TetriServe: Efficiently Serving Mixed DiT Workloads

Runyu Lu*
runyulu@umich.edu
University of Michigan
Ann Arbor, Michigan, USA

Shiqi He*
shiqihe@umich.edu
University of Michigan
Ann Arbor, Michigan, USA

Wenxuan Tan
wtan45@wisc.edu
University of Wisconsin-Madison
Madison, Wisconsin, USA

Shenggui Li
shenggui001@e.ntu.edu.sg
Nanyang Technological University
Singapore, Singapore

Ruofan Wu
ruofanw@umich.edu
University of Michigan
Ann Arbor, Michigan, USA

Jeff J. Ma
jeffjma@umich.edu
University of Michigan
Ann Arbor, Michigan, USA

Ang Chen
chenang@umich.edu
University of Michigan
Ann Arbor, Michigan, USA

Mosharaf Chowdhury
mosharaf@umich.edu
University of Michigan
Ann Arbor, Michigan, USA

Abstract

Diffusion Transformer (DiT) models excel at generating high-quality images through iterative denoising steps, but serving them under strict Service Level Objectives (SLOs) is challenging due to their high computational cost, particularly at larger resolutions. Existing serving systems use fixed-degree sequence parallelism, which is inefficient for heterogeneous workloads with mixed resolutions and deadlines, leading to poor GPU utilization and low SLO attainment.

In this paper, we propose step-level sequence parallelism to dynamically adjust the degree of parallelism of individual requests according to their deadlines. We present TetriServe, a DiT serving system that implements this strategy for highly efficient image generation. Specifically, TetriServe introduces a novel round-based scheduling mechanism that improves SLO attainment by (1) discretizing time into fixed rounds to make deadline-aware scheduling tractable, (2) adapting parallelism at the step level and minimizing GPU hour consumption, and (3) jointly packing requests to minimize late completions. Extensive evaluation on state-of-the-art DiT models shows that TetriServe achieves up to 32% higher SLO attainment compared to existing solutions without degrading image quality.

CCS Concepts: • **Computer systems organization** → **Cloud computing**.

*Both authors contributed equally to this research.

This work is licensed under a Creative Commons Attribution 4.0 International License.

ASPLOS '26, Pittsburgh, PA, USA
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2359-9/2026/03
https://doi.org/10.1145/3779212.3790233

Keywords: diffusion transformer serving, gpu resource scheduling, sequence parallelism

ACM Reference Format:
Runyu Lu, Shiqi He, Wenxuan Tan, Shenggui Li, Ruofan Wu, Jeff J. Ma, Ang Chen, and Mosharaf Chowdhury. 2026. TetriServe: Efficiently Serving Mixed DiT Workloads. In *Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '26), March 22–26, 2026, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3779212.3790233

1 Introduction

Diffusion models [3, 4, 16, 21, 34, 37, 38] have significantly advanced text-to-image and text-to-video generation, enabling photorealistic content from natural language descriptions. They now power a wide range of commercial and creative services such as OpenAI Sora [7] and Adobe Firefly [2]. At the core of these breakthroughs are *Diffusion Transformers (DiTs)* [34], which have become the backbone of leading models including Stable Diffusion 3 (SD3) [3] and FLUX.1-dev [21]. By replacing conventional UNet architectures [16, 36], DiTs achieve higher fidelity by iteratively refining a full-image latent representation over a sequence of discrete denoising steps, setting a new standard for generation quality.

As DiT models move into production, *online DiT serving* becomes a key systems challenge. Deployments such as Flux AI [13] must satisfy strict service level objectives (SLOs) in the form of a *deadline* for each request while sharing a fixed GPU pool across many users to minimize cost. Serving is particularly challenging because requests arrive with heterogeneous output resolutions and tight deadlines.

Despite advances in LLM serving [10, 20, 27–29, 33, 43, 47], these solutions are insufficient: DiTs have fundamentally different serving characteristics. Specifically, DiT inference differs from LLMs in three ways: (i) it is stateless, requiring no KV cache; (ii) it is compute-bound, as multiple denoising steps operate on the full set of latent image tokens; and (iii)

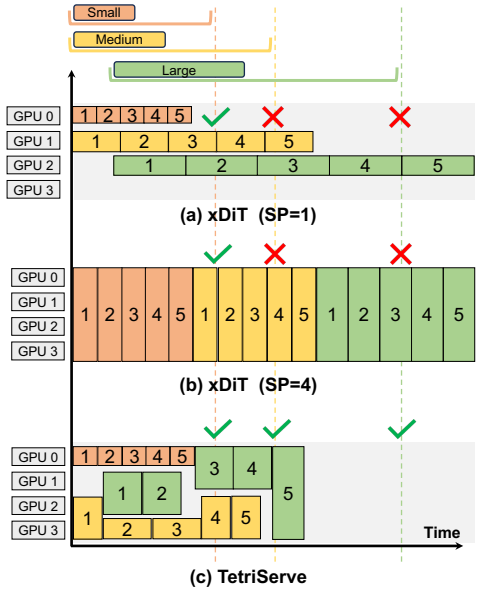


Figure 1. Three DiT serving requests—each with 5 denoising steps—arrive over time with different SLOs and output resolutions. DiT serving solutions using static parallelism cannot adapt and fail to meet multiple SLOs. TetriServe meets more SLOs via SLO-aware scheduling and packing.

model sizes are small enough to fit on a single GPU. Consequently, generating a high-resolution 2048×2048 image on a single H100 GPU can take up to a minute, while a 4096×4096 image may exceed ten minutes. To meet the stringent latency demands of online serving, parallelism is essential.

The most common approach for parallelizing DiTs is *sequence parallelism (SP)* [18, 25], which partitions the sequence of image tokens across GPUs. However, simply applying a fixed degree of SP to all requests is inefficient and leads to poor SLO attainment. This is because the optimal degree of parallelism is highly sensitive to the input image resolution; a configuration that is ideal for one resolution can be detrimental to another. As shown in the toy example in Figure 1, the fixed-degree SP approach creates a fundamental tradeoff: low degrees of parallelism (e.g., SP=1 or 2) are efficient for small inputs but underutilize the GPU cluster for large ones by leaving some GPUs idle and prolonging request runtime, while high degrees of parallelism (e.g., SP=4 or 8) accelerate large inputs but introduce excessive communication overhead for small ones, leading to head-of-line blocking. Compounding this issue, existing DiT inference engines [12] are non-preemptive: once a request begins execution with a fixed degree of parallelism, it holds its allocated GPU(s) until completion, preventing more optimal scheduling of other requests in the queue.

We observe that *step-level scheduling*, in which the degree of parallelism is adjusted across steps within each request based on its resolution and deadline, can significantly improve the serving efficiency of mixed DiT workloads. High-resolution or urgent requests can be accelerated with more GPUs, while smaller or less urgent ones conserve resources. Unfortunately, we prove that finding a globally optimal step-level schedule that maximizes deadline satisfaction under a fixed GPU budget is NP-hard (§4.1). In addition, the online arrival of requests and the need for millisecond-level scheduling decisions make exhaustive optimization infeasible.

We present *TetriServe*, a step-level DiT serving system designed to maximize SLO attainment under deadline constraints. At its core, TetriServe introduces a *deadline-aware round-based scheduler* that transforms the continuous time in the serving problem into a sequence of tractable, fixed-duration rounds. In each round, the scheduler decides which requests to serve and at what GPU parallelism degree. To make these decisions, TetriServe leverages a cost model that profiles per-step latency as a function of GPU count and identifies the *minimal feasible GPU allocation* for each request that can still meet its deadline. This allows TetriServe to construct a set of candidate allocations and perform request packing with the explicit goal of minimizing the number of requests that would otherwise become late in the next round.

TetriServe further enhances GPU efficiency while preserving request deadlines. It uses *selective continuous batching* to merge steps across small-resolution requests, reducing kernel launch overhead and boosting throughput. Meanwhile, *GPU placement preservation* and *work-conserving elastic scale-up* ensure idle GPUs are utilized without remapping distributed jobs. Together with the round-based scheduler, these techniques allow TetriServe to handle diverse DiT workloads—from small to large resolutions—while substantially improving deadline satisfaction over fixed-degree baselines.

We evaluate TetriServe on popular open-source DiT models (FLUX.1-dev and SD3) and different hardware platforms (8×H100 and 4×A40 nodes). We show that TetriServe consistently outperforms xDiT [12]—a DiT-serving engine that allows different fixed SP configurations—across diverse experimental settings by up to 32% in terms of SLO attainment ratio. TetriServe is also robust to bursty request arrival patterns, diverse workload mixes, and different model–hardware combinations.

We summarize the contributions as follows:

- We cast DiT serving as a step-level GPU scheduling problem and prove its NP-hardness.
- We present TetriServe, a deadline-aware round-based scheduler that minimizes late completions via dynamic programming.
- We show that TetriServe achieves substantial gains in SLO attainment over fixed-degree baselines on state-of-the-art DiT models while maintaining image quality.

2 Motivation

Serving DiT models has become a popular workload for modern image generation systems [2, 12]. DiT inference is both compute-intensive and latency-sensitive. To better understand the challenges of serving such workloads, in this section, we discuss DiT background, workload characteristics, and the resulting opportunities and challenges.

2.1 DiT Background

Diffusion models [7, 16, 34, 37, 38] have significantly advanced text-to-image and text-to-video generation, enabling photorealistic content from natural language descriptions. Each step operates on the full latent representation, removing noise based on a learned denoising function. Although early diffusion models used *UNet* architectures [16, 36], modern high-quality image generators use *Diffusion Transformers (DiTs)* [9, 34] as their backbone. DiTs use attention [41] to capture global context and long-range dependencies.

DiT vs. LLM Parallelism. Although both DiTs and LLMs are built upon the Transformer architecture, their inference characteristics diverge significantly, requiring different parallelism strategies. Traditional model-sharding strategies for LLMs, such as tensor and pipeline parallelism, are inefficient for DiTs. This is because DiT models are typically small enough to fit on a single GPU. For example, the largest open-source text-to-image DiT has only 12B parameters [21] and fits comfortably on a single 80GB H100 GPU. Consequently, applying model sharding introduces unnecessary communication overhead without the benefit of accommodating a larger model, resulting in poor hardware utilization.

