


# DIFFSERVE: EFFICIENTLY SERVING TEXT-TO-IMAGE DIFFUSION MODELS WITH QUERY-AWARE MODEL SCALING

**Sohaib Ahmad** [*1] **Qizheng Yang** [*1] **Haoliang Wang** [2] **Ramesh K. Sitaraman** [1] **Hui Guan** [1]



## ABSTRACT

Text-to-image generation using diffusion models has gained increasing popularity due to their ability to produce high-quality, realistic images based on text prompts. However, efficiently serving these models is challenging due to their computation-intensive nature and the variation in query demands. In this paper, we aim to address both problems simultaneously through query-aware model scaling. The core idea is to construct model cascades so that easy queries can be processed by more lightweight diffusion models without compromising image generation quality. Based on this concept, we develop an end-to-end text-to-image diffusion model serving system, DIFFSERVE, which automatically constructs model cascades from available diffusion model variants and allocates resources dynamically in response to demand fluctuations. Our empirical evaluations demonstrate that DIFFSERVE achieves up to 24% improvement in response quality while maintaining 19-70% lower latency violation rates compared to state-of-the-art model serving systems.

## 1 INTRODUCTION

Text-to-image diffusion models are a powerful class of generative models that create images from textual descriptions (queries) by progressively denoising an initial noise input into a coherent image. They have gained increasing popularity and been integrated into various interactive content creation workflows such as Adobe FireFly (Adobe, 2024) and Midjourney (Midjourney, 2024). With their growing adoption, it is essential to develop efficient diffusion model serving systems that deliver fast and accurate responses to queries. In such systems, providers must guarantee service level objectives (SLOs) to users with regard to latency deadlines. At the same time, providers aim to maximize hardware utilization by increasing throughput, i.e., serving as many queries per unit time.

Efficiently serving text-to-image diffusion models, however, presents two main challenges. First, high-quality diffusion models are computationally intensive, limiting the serving throughput of the model serving system. Here, *serving throughput* refers to the number of queries (i.e., text prompts) that can be processed by the system per unit of time (queries per second, QPS). For instance, the Stable Diffusion XL model (Podell et al., 2023) achieves a 30% improvement in generated image quality on DiffusionDB (Wang et al., 2022) dataset compared to the baseline SDXL-Lightning (Lin et al., 2024) but is $4.6\times$ slower when processing a batch of 16 queries on a Nvidia A100-80GB GPU. This example highlights the typical trade-off between accuracy and efficiency often encountered in machine learning models. Second, the query demand for a model serving system fluctuates over time (Shahrad et al., 2020; twi, 2018). Hardware resources provisioned to handle peak demand may remain largely idle during periods of low query demand, resulting in inefficient resource utilization.

This work introduces *query-aware model scaling* to tackle the challenges. The central concept of query-aware model scaling is the creation of a *diffusion model cascade*, where each query is first processed by a lightweight diffusion model (also called light model) to generate an image. If the output meets predefined quality requirements, as determined by a *discriminator*, it is used as the final response. If not, the query is routed to a more computationally intensive but higher-quality diffusion model (also called heavy model) to produce the final result. The rationale is that certain queries, known as *easy queries*, are inherently simpler and can be processed by small models with no or minor quality degradation. The approach is called *query-aware* as it routes queries based on the complexities of each query.

Model cascades address the computational burden of diffusion models by allowing easy queries to be processed exclusively by light models that execute faster. This time saving from handling easy queries enables the system to achieve higher serving throughput. In addition, our diffu-

[*]Equal contribution [1]University of Massachusetts Amherst, Amherst, MA, USA [2]Adobe Research, San Jose, CA, USA. Correspondence to: Qizheng Yang <qizhengyang@cs.umass.edu>, Sohaib Ahmad <sohaib@cs.umass.edu>.

*Proceedings of the $8^{th}$ MLSys Conference*, Santa Clara, CA, USA, 2025. 

sion model cascade uses a *confidence threshold* to control the proportion of queries classified as easy. By adjusting this threshold, the model cascade offers a way to balance image generation quality with serving throughput. Leveraging the quality-throughput trade-off in this fashion to manage query demand variations is called *model scaling*. Model scaling allows the system to adapt to demand variation and improves hardware utilization by avoiding the need to provision resources for peak demand.

The core idea of query-aware model scaling presents two major research challenges. The first challenge is to develop a discriminator that can *automatically*, *efficiently*, and *accurately* identify easy queries. The discriminator must assess whether a generated image meets quality standards without requiring manual intervention. It also needs to operate efficiently to minimize runtime overhead. Moreover, accuracy in classifying the queries is essential, as routing easy queries to the heavyweight model wastes computational resources, while handling complex queries solely with the lightweight model can degrade the quality of responses. We later show that diffusion model cascades relying on existing quantitative metrics perform no better than random classification due to their inherent limitations in capturing the nuanced image quality differences. The second challenge lies in efficiently allocating resources to optimize performance when serving diffusion model cascades. The key parameter in such cascades, the confidence threshold, must be co-optimized with other system parameters to achieve optimal performance.

This work introduces DIFFSERVE, a system that leverages query-aware model scaling to efficiently serve text-to-image diffusion models. To address the first challenge, DIFFSERVE constructs a diffusion model cascade by training a machine learning (ML) model to assess the quality of generated images. The key insight is that an ML model can be trained to accurately distinguish between images generated by diffusion models and real images. This ML model can be repurposed to differentiate whether the images produced by the lightweight model meet quality requirements based on its classification confidence. To address the second challenge, DIFFSERVE carefully models system performance as functions of key configuration parameters and formulates the resource allocation problem in a mixed integer linear programming (MILP) framework to identify the optimal allocation plan that maximizes response quality while satisfying query demand. DIFFSERVE periodically solves it and re-allocates resources to adapt to varying query demands.

We evaluate DIFFSERVE on three light-heavy diffusion model pairs using both synthetic and real traces and find that it consistently outperforms state-of-the-art systems. Compared to Proteus (Ahmad et al., 2024) which leverages model scaling but randomly routes queries to model variants based on system load, DIFFSERVE can improve system accuracy by up to 20% since its query routing considers their complexities. Compared to serving systems that use a fixed confidence threshold, DIFFSERVE demonstrates up to 24% improvement in quality and 19-70% reduction in SLO violations, owing to its better resource allocation.

We summarize the contributions of this work as follows:

- We introduce query-aware model scaling which constructs diffusion model cascades to optimize the efficiency of diffusion model serving systems.
- We leverage adversarial training to build discriminators that enable the cascading of diffusion model variants.
- We formulate the resource allocation problem as a mixed integer linear programming (MILP) framework to determine optimal configuration parameters when serving diffusion model cascades.
- We implement these techniques in the DIFFSERVE model serving system and evaluate its performance across various workload traces and diffusion models.

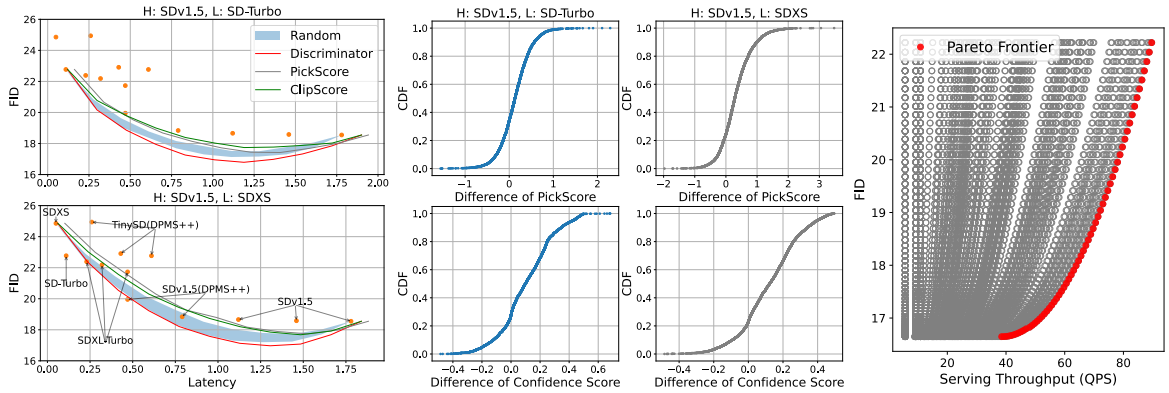
## 2 MOTIVATION AND CHALLENGES

This section motivates query-aware model scaling for serving diffusion models and explains the research challenges.

