DiFlow: A System for Micro-Serving Text-to-Image Diffusion Workflows

Lingyun Yang[*1,2,3], Suyi Li[*#1], Tianyu Feng[1], Xiaoxiao Jiang[1], Zhipeng Di[2], Weiyi Lu[2]
Kan Liu[2], Yinghao Yu[2], Tao Lan[2], Guodong Yang[2], Lin Qu[2], Liping Zhang[2], Wei Wang[1]
[1]Hong Kong University of Science and Technology [2]Alibaba Group [3]Shanghai Jiao Tong University

Abstract

Text-to-image generation executes a *diffusion workflow* comprising multiple models centered on a base diffusion model. Existing serving systems treat each workflow as an opaque monolith, provisioning, placing, and scaling all constituent models together, which obscures internal dataflow, prevents model sharing, and enforces coarse-grained resource management. In this paper, we make a case for *micro-serving* diffusion workflows with DiFlow, a system that decomposes a workflow into loosely coupled model-execution nodes that can be independently managed and scheduled. By explicitly managing individual model inference, DiFlow unlocks cluster-scale optimizations, including per-model scaling, model sharing, and adaptive model parallelism. Collectively, DiFlow outperforms existing diffusion workflow serving systems, sustaining up to 3× higher request rates and tolerating up to 8× higher burst traffic. We have open-sourced DiFlow at https://github.com/diflow-project/diflow.

CCS Concepts: • **Computer systems organization** → **Cloud computing**; • **Software and its engineering** → **Distributed systems organizing principles**; *Software performance.*

Keywords: Diffusion Workflow Serving, Model Serving, Micro-Serving

ACM Reference Format:
Lingyun Yang[*1,2,3], Suyi Li[*#1], Tianyu Feng[1], Xiaoxiao Jiang[1], Zhipeng Di[2], Weiyi Lu[2], Kan Liu[2], Yinghao Yu[2], Tao Lan[2], Guodong Yang[2], Lin Qu[2], Liping Zhang[2], Wei Wang[1]. 2026. DiFlow: A System for Micro-Serving Text-to-Image Diffusion Workflows. In *ACM SIGOPS 32nd Symposium on Operating Systems Principles (SOSP '26), September 29–October 2, 2026, Prague, Czech Republic.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3830418.3843880

[*]Equal contribution.
[#]Corresponding author: Suyi Li (slida@cse.ust.hk).

SOSP '26, Prague, Czech Republic

ACM ISBN 979-8-4007-2585-2/26/09
https://doi.org/10.1145/3830418.3843880

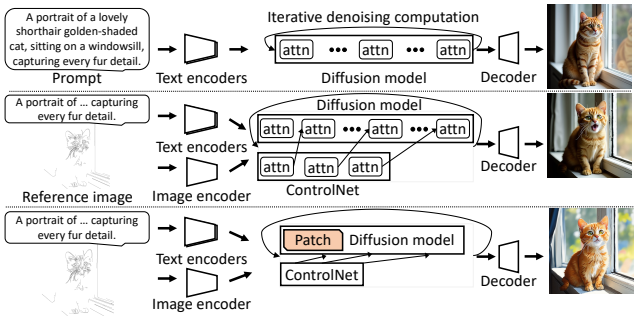

Figure 1. Top: a basic diffusion workflow using Flux-Dev [35]. **Middle**: a *ControlNet* [94] is added alongside the base diffusion model; it processes an additional reference image and passes intermediate features to the diffusion model to control image composition. **Bottom**: a *LoRA* [28] is further added to change image styles by patching the LoRA weights onto the diffusion model's weights.

1 Introduction

Text-to-image (T2I) generation using diffusion models enables the creation of high-quality, contextually accurate images from a textual description [34, 48, 54, 55, 78, 94]. A typical T2I generation *workflow* integrates a *base diffusion model* with multiple *adapter models* to form a pipeline. Execution begins with text encoders that convert prompts into embeddings (Figure 1-top). Conditioned on the embeddings, the diffusion model iteratively generates latent representations via a denoising process, which are subsequently decoded as the final image. To refine visual attributes such as composition or artistic styles, T2I workflows increasingly incorporate adapter models, such as ControlNet [94] and LoRA [28], alongside the base model [39, 42]. By augmenting the diffusion process (Figure 1), these adapters enable fine-grained alignment with user intent and aesthetics [28, 83, 94, 95].

Despite the modular nature of diffusion workflows, existing inference systems, such as HuggingFace Diffusers [16] and ComfyUI [7], primarily employ a *monolithic serving* practice, where the entire workflow, comprising the base model and all associated adapters, is encapsulated into a monolithic instance for provisioning and scheduling. The system treats these workflow instances as opaque black boxes running on a fixed set of GPUs, while remaining oblivious to the workflow's internal model executions and data exchanges.

While operationally simple, monolithic serving introduces fundamental inefficiencies. First, it forces *coarse-grained scaling*: if a single model becomes a bottleneck, the system must replicate the entire workflow. Second, isolated monoliths preclude model sharing. Production traces [39, 42] show that popular diffusion backbones and adapters are frequently reused across workflows. However, monolithic serving forces each workflow instance to maintain its own model copies, leading to redundant memory footprints. Third, treating workflows as opaque black boxes hides internal data dependencies, preventing the system from automatically optimizing resource allocation and pipelining. Finally, tight coupling increases fragility by expanding the failure domain: a failure in one component can disrupt the entire workflow.

To address these limitations, we argue that the schedulable unit for inference should be the individual model execution node, not the entire diffusion workflow. Instead of monolithic instances, the system should decompose workflows into loosely coupled microservices—each encapsulating a specific component like a text encoder, diffusion backbone, or adapter. This *micro-serving* architecture directly addresses the inefficiencies of monolithic serving. First, it enables *fine-grained scaling*: individual models can scale elastically based on real-time demand, eliminating the resource waste of replicating the entire pipeline. Second, it facilitates cross-workflow model sharing: distinct workflows can multiplex shared models, such as a common base model, avoiding redundant memory footprints. Third, by making model execution and data flow explicit, the system regains visibility into the computation graph, enabling *automated optimization* of resource allocation and pipelining. Decoupling also improves modularity and narrows the failure domain: failures can be isolated to affected workflow components rather than disrupting the entire workflow.

However, realizing micro-serving for diffusion workflows presents non-trivial systems challenges. While prior frameworks have successfully applied micro-serving to CPU-centric data analytics [63, 73, 85, 87, 91] and LLM-based agentic workflows [36, 44, 49, 66], applying this paradigm to diffusion workflows requires overcoming hurdles that these systems cannot handle. First, diffusion workflows expose semantics that cross ordinary task boundaries: ControlNet feature maps are consumed at specific model layers during iterative denoising [39, 94] and LoRA adapters mutate base model state through weight patching [28] (Figure 1). These semantics determine when a model execution can start, when intermediate tensors must be available, and whether two executions can safely share model state. Second, decoupling tightly integrated model workflows necessitates fine-grained, latency-sensitive tensor communications across GPUs whose timing follows layer-level data consumption, while their endpoints depend on runtime placement and parallelism. Existing LLM or CPU micro-serving frameworks neither capture these diffusion-specific semantics nor provide an efficient data plane for orchestrating such transfers [41, 57, 66].

In this paper, we present DiFlow, a system for micro-serving diffusion workflows. DiFlow carries diffusion-specific execution semantics through workflow composition, compilation, GPU-resident data movement, and runtime scheduling. This end-to-end integration makes model executions independently schedulable while preserving mid-execution adapter interactions and diffusion-specific optimizations. DiFlow comprises three key designs:

Programming Interface & Compilation. DiFlow provides a Python-embedded domain-specific language (DSL) for composing diffusion workflows. The DSL exposes primitives for model initialization, inference, and diffusion-specific operations for LoRA and ControlNet application [28, 94]. To provide unified support for diverse community-developed models [31], DiFlow wraps each model behind a standardized interface that encapsulates its native loading and inference logic. All primitives enforce strict input/output typing, making data dependencies explicit and catching errors at compile time. A *graph compiler* translates the workflow composition into a directed acyclic graph (DAG) of loosely coupled *workflow nodes*. DiFlow treats each model as an operator: each workflow node executes one model and serves as the unit of independent provisioning and scheduling.

Runtime & Data Plane. At runtime, DiFlow applies *lazy execution*: upon each request, it analyzes the workflow DAG and dynamically recomposes the compute graph—inserting or substituting nodes—to apply diffusion-specific optimizations. Decomposing workflows into distributed nodes, however, introduces high-bandwidth tensor transfers with complex synchronization requirements (e.g., forwarding ControlNet intermediates at specific denoising layers). To handle this, we design a *distributed data engine* atop NVSHMEM [51] that enables GPU-direct, zero-copy tensor movement over high-speed interconnects. The engine provides two fetch modes—*eager* and *deferred*—so that tensors arrive precisely when needed without stalling execution. These mechanisms are transparent to developers: once the compiler produces the workflow DAG, the data engine automatically orchestrates all inter-node tensor transfers.

Workflow Node Scheduling. The DiFlow scheduler maps workflow nodes onto distributed *executors* using three strategies that exploit micro-serving's decomposition. First, it enforces *model-granular scaling*: rather than replicating entire workflows, DiFlow scales only the bottleneck models, avoiding redundant resource provisioning. Second, because a loaded model is workflow-agnostic, the scheduler preferentially dispatches nodes to executors that already hold the required model state, enabling *multi-tenant model sharing*. Third, DiFlow employs *adaptive parallelism*: it dynamically

adjusts model parallelism at scheduling time based on real-time cluster availability, right-sizing resource allocation to maximize throughput without incurring queueing delays.

We prototyped DiFlow and evaluated it across a diverse array of diffusion workflows, encompassing SD3 [19], SD3.5-Large [65], Flux-Dev, and Flux-Schnell [35], along with their respective adapters. Our experiments show that DiFlow's micro-serving architecture significantly outperforms state-of-the-art monolithic serving systems, sustaining up to 3× higher request rates and satisfying 6× more stringent SLOs, and tolerating 8× higher burst traffic, while meeting latency SLOs for over 90% of requests. Crucially, we verify that DiFlow maintains full compatibility with emerging diffusion-specific optimizations, such as approximate caching [4], achieving performance gains consistent with their original monolithic implementations.

2 Background and Problem Statement

2.1 A Primer on Image Generation Workflow

Basic Workflows. As illustrated in Figure 1-top, a basic text-to-image (T2I) generation workflow consists of three models: a *text encoder*, a *base diffusion model*, and a *decoder-only variational autoencoder* (VAE). The process begins with the text encoder, which encodes a text prompt into a sequence of semantic token embeddings. The system then initializes a latent tensor with random Gaussian noise. Conditioned on the text embeddings, the base diffusion model iteratively refines this tensor through a series of denoising steps. Finally, the denoised latent representation is passed to the VAE decoder, which reconstructs the output image in pixel space.