DiTs adopt *sequence parallelism (SP)* [18, 23, 25], a more efficient parallel approach tailored to their compute-bound nature. In SP, token sequences (image tokens) are distributed across GPUs, enabling collaborative computation within each transformer layer. Two representative implementations are *Ulysses attention* [18], which uses all-to-all collectives to transpose tokens and heads across GPUs before local attention, and *Ring attention* [25], which arranges GPUs in a ring and passes partial Q, K, V slices peer-to-peer, overlapping communication with computation. In practice, Ulysses attention is often preferred on systems with high-bandwidth interconnects like NVLink, as its use of collective primitives can be more efficient [12].

2.2 Characteristics of DiT Workloads

DiT serving exhibits distinctive workload characteristics that affect the design of scheduling and resource management.

Heterogeneous Inputs. Unlike LLM workloads, where input text can vary widely in length, DiT serving workloads are characterized by a small, discrete set of possible input image resolutions [13, 39]. In this work, we focus on four representative resolutions common in production environments; their characteristics for the FLUX.1-dev model [21] are detailed in Table 1. Despite the small number of distinct input sizes, the substantial differences in their computational demands still lead to highly heterogeneous resource requirements across requests.

Table 1. Characteristics of representative input sizes for the FLUX.1-dev model [21], including latent tokens and computational cost (TFLOPS). Execution stability (CV) is measured over 20 steps on 8xH100 GPU for different sequence parallelism (SP) degrees.

Image Size	Tokens	TFLOPs	SP=1	SP=2	SP=4	SP=8
256 × 256	256	556.48	0.13%	0.31%	0.67%	0.62%
512 × 512	1024	1388.24	0.06%	0.15%	0.14%	0.53%
1024 × 1024	4096	5045.92	0.07%	0.12%	0.04%	0.09%
2048 × 2048	16384	24964.72	0.05%	0.11%	0.14%	0.28%

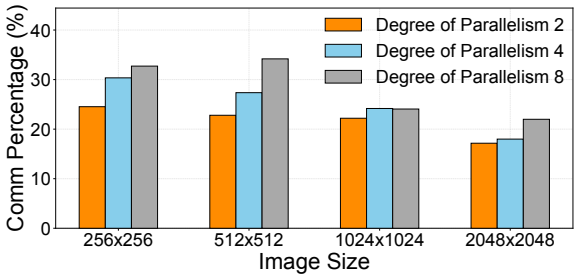
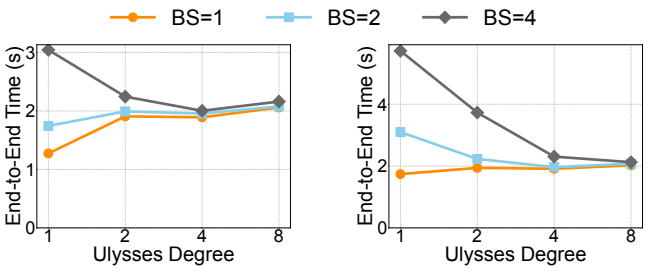


Figure 2. Percentage of time spent in communication for FLUX.1-dev for four resolutions on an 8×H100 server (Batch Size = 4). Larger resolutions benefit more from increased parallelism because of relatively less communication overhead.

Predictable Execution. Despite input diversity, DiT inference remains compute-bound and therefore exhibits stable per-step runtimes across a wide range of input resolutions. As shown in Table 1, execution time is highly stable: profiling over 100 runs with varying sequence-parallel degrees yields a coefficient of variation (CV) below 0.7% in all cases. This low variability indicates that DiT model inference is predictable across resolutions and degrees of parallelism, enabling accurate performance modeling and effective deadline-aware scheduling.

Insight 1: *DiT workloads consist of heterogeneous input requests with different output resolutions, but per-step runtime for each resolution is highly predictable.*

Scaling Efficiency of Sequence Parallelism. Sequence parallelism distributes tokens across GPUs, but its scaling efficiency is sublinear to the degree of parallelism. Two factors drive this: (i) communication overhead from collectives (all-to-all or ring exchanges) that scales with the degree of parallelism and sequence length; and (ii) reduced per-GPU



(a) Image Size: 256×256 **(b)** Image Size: 512×512

(c) Image Size: 1024×1024 **(d)** Image Size: 2048×2048

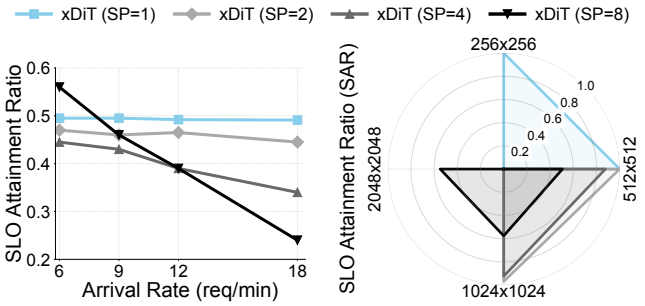
Figure 3. End-to-end scaling efficiency of FLUX.1-dev for four resolutions on an 8×H100 server for different batch size (BS). Efficiency scales sublinearly. Larger resolutions benefit more from increased parallelism, while smaller resolutions exhibit limited scalability. Note different Y-axes scales.

kernel efficiency when workloads are split, lowering occupancy and cache locality. Figure 2 quantifies this by showing the communication percentage across image sizes and degrees of parallelism. For small inputs (e.g., 256 × 256 and 512 × 512), increasing the degree of parallelism rapidly increases the communication percentage, exceeding 30% at higher degrees. In this case, communication dominates execution time, leading to poor scaling and decreasing the benefits from additional GPUs. Figure 3 shows that small inputs (e.g., 256×256, 512×512) underutilize GPUs and scale poorly, while larger inputs (e.g., 1024 × 1024, 2048 × 2048) improve efficiency though computation remains the bottleneck. This explains why in Figure 1, latency does not scale linearly with the number of GPUs.

Insight 2: *Sequence parallelism in DiT workloads scales sublinearly with the degree of parallelism and differently for each input resolution.*

2.3 Challenges and Opportunities

Limitations of Current Solutions. Conventional serving strategies using a fixed degree of parallelism are ill-suited for the heterogeneous nature of DiT workloads, a limitation illustrated in the toy example in Figure 1. With data parallelism (xDiT, SP=1), the small request meets its deadline, but the larger requests fail due to insufficient processing speed. Conversely, a high fixed degree of parallelism (xDiT, SP=4)



(a) Fixed Degree Performance **(b)** Breakdown by Resolution

Figure 4. Performance of fixed degree xDiT variants under the Uniform workload. (a) The overall SLO Attainment Ratio (SAR) is low for all fixed strategies. (b) The spider plot, shown for a representative arrival rate of 12 req/min, reveals the underlying reason: low SP degrees fail on large resolutions, while high SP degrees perform poorly on small ones.

handles the large request well, but it still misses the deadline (along with the medium one) due to head-of-line blocking and inefficient resource use of the small request.

Experimental results confirm this trade-off. As shown in Figure 4a, under a Uniform workload with a tight SLO Scale of 1.0×, no fixed-parallelism strategy achieves an SLO Attainment Ratio (SAR) above 0.6. The spider plot in Figure 4b reveals why: each fixed strategy only works well for specific resolutions. *SP=1* and *SP=2* achieve near-perfect SAR for 256 × 256 images but fail completely for 2048 × 2048, while *SP=4* and *SP=8* handle 2048 × 2048 effectively but perform poorly on smaller resolutions due to scaling inefficiency and head-of-line blocking. No single parallelism degree works across the board.

Optimization Opportunities. The limitations of fixed parallelism highlight a key opportunity: moving to dynamic, *step-level sequence parallelism*. As shown in Figure 1(c), our approach, TetriServe, meets all three deadlines by adapting the degree of parallelism for each request at the step level. It assigns fewer GPUs to the initial steps of the medium request, freeing up resources, and then scales up to meet the deadline, thus avoiding the rigid trade-offs of fixed strategies.

This flexibility to adjust the sequence parallelism degree *per step* allows a scheduler to allocate more GPUs when deadlines are tight and fewer when they are not, freeing capacity for other requests. By exploiting DiTs' predictable step execution times and heterogeneous scaling behavior, this approach enables finer-grained resource shaping and better SLO attainment than conventional fixed-SP policies.

Insight 3: *Step-level parallelism adapts GPU allocation to request deadlines, avoiding the resource waste of fixed parallelism and improving SLO attainment.*

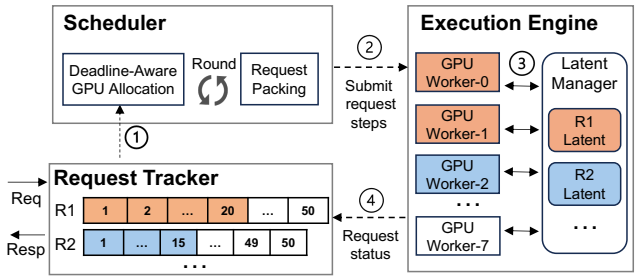


Figure 5. TetriServe architecture and request lifecycle.

3 TetriServe Overview

TetriServe allows more DiT serving requests with heterogeneous output resolutions to meet their SLOs by judiciously scheduling and packing them on shared GPU resources. In this section, we provide an overview of how TetriServe fits in the DiT serving lifecycle to help the reader follow the subsequent sections.

System Components. TetriServe is designed around a scheduler that makes deadline-aware GPU allocation decisions in a round-based manner. Its key components are:

- **Request Tracker:** Maintains metadata on active requests, including resolutions, deadlines, and execution states (e.g., remaining steps).
- **Scheduler:** The core component consists of *deadline-aware GPU allocation* and *round-based request packing*. At every round, it minimizes individual requests' GPU consumption while maximizing SLO attainment.
- **Execution Engine:** A distributed pool of GPU workers that execute assigned diffusion steps in parallel.
- **Latent Manager:** Handles intermediate latent representations across steps, reducing redundant computation and memory overhead.

Together, these components enable TetriServe to adapt resource allocation at millisecond scale, sustaining high throughput and SLO attainment for heterogeneous DiT workloads.

Request Lifecycle. When a request arrives, the *Request Tracker* records its resolution, state, and deadline. The *Scheduler* then places it into the next scheduling round ①, where a deadline-aware policy determines GPU allocations in terms of step numbers for each request for one round. For example, in Figure 5, it selects Request 1 to run 20 steps on 2 GPUs (orange) and Request 2 to run 15 steps on 1 GPU (blue) for the scheduling round. Different requests are dispatched to GPU workers in the *Execution Engine* ②, which compute diffusion steps and produce intermediate latents managed by the *Latent Manager* ③. Upon completion, workers notify the request tracker to update dependent steps ④. After all steps finish, the final output is returned to the user.