### 2.1 Background and Motivations

Model developers often train diffusion models with varying architectures and sizes to explore trade-offs between model quality and efficiency. Several quantitative metrics can be used to measure the generated image quality of text-to-image diffusion models, each with its own limitations. We highlight a few. (1) *Fréchet Inception Distance (FID) score* (Heusel et al., 2017): it quantifies diffusion model quality by comparing the distribution of generated images with that of real images given a set of prompts. A lower FID score indicates a better model quality. As FID compares two data distributions, it is not suited for assessing the quality of individual images generated from diffusion models. In this work, we use FID score to quantify the response quality of our serving system given a set of text prompts. (2) *PickScore* (Kirstain et al., 2023): It compares the quality of images generated using the same text prompt. It evaluates alignment relative to each specific prompt, not across varying prompts, making scores incomparable between different prompt-image pairs. We later use PickScore to motivate the existence of easy queries. (3) *CLIP Score* (Hessel et al., 2021): It measures the alignment between a text prompt and a generated image from it. A higher CLIP score indicates better semantic alignment. However, CLIP scores of different model variants can be very close and it does not consistently reflect the image's perceptual quality such as visual realism or aesthetic appeal.

The orange points in Figure 1a illustrate response quality-




(a) FID vs. average inference latency (b) CDF of image quality difference (c) FID vs. serving throughput

*Figure 1.* (a) The quality-latency trade-offs of systems serving independent diffusion models and diffusion model cascades with different discriminator designs with batch size one. The top panel uses diffusion model cascades built with SDv1.5 as the heavyweight model (H) and SD-Turbo as the lightweight model (L). The bottom panel uses SDXS as the lightweight model instead. Lower latency is achieved by using lighter models or treating more queries as easy in diffusion model cascades. FID is the lower the better. (b) The distribution of the difference in image quality between a lightweight model and a heavyweight model. Negative values in the x-axis mean the lightweight model's generated image quality is better than those from the heavyweight model. Top panels use PickScore as the quality metric while bottom panels use confidence score from our proposed discriminator. (c) Illustration of how different resource allocation configurations affect serving throughput (QPS) and response quality (FID). All results use the dataset MS-COCO 2017 (Lin et al., 2014).

latency trade-offs for a system serving diffusion models from HuggingFace (Hugging Face, 2024), each point representing a model variant. These model variants come from off-the-shelf diffusion models with different architectures or the same model executed with a different number of diffusion steps. Serving heavier diffusion models can deliver higher-quality images but comes at the cost of higher inference latency and thus lower serving throughput.

Query-aware model scaling is motivated by the observation that certain "easy" queries can be processed by lighter diffusion model variants without compromising the quality of the generated images. Figure 1b illustrates the distribution of the difference in image quality for two light-heavy diffusion model pairs: SD-Turbo vs. SDv1.5 and SDXS vs. SDv1.5. As we need a quality metric that compares the image quality from the same text prompt, we use PickScore (top panels) and confidence scores from our discriminator (bottom panels). The discriminator design is elaborated in §3.2. The figure shows that, for 20-40% of the queries (i.e., easy queries), the lightweight model generates images with similar or even better quality than the heavyweight model. This observation drives the design of a model serving system that dynamically serves both lightweight and heavyweight model variants, routing easy queries exclusively to the lightweight model to improve system efficiency.

### 2.2 Challenges

**Challenge 1: Model Cascading for Diffusion Models.** One effective approach in identifying easy queries involves constructing diffusion model cascades that combine models of varying sizes, guided by *discriminators* (Viola & Jones, 2001; Bolukbasi et al., 2017). The discriminator outputs a *confidence score* indicating whether the generated image meets a predefined quality standard.

However, designing the discriminator for diffusion models is non-trivial. The curves in Figure 1a show the quality-efficiency trade-offs across various model cascades, each using a unique discriminator design. We compare the following approaches: (1) *PickScore* and *ClipScore*: This design leverages widely used quantitative metrics, CLIP Score and PickScore respectively, to assess image quality. If the generated image's score exceeds a confidence threshold, the query is classified as easy. (2) *Random*: This approach uses a random classifier to assign queries, with each query classified as easy with a given probability. (3) *Discriminator*: This approach uses our discriminator design detailed in Section 3.2. The curves are generated by adjusting the confidence threshold for the PickScore, ClipScore, and Discriminator and the probability for the Random. For the Random classifier, we conducted the experiments 20 times, with the shaded area representing standard deviations across these runs.

Surprisingly, the results show that discriminators based on established metrics, CLIP Score and PickScore, underperform relative to the baseline Random classifier. This counterintuitive finding shows limitations of these metrics in cascading diffusion models, as they fail to reliably differentiate between easy and difficult queries, resulting in suboptimal routing. In contrast, our discriminator design overcomes these challenges and outperforms the Random classifier.

Another surprising observation is that FID gets worse as the latency increases in the end, implying that the overall system response quality can decrease as more queries are routed to the heavyweight model. This is consistent across different light-heavy diffusion pairs. We hypothesize the reason is that including a portion of outputs from the lightweight model yields a more balanced and diverse image representation, which better aligns with the distribution of real images and results in lower FID scores.

**Challenge 2: Resource Allocation.** The second challenge lies in effectively leveraging the time savings in processing easy queries to optimize system performance. A model serving system must co-optimize the confidence threshold with other system parameters to maximize serving throughput and response quality while meeting latency deadlines. Specifically, a higher confidence threshold imposes a stricter constraint on image quality, leading to better system response quality. However, this increased stringency results in a greater proportion of queries being redirected to the heavyweight model, thereby requiring more workers to host it. This increases system loads for a specific query demand and thus decreases the serving throughput of the system. On top of these, the batch size to execute the lightweight and heavyweight model affects both their throughput and the overall end-to-end latency of queries, which need to be adjusted accordingly depending on the system loads.

Figure 1c illustrates how different resource allocation configurations affect serving throughput and response quality. We cascade SD-Turbo and SDv1.5 using our proposed discriminator and serve it on 10 A100 GPUs. We vary three system configurations: the confidence threshold in the model cascade, batch size, and the model placement. Out of all ∼9K possible configurations, we are only interested in those at the Pareto frontier, due to the fact that for a given query demand, configurations at the Pareto frontier yield the highest possible response quality compared to other configurations.

## 3 DESIGN OF DIFFSERVE

We now present the architecture of DIFFSERVE, a diffusion model serving system that leverages query-aware model scaling to efficiently serve test-to-image diffusion models.

### 3.1 Overview

Figure 2 shows the system architecture. It has separate data and control paths. In the data path, clients send queries to the system's Load Balancer, which routes them to suitable workers and returns generated images. In the control path, the Controller periodically re-allocates resources depending on the system runtime statistics collected from the workers.

**Controller.** The Controller manages the resources in the system. It uses the Resource Manager to allocate a model variant to each worker and set its batch size and the confidence threshold for the workers hosting the lightweight model and its discriminator. It periodically collects runtime information from the workers to update model execution profiles as well as the queue lengths and demands seen by each of the workers to inform resource allocation decisions.

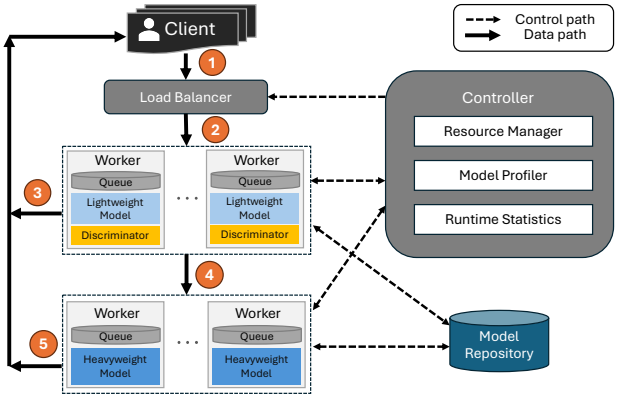



*Figure 2.* System architecture of DIFFSERVE: (1) The query from the client is sent to the load balancer, (2) The load balancer sends it to a worker with the lightweight model and the discriminator, (3) If the confidence score is greater than the threshold, the response is sent back to client, (4) Else, the query is sent to a worker with the heavyweight model, and (5) its output is sent to the client.

**Model Repository.** It manages the registration of diffusion model variants and hosts these registered variants, along with the discriminators used to cascade between them.

**Load Balancer.** The Load Balancer sits on the data path between a client and workers. Upon receiving queries from clients, the Load Balancer initially routes each query to a worker running a lightweight diffusion model. If the generated image's quality estimated by the discriminator meets the quality requirement, specified as a confidence threshold, it is returned to the Load Balancer as the response. Otherwise, the query is forwarded to a worker hosting the heavyweight diffusion model to generate the final response.

**Workers.** Each worker executes its hosted model variant to serve queries routed to it and kept in its local queue. Some workers host the lightweight models together with the discriminators while the rest host the heavyweight models. The batch size, which model variant to host, and the confidence threshold for each worker are determined by the Controller.

We next explain the two technical innovations of DIFFSERVE, the discriminator design for cascading diffusion models (§3.2) and the resource allocation algorithm (§3.3).

