CS 498 LSG A Dynamic Method to Speedup Inference Prefill through Requests Packing in Real Workload

Lei Huang
leih5@illinois.edu

Linshu Yang
linshuy2@illinois.edu

Wangjia Zhan
wangjia2@illinois.edu

Yuchen Wang
yuchen87@illinois.edu

Beichen Huang
beichen8@illinois.edu

Nikhita Punati
npunati2@illinois.edu

Abstract

During the inference of large language model, prefill is the stage that compute all the KV caches for user requests. However, performing batching in the prefill stage will require both waiting and padding, which increases the time-to-first-token (TTFT). Existing work uses packing to address this issue, but ignores the fact of temporal pattern of user requests arrival and did experiments on static and aggregated dataset only. We address this issue by designing a Designs Dynamic Controller for temporal request, fitting the needs of online serving. We uses real-world workload to evaluate packing performance. The TTFT outperforms existing solution under real-world workload up to 20%.

1 Introduction

Modern Large Language Model (LLM) services are fundamentally constrained by two user-visible metrics: **Time-to-First-Token (TTFT)** and **throughput**. Prefill—the phase that processes the input prompt, constructs the key-value (KV) cache, and determines subsequent GPU occupancy—dominates both metrics. Yet production deployments suffer from two systemic inefficiencies that make prefill particularly expensive:

1. **Temporal inefficiency: suboptimal batching under dynamic workloads.** Real-world traffic exhibits non-stationary patterns—periods of low arrival rates punctuated by sudden bursts. Fixed batching policies cannot adapt: aggressive batching during off-peak periods inflates TTFT unnecessarily, while insufficient batching during bursts leaves compute and memory bandwidth underutilized.

2. **Spatial inefficiency: padding waste from length heterogeneity.** Requests within a batch exhibit significant length variability. Without packing, shorter sequences are padded to match the longest sequence, resulting in wasted FLOPs, redundant memory transfers, and inflated kernel execution time—all without contributing useful computation.

These two axes are **orthogonal but coupled**: batching decides *when/which requests* co-execute, while packing decides *how* to arrange them to avoid padding. Today's systems treat them with static heuristics, forcing operators into a permanent trade-off: either low TTFT with poor GPU utilization or high throughput with sluggish first tokens.

This problem matters for three main reasons:

- **User experience & retention.** TTFT is the first impression; even small increases at the 95th percentile are noticeable in chat and API workloads.
- **Cost efficiency.** Padding inflates prefill FLOPs and memory traffic, directly raising $/token. In length-skewed mixes, the fraction of wasted (padded) tokens often reaches double-digit percentages, compounding inefficiency at scale.
- **Capacity planning.** Static knobs are brittle. A policy tuned for noon may break at 9 p.m. Temporal adaptivity leads to fewer emergency scale-outs and better service-level objective (SLO) compliance.

To this end, we summarize our **contribution as follows**:

- Design an AIMD-based dynamic batching controller that adaptively adjusts prefill triggering thresholds for temporal efficiency.
- Propose a DP-based optimal packing algorithm that maximizes token utilization and minimizes padding waste for spatial efficiency.
- Demonstrate up to 20% TTFT improvement over static baselines on real-world production traces with diurnal traffic patterns.

2 Background

2.1 Packing in Inference

Most existing LLM serving systems focus on optimizing the *decode phase*, where tokens are generated iteratively and GPUs often become under-utilized without dynamic scheduling. **vLLM** [1], **SGLang** [2], **Orca** [3], and **TensorRT-LLM** [4] all improve decoding throughput through techniques such as *continuous batching*, *iteration-level scheduling*, and *in-flight batching*. These methods dynamically mix requests during generation but largely leave the *prefill phase* unoptimized, typically processing incoming prompts in a first-come-first-served manner.

In contrast, recent work such as **Prepacking** [5] and our method focus specifically on improving prefill efficiency. Prepacking concatenates heterogeneous-length prompts into a single forward pass using a BINPACK-style algorithm with boundary-aware masking and position encoding, reducing padding waste but using a fixed waiting window. Our approach extends this direction by introducing an adaptive triggering mechanism based on AIMD feedback control and optimal packing via dynamic programming, achieving better throughput–latency balance without modifying the inference engine.