Workflows with Adapters. To achieve fine-grained control over visual attributes, such as spatial structure, diffusion models are frequently augmented with *adapter models* [28, 39, 83, 93–95]. These adapters can be categorized into two classes based on their execution patterns [39]:

1) Parallel Execution Adapters. The first class includes adapters that operate in tandem with the diffusion model during inference, such as ControlNet [94] (Figure 1-middle). From a systems perspective, they introduce two complications: (1) their parameter sizes are often comparable to the base model, introducing substantial model loading latency; and (2) maximizing throughput often requires parallelizing the adapter and base model across GPUs, which necessitates intricate synchronization and data transfer patterns (§2.2).

2) Weight-Patching Adapters. The second class adapts the base model through parameter-efficient fine tuning, such as LoRA [28] and IC-Light [95] (Figure 1-bottom). These adapters patch the base model's weights before inference, incurring no additional computational overhead during subsequent denoising steps. The trade-off is state management: once patched, a diffusion model replica is specialized to a specific request until its weights are restored or replaced. Serving such workflows therefore requires fetching adapter weights from remote storage on demand [39], which can bottleneck loading and complicate sharing model replicas across requests.

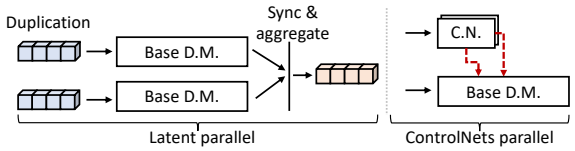

Figure 2. Latent parallelism and ControlNet parallelism.

Workflow Diversity in Production. Existing production traces contain tens of thousands of diffusion workflows and show that workflow diversity arises along two primary axes: base-model family and adapter composition [39, 42]. Adapter-driven variants are especially prevalent [42], including workflows with parallel-execution adapters such as ControlNet and weight-patching adapters such as LoRA. Recent Alibaba production traces [39] demonstrate this diversity at scale, reporting 31,133 distinct workflows over 20 days.

Data Dependencies in Diffusion Workflows. Diffusion workflows exhibit intricate data dependencies induced by performance-oriented parallelization strategies. These strategies improve performance [20, 37, 39, 45], but they also introduce model-specific data transfers and synchronizations that monolithic systems struggle to express or optimize (§4.3). We highlight three common cases:

1) Latent Parallelism. Diffusion models typically use classifier-free guidance (CFG) [27] to improve image quality by executing two denoising passes at each step: one conditioned on the prompt and one unconditional. Latent parallelism accelerates CFG by parallelizing these two computations on separate GPUs [20, 37, 39, 45]. However, this approach introduces frequent "scatter-gather" synchronizations, where partial results must be aggregated at *every* denoising step (Figure 2). These high-frequency communication barriers can erode the benefit of parallelism if not handled efficiently.

2) ControlNet Parallelism. To reduce the overhead of ControlNets, serving systems often execute them in parallel with the base diffusion model on separate GPUs [39] (Figure 2). This design introduces fine-grained data dependencies: ControlNet feature maps must be transferred to, and consumed by, specific layers of the base model during each denoising step. The exact communication pattern depends on the base model and becomes even more complex when multiple ControlNets are used, producing fan-in/fan-out transfers that are difficult to schedule efficiently.

3) Asynchronous LoRA loading. In production systems, LoRA adapters are often stored remotely and must be fetched on demand [39]. To hide this fetching cost, *asynchronous LoRA loading* overlaps adapter retrieval with the early stages of base-model inference. When the LoRA weights arrive, the

system must pause execution, hot-patch the base model in GPU memory, and then resume computation [39]. This optimization introduces *non-deterministic timing* and *dynamic state mutation*: execution now depends on I/O completion, forcing the system to coordinate mid-inference weight updates without incurring synchronization stalls.

2.2 Monolithic Serving and Its Inefficiency

Existing diffusion serving and acceleration systems largely operate at the granularity of an entire diffusion workflow. Diffusers [16, 69], vLLM-Omni [68], SGLang-Diffusion [61], and Cornserve [11] retain Diffusers-style monolithic execution [17] for diffusion workflows, while ComfyUI's node graph [12, 13] is still served as one workflow unit. DistriFusion [37], xDiT [20], Katz [39], and TetriServe [45] add diffusion-specific parallelism, but primarily embed it in monolithic Diffusers-style workflows. Consequently, they provision resources and schedule execution at the granularity of the entire workflow, without managing internal model invocations and data flows. While *monolithic serving* simplifies deployment, it has four fundamental limitations:

L1: Inefficient Scaling via Full Replication. Monolithic serving treats the entire workflow as a scaling unit, enforcing coarse-grained replication regardless of which component is the actual bottleneck. This *indiscriminate scaling* is particularly costly for diffusion workloads, where the base diffusion model is typically the sole bottleneck under load spikes. In standard pipelines [19, 35, 56], the full workflow footprint is often 1.7× to 4× larger than the base model alone. Consequently, scaling the entire monolith incurs significant overhead: our experiments on NVIDIA H800 GPUs reveal that monolithic replication using Diffusers [69] adds up to 80% in loading latency and wastes up to 75% of GPU memory compared to scaling only the bottlenecked component. Similarly, with vLLM-Omni [68] and SGLang-Diffusion [61], scaling an entire Flux-Dev pipeline adds up to 70% and 75% latency, respectively, compared to scaling only Flux-Dev model.

L2: Inability to Share Common Models. Monolithic serving enforces strict isolation between workflow instances [7, 16], precluding model sharing. This design is fundamentally inefficient given that production T2I workloads exhibit highly skewed model popularity. Alibaba's trace analyses [39, 42] indicate that popular backbones (e.g., SDXL [56], SD3 [19], and Flux-Dev [35]) appear in nearly all workflows, while the top 5 ControlNets serve 95% of generation requests. Under monolithic serving, each workflow instance must maintain independent replicas of these massive models (2–24 GiB in FP16 [39]). This redundancy prevents memory multiplexing, resulting in excessive GPU memory consumption, low GPU utilization, and load imbalance across replicas.

L3: Runtime Inefficiency. By encapsulating workflows as opaque black boxes, monolithic serving eliminates system-level visibility into internal model dependencies, data flow, and execution logic. This opacity compels the system to enforce rigid, workflow-level resource allocation, forfeiting opportunities for fine-grained runtime optimization. Specifically, because models within a workflow exhibit heterogeneous arithmetic intensities and distinct latency–throughput trade-offs (Figure 3-right) [58], a static, per-workflow configuration is inherently suboptimal. Furthermore, monolithic systems typically enforce a *fixed degree of model parallelism*. Unlike automatic tuning strategies [40, 66], this static approach prevents the system from adapting to dynamic workloads or fluctuating GPU availability, leading to performance degradation as quantified in §3.1 (Figure 4-right).

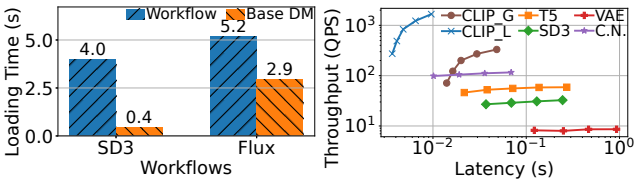

Figure 3. Left: Loading time of workflow scaling and base diffusion model (DM) scaling. **Right**: Latency-throughput tradeoff of models in an SD3 workflow. Both use H800 GPUs.

L4: System Fragility and Maintenance Overhead. Monolithic serving imposes high maintenance overheads by *violating modular systems principles*. The tight coupling of independent components creates a workflow-wide failure domain, so a failure in one component can disrupt the whole workflow. This coarse failure boundary complicates debugging, forcing developers to check the entire monolith to identify root causes. Besides, under the monolithic architecture, updating a single component necessitates holistic validation and coordination across the entire workflow, unnecessarily prolonging development and deployment cycles.

3 A Case for Micro-Serving

We advocate *micro-serving* diffusion workflows. Instead of treating an entire workflow as a schedulable unit, micro-serving decomposes the workflow into independently managed model-execution components and gives the serving system *per-model* control over scaling, sharing, and runtime configuration. This design effectively addresses the inefficiencies of current monolithic serving systems (§3.1). However, it introduces new challenges for diffusion workflows (§3.2).

3.1 Benefits of Micro-Serving

Per-Model Management. Micro-serving makes each model, rather than the entire workflow, the unit of management. This lets the serving system scale only the bottlenecked model and choose resources according to each model's latency–throughput tradeoff, directly addressing **L1** and part of **L3**. As a result, the system avoids replicating non-bottleneck components, reduces model-loading overhead, and better matches heterogeneous models to available hardware. In

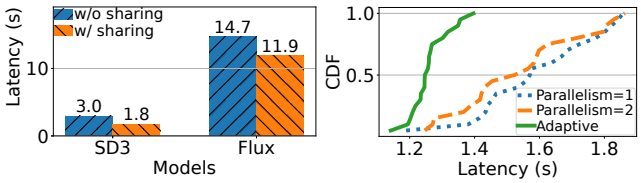

Figure 4. Left: Model sharing reduces request latency. **Right:** Adaptive parallelization reduces request latency.

Figure 3-left, we compare full-workflow scaling with scaling only the base diffusion model on H800 GPUs. Because the diffusion model is the bottleneck, full replication loads other components unnecessarily. Scaling only the diffusion model therefore reduces scaling latency by up to 90%.

Model Sharing. When different workflows invoke common models [39], micro-serving lets the system share those loaded replicas across workflows, directly addressing **L2**. Instead of binding a model replica to the workflow that loaded it, the system can multiplex compatible requests onto any resident replica. This reduces redundant replicas and improves load balance, since requests can use identical models already loaded elsewhere in the cluster. To show these benefits, we serve a pair of workflows on two H800 GPUs: one with ControlNet and one without. This setup creates model-sharing opportunities for the text encoders and diffusion models. In Figure 4-left, we compare request latency with and without model sharing for such workflow pairs, using SD3 and Flux as the base diffusion model in separate experiments. Compared with isolated workflow replicas, multiplexing already-loaded models reduces request latency by up to 40% and GPU memory footprint by up to 60%. Furthermore, base diffusion models patched with adapters (e.g., LoRA, see §2.1) can still be shared across requests requiring different LoRAs through efficient patch swapping [39]. We elaborate on this in §7.3.

Adaptive Resource Configuration. Micro-serving exposes model dependencies and execution choices to the runtime, which lets the system tune resource configurations per model instead of fixing one for the entire workflow. This directly addresses **L3**. Automatic model parallelism is one example. We deploy three SD3 workflows [19] on four H800 GPUs under three settings: *Parallelism=1* fixes the parallelism degree at 1 and leaves acceleration opportunities unused; *Parallelism=2* always applies latent parallelism proposed in [20, 39], which speeds up image generation without quality loss but requires a pair of GPUs (§2.1); and *Adaptive* selects the parallelism degree at runtime according to GPU availability. As shown in Figure 4-right, the tradeoff is clear: *Parallelism=1* yields consistently higher latency because it forgoes parallel speedup, whereas *Parallelism=2* introduces queueing when later requests wait for an available GPU pair, producing a stepped CDF curve. Compared with the static configurations, *Adaptive*'s automatic parallelization tuning accelerates average request serving by 1.3× and 1.2×, respectively.