### 3.2 Discriminator Design for Model Cascading

At the core of a diffusion model cascade is the discriminator, which evaluates the quality of images generated by a diffu-

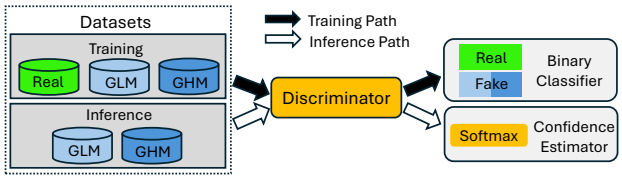



*Figure 3.* Training and inference paths of the discriminator. 'Real' refers to images sourced from real-world high-quality datasets. 'Fake' refers to the generated images from diffusion models. GLM: Generated images from Lightweight diffusion Model; GHM: Generated images from Heavyweight diffusion Model.

sion model to determine whether deferral to a heavyweight model is necessary. The discriminator must be accurate in quality estimation and efficient to minimize runtime overhead. Our approach leverages the insight that an ML model can be trained to accurately distinguish between images generated by diffusion models and real images. This ML model can then be repurposed as the discriminator to differentiate whether the images produced by the lightweight model meet quality requirements based on its classification confidence.

*Discriminator Design.* Figure 3 illustrates the offline training process for preparing the discriminator and the inference process for its use within the diffusion model cascade. The discriminator is trained on a binary classification task to distinguish between high-quality, real-world images (labeled as 'real') and generated images (labeled as 'fake'). Real images are sourced from datasets like MSCOCO (Lin et al., 2014) and DiffusionDB (Wang et al., 2022), while generated images come from both the lightweight and heavyweight diffusion models. This task enables the discriminator to detect visual differences between high-quality real images and generated ones, including variations in sharpness, texture coherence, and artifact presence, equipping it to assess image quality accurately within the cascade framework.

During inference, the discriminator receives the image produced by the lightweight model and outputs a softmax value between 0 and 1, representing the likelihood that the image belongs to the 'real' or 'fake' class. This value is referred to as the *confidence score*, representing how likely the input image is to resemble a 'real' image, implying higher quality. Our implementation uses EfficientNet as the discriminator architecture because it has much lower computation complexity compared to even the lightweight diffusion model variants while achieving high classification accuracy. Section 4.4 evaluates the impact of alternative design choices.

*Model Cascading.* Using the discriminator, cascading a light-heavy diffusion model pair is straightforward by setting a *confidence threshold* to quantify image quality requirements. If the confidence score of an image for a query exceeds the specified confidence threshold, the image is returned to the user, as it meets the quality standard.

### 3.3 Resource Manager

The Resource Manager dynamically adjusts the confidence threshold, allocates models across servers in a cluster, and configures batch sizes to respond to varying query demands. By tuning the confidence threshold, the model serving system leverages the quality-latency trade-off inherent in model cascades to adapt efficiently. For instance, during periods of low demand, a higher threshold prioritizes image quality, while at peak times, a lower threshold ensures latency deadlines are met by allowing minor quality compromises.

This model scaling approach enables a cluster to manage high query volumes gracefully, avoiding overload while tolerating minor reductions in response quality as necessary. Unlike traditional resource provisioning, which typically dedicates resources based on peak demands and results in underutilization during off-peak times, model scaling optimizes resource usage across fluctuating loads. Previous work Proteus (Ahmad et al., 2024) explored model scaling by selecting appropriate model variants to host based on loads. However, model cascades introduce dependencies between model variants for query-aware processing based on query complexities, necessitating a new resource allocation algorithm to manage these dependencies effectively.

Our resource allocation algorithm centers on building performance models for three key performance metrics: *serving throughput*, *latency*, and *response quality*. With these models, identifying the optimal resource allocation can be stated as an optimization problem and formulated within a mixed integer linear programming (MILP) framework, allowing for efficient solutions using MILP solvers. Response quality is directly influenced by the confidence threshold, denoted as $t$. Below, we describe the constraints on latency and serving throughput and then our resource allocation formulation.

*Latency Constraints.* Model dependencies in model cascades introduce complexities in estimating the total time a query spends in the system. To keep query latency within the defined SLO, the Resource Allocation component must consider two main factors: (i) execution latency and (ii) queuing delays. Appropriate batch size settings for each model are essential for managing these latencies. While larger batch sizes increase throughput, they also raise the execution latency for each query within the batch. As the execution time of text-to-prompt diffusion models is highly deterministic, execution latency can be accurately predicted and profiled across different batch sizes.

To estimate queuing delays for each model, we apply Little's law (Shortle et al., 2018), i.e. $W = \frac{L}{\lambda}$, where $W$ is the waiting or queuing time, $L$ is the length of the queue, and $\lambda$ is the query arrival rate. As mentioned in Section 3.1, the Controller maintains a record of the queue length for each worker and the demand seen by each worker. Using these

values, we can estimate the queuing delay for each model.

Mathematically, let $b_1$ and $b_2$ be the batch size used to execute the light and heavy models respectively. let $e(.)$ measures the execution latency and $q(.)$ models the queuing delays. The latency a query experienced in the system should be less than the deadline in SLO requirement $L$:

$$e(b_1) + q(b_1) + e(b_2) + q(b_2) \leq L \tag{1}$$

*Throughput Constraints.* The system's serving throughput can be limited by the throughput of workers hosting either the lightweight or heavyweight model. Mathematically, let $x_i$ represent the number of devices allocated to serve the $i$-th model variant and $T_i(.)$ represents the throughput of a single worker for the $i$-th model variant, collected through profiling. Here, $i = 1$ refers to the light model and $i = 2$ refers to the heavy model. The serving throughput of each model variant in the model cascade must meet or exceed their respective estimated query demands:

$$x_1.T_1(b_1) \geq D, \tag{2}$$
$$x_2.T_2(b_2) \geq D.f(t), \tag{3}$$
$$x_1 + x_2 \leq S, \tag{4}$$

where $D$ is the total estimated query demand entering the system (which is also the demand for the lightweight model), $f(t)$ represents the fraction of queries deferred to the heavyweight model when the confidence threshold is set to $t$, and $S$ denote the total number of devices available to the system. $f(t)$ is initialized through offline profiling and updated during model serving as $t$ changes.

*The resource allocation problem.* We formulate an optimization problem that maximizes the confidence threshold given incoming query demand. The optimization is a mixed integer linear program (MILP) that tunes server count allocated for each model variant ($x_1$, $x_2$) and batch sizes ($b_1$, $b_2$).

$$\max_{x_1,x_2,b_1,b_2} t \tag{5}$$
$$s.t. \quad \text{Constraints Eq. 1 - 4}$$

*Solving the MILP.* The optimization problem is solved periodically by invoking a MILP solver. We estimate query demand $D$ using an exponentially weighted moving average on demand history. To accommodate micro-scale variations in query arrivals, we use $\lambda D$ as the estimated query demand in Eq. 4, where $\lambda$ is the over-provisioning factor and set to 1.05 by default. The time overhead to solve the MILP does not lie on the critical path of query processing as the MILP is called asynchronously and its execution is in the control path. We compare the resource allocation algorithm with alternatives and report its runtime overhead in Section 4.5.

# 4 EVALUATION

This section evaluates the efficacy of DIFFSERVE by answering the following questions: **Q1**: How does the performance of DIFFSERVE compare to alternative approaches on both synthetic traces (§4.2) and real-world traces (§4.3)? **Q2**: How does the model cascading design (§4.4) and the resource allocation algorithm (§4.5) compare to alternative approaches? What is the runtime overhead of the MILP solvers? **Q3**: How do different SLO settings affect the performance of DIFFSERVE (§4.6)?

## 4.1 Experiment Settings

**Implementation of DIFFSERVE.** We implement DIFFSERVE in a simulator and on a testbed cluster. (1) *The simulator-based implementation* consists ∼7K lines of Python code. It uses an event queue and a timer to record the arrival and processing of queries. The execution time of queries for the diffusion models is profiled for offline usage. (2) *The cluster-based implementation* aims to test the performance of DIFFSERVE on actual GPU hardware. We use the HuggingFace (von Platen et al., 2022) and PyTorch frameworks (Paszke et al., 2019) to execute the diffusion models and discriminator for inference. Our cluster consists of 16 NVIDIA A100 GPU workers, and we use gPRC for fast and lightweight communication between system components such as the Controller and Workers. The results we reported in the paper are collected from the simulator unless noted differently. We later show that the results from our simulator closely match the results from our cluster-based implementation, with a slight discrepancy caused by variance in processing queries on actual GPUs. The detailed difference is reported in Section 4.3. We use Gurobi (Gurobi, 2024) to solve our MILP optimization.