4 Deadline-Aware Round-Based Scheduler

TetriServe introduces a deadline-aware scheduler designed to optimize SLO attainment for DiT serving. We begin with a formal definition of the GPU scheduling problem in the offline scenario and prove that it is NP-hard. We then propose a *round-based scheduling mechanism*, which maximizes goodput via minimizing GPU-hour consumption for each request. Later we proposes enhancements so that TetriServe balances utilization, latency, and scalability in DiT serving.

4.1 Problem Statement

Given a collection of GPUs and requests, the DiT serving objective for each invocation of the scheduler is the following: *Find a step-level schedule that maximizes the number of requests meeting their deadlines given a fixed number of GPUs.*

Problem Formulation. Consider an N-GPU cluster and R outstanding requests. Each request req_i consists of a sequence of S_i dependent diffusion steps $\{s_{i1}, s_{i2}, \ldots, s_{iS_i}\}$. Each step s_{ij} can be executed using $k \in \{1, 2, 4, \ldots, N\}$ GPUs, where k is a power of two. The execution time of a step, denoted $T_{ij}(k)$, is a function of k. The completion time of a request is defined as:

$$C_i = \sum_{j=1}^{S_i} \left[Q_{ij} + T_{ij}(A_{ij}) \right],$$

where Q_{ij} is the queuing delay before step s_{ij} begins and A_{ij} is the number of GPUs allocated. Then we can formulate the DiT serving objective as:

$$\text{Maximize} \quad \sum_{i=1}^{R} I_i, \quad \text{where } I_i = \begin{cases} 1 & \text{if } C_i \le D_i, \\ 0 & \text{otherwise.} \end{cases}$$

This formulation is subject to the following conditions:

1. **Step Dependency**: A step s_{ij} can start only after the previous step completes:

   $$\text{Start}(s_{ij}) \ge \text{Completion}(s_{i(j-1)}). \quad \forall i, \forall j > 1$$

   Therefore, at most one step of a request can be executed at any time.
2. **GPU Capacity**: At any time, the total number of GPUs allocated across all steps cannot exceed N:

   $$\sum_{i=1}^{R} \sum_{j=1}^{S_i} A_{ij}(t) \le N, \quad \forall t$$

   where $A_{ij}(t)$ denotes the GPUs allocated to step s_{ij} if it is running at time t, and zero otherwise.

The goal is to find a set of GPU assignments $\{A_{ij}\}$ that maximizes the number of requests meeting their deadlines.

NP-hardness. To highlight the computational complexity, we consider the special case where each request has a single non-preemptive step ($S_i = 1$). Time is discretized into slots $\mathcal{T} = \{0, 1, \ldots, T_{\max} - 1\}$. Let $\mathcal{K} = \{1, 2, 4, \ldots, N\}$ denote the

Table 2. Notations used in the GPU Scheduling Problem.

Symbol	Description
N	Total number of GPUs.
R	Number of requests.
S_i	Number of steps in request req_i.
D_i	Deadline of request req_i.
$T_{ij}(k)$	Execution time of step s_{ij} with k GPUs.
Q_{ij}	Queueing delay before step s_{ij} starts.
A_{ij}	GPU allocation for step s_{ij}.
C_i	Completion time of request req_i.

allowed GPU allocations. For each request i, start time $t \in \mathcal{T}$, and GPU count $k \in \mathcal{K}$, introduce a binary decision variable:

$$x_{i,t,k} = \begin{cases} 1 & \text{if request } i \text{ starts at time } t \text{ with } k \text{ GPUs,} \\ 0 & \text{otherwise.} \end{cases}$$

Objective. Maximize the number of requests completing by their deadlines:

$$\max \sum_i \sum_{t \in \mathcal{T}} \sum_{k \in \mathcal{K}} x_{i,t,k}.$$

Constraints.

$$\sum_{t \in \mathcal{T}} \sum_{k \in \mathcal{K}} x_{i,t,k} \le 1, \qquad \forall i, \tag{1}$$

$$arrival\_time(i) \le t, \qquad \forall i, \tag{2}$$

$$t + T_i(k) \le D_i, \qquad \forall i, t, k, \tag{3}$$

$$\sum_i \sum_{k \in \mathcal{K}} \sum_{u \in [t,\, t+T_i(k)-1]} k \cdot x_{i,t,k} \le N, \qquad \forall t, u \in \mathcal{T}, \tag{4}$$

$$x_{i,t,k} \in \{0, 1\}, \qquad \forall i, t, k. \tag{5}$$

Constraint (1) ensures each request starts at most once. Constraints (2) and (3) enforce arrival times and deadline feasibility. Constraint (4) enforces that at any time slot u, the sum of GPUs assigned to running requests does not exceed system capacity N. Constraint (5) enforces integrality.

This Zero-one Integer Linear Program (ZILP) exactly captures the offline DiT serving problem in the single-step case, where $I_i = \sum_{t \in \mathcal{T}} \sum_{k \in \mathcal{K}} x_{i,t,k}$. We show in Appendix A that solving such formulations is NP-hard [5, 14, 19, 32, 45]. Therefore, **multi-step DiT serving is NP-hard** as well.

4.2 Round-Based Scheduling

Step-level scheduling for DiT serving is NP-hard, making global optimization expensive. To enable practical scheduling, TetriServe adopts a round-based heuristic: instead of scheduling steps arbitrarily in a continuous global timeline, *we discretize execution into rounds*, where each round corresponds to a fixed-length GPU execution window. This allows us to *(i) limit the scheduling search space* and *(ii) enable efficient preemption between rounds*. Within each round, TetriServe determines the minimal required GPU allocation for requests and dynamically packs these requests to maximize SLO attainment ratio.

4.2.1 Deadline-Aware GPU Allocation. Exhaustively enumerating GPU allocations for each step is infeasible, and over-allocation wastes resources due to scaling inefficiencies in DiT models (e.g., kernel launch and communication overheads). While more GPUs reduce latency, they increase total GPU hours. To balance these trade-offs, TetriServe identifies the minimal GPU allocation needed for each request to meet its deadline at the beginning of each round. Since required allocation depends mainly on resolution and deadline, this approach avoids exploring the full allocation space.

For a step s_{ij}, the execution time $T_{ij}(k)$ is a function of the number of GPUs k. The GPU hour for executing step s_{ij} with k GPUs is $k \times T_{ij}(k)$. The goal is to minimize the total GPU hour for each request:

$$\min_{\{A_{ij}\}} \sum_{j=1}^{S_i} \big(A_{ij} \times T_{ij}(A_{ij})\big) \quad \text{s.t.} \quad \sum_{j=1}^{S_i} \big(Q_{ij} + T_{ij}(A_{ij})\big) \le D_i$$

where A_{ij} is the GPU allocation for step s_{ij}.

Offline Profiling for Cost Model. To make the optimization tractable, TetriServe profiles execution times offline. For every step type s_{ij} and GPU count $k \in \{1, 2, 4, \ldots, N\}$, we measure the actual execution time $T_{ij}(k)$. From this, we derive the GPU hour $k \times T_{ij}(k)$ and store it in a lookup table. At runtime, TetriServe simply enumerates candidate GPU assignments using these pre-profiled values.

The above process aims to assign each request the minimum number of GPUs required to meet its deadline while minimizing the total GPU hours. Figure 6 illustrates this process with a concrete example: three requests (R1–R3), each with five steps, arrive over time. R1 has a small resolution (e.g., 256) and is fixed at SP=1 since higher parallelism would reduce efficiency (see Figure 3). For R2 and R3, TetriServe identifies GPU allocations with two parallelism degrees that just meet their deadlines while minimizing overall GPU usage. The GPU allocations produced by this selection serve as the input to the subsequent request packing stage, where TetriServe schedules requests across GPUs to maximize goodput.

4.2.2 Request Packing. The objective of scheduling is to maximize the number of requests that complete before their deadlines. To make the problem tractable, we approximate it by minimizing the number of requests that become *definitely late*—those that cannot meet their deadlines even under maximal parallelism if not advanced in the current round. Deadline-aware GPU allocation determines the minimal GPU allocations needed for each request to meet its deadline while minimizing GPU hours. This makes it possible to pack more requests into each round and thereby reduce the number that would otherwise be definitely late.

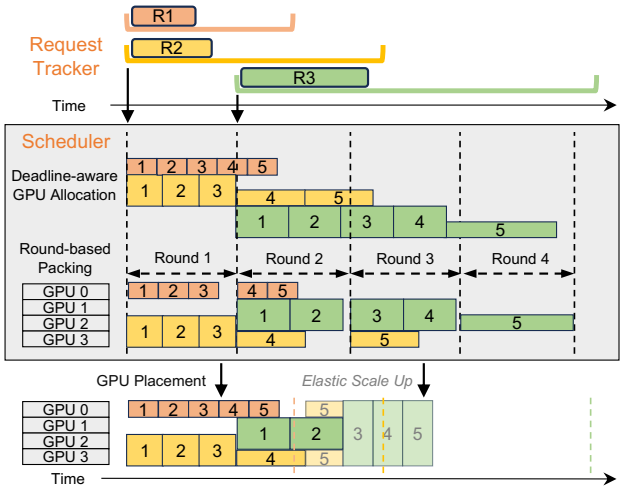


Figure 6. Illustration of TetriServe's scheduling process. The progression is shown from top to bottom: each row represents an intermediate scheduling step, while the final row shows the actual GPU allocation decision. Time is fixed across rows.

After deadline-aware GPU allocation, each request req_i is described by a set of allocations (s_i^m, A_i^m), where s_i^m is the number of steps executed with allocation A_i^m, and per-step times $T_i(A_i^m)$ are obtained from the cost model. To schedule requests across N GPUs, TetriServe divides time into *rounds* of fixed duration τ, which serves as the scheduling granularity. The choice of τ balances overhead and responsiveness: shorter rounds allow finer-grained preemption and more adaptive scheduling, while longer rounds reduce overhead but make scheduling coarser.