Modular Development. Micro-serving also improves modularity, directly addressing **L4**. By serving workflow models as independent components, it gives developers clearer failure boundaries. They can modify, validate, and debug one model without reasoning about the entire workflow, and they can test individual components in isolation. This reduces maintenance overhead and makes bugs easier to localize. This modularity isolates component failures and limits their impact to affected portions of a workflow.

3.2 Challenges of Micro-Serving

Despite these benefits, *micro-serving* diffusion workflows cannot be built by directly reusing existing microservice systems for analytics, general task runtimes, or LLM agents [15, 41, 49, 63, 66, 73, 85, 87, 91]. Diffusion workflows couple heterogeneous models through iterative denoising, adapter patching, and diffusion-specific parallelism, creating requirements these systems do not target.

First, the system needs an abstraction that is expressive for developers yet structured for backend analysis. Analytics DAG systems such as Spark [87] and generic task runtimes such as Ray [49] can express tasks and dependencies, but not adapter–base-model interactions, deferred tensor dependencies, or patching operations (§2.1).

Second, the system must compile a workflow into executable model-level tasks while preserving diffusion-specific optimizations. LLM-centric systems, such as Parrot [41] and Ayo [66], primarily optimize autoregressive inference through prefix sharing and streamed decoding [97]. Diffusion workflows instead require a compiler that preserves denoising structure and supports caching, asynchronous LoRA loading, and specialized multi-GPU parallelization [4, 20, 37, 39, 45].

Third, the runtime must be GPU-native and diffusion-aware. Data analytics and microservice systems largely assume CPU execution and host-memory dataflow [15, 63, 73, 85, 87, 91], whereas diffusion workflows exchange `CUDA` tensors across GPUs, rely on asynchronous and interleaved data movement, and require a dedicated data engine.

Finally, micro-serving only pays off if the scheduler can exploit diffusion-specific opportunities at runtime. Existing systems do not directly support model-granular scaling, cross-workflow model sharing, adaptive parallelization, or SLO-aware admission control. These challenges drive our design of a diffusion-specific programming interface, graph compiler, runtime/data engine, and scheduler in §4 and §5.

4 System Design of DiFlow

In this section, we present DiFlow, an efficient serving system for *micro-serving* diffusion workflows. DiFlow comprises four components: a programming interface for workflow composition and model integration (§4.1), a graph compiler that decomposes workflows into executable nodes and applies diffusion-specific optimizations (§4.2), a runtime with

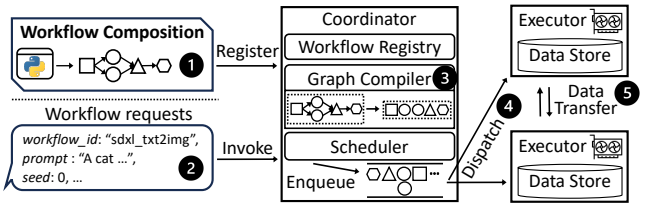

Figure 5. An overview of DiFlow.

Class	API	Description
Model	`__init__()`	Create a model instance
	`__call__()`	Express a model invocation in the frontend
	`setup_io()`	Define model inputs and outputs
	`load()`	Load the model in the backend
	`execute()`	Execute model inference in the backend
	`add_patch()`	Attach a patchable adapter to a model
	`rm_patch()`	Remove a patchable adapter from a model
Workflow	`__init__()`	Create a workflow instance
	`add_input()`	Add a workflow input placeholder
	`add_output()`	Add a workflow output placeholder

Table 1. Primitives in DiFlow's Python library for defining models and composing diffusion workflows.

a GPU-native data engine for workflow execution (§4.3), and an orchestrator for scheduling and resource management (§5). Together, these components optimize cluster-level serving performance while accommodating existing model acceleration techniques [30].

System Overview. Figure 5 presents an overview of DiFlow. At the frontend, *model developers* integrate individual models by subclassing a `Model` base class, and *workflow developers* compose these models into workflows and register them with the system (❶). End users invoke registered workflows (❷) by submitting requests with inputs such as textual prompts and random seeds.

At the backend, the graph compiler transforms a workflow into a set of loosely coupled workflow nodes (❸), which are dispatched by the scheduler across a cluster of distributed executors (❹). Each executor owns one GPU and uses an efficient data engine for inter-node communication (❺).

4.1 Programming Model for Developers

DiFlow exposes two programming-interface layers for distinct roles. The *model-integration layer* targets *model developers*, who subclass `Model` to define a model's typed I/O, loading, and execution logic without reasoning about its composition into workflows. The *workflow-composition layer* targets *workflow developers*, who express arbitrary workflows through model invocations, Python control flow, and diffusion-specific operations such as adapter attachment. Thus, workflow developers specify the intended computation and data flow, while DiFlow automatically derives the backend workflow DAG and logical data-transfer plan (§4.2). We evaluate the usability of these interfaces in §7.3.

Model Integration. To keep pace with the proliferation of models and acceleration techniques [30], DiFlow provides a `Model` base class that standardizes model and adapter integration while encapsulating all workflow-facing logic.

A model developer subclasses `Model` and implements three methods (Table 1): `setup_io()` declares the model's typed inputs and outputs, which are visible to the compiler; `load()` initializes the model on a given device; and `execute()` runs inference. Because these are the only methods a model developer writes, model integration is decoupled from workflow construction: a model developer never reasons about how the model is wired into a larger workflow.

The base class handles workflow integration automatically. Its `__call__()` method records each model invocation as a workflow node and derives data dependencies from the I/O interface declared in `setup_io()`. This separation captures a key design principle: model developers specify what a model consumes and produces; DiFlow uses that specification to place the model into an inferred workflow graph.

Figure 6 illustrates this split with a simplified Flux integration: the base `Model` class (top) is provided by the framework, while the `Flux` subclass (bottom) contains only model-specific code.

Workflow Composition. Workflow developers declare workflow inputs and outputs and express model invocations, Python control flow, and diffusion-specific operations using ordinary Python syntax. They choose the models, operations, and data flow that define workflows; they need not separately construct backend DAG nodes and edges or specify inter-node data transfers. The graph compiler (§4.2) records these invocations, resolves their dependencies and the typed I/O declared in `setup_io()`, and applies optimization rewrites.

Figure 7 shows a workflow for the Flux example in Figure 1-bottom. Creating a `Workflow` instance (line 2) establishes a scope (maintained by `WorkflowContext`); subsequent model calls within that scope are automatically recorded as workflow nodes. Workflow inputs are declared using `add_input()`, and intermediate values like `prompt_embeds` flow directly between model calls. The loop (line 23) shows that iterative denoising is expressed naturally, while DiFlow captures the structure needed for backend execution. After construction, the workflow developer registers the workflow with DiFlow for later invocation by end users.

4.2 Graph Compiler

The graph compiler (❸) lowers a registered workflow into a topologically sorted DAG of schedulable workflow nodes and applies optimization passes before execution.

DAG Construction. As described in §4.1, each model invocation during workflow composition is recorded as a workflow node with typed I/O declared in `setup_io()`. The compiler resolves data dependencies among these nodes and produces a topologically sorted DAG. For every dependency,

```python
## No need to modify it, invisible to model developers ##
class Model:
  def __init__(self, **kwargs):
    self.setup_io()
    # store associated weight-patching adapters
    self._patches = []

  # Make the class callable to create a workflow node
  def __call__(self, **kwargs):
     workflow = WorkflowContext.get_current_workflow()
     workflow_node = WorkflowNode(op=self, **kwargs)
     workflow.add_workflow_node(workflow_node)
     return workflow_node.get_outputs()

  @abstractmethod
  def setup_io(self):
    pass

  def add_patch(self, patch):
    self._patches.append(patch)

  def rm_patch(self, patch):
    self._patches.remove(patch)

## Model developers start here ##
class Flux(Model):
  def setup_io(self):
    # define inputs
    self.add_input("latents", torch.Tensor)
    self.add_input("prompt_embeds", torch.Tensor)
    # define "deferred" inputs, detailed in Sec. 4.3.2
    self.add_input("controlnet_inputs", torch.Tensor,
     deferred=True)
    # define outputs
    self.add_output("noise_pred", torch.Tensor)

  def load(self, model_path, device):
    model = SD3Transformer2DModel.from_pretrained(
        ...
    ).to(device)
    return {"transformer": model}

  @torch.no_grad()
  def execute(self, model_components, **kwargs):
    transformer = model_components["transformer"]
    noise_pred = transformer(**kwargs)[0]
    return {"noise_pred": noise_pred}
```

Figure 6. A simplified case of integrating Flux with DiFlow.

it also records the producer, consumer, and eager or deferred fetch mode, forming the logical transfer plan consumed by the data engine (§4.3.2). Topological order guarantees correct execution and exposes optimization opportunities: nodes without mutual dependencies can run in parallel to reduce latency, and nodes that invoke the same model can be batched to improve throughput (§4.3).

Optimization Passes. After constructing the DAG, the compiler applies a series of graph-rewriting passes. Each pass pattern-matches on node properties (e.g., model type, adapter attachments) and may insert, remove, or replace nodes. This design makes the compiler extensible: adding a new optimization requires only a new pass, without modifying the core lowering logic. The compiler also applies per-model optimizations such as `torch.compile()` within individual

```python
# create a workflow instance
workflow = Workflow(name="flux_txt2img_workflow")
# initialize models. All inherit the Model class
latents_generator = LatentsGenerator()
text_enc = FluxTextEncoder(model_path=model_path)
flux = Flux(model_path=model_path)
controlnet = ControlNet(model_path=controlnet_path)
lora = LoRA(model_path=lora_path)
vae = FluxVAE(model_path=model_path)
# initialize input placeholders for the workflow
seed = workflow.add_input(name="seed", data_type=int)
prompt = workflow.add_input(name="prompt", data_type=str)
num_denoising_steps = workflow.add_input(name="
     num_denoising_steps", data_type=int)
ref_image = workflow.add_input(name="ref_image",
     data_type=Image)
# add_patch() registers a LoRA adapter with the flux,
flux.add_patch(lora)
# Invoke models and establish their I/O dependencies.
# Model invocation is expressed via __call__().
latents = latents_generator(seed)
prompt_embeds = text_enc(prompt)
ref_image = vae(image=ref_image, mode="encode")
# Perform iterative denoising computation.
for i in range(num_denoising_steps):
    controlnet_outputs = controlnet(latents,
     prompt_embeds, ...)
    noise_pred = flux(latents, prompt_embeds,
     controlnet_outputs, ...)
    latents = denoise(noise_pred, latents)
output_img = vae(latents, mode="decode")
workflow.add_output(output_img, name="output_img")
```

Figure 7. A simplified diffusion workflow using Flux [35].

nodes. Below, we illustrate two diffusion-specific passes and evaluate them in §7.4.

1) Approximate caching [4] reduces the number of denoising steps by initializing from a pre-cached image of a similar prompt instead of random noise (§2.1). When a prompt cache is configured, the compiler replaces the random-latent-initialization node with a cache-lookup node, requiring no changes to the workflow definition.

2) Asynchronous LoRA loading [39] overlaps LoRA adapter loading with the early stages of diffusion model inference (§2.1). When the compiler detects an `add_patch()` attachment on a model, it can rewrite the workflow graph by inserting (1) an initial node that triggers asynchronous LoRA loading and (2) a check node after each diffusion-model node that tests whether the adapter is ready to be patched in. The workflow developer writes only `add_patch(lora)`; the compiler can insert the asynchronous-loading machinery.