**Diffusion Models and Datasets.** We construct model cascades using three light-heavy diffusion model pairs. These different diffusion pairs aim to show the generalizability of the proposed discriminator design and the effectiveness of DIFFSERVE across different model configurations. **Cascade 1:** We use SD-Turbo (Sauer et al., 2023) as the lightweight model and SDv1.5 (Rombach et al., 2022) as the heavymodel. SD-Turbo is a fast generative text-to-image model that can generate an image from a prompt in only one step. We use SDv1.5 with 50 steps. The inference latency to generate one image for a text prompt on A100-80GB for SDv1.5 and SD-Turbo is ∼1.78s and ∼0.1s respectively. We set the SLO for this Cascade to be 5s for our experiments and explore the effect of different SLO values on performance in §4.6. **Cascade 2:** We use SDXS-512-0.9 (referred to as SDXS) (Song et al., 2024) as the lightweight model and SDv1.5 as the heavymodel. SDXS takes ∼0.05s to generate an image from a prompt in one step on an A100-80G GPU. We again use an SLO of 5s for this Cascade. **Cascade 3:**

| Approach | Allocation | Query-aware |
|---|---|---|
| Clipper-Light | Static | No |
| Clipper-Heavy | Static | No |
| Proteus | Dynamic | No |
| DIFFSERVE-Static | Static | Yes |
| DIFFSERVE | Dynamic | Yes |

*Table 1.* Comparison of DIFFSERVE with baselines

We use SDXL-Lightning (Lin et al., 2024) with two steps (∼0.5s to generate an image for a prompt) as the lightweight model and SDXL (Podell et al., 2023) with 50 steps (∼6s for the same generation) as the heavyweight model. As this cascade is heavier, we use an SLO of 15s.

For datasets, we use *MS-COCO 2017* (Lin et al., 2014) for Cascades 1-2 which generate images at a resolution of 512x512, and *DiffusionDB* (Wang et al., 2022) for Cascade 3 which generates images at a resolution of 1024x1024. We select the first 5K text-image pairs from each dataset with text prompts serving as queries and images applied in calculating FID scores for evaluation. Further, we use the Microsoft Azure Functions trace (Shahrad et al., 2020) as a representative real-world workload to drive load on the system. We scale the trace using shape-preserving transformations to match the capacity of our system.

**Evaluation Metrics.** We access system performance using two key metrics. (1) Response quality (**FID**): FID measures the similarity between two distributions – generated images and ground truth images. To compute the FID score for a given system configuration, we process all text prompts in a dataset through the system and evaluate the quality of the generated images. (2) **SLO Violation Ratio**: This metric represents the proportion of queries that fail to meet the SLO latency requirement or are preemptively dropped by the system when they are predicted to miss the deadline. We vary the query demand (in QPS) entering the system and report how changes in demand affect system response quality and the SLO violation ratio.

**Counterparts for Comparison.** We evaluate DIFFSERVE against four approaches. Table 1 highlights their difference.

- **Clipper-Light** and **Clipper-Heavy** are static baselines that route all queries to the lightweight diffusion model and the heavyweight diffusion model, respectively. They implement Clipper (Crankshaw et al., 2017a) to serve these models. Although we use Clipper, this baseline is also representative of other static and query-agnostic model serving systems, such as TensorFlow-Serving (Olston et al., 2017).
- **Proteus** (Ahmad et al., 2024) is a model serving system that dynamically selects models based on changing query demand. However, its query routing strategy does not account for query complexity; instead, it randomly assigns incoming queries to model variants, disregarding the content or difficulty of each query.
- **DIFFSERVE-Static** is a variant of our system that uses a model cascade with a discriminator to estimate query difficulty and route queries appropriately. It is static as it is provisioned for peak and does not adapt confidence threshold to changing system demand. We consider this a practical baseline as it reflects a common practice in production systems where resources are provisioned to accommodate maximum anticipated demand.

### 4.2 Performance Comparison on Synthetic Traces

Figure 4 shows the performance of DIFFSERVE against the baselines on synthetic, static traces for Cascade 1. We vary the load from low to high and observe the effect on all approaches. Under static query demand, DIFFSERVE-Static and DIFFSERVE perform identically, as there is no need for dynamic adjustment of the confidence threshold. To generate multiple performance points under static query demand, we vary the over-provisioning factor (see §3.3) for both the dynamic approaches, i.e., Proteus and DIFFSERVE, to explore the quality-latency trade-off. The static approaches, i.e., Clipper-Light and Clipper-Heavy only offer a single point of performance for each graph since they cannot be tuned to navigate this quality-latency trade-off.

We see that DIFFSERVE offers the Pareto optimal trade-off between the FID (i.e., response quality) and SLO violations across all three levels of load. Although Clipper-Light achieves the smallest SLO violations because it serves the lightweight model for all queries, it suffers from poor response quality. The Clipper-Heavy counterpart offers better quality as it uses the heavyweight model for all queries, but has the highest SLO violations (45.10%-74.11%) across all approaches. Proteus offers a middle-ground performance between the two Clipper extremes by tuning the model variant based on demand. However, since it routes queries to the model variants randomly instead of considering query content, it still offers sub-optimal performance. As we can see from the Figure, DIFFSERVE offers the Pareto optimal between FID and SLO violations as the DIFFSERVE curve lies on the lower left portion of the graph across all loads. We attribute this to the query-aware nature of DIFFSERVE, as it can select the appropriate model for each query based on its content and difficulty. As we show in §4.4, the overhead of estimating the query difficulty is negligible.

Note that DIFFSERVE can even outperform Clipper-Heavy in terms of FID because, as shown in §2.1, the lightweight model offers similar or better quality for 20-40% of the queries. This phenomenon is also reported by (Ding et al., 2024) for LLMs, where small LLMs outperform large LLMs for certain queries. This further emphasizes the importance of query-awareness for model selection.

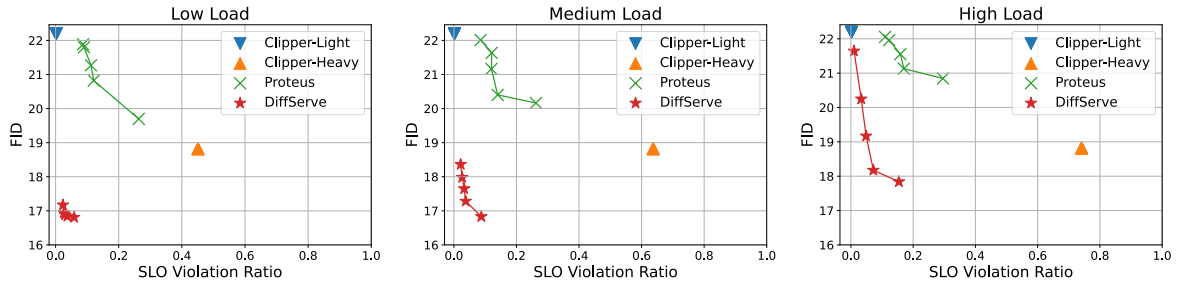



*Figure 4.* Performance comparison on static trace. DIFFSERVE offers Pareto optimal between FID and SLO violations (lower left curve).

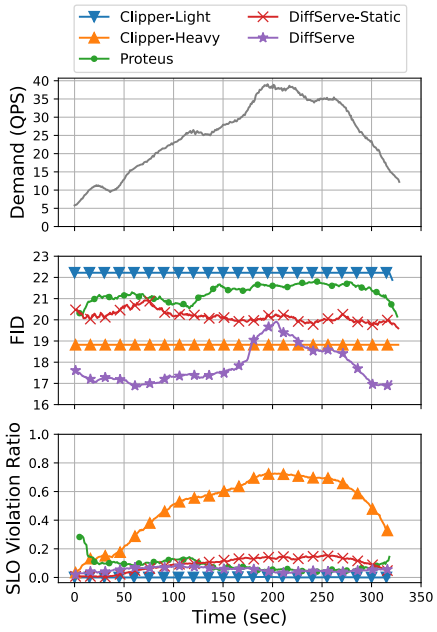



*Figure 5.* Performance comparison on real-world trace for Cascade 1. DIFFSERVE improves quality by up to 23.4% over baselines while maintaining low SLO violations. During peak, it offers similar or better quality than static approaches with significantly lower SLO violations (from 19-70%).

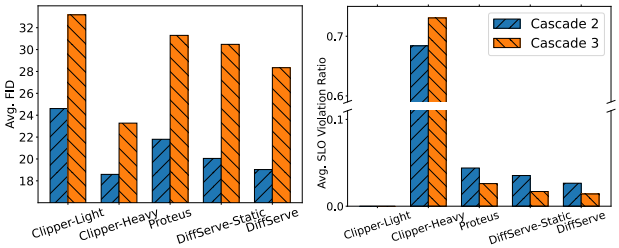


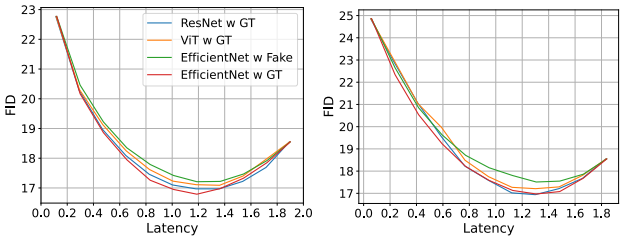

*Figure 6.* Comparison of approaches on the testbed. Across both Cascade 2 and 3, DIFFSERVE achieves an 6%-24% reduction in average FID compared to all baselines, except Clipper-Heavy which yields significantly high SLO violations. DIFFSERVE reduces the average SLO violation by up to 1.4×, 1.9× and 52× compared to DIFFSERVE-Static, Proteus and Clipper-Heavy. Clipper-Light's avg. SLO Violation Ratio is zero.