At the beginning of each round r (time t_r), the scheduler considers all pending requests and their allocations, and decides which to place within the N GPUs. Within a round of duration τ, if GPU allocation m of request i is chosen, the number of steps that can complete is

$$q_i^m = \min\left\{s_i^m,\ \left\lfloor \frac{\tau}{T_i(A_i^m)} \right\rfloor\right\}.$$

Options with $q_i^m = 0$ are discarded to avoid wasting resources. Choosing option $o \in \{\text{none}, 1, 2, \dots\}$ updates the remaining steps as

$$\tilde{s}_i^m(o) = s_i^m - \mathbb{I}[o = m]\, q_i^m,$$

clipped at zero, where $\mathbb{I}[o = m]$ equals 1 if $o = m$ and 0 otherwise. The next round begins at $t_{r+1} = t_r + \tau$.

To decide which requests must be scheduled *now*, we identify those that would become *definitely late* at t_{r+1} if not advanced in this round. Using the fastest possible step time $T_i^{\min} = \min_{k\in\{1,2,4,\dots,N\}} T_i(k)$, we define the *residual completion time lower bound* under option o as

$$\mathrm{LB}_i(o) = \left(\sum_m \tilde{s}_i^m(o)\right) T_i^{\min},$$

Algorithm 1: DP Round Scheduler

Input : Pending requests R with $\{(s_i^m, A_i^m)\}_{m\in\mathcal{M}_i}$ and $T_i(\cdot)$; capacity N; round length τ; current time t_r
Output : Selected plan

$t_{r+1} \leftarrow t_r + \tau$
foreach $i \in R$ **do**
  foreach $m \in \mathcal{M}_i$ **do**
    $q_i^m \leftarrow \min\{s_i^m, \lfloor \tau / T_i(A_i^m) \rfloor\}$
  $T_i^{\min} \leftarrow \min_{k\in K} T_i(k)$
  $O_i \leftarrow \{\text{none}\} \cup \{\, m \in \mathcal{M}_i \mid q_i^m > 0 \wedge A_i^m \le N \,\}$
  foreach $o \in O_i$ **do**
    foreach $m \in \mathcal{M}_i$ **do**
      $\tilde{s}_i^m(o) \leftarrow s_i^m - \mathbb{I}[o = m] \cdot q_i^m$
    $\mathrm{LB}_i(o) \leftarrow \left(\sum_{m\in\mathcal{M}_i} \tilde{s}_i^m(o)\right) T_i^{\min}$
    $\mathrm{sv}_i(o) \leftarrow \mathbb{I}[\, t_{r+1} + \mathrm{LB}_i(o) \le D_i \,]$
    $w_i(o) \leftarrow 0$ if $o = \text{none}$ else A_i^o
Initialize $\mathrm{dp}[0..N] \leftarrow -\infty$, $\mathrm{dp}[0] \leftarrow 0$
foreach $i \in R$ **do**
  $\mathrm{next}[0..N] \leftarrow \mathrm{dp}$
  for $c = 0$ **to** N **do**
    foreach $o \in O_i$ **do**
      if $w_i(o) \le c$ **then**
        $\mathrm{next}[c] \leftarrow \max\{\mathrm{next}[c],\ \mathrm{dp}[c - w_i(o)] + \mathrm{sv}_i(o)\}$
  $\mathrm{dp} \leftarrow \mathrm{next}$
$c^\star \leftarrow \arg\max_c \mathrm{dp}[c]$
return plan reconstructed from back-pointers at $c^\star$

where $\tilde{s}_i^m(o)$ is the updated step count. A request survives only if

$$t_{r+1} + \mathrm{LB}_i(o) \le D_i.$$

Each option o consumes $w_i(o)$ GPUs: $w_i(\text{none}) = 0$, $w_i(m) = A_i^m$. The per-round scheduling problem is therefore to select at most one option per request, with total GPU $\le N$, maximizing the number of requests that survive to the next round. For requests that have already missed their deadlines, we assign at most one GPU in a best-effort manner without impacting other requests, and scale them up later if idle GPUs become available. By anchoring scheduling decisions on the round duration τ, TetriServe balances preemption overhead and responsiveness, while ensuring urgent requests receive priority.

Dynamic Programming. Naively enumerating all per-request options O_i for feasible packings within a round is exponential in the number of requests and quickly becomes intractable. We observe that the per-round decision has the *group-knapsack* structure: for each request i (a group), we

must choose at most one option o (run one of its GPU allocation this round or none), each option consumes width (GPUs) and yields a binary "survival" value indicating whether the request is *not definitely late* at the next round start. This lets us replace exhaustive search with a dynamic program (DP) that maximizes the number of surviving requests under the round capacity N.

Concretely, the DP state $\mathrm{dp}[c]$ stores, after processing the first i requests, the maximum number of surviving requests achievable with exactly capacity $c \in \{0, \ldots, N\}$ consumed in the current round. For request i, we build its option set O_i once (group constraint): none (consume zero GPUs, no progress) and one option per allocation m that can make progress in this round, i.e., $q_i^m = \lfloor \tau / T_i(A_i^m) \rfloor > 0$ and $A_i^m \le N$. For each option $o \in O_i$, we compute:

1. **Line 9:** the updated remaining steps $\tilde{s}_i^m(o)$.
2. **Line 10:** a conservative lower bound $\mathrm{LB}_i(o)$ on the residual processing time from $t_{r+1} = t_r + \tau$.
3. **Line 12:** its width $w_i(o)$ (0 for none, A_i^m for allocation m).

We then set the survival indicator $\mathrm{sv}_i(o) = \mathbb{I}[\, t_{r+1} + \mathrm{LB}_i(o) \le D_i \,]$. The DP transition iterates options once per request (respecting the group constraint) and, for each capacity c, admits only options with $w_i(o) \le c$ (respecting the capacity constraint):

$$\mathrm{next}[c] \leftarrow \max\left\{\mathrm{next}[c],\ \mathrm{dp}[c - w_i(o)] + \mathrm{sv}_i(o)\right\}.$$

Using a rolling array yields $O(N)$ space. Since each request contributes at most $|O_i|$ options, DP runs in $O(RN)$ time and $O(N)$ space per round (rolling array), which is tractable even at millisecond-scale rounds for moderate N. This is orders of magnitude cheaper than enumerating all feasible packing combinations.

Round Duration. Algorithm 1 schedules in fixed-length rounds of duration τ. The choice of τ balances two factors: short rounds reduce admission delay for new requests but increase scheduling frequency, while long rounds amortize scheduling cost but risk larger queueing delay and deadline misses. For a given GPU configuration (e.g., NVIDIA H100), TetriServe adapts τ to the step execution times of requests across different resolutions, so that requests with heterogeneous step lengths can finish around the same round boundary. This minimizes idle bubbles while keeping τ short enough to avoid excessive queueing delay. In practice, we determine τ by the *step granularity*, which means each round executes multiple diffusion steps. We will further discuss the impact of round duration in the evaluation section (§6.4).

4.2.3 Efficient GPU Placement and Allocation.

In the round-based framework (Algorithm 1), TetriServe improves efficiency via two complementary steps: *placement preservation* and *work-conserving elastic scale-up*, illustrated in Figure 6. First, to avoid idle bubbles between rounds, TetriServe adopts a placement-aware policy: requests continue on the same GPUs across consecutive rounds whenever possible. This eliminates state-transfer delays and ensures immediate progress at round boundaries.

Second, any GPUs left idle after placement are reclaimed through a work-conserving elastic scale-up policy. Requests with sufficient remaining steps are granted additional GPUs if $T_i(k_i') < T_i(k_i)$, prioritizing those that benefit most from parallelism. This ensures no GPU remains unused within a round, reducing future load and improving deadline satisfaction. Together, placement preservation minimizes inter-round stalls, while elastic scale-up guarantees work-conserving allocation within each round.

5 Implementation

TetriServe is implemented in 5,033 lines of Python and C++ code. We reuse components from existing solutions, including the sequence parallelism engine from xDiT [12], async logic from vLLM [20], and process launcher from MuxServe [11] and SGLang [47].

Scheduler. The scheduler's core decision loop is implemented in C++ and exposed via lightweight bindings, achieving millisecond-level control-plane latency.

VAE Decoder Sequential Execution. The VAE decoder imposes a large activation-memory footprint at high resolutions and batch sizes, whereas its wall-clock cost is very small relative to diffusion steps. Accordingly, we adopt sequential per-request decoding to bound peak memory by avoiding concurrent decoder activations across a batch. Because the decoder is largely off the critical path, this design does not increase end-to-end latency. The reduced peak usage also increases headroom for model state and communication buffers, lowering the risk of out-of-memory failures under mixed workloads.

Communication Process Groups Warmup. We precreate process groups for all relevant combinations of devices (e.g., $\binom{8}{k}$ groups for degrees $k \in \{1, \ldots, 8\}$). Creating the group itself is lightweight and does not materially consume GPU memory. However, the *first* invocation on a group initializes NCCL [31] channels and allocates persistent device buffers for subsequent collectives. Proactively warming *every* group therefore inflates memory usage and can exceed available HBM. To balance startup latency and memory footprint, we warm only a compact set of commonly used, overlapping groups (e.g., [0,1,2,3], [0,2,3,4]) and defer others to on-demand warmup. Empirically, this strategy preserves performance while maintaining low peak memory.

Latent Transfer. Because TetriServe executes at step granularity, intermediate latents and lightweight metadata

must be handed off across GPU groups. We provide a Future-like abstraction for latents that enables asynchronous, non-blocking transfer between steps. Latent tensors are compact (in the compressed latent space), so transfer overhead is negligible; consequently, the scheduler excludes latent-transfer time from deadline accounting. We quantify this overhead in Section 6.4 and show it remains below 0.05% of per-step latency across all configurations.

Selective Continuous Batching. Batching in diffusion inference is only effective for identical, small-resolution requests that would otherwise underutilize GPUs. This creates a throughput-latency trade-off. Our scheduler employs a selective, step-level batching strategy that only groups requests if their SLOs are not compromised, thus improving resource utilization without harming latency.

6 Evaluation

We evaluate TetriServe against state-of-the-art baselines across diverse workloads. Key findings:

- TetriServe outperforms baselines by up to 32% across all resolutions (§6.2).
- TetriServe is robust to bursty arrivals and adapts to changing resolution mixes (§6.3).
- Sensitivity analysis confirms TetriServe's advantage holds across varying arrival rates, step granularities, and homogeneous workloads (§6.4).
- Ablation studies show that GPU placement preservation and elastic scale-up are crucial to TetriServe's performance (§6.5).