4.3 DiFlow's Runtime

4.3.1 Micro-Serving Control Plane. Given the topologically sorted DAG from the compiler, the runtime executes each request through a node-level control plane. Under *micro-serving*, each workflow node is independently schedulable on any executor once its non-deferred inputs are satisfied; the runtime can also configure parallelism and resource allocation per node based on the node's encapsulated model and available hardware.

Request Execution Lifecycle. Workflows are compiled once at registration time; the compiled DAG is instantiated only when a request arrives [3, 50, 74, 87] (❷ in Figure 5). The control plane enqueues all root nodes (those with no upstream dependencies) and enters a dispatch loop. In each cycle, the scheduler selects ready nodes and dispatches them to executors (❹). When an executor completes a node, it reports the result to the control plane, which marks downstream nodes whose inputs are now satisfied as ready. This loop continues until all nodes complete and the workflow output is returned to the end user. The scheduler's placement and batching policies are detailed in §5.

4.3.2 Distributed Data Engine.

Micro-serving introduces frequent data movement between nodes. In typical diffusion workflows such as SD3 [19] and Flux models [35], CUDA tensors account for over 99% of transferred data (see Fig. 12-right); for example, an SDXL workflow with a single ControlNet transfers 5.3 GiB [39]. Host-memory staging through PCIe is prohibitively slow at this scale.

To avoid CPU staging, DiFlow deploys a distributed data engine with a per-executor local data store (Figure 5). The stores are built on NVSHMEM [51], which provides one-sided GPU communication over NVLink and RDMA, enabling zero-copy sharing within an executor and high-speed transfers across executors.

Data Fetch Modes. Diffusion workflows exhibit diverse data-movement patterns due to their parallelization strategies (§2.1). DiFlow supports two fetch modes: *eager*, where an input must be ready before a node begins execution, and *deferred*, where a node starts execution and fetches the input at the point of consumption.

1) Eager data fetch. By default, inputs are fetched eagerly: a node cannot begin until all its eager inputs are available. In Figure 8, the text-encoder node on Executor 1 produces a prompt embedding (①) and places it in its local data store (②). When the coordinator schedules the downstream ControlNet node on Executor 2, it forwards the embedding's metadata (③). Executor 2 uses this metadata to fetch the tensor into its own store (④). The ControlNet node then reads the embedding from its local store (⑤) and begins execution—only after the fetch completes, a guarantee enforced by eager fetching.

2) Deferred data fetch. Model developers can mark an input as *deferred* (line 32 in Figure 6), indicating that it need not be ready when a node starts inference. A deferred input is implemented as a fetch function invoked at the point of consumption: it returns immediately if the data is available, or blocks until the data arrives.

This mode is tailored to diffusion workflows where ControlNet computation is interleaved with the base model (Figure 1-middle). At runtime (Figure 8), the ControlNet output consumed mid-way through Flux inference is marked as a deferred input (dashed tensor, ⑥). Flux begins execution without it. When Flux reaches the consumption point, the ControlNet output has been produced and placed in Executor 2's store (⑦); its metadata is forwarded to Executor 1 (⑧), which fetches the tensor into its local store (⑨) for Flux to consume. Without deferred fetching, Flux could not start until ControlNet completes, eliminating the parallelism between the two models.

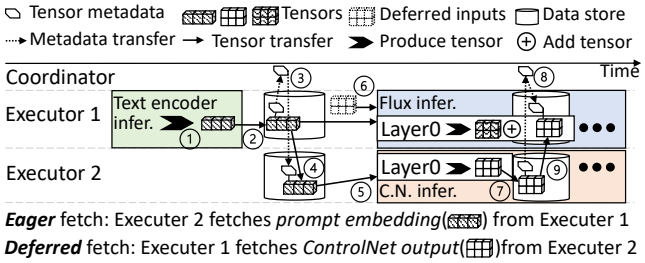

Figure 8. Data fetching in DiFlow. For clarity, we primarily illustrate tensor fetch process in data store. **C.N.**: ControlNet.

Note that tensor metadata, including a tensor's pointer, is tiny (on the order of KiB). Executors can piggyback it on node-completion notifications, allowing the coordinator to track global tensor placements with little overhead. Although illustrated with ControlNet, deferred fetch is a general asynchronous-consumption primitive: any node whose execution prefix does not require an input can start and fetch that input only at its first use. The same abstraction can express asynchronous loading [39], cache hit/miss paths in [4], and future adapter-parallel designs.

Design Properties. The data engine is transparent to both model developers and workflow developers: once the compiler lowers a workflow into nodes, the engine automatically orchestrates all inter-node data movement. All intermediate data is *immutable*—tensors produced during diffusion workflow execution are consumed once and never updated [39, 69]—which obviates consistency protocols. The engine reclaims tensors as soon as no downstream node requires them, reducing memory pressure. The data-plane design can be extended to other GPU-side communication mechanisms (e.g., rocSHMEM) while preserving the data-store and eager/deferred-fetch interfaces. Node-level execution and immutable intermediate data further narrow the failure domain: a component failure invalidates only affected nodes and their lost inputs, while unrelated components can continue running. This node granularity also provides natural units for conventional reassignment and re-execution mechanisms [49, 85, 87].

5 Workflow Node Scheduling

The scheduler is the component that translates *micro-serving* into runtime actions (❹ in Figure 5). It sits between the compiled workflow DAGs and the executor cluster, maintaining a global queue of workflow nodes and making three online

decisions in each scheduling cycle: (1) *which same-model nodes to batch together and which executors to route them to, exploiting model sharing across workflows*; (2) *how many GPUs to allocate per batch, adapting parallelism to current resource availability*; and (3) *whether to admit or reject incoming requests to preserve SLO attainment.* Algorithm 1 summarizes the scheduling loop; the remainder of this section describes how the scheduler makes each decision.

To make these decisions, the scheduler maintains two key data structures that the runtime keeps up to date. A model state table records, for every executor, which models are currently loaded in GPU memory. Executors piggyback their model states on node-completion notifications to the coordinator, so the table is updated without extra RPCs. A set of *per-model latency profiles*, collected offline, provides stable estimates [76] of data-fetch time, model-loading time, and inference time for each model under various batch sizes and parallelism degrees. Following prior work [10, 38, 40, 57, 92], the scheduler orders the ready queue by first-come-first-serve (FCFS). For nodes with the same arrival time (e.g., nodes from the same request), it further prioritizes those at shallower depths in the DAG. We intentionally use FCFS with shallow-depth tie-breaking: it requires no global optimization or future-arrival knowledge, keeps coordination overhead low, and isolates the benefits of DiFlow's microserving architecture from those of a sophisticated priority policy [24, 92]. The scheduler is pluggable, so such policies can be substituted and are orthogonal to our contribution.

Algorithm 1: Scheduling Algorithm

```
1  while True do
       // Admission control runs asynchronously (§5.3)
2      Admit or reject newly arrived requests;
       // Identify ready nodes
3      Q_ready ← nodes with satisfied dependencies from the queue;
4      E_avail ← currently available executors;
5      if Q_ready = ∅ or E_avail = ∅ then
6          continue;
7      Sort Q_ready by (arrival time, node depth);
8      n_head ← Q_ready.pop(0);
       // Batch same-model nodes (§5.1)
9      B_max ← profiled max batch size for n_head.model;
10     Batch ← {n_head} ∪ {n' ∈ Q_ready | n'.model =
         n_head.model and |Batch| < B_max};
       // Choose parallelism degree (§5.2)
11     k_max ← max useful parallelism for n_head.model;
12     k ← min(|E_avail|, k_max);
       // Score and select executors
13     for e ∈ E_avail do
14         L_data ← CalcDataFetchLatency(Batch, e);
15         L_load ← e hosts n_head.model ? 0 :
             CalcLoadTime(n_head.model);
16         L_infer ← CalcInferenceTime(Batch, e, k);
17         e.score ← L_data + L_load + L_infer;
18     E_target ← top k from E_avail with the lowest scores;
19     Dispatch Batch to E_target (triggers model load if needed);
```

5.1 Cross-Workflow Batching and Model Sharing

In each scheduling cycle, the scheduler pops the FCFS-earliest node n_{head} from the ready queue and inspects its `model` field. It then scans the remaining ready nodes for any that reference the same model, regardless of the originating workflow, and groups them into a single batch of up to B_{max} entries. The per-model B_{max} is determined offline by profiling batching efficiency: beyond a model-specific threshold, larger batches increase latency with diminishing throughput gain [10]. Because matching is by model identity rather than by workflow, a single batch may contain nodes from multiple workflows—this is how the scheduler realizes *model sharing* (§3.1).

After forming a batch, the scheduler must select an executor (Line 13 –17). For each candidate executor e, it computes a latency score from three profiled components: (1) L_{data}, the cost of fetching the batch's input tensors from their producing executors via the data engine (§4.3.2); (2) L_{load}, the cost of loading the required model into GPU memory; and (3) L_{infer}, the estimated inference time for the batch. The model state table makes L_{load} zero for any executor that already hosts the required model, so the scoring function naturally routes batches to executors with warm models. When no executor hosts the model, the scheduler selects the executor with the lowest total score and triggers a model load—loading only the single needed model, not the entire workflow, unlike monolithic scaling (**L1** in §2.2). We evaluate the throughput and latency gains from model sharing in §7.3.

5.2 Adaptive Parallelism

DiFlow exploits two forms of diffusion-specific parallelism (§2.1) through scheduling decisions.

Inter-Node Parallelism. When the compiler produces a DAG in which two nodes have no eager data dependency—for example, a ControlNet and its corresponding base-model node connected only by a *deferred* input (§4.3.2)—both nodes enter Q_{ready} simultaneously once their non-deferred inputs are satisfied. The scheduler dispatches them to separate executors in the same or adjacent loop iterations, so they execute concurrently. Runtime data exchange between the two nodes is handled by *deferred data fetch*: the base model begins execution immediately and retrieves the ControlNet output mid-inference when it becomes available (Figure 2-right).

Intra-Node Parallelism. A single model invocation can also be parallelized across multiple GPUs via latent parallelism [20, 37, 39, 45], which partitions the input tensor and distributes the shards to k executors for parallel inference (Figure 2-left). The scheduler chooses the parallelism degree k per batch by a *work-conserving* heuristic: it sets $k = \min(|E_{avail}|,\ k_{max})$, where k_{max} is the maximum useful parallelism for the model (determined offline). This rule uses

all currently available GPUs without waiting for more to free up, maximizing parallelism while avoiding extra queueing delay [40, 66]. Because the scheduler makes this decision per batch, different invocations of the same model can run at different parallelism degrees depending on instantaneous cluster load. The scheduler then selects the k lowest-scoring executors, dispatches the batch with a parallelism descriptor, and each executor processes its assigned input shard. We evaluate intra- and inter-node parallelism in §7.3.