### 4.3 Performance Comparison on Real Traces

Production systems observe significant variations in demand throughout the day. An approach that dynamically responds to demand variations can offer significant performance improvements over static approaches. We now show how DIFFSERVE performs on a real-world dynamic trace against the baselines.

Figure 5 reports the comparison using Cascade 1. We again observe that Clipper-Light has the lowest quality (highest FID) and low SLO violations due to sending all queries to the lightweight model. Clipper-Heavy offers higher quality (∼15%) but suffers from significant SLO violations at the peak (up to 75%) as the heavyweight model has a long execution time. Proteus dynamically tunes the resource allocation according to demand changes and thus experiences an almost consistent level of quality throughout the trace. However, as it is query-agnostic, its quality improvement over Clipper-Light is minimal (<5%). We also study the performance of DIFFSERVE-Static in this case. As it is query-aware but static, it also experiences a consistent level of quality (FID) throughout the trace. Due to its query-aware nature, it can improve quality over Clipper-Light and Proteus by up to 9%. However, as it cannot change its allocation to accommodate demand changes, it suffers from high SLO violations (up to 19%) during the peak.

DIFFSERVE offers the best performance throughout due to its dynamic resource allocation and query-awareness. During the off-peak, it can significantly improve quality (up to 23.4%) while guaranteeing very low SLO violations. It does so by using the heavyweight model intelligently, only sending it queries that have a low confidence score from the lightweight model, thus ensuring high quality while maintaining low latency. As demand increases and resources get constrained, it keeps SLO violations low by routing more queries to the lightweight model. At peak demand, its FID score is momentarily worse only than Clipper-Heavy, but as




(a) SD-Turbo (b) SDXS

*Figure 7.* Discriminator comparison. EfficientNet trained with ground truth images achieves the lowest FID given latency requirements, outperforming all other approaches across both cascades.

mentioned before, Clipper-Heavy suffers from significantly high SLO violations at this time. Therefore, DIFFSERVE adapts dynamically to the real-world trace and outperforms baselines in terms of both quality and SLO violations. Under imbalanced workloads dominated by easy or difficult queries, while resources would shift toward lightweight or heavyweight models respectively, DIFFSERVE would still balance response quality and SLO violations by tuning the confidence threshold.

Figure 6 presents testbed results for Cascades 2 and 3. As both configurations exhibit similar trends to Cascade 1, we report the average FID scores and SLO violation rates. We observe that DIFFSERVE reduces average FID scores by 6%-24% for Cascade 2 and by 8%-15% for Cascade 3 compared to all other baselines, except Clipper-Heavy, which incurs a high SLO violation (68.4% and 73%). DIFFSERVE outperforms all baselines in terms of SLO violations with $1.4\times$, $1.7\times$, and $26\times$ lower violation ratio than DIFFSERVE-Static, Proteus, and Clipper-Heavy, respectively, for Cascade 2, and $1.2\times$, $1.9\times$, and $52\times$ lower violation ratio for Cascade 3.

We conducted the same experiments on the simulator, observing an average difference of only 0.56% for FID and 1.1% for SLO violations compared to the testbed. This close alignment between the simulator and testbed results confirms the simulator's reliability. Therefore, we use the simulator for the remaining subsections to efficiently evaluate DIFFSERVE across a broad range of scenarios.

### 4.4 Evaluation of the Discriminator Design

We have shown in Figure 1a that using a discriminator surpasses other methods, such as PickScore, ClipScore, and Random selection, for diffusion model cascading. To further illustrate the choice of discriminator design, we examine the effectiveness of using *EfficientNet-V2* as the discriminator in our model cascading architecture by comparing it against several variants of our approach. The variants include: (1) **ResNet w GT**: using ResNet-34 (He et al., 2015) trained with ground truth images as "real" samples, (2) **ViT w GT**: using Vision Transformer (Dosovitskiy et al., 2021) (i.e., ViT_b_16) trained with ground truth images as "real" samples, (3) **EfficientNet w Fake**: using EfficientNet-V2 trained with images generated by the heavymodel as "real" samples, and (4) **EfficientNet w GT**: using EfficientNet-v2 trained with ground truth images as "real" samples, which is the final configuration applied in our paper. We evaluate all the variants across two cascade configurations, where the lightweight models are SD-Turbo and SDXS, respectively, and the heavyweight model is SDV1.5 for both. The ground truth images are taken from the MS-COCO dataset.

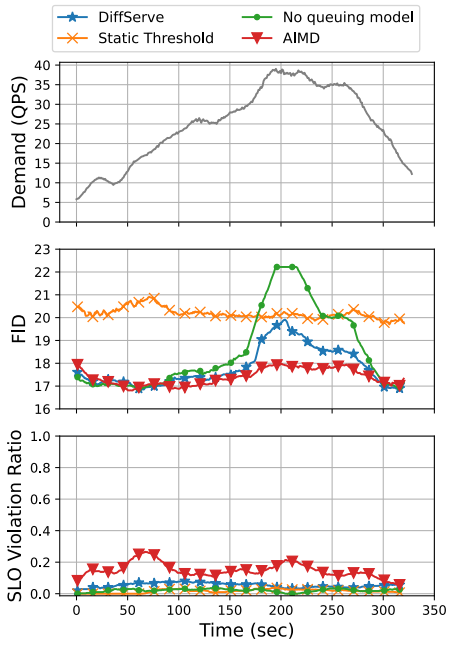



*Figure 8.* Performance of the resource allocation algorithm in DIFFSERVE and alternative approaches. DIFFSERVE reduces SLO violations by 20% and improves quality by up to 19%.

Our results reported in Figure 7 reflect that **EfficientNet w GT** consistently has the highest response quality (i.e., lowest FID scores) given a latency requirement among all the variants of approach, which means it outperforms all the other baselines. The latency of EfficientNet, ResNet, and ViT on an A100 GPU is 10ms, 2ms, and 5ms, respectively, which are negligible compared to the execution time of the diffusion models which are in the order of seconds. EfficientNet's architectural efficiency likely contributes to its superior performance, allowing it to capture complex quality features in images more effectively than the other models. Additionally, using ground truth images as "real" samples for training provides a more robust discriminator that aligns well with human-perceived quality, outperforming the configuration trained with generated images as "real" samples. These findings validate our choice of EfficientNet with ground truth images as the optimal configuration for

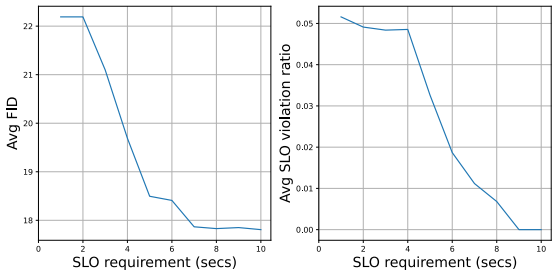



*Figure 9.* Effect of SLO on performance for Cascade 1. DIFFSERVE guarantees low SLO violations and high quality over a broad range of SLO values.

the discriminator in DIFFSERVE, leading to improved accuracy in confidence estimation and overall performance within the cascading architecture.

### 4.5 Evaluation of Resource Allocation

We now show an ablation study of our resource allocation in Figure 8 to understand the effect of various parts of our optimization such as threshold tuning, dynamic batch size selection, and our queuing model, against the following:

**Static threshold.** We fix the threshold to remain static in DIFFSERVE throughout the experiment while letting the optimization tune server allocation and batch size. Note that this approach is different from DIFFSERVE-Static where all the optimization parameters are fixed and provisioned for the peak. As the threshold is fixed, this approach cannot adapt to demand changes, and thus loses out on the quality improvement offered by DIFFSERVE (up to 19%) during off-peak by sending more queries to the heavyweight model.

**AIMD batching.** Instead of letting our optimization tune the batch size, we use a heuristic used by prior work such as Clipper (Crankshaw et al., 2017b): additive-increase multiplicative decrease. This heuristic decreases the batch size by a multiplicative factor upon experiencing an SLO timeout and increases it additively otherwise. As this approach is reactive based on SLO violations as signal, it experiences a significantly higher SLO violation ratio throughout the experiment. DIFFSERVE proactively sets the batch size based on the system demand and available resources, and thus experiences significantly lower SLO violations (up to 20%).

**No queuing model.** Since DIFFSERVE uses a queuing model to estimate the queuing delays, we replace the theoretical model with a heuristic used by prior work such as Proteus to estimate queuing delays by assuming it to be twice the execution delay. This approach works well when query load is low and a query can always be executed in the next batch after it arrives. However, during off-peak times, it experiences significantly lower quality (up to 12%) due to under-estimation of queuing delay. DIFFSERVE avoids this problem by estimating queuing delay based on real-time queue length and query arrival information.