6.1 Methodology

Testbed. We conduct experiments on two GPU clusters. The first comprises nodes with 8 NVIDIA H100-80GB HBM3 GPUs interconnected via NVLink 4.0 (900 GB/s inter-GPU bandwidth). The second features nodes with 4 NVIDIA A40-48GB GPUs connected in pairs via NVLink and interfaced to the host via PCIe 4.0. Our software environment is based on NVIDIA's NGC container with CUDA 12.5, NCCL 2.22.3 [31], PyTorch 2.4.0 [46], and xDiT [12] (git-hash 8f4b9d30).

Models and Metrics. We select *FLUX.1-dev* [21] and *Stable Diffusion 3 Medium* (SD3) [3] as representative models, evaluating them on H100 and A40 clusters, respectively. We report SLO Attainment Ratio (SAR; fraction of requests finishing within SLO) as our primary metric and plot end-to-end latency CDFs to show the latency distribution.

Baselines. We compare TetriServe against:

- **xDiT (SP=1/2/4/8).** Fixed sequence parallelism degree; each request uses a constant number of GPUs.
- **Resolution-Specific SP (RSSP).** Selects the best SP degree per resolution via offline profiling: SP=1 for 256×256 and 512×512, SP=2 for 1024×1024, and SP=8 for 2048×2048. Represents an oracle static configuration.

SLO Settings. We adopt resolution-specific latency targets grounded in user-perceived responsiveness. Prior research [1] reports that 63% of users prefer a maximum response delay of 5 seconds in interactive settings. Accordingly, we cap the target at 1.5 seconds for small images and set an upper bound of 5.0 seconds for the largest resolution: $(256, 256)$ = 1.5 s, $(512, 512)$ = 2.0 s, $(1024, 1024)$ = 3.0 s, and $(2048, 2048)$ = 5.0 s. We sweep SLO Scale from 1.0× to 1.5× relative to each resolution's baseline.

Workload and Dataset. We sample 300 prompts from DiffusionDB [42] to generate requests. By default, requests arrive as a Poisson process at 12 requests/minute.

We consider two resolution mixes:

- *Uniform*: equal number of requests across resolutions {256, 512, 1024, 2048}.
- *Skewed*: resolutions sampled with exponential weight over latent length, $p_i \propto \exp(\alpha \cdot L_i / L_{\max})$, with $\alpha = 1.0$ and $L_i = (H_i \cdot W_i)/16^2$, biasing toward larger resolutions.

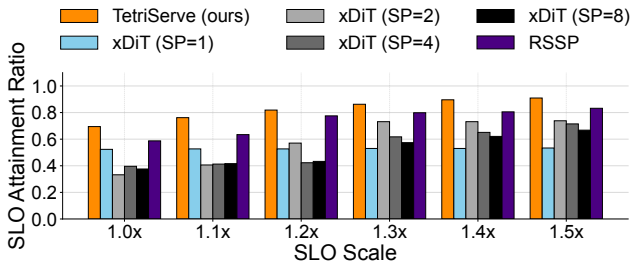
6.2 End-to-End Performance

TetriServe Improves SAR.. Figures 7 and 8 show the end-to-end SLO Attainment Ratio (SAR) of TetriServe compared to fixed-parallelism baselines for FLUX on H100s for both the Uniform and Skewed workload mixes at an arrival rate of 12 requests per minute. As shown in Figures 7a and 8a, TetriServe consistently achieves the highest SAR across all SLO scales and both workload distributions. This demonstrates the effectiveness of its step-level parallelism control and request packing, which allow it to dynamically adapt to the workload and outperform the rigid strategies of the baselines.

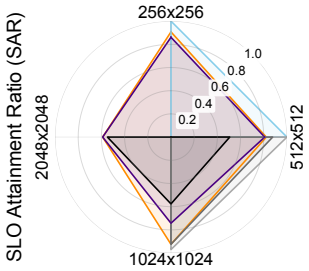
On average, TetriServe outperforms the best fixed parallelism strategy by 10% for the Uniform mix and 15% for the Skewed mix. The performance gap is particularly pronounced at tighter SLOs. For instance, with an SLO scale of 1.1× in the Uniform mix, TetriServe outperforms the best baseline by 28%. Similarly, in the Skewed mix with a 1.2× SLO scale, TetriServe's SAR is 32% higher than the best-performing fixed strategy.

Notably, this advantage holds even when compared against RSSP, a strong per-resolution baseline that selects the best fixed parallelism degree for each input resolution. Despite this, RSSP remains fundamentally limited by its lack of deadline awareness and runtime adaptation, whereas TetriServe dynamically adjusts parallelism at the step level to meet per-request SLOs. This highlights TetriServe's superior performance under challenging, tightly constrained Workloads.

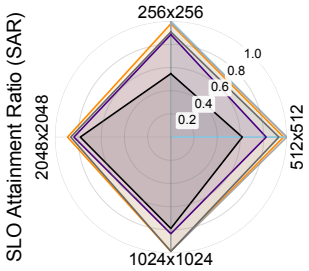
TetriServe Benefits All Resolutions. TetriServe's strength lies in its ability to deliver high SAR across all request resolutions, unlike fixed strategies that only excel at specific



(a) SLO Attainment Ratio (SAR) of Uniform Workload

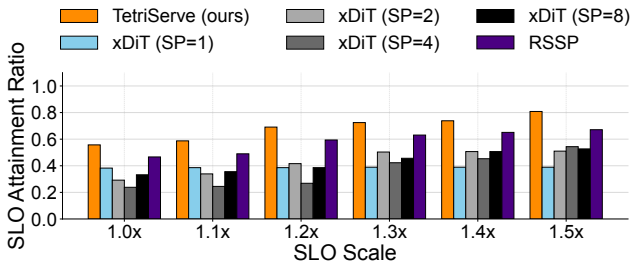


(b) Uniform, SLO Scale=1.0×

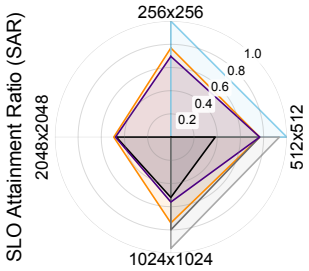


(c) Uniform, SLO Scale=1.5×

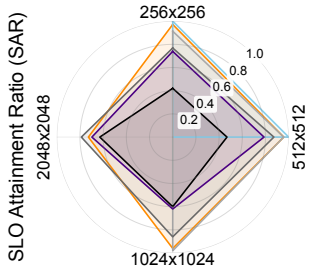
Figure 7. End-to-end performance on the Uniform workload at 12 req/min. **(Top)** TetriServe achieves the highest SLO Attainment Ratio (SAR) across all SLO scales. **(Bottom)** The spider plots show that xDiT variants only perform well for specific resolutions, TetriServe delivers high SAR across all resolutions no matter tight or loose SLO Setting.



(a) SAR of Skewed Workload



(b) Skewed, SLO Scale=1.0×



(c) Skewed, SLO Scale=1.5×

Figure 8. End-to-end performance on the Skewed workload at 12 req/min. **(Top)** TetriServe again achieves the highest SLO Attainment Ratio (SAR) across all SLO scales. **(Bottom)** The spider plots confirm that TetriServe's adaptive parallelism provides robust performance across all resolutions, even in a workload dominated by large images

ones. The spider plots in the bottom row of Figures 7 and 8 break down SAR by resolution. With a relaxed SLO of 1.5× (Figures 7c and 8c), TetriServe achieves near-perfect SAR across all resolutions for both workload mixes, consistently outperforming all xDiT baselines. Under the tightest SLO of 1.0× (Figures 7b and 8b), TetriServe provides the best overall performance. While some fixed-parallelism strategies may marginally outperform TetriServe on a single resolution (e.g., xDiT SP=1 on 256px), they perform poorly on others. In contrast, TetriServe dynamically adapts its parallelism, providing high SAR across the entire spectrum of resolutions.

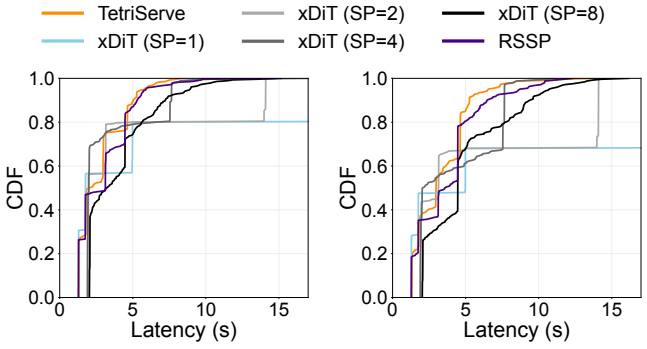
Conceptually, RSSP is a restricted variant of TetriServe in which the scheduler cannot adjust parallelism beyond a fixed configuration. Since RSSP explores only a subset of TetriServe's decision space, it cannot exploit additional parallelism for deadline-critical requests, resulting in uniformly lower SAR across resolutions. In contrast, TetriServe avoids over parallelization for less urgent requests and prioritizes more GPU resources for more urgent requests, thus performing well on all resolutions.

Tail Latency. Figure 9 plots the CDF of end-to-end request latency under the tightest SLO setting (SLO scale = 1.0×) for both the Uniform and Skewed mixes. We compute the CDF over completed requests only, i.e., requests that finish execution at least once (those that miss the deadline and are dropped/timeout are excluded from the latency distribution). Across both workload mixes, TetriServe produces a consistently more favorable tail distribution than fixed-parallelism baselines and RSSP. Compared to fixed SP baselines, TetriServe shifts the latency distribution left and reaches high completion probability at lower latency, indicating that most served requests finish quickly even under strict deadlines. Compared to RSSP, which restricts scheduling to a smaller decision space, TetriServe further reduces tail latency by dynamically reallocating GPUs toward more urgent requests and avoiding over-parallelization on less critical ones. Overall, these results show that TetriServe improves not only SAR but also keep the steady long tail latency under tight SLO scale.

Compatibility with Cache-Based Diffusion Acceleration. TetriServe is orthogonal and compatible with cache-based diffusion acceleration techniques. To demonstrate this, we integrate Nirvana [2] into our system. Nirvana accelerates diffusion inference by reusing intermediate denoising latents from prior requests. Each incoming prompt is embedded using CLIP [35] and matched against a cache of previously served prompts. Based on prompt similarity, the system determines how many initial diffusion steps can be skipped, yielding an effective diffusion length of $N - k$ steps, where

Table 3. SAR with Nirvana Integration. SLO Attainment Ratio (SAR) under uniform and skewed workload mixes (12 req/min, SLO Scale = 1.0×). TetriServe combined with Nirvana [2] achieves the highest SAR by jointly exploiting cache-based step reduction and adaptive GPU parallelism.