5.3 SLO-Aware Admission Control

Admitting requests beyond system capacity inflates queueing delays and causes cascading SLO violations. DiFlow prevents this with an *early-abort* admission policy that leverages micro-serving's per-node visibility into request progress.

When a new request arrives, the scheduler estimates its end-to-end completion time. Because the control plane tracks which nodes of each inflight request have completed, the scheduler can compute, for every inflight request, the sum of profiled latencies along its remaining critical path. It admits the new request only if the estimated completion time—accounting for current queueing depth—satisfies the request's latency SLO. Otherwise, the request is rejected immediately, preserving resources for already-admitted requests. This early-abort policy is feasible only under micro-serving: in monolithic serving, the system has no visibility into sub-workflow progress and cannot estimate remaining work at fine granularity. The admission control runs asynchronously and does not block the scheduling loop. We quantify its effect on SLO attainment in §7.3.

6 Implementation

We have implemented DiFlow with a FastAPI [21] frontend and a distributed GPU-based inference engine. The frontend (approx. 1,000 LoC in Python) exposes an intuitive programming interface for users to compose and register diffusion workflows (§4.1). Users can invoke registered workflows with customized image generation parameters, such as prompts and reference images, similar to the OpenAI API [53]. We currently support diffusion workflows for popular models including the SD3 [19] and Flux families [35]. DiFlow's backend runtime consists of a coordinator and distributed executors (Figure 5), totaling 4,000 lines of Python code. The data engine is implemented in 1,000 lines of C++/CUDA code using NVSHMEM [51]. Aside from CUDA tensors, communication between the coordinator and distributed executors is facilitated via ZeroMQ [2].

7 Evaluation

We evaluate DiFlow with the following highlights:

- DiFlow outperforms state-of-the-art baselines, sustaining up to 3× higher request rates, satisfying 6× more stringent SLOs, reducing GPU requirements by up to 3×, and tolerating 8× higher burst traffic, all while maintaining over 90% SLO attainment (§7.2).
- Microbenchmarks isolate the contribution of key scheduling mechanisms and validate compatibility with emerging diffusion optimizations (§7.3, §7.4).
- DiFlow introduces negligible system overhead (§7.5).

Table 2. Evaluation Settings. We use workflows of representative diffusion models. S1–S4 represent single-model deployments, each including three workflow variants: a *Basic* workflow (text encoders, diffusion model, and decoder), plus two using adapters (*Basic + C.N. 1* and *Basic + C.N. 2*). S5–S6 represent mixed-model deployments. **C.N.**: ControlNet.

Setting	Diffusion Model	Workflow
Single-model Deployments (3 workflows each)		
S1	SD3 [19]	(Basic, +C.N. 1, +C.N. 2)
S2	SD3.5-Large [65]	(Basic, +C.N. 1, +C.N. 2)
S3	Flux-Schnell [35]	(Basic, +C.N. 1, +C.N. 2)
S4	Flux-Dev [35]	(Basic, +C.N. 1, +C.N. 2)
Mixed-model Deployments (6 workflows each)		
S5	SD3 + SD3.5-Large	S1's + S2's
S6	Flux-Schnell + Flux-Dev	S3's + S4's

7.1 Experimental Setup

Diffusion Workflows and Testbed. We use 12 diffusion workflows composed from four popular base models: SD3 [19], SD3.5-Large [65], Flux-Dev [35], and Flux-Schnell [35]. They exhibit diverse computational characteristics, with parameter counts spanning 2.5B to 12B and denoising steps ranging from 4 to 50. In Table 2, we categorize these workflows into six evaluation settings (S1–S6) to assess system performance under varying degrees of workload heterogeneity. We use a real testbed of 8–32 NVIDIA H800 GPUs and a 256-GPU simulator for scalability experiments. Each server contains eight GPUs in one NVLink domain; runs with 16–32 GPUs span two to four such servers, with GPU tensor traffic using RDMA across servers.

Baselines. We primarily compare DiFlow with Diffusers, the most representative *monolithic-serving* system (§2.2)[16, 69], as our goal is to compare *micro-serving* with the prevailing monolithic design. While other systems [11, 20, 45, 61, 68] support more parallelism methods and high-performance kernels, they largely inherit Diffusers's monolithic pipeline design and these optimizations are orthogonal to DiFlow.

To compare against a broader monolithic design space, we include *monolithic-serving* system variants by adopting techniques from multi-model serving systems [24, 92] for workflow orchestration. For a fair comparison, the baselines use FCFS scheduling and workflow-level admission control.

- Diffusers represents a static deployment strategy. Each workflow is executed monolithically and statically bound

to dedicated GPUs [16, 69]. It cannot share models across workflows or adapt parallelism at runtime.

- Diffusers-C implements a swap-based serving strategy by adapting Clockwork [24] to Diffusers. Leveraging the predictable end-to-end latency of diffusion workflows, it treats each monolithic workflow as a swappable DNN model unit, dynamically loading and unloading workflows into GPU memory on demand. Because the swap unit is an entire workflow, it cannot share individual models across workflows or adapt parallelism within a workflow.
- Diffusers-S incorporates the planning-and-scheduling framework of Shepherd [92] to orchestrate instances, modeling each monolithic diffusion workflow as a distinct model unit. Like Diffusers-C, it schedules whole workflows and thus cannot exploit per-model sharing or adaptive parallelism.

Workloads. We use a real-world T2I production trace [39]. To rigorously evaluate under diverse conditions, we vary request arrival rates, SLO targets, traffic burstiness, and testbed sizes, effectively simulating a wide spectrum of real-world traffic patterns and performance requirements.

Metrics. Our primary metric is *SLO attainment*: the fraction of all submitted requests completed within their specified latency deadline. We set the default deadline to 2× the solo inference latency of each workflow (SLO Scale = 2), which is tight given that any queueing or resource contention will cause violations. The denominator includes rejected requests, so admission control (§5.3) cannot increase attainment merely by rejecting more requests. DiFlow does not modify the original workflow computation; it only changes how model executions are scheduled. We verified this computational equivalence by checking that DiFlow and Diffusers generate identical images for the same inputs.

7.2 End-to-End Performance

As Figure 9 shows, DiFlow consistently outperforms all baselines across six settings, four traffic dimensions, and a range of testbed sizes. At high request rates, DiFlow achieves over 90% SLO attainment in settings where the strongest baseline drops below 3%.

SLO Attainment vs. Rate. We evaluate DiFlow and the baselines by varying the request rate while keeping the SLO scale (2.0) and traffic burstiness fixed. As shown in Figure 9 (a)–(f) and (j), DiFlow consistently achieves higher SLO attainment across varying rate scales. Compared to Diffusers-S, the strongest baseline, DiFlow sustains up to a 3× higher request rate while meeting a 90% SLO attainment target. At low request rates, all systems achieve high SLO attainment; however, as the rate increases, baseline performance plunges due to coarse-grained workflow scaling and inability to share common models (§2.2). In contrast, DiFlow's gains stem from two mechanisms: model sharing (§5.1) enables batching nodes from all three workflows onto shared model replicas, avoiding redundant loading; adaptive parallelism (§5.2) further reduces per-request latency at low-to-moderate rates by distributing inference across idle GPUs.

Next, we present evaluations that vary the SLO scale, traffic burstiness, and testbed size, respectively. We focus on the widely adopted Flux model family (S6), which represents recent advances in the field [32].

SLO Attainment vs. SLO Scale (16 GPUs). In Figure 9(g), we fix the rate scale at 1.0 and evaluate using the original production trace. Even at a strict SLO scale of 1.0, DiFlow achieves substantially higher SLO attainment than the baselines. At this scale, the deadline is tight enough that only intra-node parallelism—splitting the base model across two GPUs (§5.2)—can bring per-request latency below the target; baselines, which run each workflow on a single GPU, cannot meet this deadline regardless of scheduling policy. At an SLO scale of 2.0, DiFlow satisfies the SLO for over 90% of requests, whereas the baselines require an SLO scale of 12.0 to reach the same level. We observe a sharp increase in baseline SLO attainment when the SLO scale rises from 1.0 to 2.0, because this relaxation begins to absorb the inherent overheads of monolithic serving (§2.2). Beyond that point, baseline improvements are gradual. In contrast, DiFlow benefits more effectively from relaxed SLOs, achieving 1.4× higher SLO attainment than the strongest baseline at an SLO scale of 4.0.

SLO Attainment vs. CV (16 GPUs). In Figure 9 (h), we fix the rate scale at 0.25 and the SLO scale at 2.0. Following prior work [24, 40], we slice the original trace into time windows and fit the arrivals to a Gamma Process parameterized by the coefficient of variation (CV). Scaling the CV and resampling allows us to control traffic burstiness. We use CV values of $\{2, 4, 8, 16\}$ while holding the mean arrival rate fixed. Higher CVs indicate burstier traffic, which exacerbates queueing delays and increases SLO violations. As shown, DiFlow gracefully handles highly bursty traffic, sustaining high attainment even at an 8× larger CV compared to the baselines. When traffic subsides, the work-conserving parallelism heuristic (§5.2) drains the queue faster by assigning more GPUs per request, creating headroom before the next burst. Admission control (§5.3) then protects admitted requests during the spike itself by rejecting those that would violate their SLOs.

SLO Attainment vs. Testbed Size. Finally, in Figure 9 (i), we fix the rate scale at 0.5 and the SLO scale at 2.0, varying the testbed size under the original production trace. DiFlow requires up to 3× fewer GPUs to achieve a 90% SLO attainment target. This high resource efficiency is driven by our *micro-serving* design. Unlike monolithic serving systems that rigidly partition resources at the workflow level, DiFlow enables fine-grained model scaling and serving, as well as model sharing, which effectively treats all GPUs as a unified

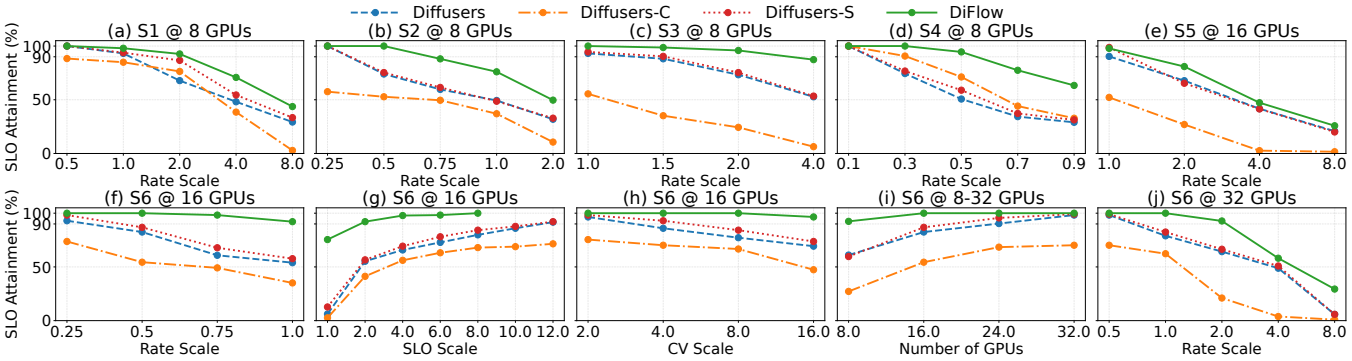

Figure 9. End-to-end performance across six settings (S1–S6 in Table 2), evaluated under varying request traffic rates (a–f, j), SLO requirements (g), traffic burstiness (h), and testbed sizes (i).

