g., dynamic batch size and sequence length for LM inputs)[8], we focus on resource scheduling for dynamic request length, with a fixed small batch size. Batch size often trades off between throughput and per-request latency. A small batch size is conservative and reasonable in latency-sensitive scenarios, complemented by the excellent performance and resource utilization afforded by SOTA compilers. However, ideally, batch size should be dynamic in response to traffic load. This requires advanced support from compilers or more fine-grained scheduling in both batch size and length dimensions, which could result in potential overheads from frequent instance replacements. We leave these as future work.

Large models with multiple GPUs. Arlo is primarily optimized for serving small models that can be accommodated within a single GPU. Nonetheless, when faced with models too large to fit in one GPU, model parallelism becomes a necessity. Such parallelism can be implemented either through inter-operator or intra-operator strategies. Regardless of the parallelism technique employed, the computational load remains dependent on the input shape. Consequently, Arlo can still be effectively leveraged to schedule resources and manage the serving of these larger models.

7 RELATED WORK

DL inference system. Previous work has focused on resource autoscaling and scheduling to handle diverse model request streams and meet their overall demands, which can be categorized into two strategies: per-stream approaches [16, 32, 46] and global strategies [21, 47]. Additionally, some studies have explored automatic model selection and model-less services [22, 30, 38]. Besides, certain systems have been optimized with customized GPU kernel, specifically for transformer models [20, 45]. Recent generative LM-specific inference systems [23, 26, 43, 50] have been developed to address the unique challenges of autoregressive generation, optimizing performance from various angles. In contrast, Arlo takes a distinct approach, focusing on discriminative LMs and optimizing resource allocation for specific request streams with varying input lengths, allowing for seamless integration into most existing systems.

DL compilers. Existing DL compilers [8, 10, 12, 13, 31] provide excellent support for static shapes. They rely on pre-defined input shapes to determine the tensor shapes, allocate memory, and optimize kernel performance. Several of them [8, 10, 33] also add support for dynamic shapes, albeit at the cost of performance. Besides, the dynamic sparsity optimization is also explored [48, 49]. Arlo schedules statically-compiled runtimes and achieves a balance between input flexibility and performance, enabling it to efficiently handle length-variable requests without any special dynamic compilation support.

8 CONCLUSION

In this paper, we present Arlo, an inference scheduler specifically designed for request streams with varying input lengths. Arlo realizes the idea of *polymorphing*, which involves compiling and scheduling multiple runtimes that are statically compiled for different lengths. This aims to strike a balance between zero-padding size and runtime performance. A multi level queue-based heuristic is employed to address the online request dynamics and dispatch requests more intelligently. Our evaluation demonstrates that Arlo outperforms existing schemes by a significant margin.

ACKNOWLEDGMENTS

This work is supported in part by funding from the Research Grants Council of Hong Kong (CRF C7004-22G) and from CUHK (4055199).