2.2 Packing in Training

2.2.1 Supervised Fine-Tuning (SFT)

Packing, which concatenates multiple variable-length sequences into longer fixed-length examples, has been widely adopted in supervised fine-tuning (SFT) to reduce padding overhead and improve compute efficiency. A recent empirical study on packing in SFT [6] systematically evaluates its impact across different model sizes (8B–70B) and dataset scales (69K–1.2M examples). The results show that packing often leads to better performance than padding under similar compute budgets, its benefits become more pronounced for larger models or datasets, and naive concatenation of unrelated samples may introduce context contamination that hurts generalization.

Another line of work integrates packing with efficient attention kernels. A HuggingFace blog post on packing with Flash Attention [7] points out that naive packing can cause cross-example attention leakage, and proposes boundary masking to preserve per-example independence. This method achieves up to a $2\times$ throughput improvement in instruction tuning without loss of model quality. The same ideas have also been implemented in HuggingFace Transformers, for example through the `DataCollatorWithFlattening` and a related pull request [8], which add explicit support for packing with Flash Attention 2.

While these works mainly focus on packing offline, a more recent work [9] has also supported online packing with flexible recipe. It is also interruptible and has been merged into huggingface Transformers since 4.44.

2.2.2 Multimodal Packing

While packing has been extensively studied in the context of supervised fine-tuning with text-only data, its extension to multimodal inputs is still less explored, particularly for general-purpose large language models. HuggingFace's Efficient MultiModal Data Pipeline (MMDP) [10] is one of the most concrete references in this area. In MMDP, packing evolves from naive padding to knapsack-based strategies. Stage 4 introduces knapsack packing for text sequences, where shorter examples are combined to reduce padding waste, and Stage 5 extends this idea to multimodal data by adding dual constraints, such as a maximum token length and a limit on the number of images per batch, with a balanced greedy knapsack algorithm to balance textual and visual modalities.

However, MMDP remains oriented toward batch construction in training pipelines. It does not address challenges such as online or streaming packing, unbiased sample exposure, or resumability across checkpoints. In addition, MMDP does not provide a detailed treatment of modality cost weighting, cross-modal adjacency bias, or integration with model inference and serving.

3 Design

3.1 Overview

We propose a *workload-aware prefill optimizer* that jointly optimizes two orthogonal dimensions:

1. A **dynamic batching controller** that adaptively determines *when* to trigger prefill and *how many* requests to accumulate based on real-time latency feedback and workload telemetry.

2. A **spatial packing procedure** that arranges variable-length sequences within each batch to minimize padding waste while respecting engine constraints.

The temporal controller employs an Additive-Increase Multiplicative-Decrease (AIMD) rule to balance TTFT and throughput across varying load conditions, while the spatial optimizer performs offline packing (solved via dynamic programming) to maximize effective token utilization. Together, they enable the system to dynamically navigate the throughput–TTFT trade-off frontier without requiring invasive engine modifications. Unlike prior approaches that treat batching and packing as independent heuristics with fixed parameters, our design explicitly couples temporal admission control with spatial layout optimization. This coupling enables the system to adapt both *when* batches are formed and *how* requests are arranged within each batch, yielding robust performance across highly non-stationary workloads.

3.2 Temporal Controller: AIMD-Based Dynamic Batching

Let S_t denote a smoothed TTFT signal at control epoch t, computed as $S_t = \text{EWMA}_\gamma(\text{TTFT}_{\text{p95}})$ with smoothing factor $\gamma \in (0,1)$. We define a target hysteresis band $[\tau_{\text{low}}, \tau_{\text{high}}]$ around the desired TTFT SLO to prevent control oscillations.

The controller maintains a *batching threshold* $N_t \in [N_{\min}, N_{\max}]$ representing the number of requests to accumulate before triggering prefill. At each control epoch, N_t is updated according to:

$$N_{t+1} = \begin{cases} \min\{N_{\max}, N_t + \alpha\}, & \text{if } S_t \le \tau_{\text{low}} \quad \text{(AI)} \\ \max\{N_{\min}, \lceil \beta \cdot N_t \rceil\}, & \text{if } S_t \ge \tau_{\text{high}} \quad \text{(MD)} \\ N_t, & \text{otherwise} \quad \text{(stable region)} \end{cases} \tag{1}$$

where $\alpha \in \mathbb{N}^+$ is the additive increase step and $\beta \in (0,1)$ is the multiplicative decrease factor.