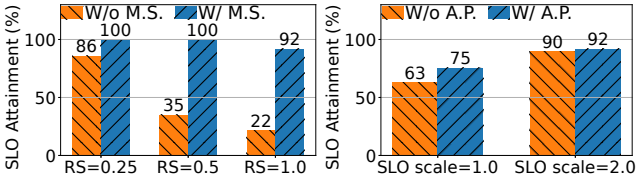

Figure 10. Left: Effectiveness of model sharing (M.S.) in DiFlow in setting S6. **RS**: Rate Scale. **Right:** Effectiveness of adaptive parallelism (A.P.) in DiFlow in setting S6.

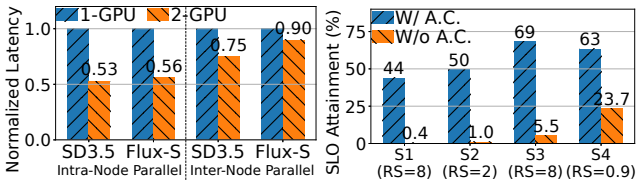

Figure 11. Left: Normalized latency of DiFlow across different numbers of available GPUs with intra-/inter-node parallelism (§5). **Flux-S:** Flux-Schnell. **Right:** Effectiveness of admission control (A.C.) in settings S1-4. **RS**: Rate Scale.

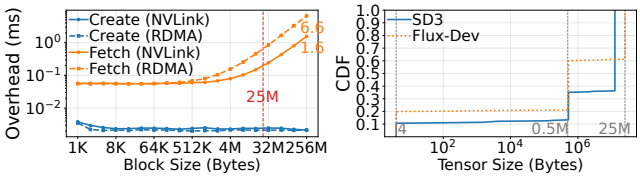

Figure 12. Left: Data fetching latency of varying sizes of tensor blocks. **Right:** The distribution of tensor block sizes found in typical SD3 and Flux workflows.

pool. This eliminates resource over-provisioning and ensures every available GPU cycle is utilized efficiently.

7.3 Microbenchmarks

We isolate the benefits of DiFlow's micro-serving (§3).

Model Sharing. DiFlow's micro-serving mechanism enables model sharing across workflows (§3.1), avoiding redundant model loading by allowing requests to reuse identical models already loaded in the cluster. In Figure 10-left, we evaluate its effectiveness by toggling model sharing on and off in S6 while keeping the SLO scale fixed at 2.0 and using 16 GPUs. When disabled, requests cannot reuse a model replica across workflow identities; all other scheduling mechanisms remain unchanged. The results show that enabling model sharing consistently improves SLO attainment. In particular, its benefit becomes more pronounced under heavier loads: SLO attainment improves from 86% to 100%, 35% to 100%, and 22% to 92% as the rate scale increases from 0.25 to 1.0. This is because model sharing allows different workflows to reuse common models, eliminating substantial model loading overhead caused by model switches when requests exercise different workflows.

Separately, diffusion models patched with LoRAs can be shared across requests, significantly reducing the memory footprint and latency overhead compared to loading a new model. We validate this using SD3 and a typical LoRA [43]. While the LoRA occupies 886 MiB of memory and takes 100 ms to swap [39], this saves the 3.9 GiB of memory and 430 ms of latency incurred by loading a fresh SD3 model.

Adaptive Parallelism. DiFlow's micro-serving mechanism enables the system to adaptively adjust the parallelism of each model inference (§3.1). In Figure 10-right, we evaluate the effectiveness of adaptive parallelism by toggling it on and off in S6 under different SLO scales, while keeping the rate scale fixed at 1.0 and using 16 GPUs. Disabling it fixes every model invocation to one GPU; all other mechanisms remain enabled. The results show that enabling adaptive parallelism improves SLO attainment by 19% under the stringent SLO scale of 1.0, as it allows DiFlow to effectively leverage additional GPUs to accelerate inference. With a more relaxed SLO scale of 2.0, the benefit diminishes because the additional SLO slack can already tolerate slower inference under the rate scale of 1.0, reducing the need for increased parallelism.

Table 3. Effective LOC and adaptive-runtime support.

Technique	LOC / (Supports adaptive adjustment?)		
	Katz [39]	xDiT [20]	DiFlow
Latent parallelism	92 (No)	68 (No)	74 (Yes)
ControlNet parallelism	127 (No)	N.A.	79 (Yes)
Async LoRA loading	182 (Yes)	N.A.	61 (Yes)

Intra-Node Parallelism. DiFlow natively integrates latent parallelism [20, 37, 39, 45] with intra-node parallelism (§5.2), enabling accelerated diffusion model inference across two GPUs. As shown in Figure 11-left, our intra-node implementation achieves a speedup of up to 1.9×. This aligns with findings in [37, 39] and validates DiFlow's capability to support state-of-the-art optimizations.

Inter-Node Parallelism. As discussed in §4.3.2, DiFlow implements a *deferred* data fetch mechanism to enable ControlNet parallelization [39], a key form of inter-node parallelism described in §5.2. As shown in Figure 11-left, DiFlow's inter-node parallelism accelerates workflow execution across different models by up to 1.3×, consistent with results in [39]. Note that the gains with Flux models are limited because their ControlNets are small (only 6% of the base model size) and have negligible latency compared to the base model.

Admission Control. We evaluate the effectiveness of DiFlow's admission control (§5.3) in optimizing SLO attainment. In Figure 11-right, enabling admission control across the four settings (Table 2) prevents system overload under high request rates. By proactively aborting requests that are destined to violate SLOs, DiFlow increases SLO attainment from a mere 0.4% to 44% in setting S1.

Programmability. DiFlow provides an intuitive programming model for composing complex workflows. Following [97], we quantify developer productivity using effective Lines of Code (LoC). We compare DiFlow against Katz [39] and xDiT [20], two popular diffusion model serving engines that support parallel acceleration. As shown in Table 3, DiFlow requires comparable or lower implementation effort to express these optimizations, while additionally supporting adaptive runtime behavior that the baselines do not.

7.4 Case Study

We next show that DiFlow supports emerging optimizations tailored for diffusion models described in §4.2.

Approximate Caching. In DiFlow, we implement Nirvana's [4] approximate caching optimization (§2.1). Following prior work [4, 39], we use an SDXL workflow and configure it to reduce denoising computation by 20% and 40%, respectively. The optimization achieves speedups of 1.13× and 1.43× with its original implementation on Diffusers, and comparable speedups of 1.17× and 1.42× on DiFlow, evidencing DiFlow's effective support for the optimization.

Async LoRA Loading. We implement Katz's [39] asynchronous LoRA loading design in DiFlow and evaluate it against the original Diffusers implementation, using SDXL [56] with a papercut-style LoRA [67]. Our implementation reduces LoRA loading overhead from 0.5 seconds to 0.05 seconds, matching the results reported in [39].

7.5 System Overhead

Execution Overhead. Micro-serving introduces additional overhead due to inter-node communication and control-plane coordination. We quantify this overhead by comparing DiFlow against monolithic baselines on four workflows: SD3, SD3.5-Large, Flux-Dev, and Flux-Schnell. Across all cases, the maximum end-to-end overhead is 150 ms. Given that these diffusion workloads typically take 2–20 seconds to complete, this additional cost is negligible.

Control-Plane Scalability. To show DiFlow's control plane remains efficient at large scale, we conduct simulation-based experiments on a 256-GPU setup under high concurrency, with 500 inflight requests. Across two representative workflows, Flux-Dev and SD3.5-Large, the coordinator accounts for only 3.4% and 2.7% of total execution time, respectively, indicating that the control plane does not emerge as the dominant bottleneck at this scale. Control-plane load is determined primarily by the rate at which nodes become ready for scheduling, rather than by workflow complexity alone; Flux-Dev and SD3.5-Large represent the largest DAGs within their respective model families.

Data Transmission Latency. We further isolate the cost of intermediate tensor movement in Figure 12. The left panel shows the latency of tensor serialization and transmission over a range of tensor sizes, and the right panel reports the actual intermediate tensor sizes produced by SD3 and Flux-Dev workflows with ControlNet. Even for the largest intermediate tensors, transmission latency remains below 1 ms, confirming that the inter-GPU bandwidth is not a bottleneck for DiFlow's fine-grained execution model.

8 Discussion and Related Work

Scalability and Fault Tolerance. In DiFlow, one coordinator manages N executors (Figure 5). To avoid a coordinator bottleneck as N grows, DiFlow shards executors across multiple coordinators, each managing a disjoint subset of workflows that share models, preserving sharing opportunities. A cluster management service [1, 33] provides conventional coordinator discovery and failure detection. When an executor fails, its coordinator reassigns the affected nodes to healthy executors for re-execution. Because workflow nodes are independently schedulable, recovery is limited to these nodes rather than restarting the entire workflow, while unaffected executors continue serving.

Diffusion Model Serving Systems. vLLM-Omni [68], SGLang-Diffusion [61], and Cornserve [11] target omni-modal pipelines, coarsely separating autoregressive and diffusion stages. However, they largely retain Diffusers-style monolithic execution within each diffusion workflow. Concurrent with this work, SGLang-Diffusion also introduced a preliminary deployment option that statically partitions a diffusion workflow at predefined boundaries [62]. However, this option does not provide native scheduling of arbitrary model nodes, adaptive parallelism, or cross-workflow model sharing; these systems also currently provide limited adapter support [59, 60, 68]. In contrast, DiFlow natively micro-serves diffusion workflows at model-node granularity. Their kernel-level optimizations are complementary and can be incorporated into a model node's `execute()` method (§4.1) without changing DiFlow's micro-serving runtime.

DistriFusion [37], xDiT [20], Katz [39], and TetriServe [45] propose diffusion-specific parallelism techniques, but primarily embed them in monolithic workflows with static parallelization strategies. DiFlow does not claim a new parallelism technique; instead, it expresses parallelism across workflow models (e.g., Katz's ControlNet parallelism) as inter-node parallelism and parallelism within a model (e.g., DistriFusion and xDiT) as intra-node parallelism (§5.2). This makes these techniques dynamically schedulable: DiFlow selects a parallelism strategy for each batch according to current resource availability. Thus, these acceleration techniques are complementary to DiFlow (§4.2, §7.4).

Several works accelerate individual workflow execution: Nirvana [4] reduces denoising steps via cached images, while TetriServe [45] and TridentServe [76] adapt sequence parallelism to meet latency SLOs. However, several of these systems [4, 37, 45, 76] lack support for adapters that are prevalent in production [39, 42], and none targets cluster-level, multi-workflow deployment. DiFlow's micro-serving approach is complementary to these techniques (§4.2, §7.4).