**Overhead of MILP Solver.** We measure the average runtime of the MILP solver to be $\sim$10 milliseconds. This incurs a minimal overhead on the system for periodically changing resource allocation in real-time. Moreover, since the MILP solver does not lie on the critical path of query serving, this overhead does not affect individual queries.

### 4.6 Sensitivity to SLO

Figure 9 explores the effect of different SLO values on the accuracy and SLO violation ratios. We note that DIFFSERVE consistently guarantees low SLO violations and high quality across a broad range of SLO values.

## 5 DISCUSSION

**Scalability of DIFFSERVE.** DIFFSERVE is scalable and can be generalized to various situations. For longer pipelines, DIFFSERVE can be extended by applying a discriminator after each model, with adjustments to the MILP formulation to include multiple confidence thresholds as optimization variables. For higher-resolution image generation, the overhead of the discriminator remains negligible compared to the overall pipeline as the computational complexities of the discriminator and the diffusion models grow with the image resolution. For other models and tasks (e.g., LLMs), the basic technique of model cascading is still applicable with specific quality metrics like BARTScore (Yuan et al., 2021) used in the discriminator. For deploying DIFFSERVE in heterogeneous GPU clusters, a slightly more complex MILP formulation would be required to account for different server classes and model runtimes on each class. Although this adjustment would increase the runtime complexity of the MILP, there is no fundamental limitation that prevents its implementation.

**Reuse Opportunities.** A potential optimization to improve DIFFSERVE is the *reuse* of intermediate outputs from lightweight models during the execution of heavyweight models. This approach compensates for the overhead of both difficulty estimation and the initial execution of lightweight models, as the heavyweight model can directly build upon the results from the lightweight model instead of starting from scratch. While reuse opportunities exist in cascaded inference, they introduce additional complexity in selecting compatible lightweight and heavyweight models, as reuse may even potentially negatively impact the image generation quality of the heavyweight model. For instance, in our experiments, with 50 denoising steps, reusing images from SD-Turbo with SDv1.5 showed no significant change in FID, but reusing those from SDXS increased FID

from 18.55 to 19.75 on the MS-COCO dataset, implying worse image quality. While reuse is feasible, ensuring compatibility between models is critical to maintaining output quality.

**Design of Predictive Router.** While DIFFSERVE relies on a discriminator to assess the quality of generated images for routing decisions, an alternative approach is to use the query itself to make routing decisions before executing any diffusion models. However, predicting image generation quality solely from text inputs is challenging, as image quality is highly dependent on the specific diffusion models used, making it non-trivial to accurately estimate outcomes based on the query alone. Thus, it remains an open question whether a query-based routing strategy would yield better performance in image-generation pipelines.

## 6 RELATED WORK

**Model Serving.** Existing model serving systems generally fall into two categories, neither of which addresses the resource management challenges with query-aware model scaling. The first category requires users to specify the models for inference, leaving the system to handle resource management alone. This category includes production systems like SageMaker (sag, 2020), Triton (tri, 2022), TensorFlow Serving (TensorFlow Authors, 2023), and TorchServe (PyTorch Authors, 2023), as well as academic prototypes such as Nexus (Shen et al., 2019), BATCH (Ali et al., 2020), and Clockwork (Gujarati et al., 2020). These systems are limited in that they do not automatically manage model variants for the same task, providing only a narrow scope of services. The second category improves upon the first by automatically managing both resources and model variants. Works such as Clipper (Crankshaw et al., 2017b), Rafiki (Wang et al., 2018), and Cocktail (Gunasekaran et al., 2022) use model ensembles to enhance response quality. INFaaS (Romero et al., 2021a), Model Switching (Zhang et al., 2020), and Sommelier (Guo et al., 2022) dynamically select model variants based on system load to process queries. Proteus (Ahmad et al., 2024) is the first to formalize and address the resource management problem when serving model variants with different accuracy-efficiency tradeoffs. However, these systems route queries based primarily on system workload, overlooking the optimization potential inherent in varying query difficulties.

**Cascaded Inference.** A cascading architecture for model inference has been explored across various machine learning domains in recent years. CascadeBERT (Li et al., 2021) and Tabi (Wang et al., 2023) speed up inference by cascading progressively larger pre-trained language models, using calibrated confidence scores to decide whether to return results or re-route inputs to complex models. (Shen et al., 2017) introduces a cascade of CNN models with adaptive decision-making for efficient video classification. WILLUMP (Kraft et al., 2020) cascades feature computation by classifying simple inputs with inexpensive key features while routing complex ones to a more powerful model. For generative language models, (Gupta et al., 2024) leverages token-level uncertainty for deferral rules in LM cascades, and FrugalGPT (Chen et al., 2023) proposes a flexible LLM cascade that learns optimal combinations of models to use for different queries. Nonetheless, unlike discriminative inference and language generation tasks, diffusion models lack clear intermediate decision points to evaluate and cascade based on partial outputs or confidence, making these methods incompatible with text-to-image generation pipelines.

**Pipeline Serving.** Works that serve inference pipelines are relevant as model cascades are a type of inference pipeline. InferLine (Crankshaw et al., 2020) minimizes the cost of inference serving by scaling hardware in response to changes in demand. A lot of work has been specifically related to video analytics pipelines such as VideoStorm (Zhang et al., 2017), Scrooge (Hu et al., 2021), Llama (Romero et al., 2021b), and Nexus (Shen et al., 2019). DIFFSERVE differs in its focus on cascading models within the pipeline, using a confidence-based decision process to dynamically switch between lightweight and heavyweight models, balancing both computational efficiency and response quality.

## 7 CONCLUSION

This work proposed DIFFSERVE, an innovative system that optimizes the efficiency of text-to-image diffusion model serving by leveraging query-aware model scaling. DIFFSERVE employs a dynamic approach to model placement, implementing a cascading framework in which an ML-based discriminator routes queries based on their complexity, thereby preserving image generation quality. By framing resource allocation as a mixed integer linear programming (MILP) problem, DIFFSERVE efficiently manages real-time query demand with optimized response quality. Our evaluations across various benchmarks indicate that DIFFSERVE enhances response quality by up to 24% over existing systems while decreasing SLO violations by 19-70%, demonstrating that query-aware scaling is both a robust and flexible solution for diffusion model serving.

## ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under grants CNS-2106463, CNS-1901137, CNS-2312396, CNS-2338512, CNS-2224054, and DMS-2220211. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation. Part of this work is also supported by Adobe gift funding.

## REFERENCES


Twitter streaming traces. https://archive.org/details/archiveteam-twitter-stream-2018-04, 2018.

Amazon sagemaker. build, train, and deploy machine learning models at scale. https://aws.amazon.com/sagemaker/, 2020. Accessed: 2021-06-23.

Triton inference server. https://developer.nvidia.com/nvidia-triton-inference-server, 2022.

Adobe. Adobe firefly. https://www.adobe.com/products/firefly.html, 2024. Accessed: 2024-10-07.

Ahmad, S., Guan, H., Friedman, B. D., Williams, T., Sitaraman, R. K., and Woo, T. Proteus: A high-throughput inference-serving system with accuracy scaling. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, pp. 318–334, 2024.

Ali, A., Pinciroli, R., Yan, F., and Smirni, E. Batch: Machine learning inference serving on serverless platforms with adaptive batching. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–15. IEEE, 2020.

Bolukbasi, T., Wang, J., Dekel, O., and Saligrama, V. Adaptive neural networks for efficient inference. In *International Conference on Machine Learning*, pp. 527–536. PMLR, 2017.

Chen, L., Zaharia, M., and Zou, J. Frugalgpt: How to use large language models while reducing cost and improving performance, 2023. URL https://arxiv.org/abs/2305.05176.

Crankshaw, D., Wang, X., Zhou, G., Franklin, M. J., Gonzalez, J. E., and Stoica, I. Clipper: A low-latency online prediction serving system. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pp. 613–627, Boston, MA, March 2017a. USENIX Association. ISBN 978-1-931971-37-9. URL https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/crankshaw.

Crankshaw, D., Wang, X., Zhou, G., Franklin, M. J., Gonzalez, J. E., and Stoica, I. Clipper: A low-latency online prediction serving system. In *NSDI*, volume 17, pp. 613–627, 2017b.

Crankshaw, D., Sela, G.-E., Mo, X., Zumar, C., Stoica, I., Gonzalez, J., and Tumanov, A. Inferline: latency-aware provisioning and scaling for prediction serving pipelines. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, pp. 477–491, 2020.

Ding, D., Mallick, A., Wang, C., Sim, R., Mukherjee, S., Ruhle, V., Lakshmanan, L. V. S., and Awadallah, A. H. Hybrid llm: Cost-efficient and quality-aware query routing, 2024. URL https://arxiv.org/abs/2404.14618.