Control intuition. The AIMD rule allows the controller to cautiously probe available batching slack while reacting quickly to SLO violations. Additive increase gradually expands the batching threshold when latency is comfortably below the target, enabling better packing and higher throughput. Multiplicative decrease, in contrast, rapidly shrinks the window upon SLO violations, bounding worst-case waiting time and preventing runaway TTFT under sudden workload shifts. Figure 1 illustrates the controller structure.

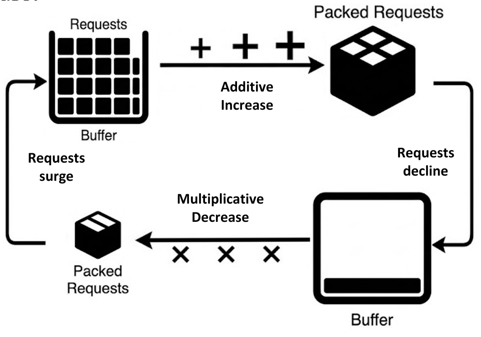


Figure 1: AIMD-based dynamic batching. The controller adaptively adjusts the prefill admission threshold to balance packing efficiency and TTFT under non-stationary workloads.

Workload-aware safety guards. To prevent SLO violations during traffic bursts, we monitor two auxiliary signals:

- **Queue depth** q_t: current number of pending requests
- **Arrival rate estimate** $\hat{\lambda}_t$: exponentially weighted moving average of recent arrival rate

When $q_t \ge q_{\text{thresh}}$ or $\hat{\lambda}_t \ge \lambda_{\text{thresh}}$, the controller immediately triggers prefill regardless of N_t (burst override), preventing queue buildup. Additionally, we gate upward adjustments by GPU utilization: increases are only applied when streaming multiprocessor (SM) utilization falls below a threshold u^* (e.g., 85%), avoiding over-batching on saturated devices. These guards decouple correctness from control convergence: even if the AIMD controller reacts slowly or encounters transient noise, burst overrides provide a hard safety net that bounds queueing delay and preserves SLO compliance.

3.3 Spatial Optimizer: Optimal Packing via Dynamic Programming

Traditional continuous batching systems perform *online* packing: requests are admitted in arrival order with no lookahead, forcing greedy first-fit strategies. In contrast, our temporal controller deliberately buffers requests, converting the problem to *offline bin-packing* where all sequence lengths $\{L_i\}_{i \in \mathcal{R}_t}$ are known in advance.

Given a candidate set $\mathcal{R}_t$ at trigger time t, we partition it into mini-batches $\mathcal{B}_t = \{b_1, \ldots, b_K\}$ to maximize useful tokens:

$$\max_{\mathcal{B}_t} \sum_{k=1}^{K} \sum_{i \in b_k} L_i \quad \text{s.t.} \quad \sum_{i \in b_k} L_i \le T_{\max}, \quad |b_k| \le B_{\max}, \quad \forall k \tag{2}$$

We solve this optimally via **dynamic programming**. Define $\text{DP}[i][k]$ as the maximum useful tokens achievable by packing the first i requests into k bins. For each state, we enumerate all feasible subsets $S \subseteq \{1, \ldots, i\}$ that fit into a single bin (i.e., $\sum_{j \in S} L_j \le T_{\max}$ and $|S| \le B_{\max}$), yielding:

$$\text{DP}[i][k] = \max_{S \subseteq \{1,\ldots,i\}} \left\{ \text{DP}[i - |S|][k-1] + \sum_{j \in S} L_j \right\} \tag{3}$$

The optimal solution is $\max_k \text{DP}[|\mathcal{R}_t|][k]$. With complete length information, optimal packing is expected to significantly reduce wasted padding versus online first-fit by enabling tighter bin utilization, and potentially decrease the number of kernel launches by consolidating requests more efficiently. The DP overhead is expected to be negligible relative to GPU pre-fill compute for typical production batch sizes.

Motivating Example: As depict in Figure 2, imagine a GPU batch capacity of 10 token capacity and four incoming requests with lengths: [7, 6, 4, 3]. In the greedy approach, the first batch would take 7. The next request is 6, which would be too big to fit within the GPU batch (7 + 6 = 13 > 10) so the first batch finishes with 7 tokens, wasting 3. The second batch includes 6 and 4 which fit perfectly to 10 tokens and waste 0 tokens. The final third batch takes 3 tokens and wastes 7. In the greedy approach, we take 3 batches and waste 10 tokens total.