Workload	RSSP	TetriServe	RSSP + Nirvana	TetriServe + Nirvana
Uniform	0.32	0.42	0.77	**0.88**
Skewed	0.04	0.19	0.53	**0.75**



(a) Uniform mix **(b)** Skewed mix

Figure 9. End-to-end latency CDF under strict SLOs (FLUX on H100, SLO scale = 1.0×). TetriServe shows more consistent and better tail latency distribution than other baselines under strict SLO settings. The x-axis is truncated at 17s for readability; the SP=1 baseline has a much heavier tail beyond this range.

$k \in \{5, 10, 15, 20, 25\}$ and $N = 50$ by default. We warm up the cache using the first 10K requests and then maintain a fixed-size cache with LRU eviction for online requests.

Table 3 compares four configurations: RSSP, TetriServe, RSSP combined with Nirvana, and TetriServe combined with Nirvana, under both Uniform and Skewed mix workloads under the SLO Scale of 1.0×. While Nirvana alone substantially improves SLO attainment by reducing per-request computation, it does not address resource fragmentation caused by heterogeneous request resolutions. By contrast, TetriServe further improves SLO attainment by dynamically adjusting GPU parallelism to match the reduced and variable step counts introduced by caching. As a result, the combined system achieves the highest SLO attainment across both mixes, confirming that cache-based step reduction and TetriServe's scheduling operate on complementary and orthogonal dimensions.

6.3 Performance Stability under Bursty Traffic

TetriServe maintains a high and stable SAR even under bursty arrival patterns, whereas fixed-parallelism approaches exhibit significant performance oscillations. For instance, Figure 10 plots the SAR over time for the Uniform mix (12 req/min, SLO Scale=1.5×). TetriServe's SAR remains consistently high with low variance. In contrast, the fixed xDiT variants suffer from periodic drops in SAR, a result of utilization bubbles and subsequent queueing delays when bursty arrivals create contention.

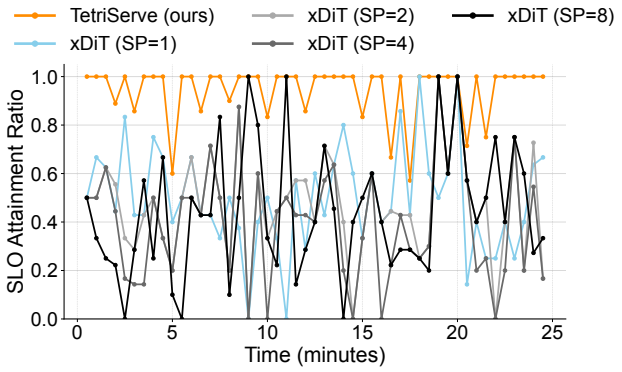


Figure 10. Performance stability under the Uniform workload at 12 req/min with a 1.5x SLO Scale. TetriServe maintains a high and stable SLO Attainment Ratio (SAR) over time, which handles burstiness well.

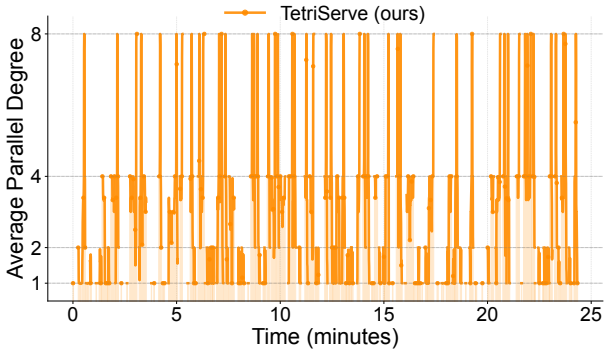
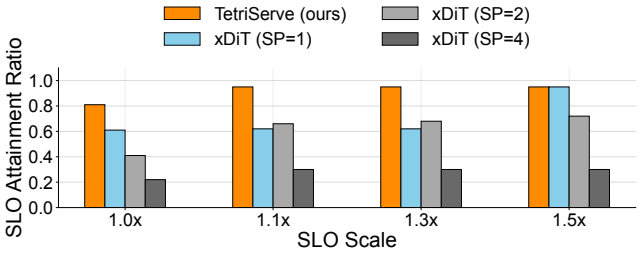
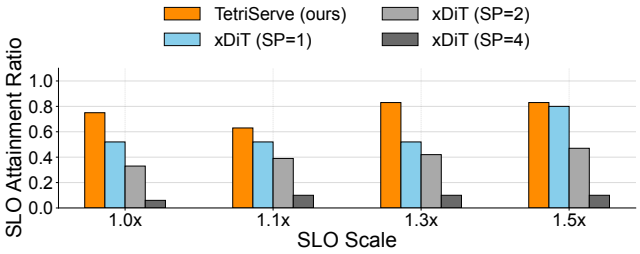


Figure 11. Average parallel degree of TetriServe during serving under the Uniform workload (1.5× SLO Scale). TetriServe dynamically adjusts sequence parallelism (SP) per request, assigning more GPUs to intensive requests (longer bars) to meet deadlines.

The key to TetriServe's stability is its ability to adapt the degree of sequence parallelism (SP) at the step level. As shown in Figure 11, when bursty arrivals create contention, TetriServe dynamically raises the SP degree for computationally intensive, urgent requests to shorten their critical path and reduce SLO violation risk. Conversely, it scales down the degree for less urgent requests steps while maintain SLO Attainment Ratio. This fine-grained, adaptive parallelism is how TetriServe handles burstiness and achieves superior efficiency and responsiveness compared to rigid, fixed-degree systems.



(a) SAR vs. SLO Scale (SD3, Uniform mix)



(b) SAR vs. SLO Scale (SD3, Skewed mix)

Figure 12. TetriServe's performance on the Stable Diffusion 3 (SD3) model. The plots show the SLO Attainment Ratio (SAR) as a function of SLO Scale for the Uniform mix (left) and Skewed mix (right) on 4×A40 GPUs. In both workloads, TetriServe consistently outperforms all xDiT variants

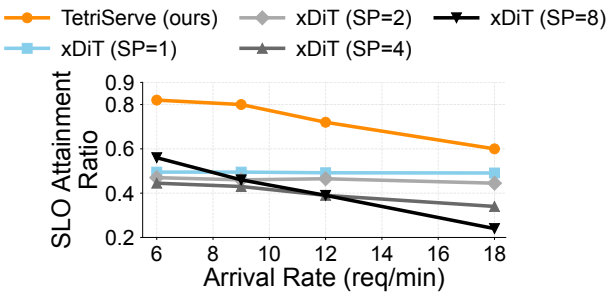


Figure 13. SLO Attainment Ratio vs. arrival rate under the Uniform mix (SLO Scale=1.0x). TetriServe gracefully handles increasing load, maintaining a high SAR.

6.4 Sensitivity Analysis

Different GPU Settings and Models. On SD3, trends align with FLUX. In both the Uniform mix (Figure 12a) and Skewed mix (Figure 12b), TetriServe achieves the highest SAR across all SLO scales, with the largest margins at tight SLOs (1.0×). As SLOs loosen, fixed SP2 and SP4 improve but remain below TetriServe, while fixed SP1 underutilize and plateau. This indicates the benefits generalize to a different DiT architecture. On the A40 cluster, NVLink links GPUs only in pairs; at SP=4, collectives traverse PCIe, and even at SP=2 poor placement can cross PCIe. For SD3 this communication path becomes the bottleneck, so SP2 and SP4 perform notably worse than on H100.

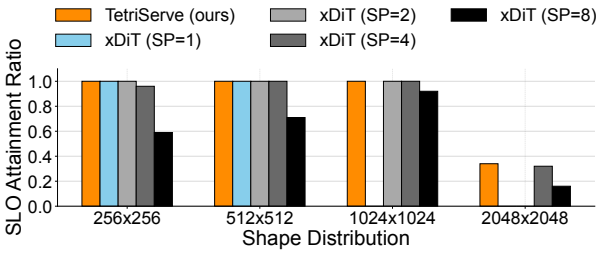


Figure 14. SLO Attainment Ratio for homogeneous workloads at 12 req/min with a 1.5x SLO Scale. Each group of bars represents a workload with only one resolution type. TetriServe consistently achieves the highest SAR across all resolutions.

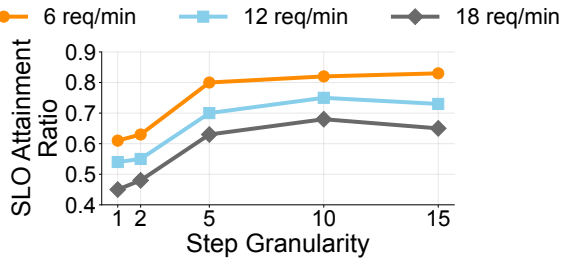


Figure 15. Sensitivity of SLO Attainment Ratio to step granularity and arrival rate under the Uniform mix (SLO Scale=1.0x). A moderate granularity (5/10 steps) provides the most robust performance as system load increases, balancing scheduling flexibility and overhead.

Arrival Rate. Figure 13 shows the SAR of different scheduling strategies under the Uniform mix with a tight SLO of 1.0× as the arrival rate increases from 6 to 18 req/min. TetriServe demonstrates superior performance across the full range of arrival rates. At low-to-medium rates, TetriServe maintains a consistently high SAR, while fixed-parallelism strategies already show signs of degradation. At high arrival rates, where the system is under heavy load, TetriServe's SAR remains relatively high, showcasing graceful degradation.

Homogeneous Resolutions. To isolate the effect of input resolution on parallelism strategies, we evaluate homogeneous workloads containing only a single resolution. Figure 14 shows the SLO Attainment Ratio (SAR) for workloads consisting of only one resolution type at an arrival rate of 12 req/min and an SLO Scale of 1.5x. Even in these simplified scenarios, TetriServe still achieves the highest SAR across all resolution types. This demonstrates that TetriServe's adaptive scheduling is effective not only for mixed workloads but also for homogeneous ones, as it can still optimize resource allocation to better meet deadlines.

Step Granularity. We examine the impact of step granularity, which defines how frequently TetriServe can reschedule and change the degree of parallelism for an in-flight

Table 4. Latent transfer overhead as a percentage of inference step latency. Across all configurations, the overhead is negligible (< 0.05%).