Dosovitskiy, A., Beyer, L., Kolesnikov, A., Weissenborn, D., Zhai, X., Unterthiner, T., Dehghani, M., Minderer, M., Heigold, G., Gelly, S., Uszkoreit, J., and Houlsby, N. An image is worth 16x16 words: Transformers for image recognition at scale, 2021. URL https://arxiv.org/abs/2010.11929.

Gujarati, A., Karimi, R., Alzayat, S., Hao, W., Kaufmann, A., Vigfusson, Y., and Mace, J. Serving {DNNs} like clockwork: Performance predictability from the bottom up. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pp. 443–462, 2020.

Gunasekaran, J. R., Mishra, C. S., Thinakaran, P., Sharma, B., Kandemir, M. T., and Das, C. R. Cocktail: A multi-dimensional optimization for model serving in cloud. In *USENIX NSDI*, pp. 1041–1057, 2022.

Guo, P., Hu, B., and Hu, W. Sommelier: Curating dnn models for the masses. In *Proceedings of the 2022 International Conference on Management of Data*, pp. 1876–1890, 2022.

Gupta, N., Narasimhan, H., Jitkrittum, W., Rawat, A. S., Menon, A. K., and Kumar, S. Language model cascades: Token-level uncertainty and beyond, 2024. URL https://arxiv.org/abs/2404.10136.

Gurobi, 2024. URL https://www.gurobi.com/.

He, K., Zhang, X., Ren, S., and Sun, J. Deep residual learning for image recognition, 2015. URL https://arxiv.org/abs/1512.03385.

Hessel, J., Holtzman, A., Forbes, M., Le Bras, R., and Choi, Y. CLIPScore: A reference-free evaluation metric for image captioning. In Moens, M.-F., Huang, X., Specia, L., and Yih, S. W.-t. (eds.), *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pp. 7514–7528, Online and Punta Cana, Dominican Republic, November 2021. Association for Computational Linguistics. doi: 10.18653/v1/2021.emnlp-main.595. URL https://aclanthology.org/2021.emnlp-main.595.

Heusel, M., Ramsauer, H., Unterthiner, T., Nessler, B., and Hochreiter, S. Gans trained by a two time-scale update rule converge to a local nash equilibrium. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, pp. 6629–6640, Red Hook, NY, USA, 2017. Curran Associates Inc. ISBN 9781510860964.

Hu, Y., Ghosh, R., and Govindan, R. Scrooge: A cost-effective deep learning inference system. In *Proceedings of the ACM Symposium on Cloud Computing*, pp. 624–638, 2021.

Hugging Face. Hugging face models hub, 2024. URL https://huggingface.co/models. Accessed: 2024-10-30.

Kirstain, Y., Polyak, A., Singer, U., Matiana, S., Penna, J., and Levy, O. Pick-a-pic: An open dataset of user preferences for text-to-image generation, 2023. URL https://arxiv.org/abs/2305.01569.

Kraft, P., Kang, D., Narayanan, D., Palkar, S., Bailis, P., and Zaharia, M. Willump: A statistically-aware end-to-end optimizer for machine learning inference, 2020. URL https://arxiv.org/abs/1906.01974.

Li, L., Lin, Y., Chen, D., Ren, S., Li, P., Zhou, J., and Sun, X. Cascadebert: Accelerating inference of pre-trained language models via calibrated complete models cascade, 2021. URL https://arxiv.org/abs/2012.14682.

Lin, S., Wang, A., and Yang, X. Sdxl-lightning: Progressive adversarial diffusion distillation, 2024. URL https://arxiv.org/abs/2402.13929.

Lin, T.-Y., Maire, M., Belongie, S., Hays, J., Perona, P., Ramanan, D., Dollár, P., and Zitnick, C. L. Microsoft coco: Common objects in context. In *Computer Vision–ECCV 2014: 13th European Conference, Zurich, Switzerland, September 6-12, 2014, Proceedings, Part V 13*, pp. 740–755. Springer, 2014.

Midjourney. Midjourney. https://www.midjourney.com/home, 2024. Accessed: 2024-10-07.

Olston, C., Li, F., Harmsen, J., Soyke, J., Gorovoy, K., Lao, L., Fiedel, N., Ramesh, S., and Rajashekhar, V. Tensorflow-serving: Flexible, high-performance ml serving. In *Workshop on ML Systems at NIPS 2017*, 2017.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Köpf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., and Chintala, S. Pytorch: An imperative style, high-performance deep learning library, 2019. URL https://arxiv.org/abs/1912.01703.

Podell, D., English, Z., Lacey, K., Blattmann, A., Dockhorn, T., Müller, J., Penna, J., and Rombach, R. Sdxl: Improving latent diffusion models for high-resolution image synthesis. *arXiv preprint arXiv:2307.01952*, 2023.

PyTorch Authors. Torchserve. https://pytorch.org/serve/, 2023. Accessed: 2024-09-26.

Rombach, R., Blattmann, A., Lorenz, D., Esser, P., and Ommer, B. High-resolution image synthesis with latent diffusion models. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 10684–10695, June 2022.

Romero, F., Li, Q., Yadwadkar, N. J., and Kozyrakis, C. {INFaaS}: Automated model-less inference serving. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pp. 397–411, 2021a.

Romero, F., Zhao, M., Yadwadkar, N. J., and Kozyrakis, C. Llama: A heterogeneous & serverless framework for auto-tuning video analytics pipelines. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '21, pp. 1–17, New York, NY, USA, 2021b. Association for Computing Machinery. ISBN 9781450386388. doi: 10.1145/3472883.3486972. URL https://doi.org/10.1145/3472883.3486972.

Sauer, A., Lorenz, D., Blattmann, A., and Rombach, R. Adversarial diffusion distillation, 2023. URL https://arxiv.org/abs/2311.17042.

Shahrad, M., Fonseca, R., Goiri, Í., Chaudhry, G., Batum, P., Cooke, J., Laureano, E., Tresness, C., Russinovich, M., and Bianchini, R. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pp. 205–218, 2020.

Shen, H., Han, S., Philipose, M., and Krishnamurthy, A. Fast video classification via adaptive cascading of deep models. In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 2197–2205, 2017. doi: 10.1109/CVPR.2017.236.

Shen, H., Chen, L., Jin, Y., Zhao, L., Kong, B., Philipose, M., Krishnamurthy, A., and Sundaram, R. Nexus: A gpu cluster engine for accelerating dnn-based video analysis. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pp. 322–337, 2019.

Shortle, J. F., Thompson, J. M., Gross, D., and Harris, C. M. *Fundamentals of queueing theory*, volume 399. John Wiley & Sons, 2018.

Song, Y., Sun, Z., and Yin, X. Sdxs: Real-time one-step latent diffusion models with image conditions, 2024. URL https://arxiv.org/abs/2403.16627.

TensorFlow Authors. Tensorflow serving. https://www.tensorflow.org/tfx/guide/serving, 2023. Accessed: 2024-09-26.

Viola, P. and Jones, M. Rapid object detection using a boosted cascade of simple features. In *Proceedings of the 2001 IEEE computer society conference on computer vision and pattern recognition. CVPR 2001*, volume 1, pp. I–I. Ieee, 2001.

von Platen, P., Patil, S., Lozhkov, A., Cuenca, P., Lambert, N., Rasul, K., Davaadorj, M., Nair, D., Paul, S., Berman, W., Xu, Y., Liu, S., and Wolf, T. Diffusers: State-of-the-art diffusion models. https://github.com/huggingface/diffusers, 2022.

Wang, W., Wang, S., Gao, J., Zhang, M., Chen, G., Ng, T. K., and Ooi, B. C. Rafiki: Machine learning as an analytics service system. *arXiv preprint arXiv:1804.06087*, 2018.

Wang, Y., Chen, K., Tan, H., and Guo, K. Tabi: An efficient multi-level inference system for large language models. In *Proceedings of the Eighteenth European Conference on Computer Systems*, EuroSys '23, pp. 233–248, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9781450394871. doi: 10.1145/3552326.3587438. URL https://doi.org/10.1145/3552326.3587438.

Wang, Z. J., Montoya, E., Munechika, D., Yang, H., Hoover, B., and Chau, D. H. DiffusionDB: A large-scale prompt gallery dataset for text-to-image generative models. *arXiv:2210.14896 [cs]*, 2022. URL https://arxiv.org/abs/2210.14896.

Yuan, W., Neubig, G., and Liu, P. Bartscore: Evaluating generated text as text generation. In Ranzato, M., Beygelzimer, A., Dauphin, Y., Liang, P., and Vaughan, J. W. (eds.), *Advances in Neural Information Processing Systems*, volume 34, pp. 27263–27277. Curran Associates, Inc., 2021. URL https://proceedings.neurips.cc/paper/2021/file/e4d2b6e6fdeca3e60e0f1a62fee3d9dd-Paper.pdf.

Zhang, H., Ananthanarayanan, G., Bodik, P., Philipose, M., Bahl, P., and Freedman, M. J. Live video analytics at scale with approximation and Delay-Tolerance. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pp. 377–392, Boston, MA, March 2017. USENIX Association. ISBN 978-1-931971-37-9. URL https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/zhang.