On the other hand, our DP approach knows a set of request token lengths in advance, it can determine how to fit them into slots such that it would reduce the amount of waste. In the previous example, our DP approach would combine 3 and 7 into one batch and 6 and 4 into another batch, resulting in 2 batches and 0 tokens wasted in total.

Complexity and practicality. Although the formulation is optimal, the effective search space is small in practice. The number of buffered requests $|\mathscr{R}_t|$ is bounded by $N_{\max}$, which is typically on the order of tens. Moreover, sequence lengths are known and highly skewed, allowing aggressive pruning of infeasible subsets. As a result, the DP overhead is negligible compared to the GPU prefill cost, and packing can be performed entirely on the CPU without impacting the critical path. Crucially, offline packing is only possible because the temporal controller intentionally introduces batching boundaries. Without this coordination, continuous admission forces strictly online heuristics that leave substantial packing efficiency on the table.

3.4 Integration with Continuous Batching Engines

Modern LLM serving systems (e.g., vLLM, TensorRT-LLM) employ *continuous batching*: incoming requests are immediately admitted to prefill upon arrival, and new requests can join ongoing decode iterations dynamically. While this maximizes responsiveness, it fundamentally precludes packing—without explicit batching boundaries, there is no opportunity to inspect multiple request lengths simultaneously and optimize their spatial arrangement.

Our key insight is to introduce a **staging queue** Q that temporarily buffers requests before admission, effectively converting continuous batching into *controlled micro-batching*. This creates explicit batching boundaries where packing can be applied. The integration involves four components:

1. **Request buffering**: Incoming requests enter staging queue Q rather than directly entering the engine's scheduler.

2. **Trigger policy**: Prefill is triggered when $|Q| \geq N_t$ (temporal threshold), or burst override activates ($q_t \geq q_{\text{thresh}}$ or $\hat{\lambda}_t \geq \lambda_{\text{thresh}}$), or a timeout expires (SLO safety).

3. **Optimal packing**: Upon trigger, extract requests from Q, apply DP-based packing to partition them into mini-batches respecting $(T_{\max}, B_{\max})$, then dispatch each mini-batch to the engine.

4. **AIMD feedback**: Observe TTFT metrics from completed prefills, update S_t, and adjust N_t accordingly.

Crucially, this design requires no modifications to kernel implementations or memory management—only a lightweight queue layer before the existing scheduler. Decode remains continuous and unchanged. Importantly, the staging queue only affects the prefill admission path. Once requests enter decoding, they fully participate in the engine's native continuous batching mechanism. This separation preserves the low-latency benefits of continuous decoding while enabling controlled batching exclusively where packing is beneficial. Figure 3 provides an example showing why delaying admission can reduce end-to-end prefill cost under packing.

3.5 Stability and SLO Guarantees

The AIMD controller inherits stability properties from TCP congestion control literature. Key design elements ensure robust behavior:

- **Hysteresis band** $[\tau_{\text{low}}, \tau_{\text{high}}]$: prevents oscillations from measurement noise by requiring sustained SLO deviations before control actions.

- **Hard bounds** $[N_{\min}, N_{\max}]$: enforce minimum responsiveness ($N_{\min}$) and maximum batching ($N_{\max}$) regardless of feedback.

- **Burst overrides**: queue-depth and arrival-rate guards provide hard SLO protection during traffic spikes, overriding the AIMD state when necessary.

- **Cold-start initialization**: $N_0 = N_{\min}$ ensures low initial TTFT before sufficient telemetry accumulates.

- **Utilization gating**: prevents pathological over-batching when GPUs are already saturated, ensuring the controller adapts to system capacity.

These mechanisms collectively yield a controller that adapts to diurnal traffic patterns while maintaining p95 TTFT SLOs across diverse workload regimes. Together, these mechanisms ensure that the controller converges to a stable operating region under stationary workloads, while retaining the ability to react sharply to transient bursts. In effect, the system achieves both long-term efficiency and short-term SLO protection.