Batch Size	256×256	512×512	1024×1024	2048×2048
BS = 1	0.03%	0.03%	0.04%	0.01%
BS = 2	0.04%	0.03%	0.05%	0.02%
BS = 4	0.04%	0.05%	0.03%	0.01%

request. This presents a fundamental trade-off: fine-grained control (e.g., every 1-2 steps) offers maximum flexibility at the cost of high scheduling overhead, while coarse-grained control (e.g., every 10 steps) minimizes overhead but creates longer, non-preemptible execution blocks that reduce adaptability. Figure 15 illustrates this trade-off under the Uniform mix (SLO Scale=1.0x) across different arrival rates. At low rates, performance is less sensitive to granularity. However, as load increases, a moderate granularity of 5 steps proves most robust, balancing adaptability and overhead. Very fine-grained control (1 step) suffers from excessive overhead, while coarse-grained control (10 steps) is too inflexible to handle preemption, leading to lower SLO attainment.

Parallel Reconfiguration Overhead. TetriServe performs step-level scheduling, which requires transferring intermediate latent representations and metadata across GPU groups when parallelism changes between steps. Table 4 quantifies this parallel reconfiguration overhead as a percentage of per step inference latency across varying resolutions and batch sizes. We observe that the overhead is consistently negligible, accounting for at most 0.05% of step latency in all configurations. As a result, TetriServe's scheduler can safely ignore latent transfer time in deadline accounting without affecting SLO accuracy.

6.5 Ablation Study

TetriServe includes two practical mechanisms on top of the round-based DP scheduler: (i) *GPU Placement Preservation*, which keeps a request on the same GPU set across rounds whenever possible to avoid remapping stalls; and (ii) *Elastic Scale-up*, which makes use of idle GPUs after placement and temporarily grants extra GPUs to requests that benefit from higher parallelism. To quantify their impact, we ablate these components under two SLO scales (1.0× and 1.5×) on two workload mixes: Uniform and Skewed. Table 5 reports the SLO Attainment Ratio and mean latency.

Overall, both mechanisms are important for improving serving efficiency. GPU Placement Preservation improves SAR and/or mean latency in most settings by avoiding remapping overhead and enabling immediate progress at round boundaries, while Elastic Scale-up consistently increases SAR (up to +0.11 absolute on Skewed mix at 1.5×) and typically further reduces mean latency by utilizing idle GPUs. Consequently, enabling both GPU placement preservation and Elastic Scale-up achieves the best SLO Attainment Ratio across all tested scenarios, while also improving latency compared to disabling these optimizations.

Table 5. Ablation of scheduling mechanisms. GPU Placement Preservation reduces inter-round stalls by keeping requests on the same GPU set; Elastic Scale-up opportunistically reallocates idle GPUs to requests that benefit from extra parallelism.

(a) Uniform Mix.

Variant	SLO = 1.0× SAR ↑ / Mean Lat. ↓	SLO = 1.5× SAR ↑ / Mean Lat. ↓
TetriServe schedule	0.54 / 4.45	0.74 / **4.81**
+ Placement	0.56 / 3.96	0.69 / 5.14
+ Elastic Scale-Up	**0.63** / **3.89**	**0.78** / 4.83

(b) Skewed Mix.

Variant	SLO = 1.0× SAR ↑ / Mean Lat. ↓	SLO = 1.5× SAR ↑ / Mean Lat. ↓
TetriServe schedule	0.27 / 8.43	0.38 / 9.92
+ Placement	0.31 / **7.64**	0.45 / 8.16
+ Elastic Scale-Up	**0.36** / 7.68	**0.55** / **7.71**

7 Related Work

LLM Serving Frameworks. LLM serving systems [20, 47] are not directly applicable to DiT workloads. LoongServe [43] optimizes prefill-decode stages for long-context LLMs, while PrefillOnly [10] targets memory efficiency for short, prefill-intensive requests. Neither suits the multi-step, stateless inference pattern of DiTs.

DiT Inference and Serving. DiT-specific serving systems are still emerging. xDiT [12] uses fixed sequence parallelism, which is inefficient for heterogeneous workloads. DDiT [17] targets video generation and maximizes throughput rather than meeting SLOs. TetriServe uniquely prioritizes SLO attainment for heterogeneous requeststhrough cost-model-driven scheduling.

Text-to-Image Caching. Several systems accelerate text-to-image diffusion via caching. AsyncDiff [8] parallelizes diffusion through asynchronous denoising cross requests. Caching-based approaches exploit reuse across prompts or adapters, including approximate latent caching in Nirvana [2], layer-level caching [26], final image caching [44], workflow-aware reuse [24], and patch-level reuse [40]. These techniques reduce redundant computation; TetriServe addresses an orthogonal dimension by scheduling GPU parallelism across concurrent requests and could integrate these methods for further gains.

Resource Scheduling. In VM allocation frameworks [6], machine count is fixed at admission. GPU schedulers like

Gavel [30], Tiresias [15], and AlloX [22] focus on job placement and fairness but require users to specify parallelism. In contrast, TetriServe treats parallelism as a scheduling decision, dynamically adjusting GPU degree at step granularity based on deadlines and scaling efficiency.

8 Conclusion

We presented TetriServe, a deadline-aware round-based DiT serving system that addresses the challenge of meeting SLOs under heterogeneous workloads. TetriServe dynamically adapts parallelism at the *step level*, guided by a profiling-driven cost model and a deadline-aware scheduling algorithm. Extensive evaluation shows that TetriServe consistently outperforms fixed-parallelism baselines, achieving up to 32% higher SLO attainment and robust performance across varying resolutions, workload distributions, and arrival rates.

Acknowledgements

We thank the ASPLOS reviewers, as well as members of SymbioticLab and UseSysLab, for their helpful feedback. This work was supported in part by NSF grants CCF-2450085, CNS-2106184, CNS-2214272 and CNS-2106751, and by grants from Ford and Cisco.

References

[1] Tahir Abbas, Ujwal Gadiraju, Vassilis-Javed Khan, and Panos Markopoulos. 2022. Understanding User Perceptions of Response Delays in Crowd-Powered Conversational Systems. *Proceedings of the ACM on Human-Computer Interaction* (2022).

[2] Shubham Agarwal, Subrata Mitra, Sarthak Chakraborty, Srikrishna Karanam, Koyel Mukherjee, and Shiv Kumar Saini. 2024. Approximate Caching for Efficiently Serving Text-to-Image Diffusion Models. In *NSDI*.

[3] Stability AI. 2024. Stable Diffusion 3 Medium. https://huggingface.co/stabilityai/stable-diffusion-3-medium.

[4] Stability AI. 2024. Stable Diffusion 3.5 Large. https://huggingface.co/stabilityai/stable-diffusion-3.5-large.

[5] Amotz Bar-Noy, Sudipto Guha, Joseph Naor, and Baruch Schieber. 1999. Approximating the Throughput of Multiple Machines under Real-Time Scheduling. In *STOC*.

[6] Hugo Barbalho, Patricia Kovaleski, Beibin Li, Luke Marshall, Marco Molinaro, Abhisek Pan, Eli Cortez, Matheus Leao, Harsh Patwari, Zuzu Tang, et al. 2023. Virtual Machine Allocation with Lifetime Predictions. In *MLSys*.

[7] Tim Brooks, Bill Peebles, Connor Holmes, Will DePue, Yufei Guo, Li Jing, David Schnurr, Joe Taylor, Troy Luhman, Eric Luhman, et al. 2024. Video Generation Models as World Simulators. *OpenAI Blog* (2024).

[8] Zigeng Chen, Xinyin Ma, Gongfan Fang, Zhenxiong Tan, and Xinchao Wang. 2024. AsyncDiff: Parallelizing Diffusion Models by Asynchronous Denoising. *NeurIPS*.

[9] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby. 2021. An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale. In *ICLR*.

[10] Kuntai Du, Bowen Wang, Chen Zhang, Yiming Cheng, Qing Lan, Hejian Sang, Yihua Cheng, Jiayi Yao, Xiaoxuan Liu, Yifan Qiao, Ion Stoica, and Junchen Jiang. 2025. PrefillOnly: An Inference Engine for Prefill-only Workloads in Large Language Model Applications. In *SOSP*.

[11] Jiangfei Duan, Runyu Lu, Haojie Duanmu, Xiuhong Li, Xingcheng Zhang, Dahua Lin, Ion Stoica, and Hao Zhang. 2024. MuxServe: Flexible Spatial-Temporal Multiplexing for Multiple LLM Serving. In *ICML*.

[12] Jiarui Fang, Jinzhe Pan, Xibo Sun, Aoyu Li, and Jiannan Wang. 2024. xDiT: an Inference Engine for Diffusion Transformers (DiTs) with Massive Parallelism. *arXiv preprint arXiv:2411.01738* (2024).

[13] Flux.1 AI. 2025. *Flux.1 AI Image Generator.* https://flux1.ai/create

[14] Michael R Garey and David S. Johnson. 1977. Two-Processor Scheduling with Start-Times and Deadlines. *SIAM journal on Computing* (1977).

[15] Juncheng Gu, Mosharaf Chowdhury, Kang G. Shin, Yibo Zhu, Myeongjae Jeon, Junjie Qian, Hongqiang Harry Liu, and Chuanxiong Guo. 2019. Tiresias: A GPU Cluster Manager for Distributed Deep Learning. In *NSDI*.

[16] Jonathan Ho, Ajay Jain, and Pieter Abbeel. 2020. Denoising diffusion probabilistic models. In *NeurIPS*.

[17] Heyang Huang, Cunchen Hu, Jiaqi Zhu, Ziyuan Gao, Liangliang Xu, Yizhou Shan, Yungang Bao, Sun Ninghui, Tianwei Zhang, and Sa Wang. 2025. DDiT: Dynamic Resource Allocation for Diffusion Transformer Model Serving. *arXiv preprint arXiv:2506.13497* (2025).

[18] Sam Ade Jacobs, Masahiro Tanaka, Chengming Zhang, Minjia Zhang, Shuaiwen Leon Song, Samyam Rajbhandari, and Yuxiong He. 2023. DeepSpeed Ulysses: System Optimizations for Enabling Training of Extreme Long Sequence Transformer Models. *arXiv preprint arXiv:2309.14509* (2023).

[19] Alind Khare, Dhruv Garg, Sukrit Kalra, Snigdha Grandhi, Ion Stoica, and Alexey Tumanov. 2025. SuperServe: Fine-Grained Inference Serving for Unpredictable Workloads. In *NSDI*.