Other Model Serving Systems. Prior work on model serving has improved latency [14, 71], throughput [6, 80], and resource efficiency [25, 70, 72, 79, 89] across DNNs and LLMs [5, 8, 9, 18, 23, 26, 29, 47, 52, 64, 75, 81, 82, 84, 86, 88, 96]. DiFlow complements them by focusing on text-to-image serving, which has distinct computational and workflow characteristics. Prior work on online multi-model serving has further explored model placement [40, 92], request scheduling [24, 92], and dynamic scaling [22, 90]. In §7, we integrate representative techniques from them with existing diffusion workflow serving systems [16]. Despite their effectiveness, DiFlow outperforms them with its micro-serving approach, which fundamentally addresses the limitations of monolithic serving. Also, DiFlow is compatible with these optimizations.

Global Planning. DiFlow currently makes online scheduling decisions using per-model profiles and current executor state; it does not optimize long-term replica placement. A complementary global planner could forecast model demand and determine replica counts and placements over longer time horizons, improving resource efficiency by exploiting model sharing and data locality without changing DiFlow's runtime. While such planners have been proposed for LLM serving [77] and omni-modal workflow serving [46], designing one for shared, adapter-rich diffusion workflows remains an open problem.

9 Conclusion

We presented DiFlow, an efficient micro-serving system for diffusion workflows. DiFlow has three key designs: (1) a programming model that transforms workflow compositions into loosely coupled nodes; (2) a specialized runtime and data plane that streamline data communication to facilitate micro-serving; and (3) a scheduler that realizes the benefits of micro-serving at the cluster level. Collectively, DiFlow outperforms existing *monolithic* serving systems, sustaining up to 3× higher request rates and tolerating 8× higher burst traffic, with the same performance requirements.

Acknowledgments

We would like to thank our shepherd, Mosharaf Chowdhury, and the anonymous reviewers of OSDI 2026 and SOSP 2026 for their valuable feedback that helped improve the quality of this work. Lingyun Yang was supported in part by the Joint Postdoctoral Research Workstation established by Alibaba DAMO Academy and Shanghai Jiao Tong University. This work was supported by the HKUST-Alibaba Joint Laboratory on Big Data and AI, RGC CRF Grant (Ref. #C6015-23G), RGC GRF Grant (Ref. #16217124), and NSFC/RGC CRS Grant (Ref. #CRS_HKUST601/24).

References

[1] 2025. Apache ZooKeeper. https://zookeeper.apache.org/.
[2] 2025. ZeroMQ. https://github.com/zeromq/pyzmq.
[3] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A system for large-scale machine learning. In *Proc. USENIX OSDI*.
[4] 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 *Proc. USENIX NSDI*.
[5] Amey Agrawal, Nitin Kedia, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav Gulavani, Alexey Tumanov, and Ramachandran Ramjee. 2024. Taming throughput-latency tradeoff in LLM inference with Sarathi-Serve. In *Proc. USENIX OSDI*.
[6] Sohaib Ahmad, Hui Guan, Brian D. Friedman, Thomas Williams, Ramesh K. Sitaraman, and Thomas Woo. 2024. Proteus: A high-throughput inference-serving system with accuracy scaling. In *Proc. ACM ASPLOS*.

[7] BentoML. 2025. comfy-pack: Serving ComfyUI workflows as APIs. https://www.bentoml.com/blog/comfy-pack-serving-comfyui-workflows-as-apis.

[8] Hongtao Chen, Weiyu Xie, Boxin Zhang, Jingqi Tang, Jiahao Wang, Jianwei Dong, Shaoyuan Chen, Ziwei Yuan, Chen Lin, Chengyu Qiu, Yuening Zhu, Qingliang Ou, Jiaqi Liao, Xianglin Chen, Zhiyuan Ai, Yongwei Wu, and Mingxing Zhang. 2025. KTransformers: Unleashing the full potential of CPU/GPU hybrid inference for MoE models. In *Proc. ACM SOSP*.

[9] Le Chen, Dahu Feng, Erhu Feng, Yingrui Wang, Rong Zhao, Yubin Xia, Pinjie Xu, and Haibo Chen. 2025. Characterizing mobile SoC for accelerating heterogeneous LLM inference. In *Proc. ACM SOSP*.

[10] Lequn Chen, Zihao Ye, Yongji Wu, Danyang Zhuo, Luis Ceze, and Arvind Krishnamurthy. 2024. Punica: Multi-tenant LoRA serving. In *Proc. MLSys*.

[11] Jae-Won Chung, Jeff J. Ma, Jisang Ahn, Yizhuo Liang, Akshay Jajoo, Myungjin Lee, and Mosharaf Chowdhury. 2026. Cornserve: A distributed serving system for any-to-any multimodal models. In *Proc. ACM CAIS*.

[12] ComfyUI. 2025. ComfyUI: The most powerful and modular visual AI engine and application. https://github.com/comfyanonymous/ComfyUI.

[13] ComfyUI. 2025. Understand the concept of a node in ComfyUI. https://docs.comfy.org/essentials/core-concepts/nodes.

[14] Daniel Crankshaw, Xin Wang, Guilio Zhou, Michael J. Franklin, Joseph E. Gonzalez, and Ion Stoica. 2017. Clipper: A low-latency online prediction serving system. In *Proc. USENIX NSDI*.

[15] Jeffrey Dean and Sanjay Ghemawat. 2004. MapReduce: Simplified data processing on large clusters. In *Proc. USENIX OSDI*.

[16] HuggingFace Diffusers. 2025. Create a server. https://github.com/huggingface/diffusers/blob/main/docs/source/en/using-diffusers/create_a_server.md.

[17] HuggingFace Diffusers. 2025. Philosophy. https://huggingface.co/docs/diffusers/en/conceptual/philosophy.

[18] 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 *Proc. ICML*.

[19] Patrick Esser, Sumith Kulal, Andreas Blattmann, Rahim Entezari, Jonas Müller, Harry Saini, Yam Levi, Dominik Lorenz, Axel Sauer, Frederic Boesel, Dustin Podell, Tim Dockhorn, Zion English, and Robin Rombach. 2024. Scaling rectified flow transformers for high-resolution image synthesis. In *Proc. ICML*.

[20] 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).

[21] FastAPI. 2025. FastAPI. https://github.com/fastapi/fastapi.

[22] Yao Fu, Leyang Xue, Yeqi Huang, Andrei-Octavian Brabete, Dmitrii Ustiugov, Yuvraj Patel, and Luo Mai. 2024. ServerlessLLM: Low-latency serverless inference for large language models. In *Proc. USENIX OSDI*.

[23] Shiwei Gao, Qing Wang, Shaoxun Zeng, Youyou Lu, and Jiwu Shu. 2025. WEAVER: Efficient multi-LLM serving with attention offloading. In *Proc. USENIX ATC*.

[24] Arpan Gujarati, Reza Karimi, Safya Alzayat, Wei Hao, Antoine Kaufmann, Ymir Vigfusson, and Jonathan Mace. 2020. Serving DNNs like Clockwork: Performance predictability from the bottom up. In *Proc. USENIX OSDI*.

[25] Jashwant Raj Gunasekaran, Cyan Subhra Mishra, Prashanth Thinakaran, Bikash Sharma, Mahmut Taylan Kandemir, and Chita R. Das. 2022. Cocktail: A multidimensional optimization for model serving in cloud. In *Proc. USENIX NSDI*.

[26] Yongjun He, Haofeng Yang, Yao Lu, Ana Klimović, and Gustavo Alonso. 2025. Resource multiplexing in tuning and serving large language models. In *Proc. USENIX ATC*.

[27] Jonathan Ho and Tim Salimans. 2021. Classifier-free diffusion guidance. In *Proc. NeurIPS 2021 Workshop on Deep Generative Models and Downstream Applications*.

[28] Edward J Hu, yelong shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2022. LoRA: Low-rank adaptation of large language models. In *Proc. ICLR*.

[29] Zhengding Hu, Vibha Murthy, Zaifeng Pan, Wanlu Li, Xiaoyi Fang, Yufei Ding, and Yuke Wang. 2025. HedraRAG: Co-optimizing generation and retrieval for heterogeneous RAG workflows. In *Proc. ACM SOSP*.

[30] HuggingFace. 2025. Accelerate inference of text-to-image diffusion models. https://huggingface.co/docs/diffusers/en/tutorials/fast_diffusion.

[31] HuggingFace. 2025. Hugging Face models. https://huggingface.co/models?pipeline_tag=text-to-image&sort=downloads.

[32] HuggingFace. 2025. Hugging Face models. https://huggingface.co/models?pipeline_tag=text-to-image&sort=likes.

[33] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. 2010. ZooKeeper: Wait-free coordination for Internet-scale systems. In *Proc. USENIX ATC*.

[34] Xuan Ju, Xian Liu, Xintao Wang, Yuxuan Bian, Ying Shan, and Qiang Xu. 2024. BrushNet: A plug-and-play image inpainting model with decomposed dual-branch diffusion. In *Proc. ECCV*.

[35] Black Forest Labs. 2024. FLUX. https://github.com/black-forest-labs/flux.

[36] LangChain. 2025. LangChain. https://www.langchain.com.

[37] Muyang Li, Tianle Cai, Jiaxin Cao, Qinsheng Zhang, Han Cai, Junjie Bai, Yangqing Jia, Ming-Yu Liu, Kai Li, and Song Han. 2024. DistriFusion: Distributed parallel inference for high-resolution diffusion models. In *Proc. IEEE/CVF CVPR*.

[38] Suyi Li, Hanfeng Lu, Tianyuan Wu, Minchen Yu, Qizhen Weng, Xusheng Chen, Yizhou Shan, Binhang Yuan, and Wei Wang. 2025. Toppings: CPU-assisted, rank-aware adapter serving for LLM inference. In *Proc. USENIX ATC*.

[39] Suyi Li, Lingyun Yang, Xiaoxiao Jiang, Hanfeng Lu, 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 *Proc. USENIX ATC*.

[40] Zhuohan Li, Lianmin Zheng, Yinmin Zhong, Vincent Liu, Ying Sheng, Xin Jin, Yanping Huang, Zhifeng Chen, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. 2023. AlpaServe: Statistical multiplexing with model parallelism for deep learning serving. In *Proc. USENIX OSDI*.

[41] Chaofan Lin, Zhenhua Han, Chengruidong Zhang, Yuqing Yang, Fan Yang, Chen Chen, and Lili Qiu. 2024. Parrot: Efficient serving of LLM-based applications with semantic variable. In *Proc. USENIX OSDI*.

[42] Yanying Lin, Shuaipeng Wu, Shutian Luo, Hong Xu, Haiying Shen, Chong Ma, Min Shen, Le Chen, Chengzhong Xu, Lin Qu, and Kejiang Ye. 2025. Understanding diffusion model serving in production: A top-down analysis of workload, scheduling, and resource efficiency. In *Proc. ACM SoCC*.

[43] linoyts. 2025. Yarn_art_SD3_LoRA. https://huggingface.co/linoyts/Yarn_art_SD3_LoRA.

[44] LlamaIndex. 2025. LlamaIndex. https://www.llamaindex.ai.

[45] 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 *Proc. ACM ASPLOS*.

[46] Jeff J. Ma, Jae-Won Chung, Jisang Ahn, Yizhuo Liang, Runyu Lu, Akshay Jajoo, Myungjin Lee, and Mosharaf Chowdhury. 2025. Cornfigurator: Automated planning for any-to-any multimodal model serving. *arXiv preprint arXiv:2512.14098* (2025).