4 Evaluation

In this section, we evaluate our method in comparison with Prepacking [5] using real-world user requests, demonstrating that it achieves significant improvement

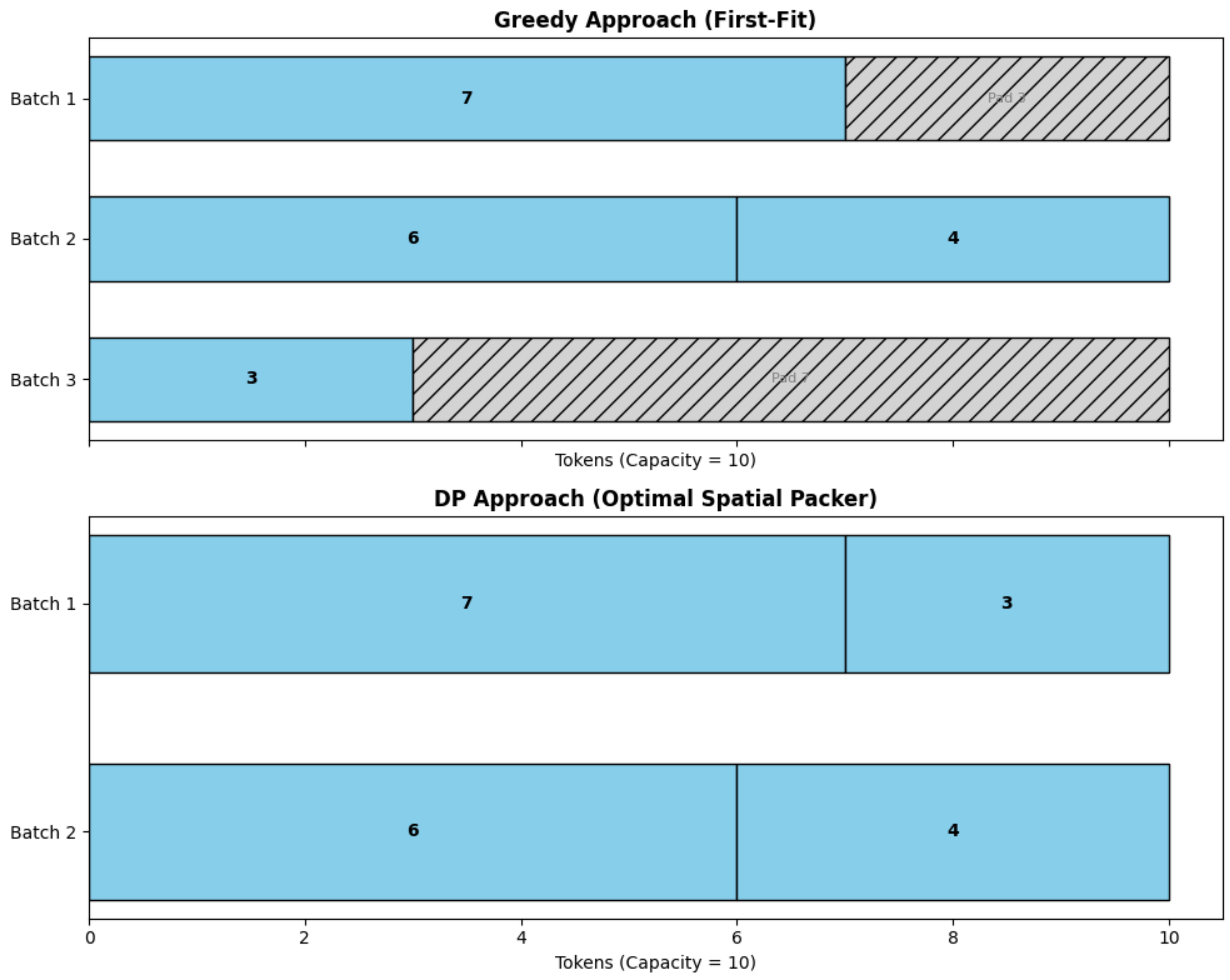


Figure 2: Illustration of an example where DP performs better than greedy first-fit approach.