Zhang, J., Elnikety, S., Zarar, S., Gupta, A., and Garg, S. {Model-Switching}: Dealing with fluctuating workloads in {Machine-Learning-as-a-Service} systems. In *12th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 20)*, 2020.

# A ARTIFACT APPENDIX

## A.1 Abstract

This artifact describes the complete workflow to set up the cluster testbed experiments for DIFFSERVE. We introduce the hardware and software requirements of the cluster testbed. We then describe how to obtain the code, and then describe how to install the dependencies. Finally, we explain how to run the experiments and plot the results.

## A.2 Artifact check-list (meta-information)

- **Algorithm:** Optimization using a trained DNN model as a discriminator, combined with mixed integer linear programming (Gurobi is used).
- **Data set:** Multiple text files included in the repository.
- **Run-time environment:** Linux Ubuntu
- **Hardware:** Multiple powerful GPUs (A100-40G/80G, L40s, etc.) for execution of diffusion model, or multiple Nvidia GPUs (1080Ti, V100, etc.) for simulated execution.
- **Metrics:** Confidence threshold, FID score, SLO violation ratio.
- **Output:** Log files are output by the testbed, which are then used by the plotting scripts to generate graphs.
- **Experiments:** End-to-end experiment of DIFFSERVE for three cascaded pipelines shown in the paper on dynamic traces.
- **How much disk space required (approximately)?:** Approximately 15GB of disk space to download all diffusion models.
- **How much time is needed to prepare workflow (approximately)?:** 30 minutes.
- **How much time is needed to complete experiments (approximately)?:** 10 minutes for each cascaded pipeline.
- **Publicly available?:** Yes. See Section A.3.1 for details.

## A.3 Description

### *A.3.1 How delivered*

The cluster testbed code and workload trace can be accessed at https://github.com/qizhengyang98/DiffServe. It is also accessible at the following DOI: https://doi.org/10.5281/zenodo.14984970.

### *A.3.2 Hardware dependencies*

The DIFFSERVE testbed requires a CPU server and multiple GPU servers.

- **CPU:** a CPU server with 10 cores, 16G RAM for the controller, load_balancer, sink worker, and client processes.
- **GPU:** multiple servers with powerful GPUs (e.g., A100-40G/80G, L40s, etc) to execute diffusion models. The number of servers depends on how many workers are used in the experiments. At a minimum, 4 servers are needed. In the paper, we used 16 servers with A100-80G for 16 workers.

Alternatively, we provide simulated execution in the artifact, which simulates the execution of diffusion models. In this case, any Nvidia GPUs can be used, and multiple workers can be created on a single GPU server by using *tmux* such that the experiments can be done with fewer GPUs.

### *A.3.3 Software dependencies*

- Linux OS, conda environment with Python=3.8, and several Python packages listed in the *requirements.txt*.
- Gurobi optimization software.
- A Gurobi license.

### *A.3.4 Datasets*

We have three cascaded pipelines: *sdturbo&sdv1.5*, *sdxs&sdv1.5*, and *sdxl-lightning&sdxl*. The name of the cascade indicates the lightweight and heavyweight models used. Each cascade uses a specific dataset and trace.

For query content, We use the MS-COCO 2017 dataset for Cascades 1 and 2, and DiffusionDB for Cascade 3. We select the first 5K text-image pairs from each dataset with text prompts serving as queries and the respective images used to calculate FID scores for evaluation. We provide text files in the artifact that are required to run the end-to-end experiments.

For query arrivals, we use the Microsoft Azure Functions trace and scale it using shape-preserving transformations to match the capacity of our system. All traces used in the paper are included in the artifact. The trace files follow the naming format *trace_{A}to{B}qps.txt*, where A and B represent the minimum and maximum query rates in the trace, respectively. For cascade 1 and 2, We used *4to32qps*, while for cascade 3, we used *1to8qps*, on a cluster of 16 GPUs.

## A.4 Installation

To set up the environments, first clone the repository via the link in Section A.3.1. Then, go to the root folder and create a conda environment:

- conda create -n diffserve python=3.8
- conda activate diffserve
- pip install -r requirements.txt

We implement the MILP optimization of DIFFSERVE using Gurobi. Therefore, it is required to obtain a Gurobi license as follows:

- Follow the instructions on the official website to get a commercial or a free academic license for Gurobi.
- Once the license is obtained, Gurobi will provide a *gurobi.lic* file.
- Place the license file under the path *gurobi/gurobi.lic*.

Then, download the pretrained discriminators and image datasets by running "*python prepare_ds_mod.py*" under the conda environment. Note that this step can be skipped if you use simulated execution.

### A.5 Experiment workflow

In this section, we provide instructions on how to execute experiments using the scripts in the artifact. In the following steps, {*Step Num.-R*} means steps for doing real execution of diffusion models, while {*Step Num.-S*} means steps for doing simulated execution.

- **Step 1-R** For preparation, open one terminal on each GPU server and 4 terminals on the CPU server.
- **Step 2-R** In the terminals of the CPU server, run {*tmux new -s contr*}, {*tmux new -s loadb*}, {*tmux new -s sink*}, {*tmux new -s client*}, respectively.
- **Step 3-R** Activate the conda environment in each terminal by running {*conda activate diffserve*}.
- **Step 4-R** To run experiments of cascade-1, in **tmux contr** terminal, run the script *start_controller.sh* which starts the controller process. Copy the IP address printed in the console and replace the original IP address after *-cip* in *start_worker.sh* and *start_worker_sink.sh*.
- **Step 5-R** In **tmux loadb** terminal, run *start_load_balancer.sh* which starts the load_balancer process.
- **Step 6-R** In **tmux sink** terminal, run *start_worker_sink.sh* which starts the sink worker process.
- **Step 7-R** In each terminal of GPU server, modify the number after *-p* in *start_worker.sh*, then run the script. This number is the port number of each worker. Make sure the number you assign to each worker is unique, and is between [50051, 50066]. Note that if you execute the worker for the first time, diffusion models will be downloaded automatically to *models* folder.
- **Step 8-R** If all the processes have been set up successfully, there will be logs corresponding to each process under the folder *logs*, and worker processes will report "Worker is ready" in the console. The logs include *controller*, *load_balancer*, *worker_*{*port number*}, and *model_*{*port number*}.
- **Step 9-R** Then in **tmux client**, run *start_client.sh* to start the client process, which keeps sending queries in 6 minutes. Modify the flag *-trace* given the number of workers you allocate. Use *1to8qps*, *2to16qps*, *2_5to20qps*, *3to24qps*, *4to32qps* if you have ∼4, 8, 10, 12, 16 workers, respectively.
- **Step 10-R** The Client process will report *"Trace ended"* when it stops sending queries. Then stop all the processes.
- **Step 11-R** Under folder *logs*, there will be three *csv* files which contains the end-to-end experiment results. To generate graphs, go to folder *plotting*, and run the script *run_plot_results.sh*.
- **Step 12-R** Be sure to remove all the log files before starting a new experiment.

For simulated execution,

- **Step 1-S** For preparation, open one or multiple terminals on each GPU server and 4 terminals on the CPU server. The total number of terminals on GPU servers should be equal to the total number of workers you want to allocate.
- **Step 2.1-S** In the terminals of the CPU server, run {*tmux new -s contr*}, {*tmux new -s loadb*}, {*tmux new -s sink*}, {*tmux new -s client*}, respectively.
- **Step 2.2-S** In each terminal of the GPU servers, run {*tmux new -s workerX*} respectively, where *X* is a number or character. Make sure *X* is unique to each terminal on a single server.
- **Step 2.3-S** In *start_worker.sh*, add a flag *--do_simulate* at the end of the python command.
- **Step (3-12)-S** Steps 3-12 are the same as Step 3-R to Step 12-R explained above.

The above steps describe the end-to-end experiment flow of cascade 1. To run experiments for Cascades 2 and 3, simply replace the flag *-c sdturbo* with *-c sdxs* and *-c sdxlltn* in all shell scripts in the root folder, then follow the same steps. For Cascade 3, it is recommended to use simulated execution when the number of GPUs is less than 16, as the controller may struggle to find a solution due to insufficient available workers.

### A.6 Evaluation and expected result

The testbed produces log files in the *logs* folder. The log files contain snapshots of the system at regular intervals, including the resource allocation, user demands, system capacity, queries served/dropped/late, and confidence thresholds set given the demand changes.

To generate the graphs from the logs, the Python script *plotting/plot_results.py* can be used to plot the following: confidence threshold over time, SLO violation ratio over time, and FID score over time, which should be similar to the one in Figure 5. The script also prints the average SLO violation ratio and the average FID score, which should be similar to those in Figure 6. The results can vary slightly given different hardware and trace files in use. To generate the graphs, simply modify the flag *--cascade* with [sdturbo, sdxs, sdxlltn] for different cascades, then run the script *run_plot_results.sh*.