[47] 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 *Proc. ACM ASPLOS*.

[48] Modal. 2025. How OpenArt scaled their gen AI art platform on hundreds of GPUs. https://modal.com/blog/openart-case-study.

[49] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. 2018. Ray: A distributed framework for emerging AI applications. In *Proc. USENIX OSDI*.

[50] Dung Nguyen and Stephen B. Wong. 2000. Design patterns for lazy evaluation. In *Proc. ACM SIGCSE*.

[51] NVIDIA. 2025. NVIDIA OpenSHMEM library (NVSHMEM) documentation. https://docs.nvidia.com/nvshmem/api/index.html.

[52] Gabriele Oliaro, Xupeng Miao, Xinhao Cheng, Vineeth Kada, Ruohan Gao, Yingyi Huang, Remi Delacourt, April Yang, Yingcheng Wang, Mengdi Wu, Colin Unger, and Zhihao Jia. 2025. FlexLLM: A system for co-serving large language model inference and parameter-efficient finetuning. *arXiv preprint arXiv:2402.18789* (2025).

[53] OpenAI. 2020. OpenAI API. https://openai.com/index/openai-api/.

[54] OpenAI. 2025. Introducing 4o image generation. https://openai.com/index/introducing-4o-image-generation/.

[55] OpenAI. 2025. OpenAI DALL·E 2. https://openai.com/index/dall-e-2/.

[56] Dustin Podell, Zion English, Kyle Lacey, Andreas Blattmann, Tim Dockhorn, Jonas Müller, Joe Penna, and Robin Rombach. 2024. SDXL: Improving latent diffusion models for high-resolution image synthesis. In *Proc. ICLR*.

[57] Ruoyu Qin, Zheming Li, Weiran He, Jialei Cui, Feng Ren, Mingxing Zhang, Yongwei Wu, Weimin Zheng, and Xinran Xu. 2025. Mooncake: Trading more storage for less computation — A KVCache-centric architecture for serving LLM chatbot. In *Proc. USENIX FAST*.

[58] Pol G. Recasens, Yue Zhu, Chen Wang, Eun Kyung Lee, Olivier Tardieu, Alaa Youssef, Jordi Torres, and Josep Ll. Berral. 2024. Towards Pareto optimal throughput in small language model serving. In *Proc. ACM EuroMLSys*.

[59] sgl project. 2026. diffusion: Add-ons support (LoRA & ControlNet). https://github.com/sgl-project/sglang/issues/13790.

[60] sgl project. 2026. [Roadmap] diffusion (2025 Q4). https://github.com/sgl-project/sglang/issues/12799.

[61] sgl project. 2026. SGLang diffusion. https://github.com/sgl-project/sglang/tree/main/python/sglang/multimodal_gen.

[62] sgl project. 2026. SGLang [diffusion] feat: Disaggregated diffusion. https://github.com/sgl-project/sglang/commit/9da998a882c810cad5bb739e691a457fa61eb1f1.

[63] Arjun Singhvi, Arjun Balasubramanian, Kevin Houck, Mohammed Danish Shaikh, Shivaram Venkataraman, and Aditya Akella. 2021. Atoll: A scalable low-latency serverless platform. In *Proc. ACM SoCC*.

[64] Vikranth Srivatsa, Zijian He, Reyna Abhyankar, Dongming Li, and Yiying Zhang. 2025. Preble: Efficient distributed prompt scheduling for LLM serving. In *Proc. ICLR*.

[65] stabilityai. 2025. stable-diffusion-3.5-large. https://huggingface.co/stabilityai/stable-diffusion-3.5-large.

[66] Xin Tan, Yimin Jiang, Yitao Yang, and Hong Xu. 2025. Towards end-to-end optimization of LLM-based applications with Ayo. In *Proc. ACM ASPLOS*.

[67] TheLastBen. 2025. Papercut style, SDXL LoRA. https://huggingface.co/TheLastBen/Papercut_SDXL.

[68] vllm project. 2026. vLLM Omni. https://github.com/vllm-project/vllm-omni.

[69] Patrick von Platen, Suraj Patil, Anton Lozhkov, Pedro Cuenca, Nathan Lambert, Kashif Rasul, Mishig Davaadorj, Dhruv Nair, Sayak Paul, William Berman, Yiyi Xu, Steven Liu, and Thomas Wolf. 2022. Diffusers: State-of-the-art diffusion models. https://github.com/huggingface/diffusers.

[70] Luping Wang, Lingyun Yang, Yinghao Yu, Wei Wang, Bo Li, Xianchao Sun, Jian He, and Liping Zhang. 2021. Morphling: Fast, near-optimal auto-configuration for cloud-native model serving. In *Proc. ACM SoCC*.

[71] Yiding Wang, Kai Chen, Haisheng Tan, and Kun Guo. 2023. Tabi: An efficient multi-level inference system for large language models. In *Proc. ACM EuroSys*.

[72] Yuke Wang, Boyuan Feng, Zheng Wang, Tong Geng, Kevin Barker, Ang Li, and Yufei Ding. 2023. MGG: Accelerating graph neural networks with fine-grained intra-kernel communication-computation pipelining on multi-GPU platforms. In *Proc. USENIX OSDI*.

[73] Zibo Wang, Pinghe Li, Chieh-Jan Mike Liang, Feng Wu, and Francis Y. Yan. 2024. Autothrottle: A practical bi-level approach to resource management for SLO-targeted microservices. In *Proc. USENIX NSDI*.

[74] Wikipedia. 2025. Lazy evaluation. https://en.wikipedia.org/wiki/Lazy_evaluation.

[75] 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 *Proc. ACM SOSP*.

[76] Yifei Xia, Fangcheng Fu, Hao Yuan, Hanke Zhang, Xupeng Miao, Yijun Liu, Suhan Ling, Jie Jiang, and Bin Cui. 2025. TridentServe: A stage-level serving system for diffusion pipelines. *arXiv preprint arXiv:2510.02838* (2025).

[77] Tianhao Xu, Yiming Liu, Xianglong Lu, Yijia Zhao, Xuting Zhou, Aichen Feng, Yiyi Chen, Yi Shen, Qin Zhou, Xumeng Chen, Ilya Sherstyuk, Haorui Li, Rishi Thakkar, Ben Hamm, Yuanzhe Li, Xue Huang, Wenpeng Wu, Anish Shanbhag, Harry Kim, Chuan Chen, and Junjie Lai. 2026. AIConfigurator: Lightning-fast configuration optimization for multi-framework LLM serving. *arXiv preprint arXiv:2601.06288* (2026).

[78] Yuhao Xu, Tao Gu, Weifeng Chen, and Arlene Chen. 2025. OOTDiffusion: Outfitting fusion based latent diffusion for controllable virtual try-on. In *Proc. AAAI*.

[79] Lingyun Yang, Yongchen Wang, Yinghao Yu, Qizhen Weng, Jianbo Dong, Kan Liu, Chi Zhang, Yanyi Zi, Hao Li, Zechao Zhang, Nan Wang, Yu Dong, Menglei Zheng, Lanlan Xi, Xiaowei Lu, Liang Ye, Guodong Yang, Binzhang Fu, Tao Lan, Liping Zhang, Lin Qu, and Wei Wang. 2025. GPU-disaggregated serving for deep learning recommendation models at scale. In *Proc. USENIX NSDI*.

[80] Yanan Yang, Laiping Zhao, Yiming Li, Huanyu Zhang, Jie Li, Mingyang Zhao, Xingzhen Chen, and Keqiu Li. 2022. INFless: A native serverless system for low-latency, high-throughput inference. In *Proc. ACM ASPLOS*.

[81] Jiayi Yao, Hanchen Li, Yuhan Liu, Siddhant Ray, Yihua Cheng, Qizheng Zhang, Kuntai Du, Shan Lu, and Junchen Jiang. 2025. CacheBlend: Fast large language model serving for RAG with cached knowledge fusion. In *Proc. ACM EuroSys*.

[82] Xiaozhe Yao, Qinghao Hu, and Ana Klimovic. 2025. DeltaZip: Efficient serving of multiple full-model-tuned LLMs. In *Proc. ACM EuroSys*.

[83] Hu Ye, Jun Zhang, Sibo Liu, Xiao Han, and Wei Yang. 2023. IP-Adapter: Text compatible image prompt adapter for text-to-image diffusion models. *arXiv preprint arXiv:2308.06721* (2023).

[84] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. 2022. Orca: A distributed serving system for transformer-based generative models. In *Proc. USENIX OSDI*.

[85] Minchen Yu, Tingjia Cao, Wei Wang, and Ruichuan Chen. 2023. Following the data, not the function: Rethinking function orchestration in serverless computing. In *Proc. USENIX NSDI*.

[86] Yifan Yu, Yu Gan, Nikhil Sarda, Lillian Tsai, Jiaming Shen, Yanqi Zhou, Arvind Krishnamurthy, Fan Lai, Hank Levy, and David Culler. 2025. IC-Cache: Efficient large language model serving via in-context caching. In *Proc. ACM SOSP*.

[87] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proc. USENIX NSDI*.

[88] Shaoxun Zeng, Minhui Xie, Shiwei Gao, Youmin Chen, and Youyou Lu. 2025. Medusa: Accelerating serverless LLM inference with materialization. In *Proc. ACM ASPLOS*.

[89] Chengliang Zhang, Minchen Yu, Wei Wang, and Feng Yan. 2019. MArk: Exploiting cloud services for cost-effective, SLO-aware machine learning inference serving. In *Proc. USENIX ATC*.

[90] Dingyan Zhang, Haotian Wang, Yang Liu, Xingda Wei, Yizhou Shan, Rong Chen, and Haibo Chen. 2025. Fast and live model auto scaling without caching. In *Proc. USENIX OSDI*.

[91] Hong Zhang, Yupeng Tang, Anurag Khandelwal, Jingrong Chen, and Ion Stoica. 2021. Caerus: NIMBLE task scheduling for serverless analytics. In *Proc. USENIX NSDI*.

[92] Hong Zhang, Yupeng Tang, Anurag Khandelwal, and Ion Stoica. 2023. Shepherd: Serving DNNs in the wild. In *Proc. USENIX NSDI*.

[93] Lvmin Zhang. 2025. Fooocus. https://github.com/lllyasviel/Fooocus.

[94] Lvmin Zhang, Anyi Rao, and Maneesh Agrawala. 2023. Adding conditional control to text-to-image diffusion models. In *Proc. IEEE/CVF ICCV*.

[95] Lvmin Zhang, Anyi Rao, and Maneesh Agrawala. 2025. Scaling in-the-wild training for diffusion-based illumination harmonization and editing by imposing consistent light transport. In *Proc. ICLR*.

[96] Wei Zhang, Ziyu Wu, Yi Mu, Rui Ning, Banruo Liu, Nikhil Sarda, Myungjin Lee, and Fan Lai. 2026. JITServe: SLO-aware LLM serving with imprecise request information. In *Proc. USENIX NSDI*.

[97] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark Barrett, and Ying Sheng. 2024. SGLang: Efficient execution of structured language model programs. In *Proc. NeurIPS*.