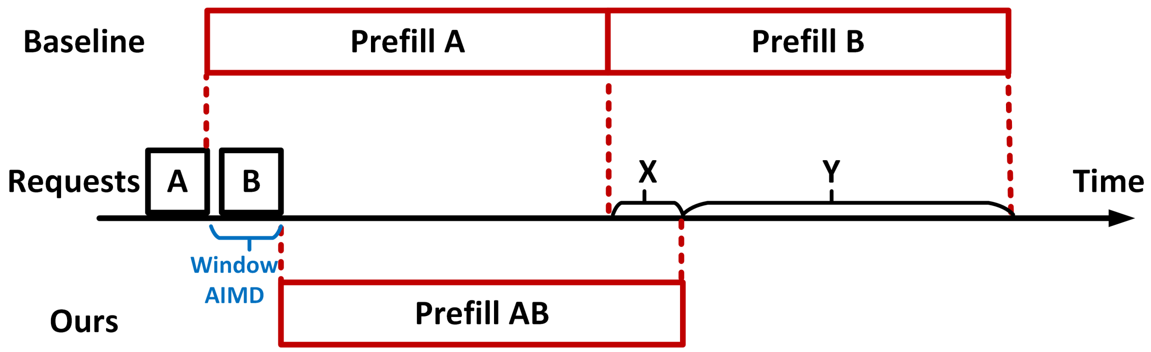


Figure 3: Illustrative example comparing baseline immediate prefilling and our controlled micro-batching. Request A incurs extra waiting time X, but request B avoids a separate prefill and gains a larger reduction Y when executed jointly.

on serving efficiency and TTFT. We also show the impact of different packing algorithms within the system on performance.

4.1 Methodology

Hardware. We conduct evaluation on the *Delta* supercomputer [11]. The 4-Way A40 GPU node in *Delta* is

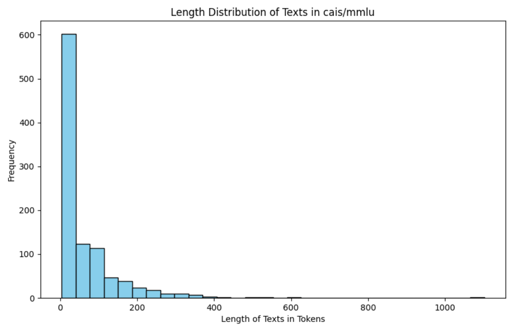


Figure 4: Synthetic User Request Length Histogram used in Prepacking [5]

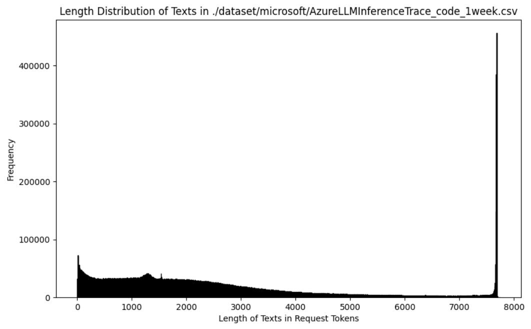


Figure 5: Real User Request Length Histogram (Code)

equipped with 4× NVIDIA A40 GPU with 48 GB DDR6 ECC RAM. We use 1 node for the experiments.

Evaluation Setup. We implement our method on top of the codebase of Prepacking. We include the implementation and environment details in the submitted zip file. If not specified, we use the Sheared-LLaMA-1.3B model [12] for inference, with a maximum tokens of 20K.

4.2 Dataset

We use the real-world user requests pattern published in DynamoLLM [13] as the dataset for the evaluation. The dataset contains user requests of coding and conversation workloads, in a time span of 7 days, with the *timestamp*, *reqeust token length* for each requests. We randomly select 5 distinct time windows and replay them at a scale factor of 4 to simulate real-world serving scenarios in the evaluation.

Compared to the synthetic and static dataset of Prepacking, the length distribution of the requests is hugely different, for both coding and conversation workloads, as depicted in Figs. 4 – 8. In the temporal dimension, which Prepacking did not take into consideration, both workloads exhibit strong weekday diurnal cycles, with observable peaks and troughs, and both decline on weekends.

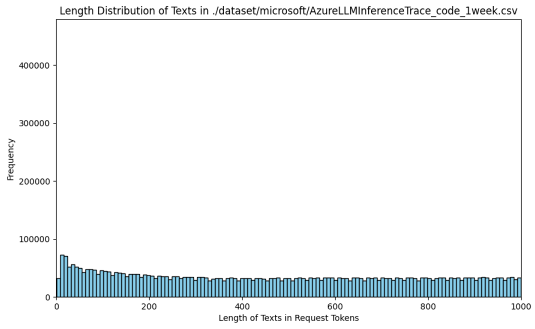


Figure 6: Real User Request Length Histogram (Code) (closer look)