[20] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient Memory Management for Large Language Model Serving with PagedAttention. In *SOSP*.

[21] Black Forest Labs. 2024. *FLUX.1-dev: Text-to-Image Generation Model.*

[22] Tan N. Le, Xiao Sun, Mosharaf Chowdhury, and Zhenhua Liu. 2020. AlloX: Compute Allocation in Hybrid Clusters. In *EuroSys*.

[23] Shenggui Li, Fuzhao Xue, Chaitanya Baranwal, Yongbin Li, and Yang You. 2023. Sequence Parallelism: Long Sequence Training from System Perspective. In *ACL*.

[24] Suyi Li, Lingyun Yang, Xiaoxiao Jiang, Hanfeng Lu, Dakai An, Zhipeng Di, Weiyi Lu, Jiawei Chen, Kan Liu, Yinghao Yu, Tao Lan, Guodong Yang, Lin Qu, Liping Zhang, and Wei Wang. 2025. Katz: Efficient Workflow Serving for Diffusion Models with Many Adapters. In *ATC*.

[25] Hao Liu, Matei Zaharia, and Pieter Abbeel. 2023. Ring Attention with Blockwise Transformers for Near-Infinite Context. *arXiv preprint arXiv:2310.01889* (2023).

[26] Xinyin Ma, Gongfan Fang, Michael Bi Mi, and Xinchao Wang. 2024. Learning-to-Cache: Accelerating Diffusion Transformer via Layer Caching. *NeurIPS*.

[27] Yixuan Mei, Yonghao Zhuang, Xupeng Miao, Juncheng Yang, Zhihao Jia, and Rashmi Vinayak. 2025. Helix: Serving Large Language Models over Heterogeneous GPUs and Network via Max-Flow. In *ASPLOS*.

[28] Xupeng Miao, Gabriele Oliaro, Zhihao Zhang, Xinhao Cheng, Zeyu Wang, Zhengxin Zhang, Rae Ying Yee Wong, Alan Zhu, Lijie Yang, Xiaoxiang Shi, Chunan Shi, Zhuoming Chen, Daiyaan Arfeen, Reyna Abhyankar, and Zhihao Jia. 2024. SpecInfer: Accelerating Large Language Model Serving with Tree-based Speculative Inference and Verification. In *ASPLOS*.

[29] Xupeng Miao, Chunan Shi, Jiangfei Duan, Xiaoli Xi, Dahua Lin, Bin Cui, and Zhihao Jia. 2024. SpotServe: Serving Generative Large Language Models on Preemptible Instances. In *ASPLOS*.

[30] Deepak Narayanan, Keshav Santhanam, Fiodar Kazhamiaka, Amar Phanishayee, and Matei Zaharia. 2020. Heterogeneity-Aware Cluster

Scheduling Policies for Deep Learning Workloads. In *OSDI*.
[31] NVIDIA. 2022. NVIDIA Collective Communication Library (NCCL) Documentation. https://docs.nvidia.com/deeplearning/nccl/user-guide/docs/index.html.
[32] Christos H Papadimitriou and Kenneth Steiglitz. 1998. *Combinatorial Optimization: Algorithms and Complexity*. Courier Corporation.
[33] Pratyush Patel, Esha Choukse, Chaojie Zhang, Aashaka Shah, Íñigo Goiri, Saeed Maleki, and Ricardo Bianchini. 2024. Splitwise: Efficient Generative LLM Inference Using Phase Splitting. In *ISCA*.
[34] William Peebles and Saining Xie. 2023. Scalable Diffusion Models with Transformers. In *ICCV*.
[35] Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, et al. 2021. Learning Transferable Visual Models From Natural Language Supervision. In *ICML*.
[36] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. 2015. U-Net: Convolutional networks for biomedical image segmentation. In *MICCAI*.
[37] Jascha Sohl-Dickstein, Eric A. Weiss, Niru Maheswaranathan, and Surya Ganguli. 2015. Deep unsupervised learning using nonequilibrium thermodynamics. *arXiv preprint arXiv:1503.03585* (2015).
[38] Yang Song and Stefano Ermon. 2021. Score-Based Generative Modeling through Stochastic Differential Equations. In *ICLR*.
[39] Stability AI. 2024. Stability AI Platform API Reference. https://platform.stability.ai/docs/api-reference Accessed: 2024-11-26.
[40] Desen Sun, Zepeng Zhao, and Yuke Wang. 2026. MixFusion: A Patch-Level Parallel Serving System for Mixed-Resolution Diffusion Models. In *PPoPP*.
[41] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is All You Need. In *NeurIPS*.
[42] Zijie J Wang, Evan Montoya, David Munechika, Haoyang Yang, Benjamin Hoover, and Duen Horng Chau. 2023. DiffusionDB: A Large-Scale Prompt Gallery Dataset for Text-to-Image Generative Models. In *ACL*.
[43] Bingyang Wu, Shengyu Liu, Yinmin Zhong, Peng Sun, Xuanzhe Liu, and Xin Jin. 2024. LoongServe: Efficiently Serving Long-Context Large Language Models with Elastic Sequence Parallelism. In *SOSP*.
[44] Yuchen Xia, Divyam Sharma, Yichao Yuan, Souvik Kundu, and Nishil Talati. 2026. MoDM: Efficient Serving for Image Generation via Mixture-of-Diffusion Models. In *ASPLOS*.
[45] Hong Zhang, Yupeng Tang, Anurag Khandelwal, and Ion Stoica. 2023. SHEPHERD: Serving DNNs in the Wild. In *NSDI*.
[46] Yanli Zhao, Andrew Gu, Rohan Varma, Liang Luo, Chien-Chin Huang, Min Xu, Less Wright, Hamid Shojanazeri, Myle Ott, Sam Shleifer, et al. 2023. PyTorch FSDP: Experiences on Scaling Fully Sharded Data Parallel. In *VLDB*.
[47] 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. 2024. SGLang: Efficient Execution of Structured Language Model Programs. In *NeurIPS*.

A NP-Hardness of DiT Serving

We prove NP-hardness for the DiT serving problem defined in TetriServe, which maximizes the number of requests that complete by deadlines under GPU capacity constraints.

Let us first define the decision problem DiT-Serving-Decision: given an instance, and an integer target B, decide whether there exists a schedule in which at least B requests meet their deadlines. This is the natural decision version of TetriServe's objective max $\sum_i I_i$.

# Reqs	Time (s)
1	*<0.01*
2	*0.27*
3	*52.56*
4	*>60.00*

(a) 4 GPUs

# Reqs	Time (s)
1	*0.02*
2	*11.12*
3	*>60.00*
4	*>60.00*

(b) 8 GPUs

Table 6. Scheduling overhead of exhaustive search. Control plane scheduling time under different GPU budgets and queue sizes. TetriServe remains lightweight: it takes ***<0.01 s*** compared to exhaustive search following the same settings, enabling online scheduling in practice.

Bar-Noy et al. [5, 14] state that the following real-time (RT) scheduling feasibility decision problem (RT-Feasibility) is NP-hard in the strong sense: on a *single* machine, given jobs with release times r_i, deadlines d_i, and processing times l_i, decide whether *all* jobs can be scheduled within their time windows. Since RT-Feasibility is strongly NP-hard, it remains NP-hard even when all numeric parameters are bounded by a polynomial in the input size. Therefore, $T_{\max} = \max_i d_i$ is polynomially bounded, and our time-indexed reduction is polynomial-time.

Reduction to DiT serving with $\mathcal{K} = \{1\}$. Given a RT-Feasibility instance [5] with jobs $i = 1, \ldots, n$ and parameters (r_i, d_i, l_i), let us construct a single-step DiT instance as follows: $N := 1, R := n, S_i := 1, K := \{1\}, \texttt{arrival\_time}(i) := r_i, D_i := d_i, T_i(1) := l_i$. Set the throughput target $B := n$.

Equivalently, in TetriServe's single-step time-indexed formulation with variables $x_{i,t,k}$ and constraints (1)–(5), we restrict to $k = 1$ and $N = 1$, and disallow infeasible start times by setting $x_{i,t,1} = 0$ whenever $t < r_i$ or $t + l_i > d_i$.

Correctness. ($\Rightarrow$) If the RT-Feasibility instance is feasible, let s_i be the start time of job i in a feasible single-machine schedule. Schedule each corresponding DiT request i to start at time s_i using one GPU. All requests meet deadlines, so $\sum_i I_i = n \geq B$.

($\Leftarrow$) If the constructed DiT instance has a schedule with $\sum_i I_i \geq n$, then all n requests meet deadlines. Since $N = 1$ and each request uses one GPU, the capacity constraint implies no two requests overlap. Thus the chosen start times form a feasible non-preemptive single-machine schedule for all jobs in the original RT-Feasibility instance.

Therefore, we can convert any RT-Feasibility instance into a DiT-Serving-Decision instance in polynomial time such that a feasible schedule exists in the former iff one exists in the latter. DiT-Serving-Decision is NP-hard even for the restricted case $S_i = 1$ and $\mathcal{K} = \{1\}$; consequently, the general multi-step DiT serving problem is NP-hard.

B Scheduling Overhead Analysis

To validate the necessity of TetriServe's heuristic approach, we quantify the computational cost of finding a globally optimal schedule via exhaustive search. As established in Appendix A, the underlying step-level scheduling problem is NP-hard.

Experimental Setup. We implement an exact baseline solver that enumerates the complete decision space to maximize SLO attainment. The solver explores two dimensions of complexity for each request: (1) all feasible sequence-parallel degrees per diffusion step (e.g., $k \in \{1, 2, 4, 8\}$), and (2) all valid permutations of physical GPU mapping for those degrees. The objective is to identify the schedule with the highest SLO attainment, using minimum total GPU hours as a tie-breaker. We measure the wall clock latency required to generate a single scheduling plan using an AMD EPYC 7513 32-Core CPU, varying the queue depth (R) under fixed GPU budgets of $N \in \{4, 8\}$.

Results. Table 6 presents the scheduling overhead. The baseline exhibits immediate combinatorial explosion: with a budget of 8 GPUs, optimally scheduling merely three requests exceeds a 60-second timeout. This intractability stems from the factorial growth of permutation possibilities as the number of available GPUs increases. In contrast, TetriServe maintains a decision latency of *<10 ms*. These results confirm that exhaustive optimization is prohibitive for online serving, necessitating the efficient round-based planning strategy employed by TetriServe.