REFERENCES

[1] Amazon autosacling. https://docs.aws.amazon.com/autoscaling/.
[2] ChatGPT. https://chat.openai.com/.
[3] Dolly. https://www.databricks.com/blog/2023/04/12/dolly-first-open-commercially-viable-instruction-tuned-llm.
[4] FasterTransformer. https://github.com/NVIDIA/FasterTransformer.
[5] Gurobi Optimization. https://www.gurobi.com/.
[6] HuggingFace. https://huggingface.co/.
[7] HuggingfaceTokenizers. https://github.com/huggingface/tokenizers.
[8] TensorRT. https://developer.nvidia.com/tensorrt.
[9] Triton inference server. https://github.com/triton-inference-server/server.
[10] TVM Unity. https://github.com/apache/tvm/tree/unity.
[11] Twitter streaming traces. https://archive.org/details/twitterstream.
[12] XLA: Accelerated Linear Algebra. https://github.com/openxla/xla.
[13] Tianqi Chen et al. 2018. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *Proc. USENIX OSDI.*
[14] Seungbeom Choi et al. 2022. Serving heterogeneous machine learning models on Multi-GPU servers with Spatio-Temporal sharing. In *Proc. USENIX ATC.*
[15] Daniel Crankshaw et al. 2017. Clipper: A Low-Latency Online Prediction Serving System.. In *Proc. USENIX NSDI.*
[16] Daniel Crankshaw et al. 2020. InferLine: latency-aware provisioning and scaling for prediction serving pipelines. In *Proc. ACM SoCC.*
[17] Jacob Devlin et al. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proc. NAACL-HLT.*
[18] Tianyu Ding et al. 2023. The Efficiency Spectrum of Large Language Models: An Algorithmic Survey. *arXiv preprint arXiv:2312.00678* (2023).
[19] Yingtong Dou et al. 2021. User preference-aware fake news detection. In *Proc. ACM SIGIR.*
[20] Jiarui Fang et al. 2021. Turbotransformers: an efficient gpu serving system for transformer models. In *Proc. ACM PPoPP.*
[21] Arpan Gujarati et al. 2020. Serving dnns like clockwork: Performance predictability from the bottom up. *arXiv preprint arXiv:2006.02464* (2020).
[22] Jashwant Raj Gunasekaran et al. 2022. Cocktail: A Multidimensional Optimization for Model Serving in Cloud. In *Proc. USENIX NSDI.*
[23] Woosuk Kwon et al. 2023. Efficient Memory Management for Large Language Model Serving with PagedAttention. In *Proc. ACM SOSP.*
[24] Mike Lewis et al. 2020. BART: Denoising Sequence-to-Sequence Pre-training for Natural Language Generation, Translation, and Comprehension. In *Proc. ACL.*
[25] Wenhao Lu et al. 2020. Twinbert: Distilling knowledge to twin-structured compressed bert models for large-scale retrieval. In *Proc. ACM CIKM.*
[26] Xupeng Miao et al. 2024. SpecInfer: Accelerating Large Language Model Serving with Tree-based Speculative Inference and Verification. In *Proc. ACM ASPLOS.*
[27] Christopher Olston et al. 2017. TensorFlow-Serving: Flexible, High-Performance ML Serving. In *Workshop on ML Systems at NIPS.*
[28] OpenAI. 2023. GPT-4 Technical Report. *arXiv preprint arXiv:2303.08774* (2023).
[29] Colin Raffel et al. 2020. Exploring the limits of transfer learning with a unified text-to-text transformer. *The Journal of Machine Learning Research* (2020).
[30] Francisco Romero et al. 2021. INFaaS: Automated model-less inference serving. In *Proc. USENIX ATC.*
[31] Nadav Rotem et al. 2018. Glow: Graph lowering compiler techniques for neural networks. *arXiv preprint arXiv:1805.00907* (2018).
[32] Haichen Shen et al. 2019. Nexus: A GPU cluster engine for accelerating DNN-based video analysis. In *Proc. ACM SOSP.*
[33] Haichen Shen et al. 2021. Nimble: Efficiently compiling dynamic neural networks for model inference. In *Proc. Machine Learning and Systems.*
[34] Shivangi Singhal et al. 2019. Spotfake: A multi-modal framework for fake news detection. In *Proc. IEEE BigMM.*
[35] Cheng Tan et al. 2021. Serving DNN models with multi-instance gpus: A case of the reconfigurable machine scheduling problem. *arXiv preprint arXiv:2109.11067* (2021).
[36] Ashish Vaswani et al. 2017. Attention is all you need. In *Proc. ACM NIPS.*
[37] Guanhua Wang et al. 2021. sensai: Convnets decomposition via class parallelism for fast inference on live data. In *Proc. Machine Learning and Systems.*
[38] Yiding Wang et al. 2023. Tabi: An Efficient Multi-Level Inference System for Large Language Models. In *Proc. ACM EuroSys.*
[39] Zhiguo Wang et al. 2019. Multi-passage bert: A globally normalized bert model for open-domain question answering. *arXiv preprint arXiv:1908.08167* (2019).
[40] Bingyang Wu et al. 2023. Fast Distributed Inference Serving for Large Language Models. *arXiv preprint arXiv:2305.05920* (2023).
[41] Fei Xu et al. 2022. igniter: Interference-aware gpu resource provisioning for predictable dnn inference in the cloud. *IEEE TPDS* (2022).
[42] Shaowei Yao et al. 2022. ReprBERT: Distilling BERT to an Efficient Representation-Based Relevance Model for E-Commerce. In *Proc. ACM SIGKDD.*
[43] Gyeong-In Yu et al. 2022. Orca: A distributed serving system for Transformer-Based generative models. In *Proc. USENIX OSDI.*
[44] Matei Zaharia et al. 2012. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for in-Memory Cluster Computing. In *Proc. USENIX NSDI.*
[45] Yujia Zhai et al. 2023. Bytetransformer: A high-performance transformer boosted for variable-length inputs. In *Proc. IEEE IPDPS.*
[46] Chengliang Zhang et al. 2019. MArk: Exploiting Cloud Services for Cost-Effective, SLO-Aware Machine Learning Inference Serving.. In *Proc. USENIX ATC.*
[47] Hong Zhang et al. 2023. SHEPHERD: Serving DNNs in the Wild. In *Proc. USENIX NSDI.*
[48] Ningxin Zheng et al. 2022. {SparTA}:{Deep-Learning} Model Sparsity via {Tensor-with-Sparsity-Attribute}. In *Proc. USENIX OSDI.*
[49] Ningxin Zheng et al. 2023. SparDA: Accelerating Dynamic Sparse Deep Neural Networks via Sparse-Dense Transformation. *arXiv preprint arXiv:2301.10936* (2023).
[50] Yinmin Zhong et al. 2024. DistServe: Disaggregating Prefill and Decoding for Goodput-optimized Large Language Model Serving. *arXiv preprint arXiv:2401.09670* (2024).
[51] Kai Zhu et al. 2021. DISC: A dynamic shape compiler for machine learning workloads. In *Proc. the 1st Workshop on Machine Learning and Systems.*
[52] Liu Zhuang et al. 2021. A Robustly Optimized BERT Pre-training Approach with Post-training. In *Proc. Chinese National Conference on Computational Linguistics.*