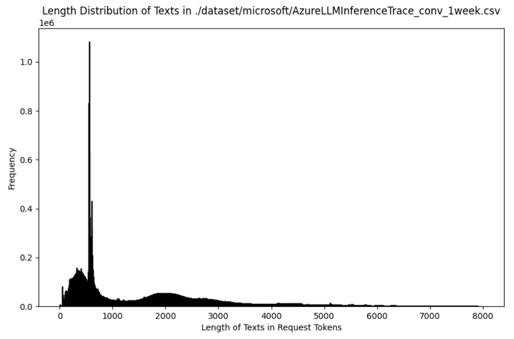


Figure 7: Real User Request Length Histogram (Conversation)

4.3 Main Results

We demonstrate that our method improves the end-to-end TTFT for real-world user requests effectively. The end-to-end TTFT is defined as the time between when the request comes in and the first token output, which includes the waiting time and prefill time. We compare our method against padding and Prepacking as baselines. For padding, the requests are prefill-ed as-is, without additional pre-processing. For Prepacking, the bin-pack algorithm is applied to the requests for pre-processing, but no additional control is involved.

Table 1: End-to-end TTFT results for serving real-world user requests with padding, Prepacking, our method using greedy algorithm for packing, and our method using DP algorithm for packing.

Workload	Coding (s)						Conversation (s)					
Time Window	1	2	3	4	5	Mean	1	2	3	4	5	Mean
Padding	25.42	1.10	0.19	0.73	137.92	33.07	50.39	49.58	45.99	49.87	52.57	49.68
Prepacking	4.59	0.88	0.32	0.97	81.57	17.66	34.89	28.82	28.56	30.63	28.42	30.26
Ours (Greedy)	3.94	0.77	0.27	0.83	75.45	16.25	28.03	24.19	25.82	27.77	22.70	25.70
Ours (DP)	3.99	0.86	0.27	0.84	77.19	16.63	29.04	25.36	27.53	28.30	24.52	26.95

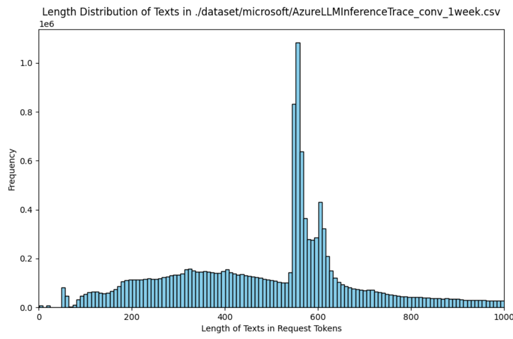


Figure 8: Real User Request Length Histogram (Conversation) (closer look)

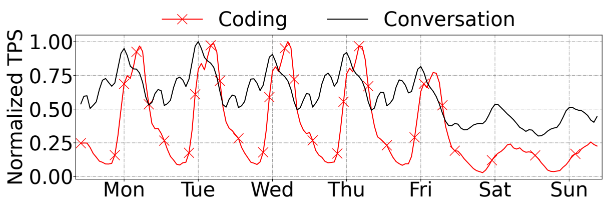


Figure 9: Load over a week for Coding and Conversation LLM inference workloads (from [13])

End-to-end TTFT improvement. As shown in Table 1, for both workloads, we can observe significant improvements for our method over Prepacking across the five randomly chose time windows. For the conversation workload, our method archives up to 51% lower TTFT compared to padding, and up to 20% compared to Prepacking. For the coding workload, the improvement is smaller, but still reaches up to a 14% reduction. We also observe that within certain time window of the coding workload, applying packing results in worse TTFT. We also observe that in some coding time windows, packing can increase TTFT. We attribute this to the fact that the coding workload is much lower during trough periods than the conversation workload, leaving less opportunity for packing to help and making any packing overhead more noticeable.

Impact of different packing algorithms. For both coding and conversation workloads, as depicted in Table 1, the DP algorithm for packing results in worse performance. We believe this is because, for real-world user requests, the marginal improvement over a strong greedy baseline is small, while DP incurs additional complexity. Despite this, our method with DP as the packing algorithm still offers better TTFT compared to Prepacking.

5 Conclusion

We presented a workload-aware prefill optimizer that addresses both temporal and spatial inefficiencies in LLM serving systems. Our approach makes three key contributions:

1. **Dynamic temporal control**: An AIMD-based batching controller that adaptively adjusts prefill triggering thresholds based on real-time TTFT feedback, enabling automatic navigation of the throughput-latency trade-off under non-stationary workloads.

2. **Optimal spatial packing**: A dynamic programming algorithm that partitions buffered requests into mini-batches to maximize token utilization and minimize padding waste, achieving provably optimal packing given complete length information.

3. **Real-world validation**: Evaluation on production-trace datasets from DynamoLLM demonstrates up to 20% end-to-end TTFT improvements over static baselines across diverse coding and conversation workloads with diurnal traffic patterns.

By jointly optimizing temporal batching and spatial arrangement without modifying inference kernels, our system achieves balanced performance across varying load conditions while maintaining SLO compliance.

Team Member Contribution

- *leih5*: Contribute to date: attended all group meetings and contributed to the proposal and midterm, made a decision to change research scope, reproduced the experiments in Prepacking, found and validated the dataset this paper would rely on. Planned contribution toward final report and presentation: find computing resource, reproduce baselines and do some experiments.

- *wangjia2*: Actively participated in all group meetings and contributed to all written materials; set up and maintained the project codebase; took primary responsibility for designing and implementing the temporal controller and helped with the spatial packing; and created the illustrative figures used in the presentation.

- *yuchen87*: Actively participated in all group meetings; contributed to the drafting of the Related Work section and polishing of the Design section; assisted with revisions across other sections of the proposal and midterm report; participated in the implementation of parts of the proposed algorithm and contributed to the writing of the final report.

- *linshuy2*: Actively participated in all group meetings and contributed to the literature survey related to sequence packing, reproducing the results of Prepacking, preparing the final presentation materials, and the writing of the final report.

- *npunati2*: Actively participated in all group meetings and contributed to proposal; Planned contribution toward final report and presentation: responsible for writing DP portion of the report.

- *bhuang4*: Actively participated in all group meetings and contributed to proposal; contributed to the slides of the final presentation and gave the presentation as the presentor.

References

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

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

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

[4] NVIDIA Corporation. Nvidia tensorrt-llm: A tensorrt toolbox for optimized large language model inference. `https://github.com/NVIDIA/TensorRT-LLM`, 2023.

[5] Siyan Zhao, Daniel Israel, Guy Van den Broeck, and Aditya Grover. Prepacking: A simple method for fast prefilling and increased throughput in large language models. *arXiv preprint arXiv:2404.09529*, 2024.

[6] Zongxian Yang, Jiayu Qian, Zegao Peng, Haoyu Zhang, and Zhi-An Huang. Med-refl: Medical reasoning enhancement via self-corrected fine-grained reflection. *arXiv preprint arXiv:2506.13793*, 2025.

[7] HuggingFace. Enhancing training efficiency using packing with flash attention. HuggingFace Blog, 2024.

[8] HuggingFace. Transformers pull request #31629: Add support for packing with flash attention. GitHub Pull Request, 2024.

[9] Achintya Kundu, Rhui Dih Lee, Laura Wynter, Raghu Kiran Ganti, and Mayank Mishra. Enhancing training efficiency using packing with flash attention, 2024.

[10] HuggingFace. Efficient multimodal data pipeline. HuggingFace Blog, 2024.

[11] William Gropp, Tim Boerner, Brett Bode, and Greg Bauer. Delta: Balancing gpu performance with advanced system interfaces. 2023.

[12] Mengzhou Xia, Tianyu Gao, Zhiyuan Zeng, and Danqi Chen. Sheared llama: Accelerating language model pre-training via structured pruning, 2024.

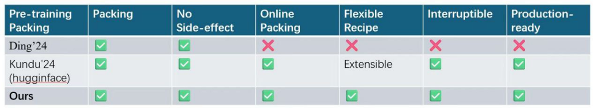
[13] Jovan Stojkovic, Chaojie Zhang, Íñigo Goiri, Josep Torrellas, and Esha Choukse. Dynamollm: Designing llm inference clusters for performance and energy efficiency. In *2025 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, page 1348–1362. IEEE, March 2025.

6 Appendix

6.1 Reasons behind Changing the Scope

We have found a recent work has contributed what we intended to do, [9] and they have been merged to Huggingface Transformers 4.44.

Since the same contributions have been made, we change the scope from training time packing to inference time packing.



Pre-training Packing	Packing	No Side-effect	Online Packing	Flexible Recipe	Interruptible	Production-ready
Ding'24	✅	✅	❌	❌	❌	❌
Kundu'24 (hugginface)	✅	✅	✅	Extensible	✅	✅
Ours	✅	✅	✅	✅	✅	✅

Figure 10: Survey Regarding Packing in Training Time