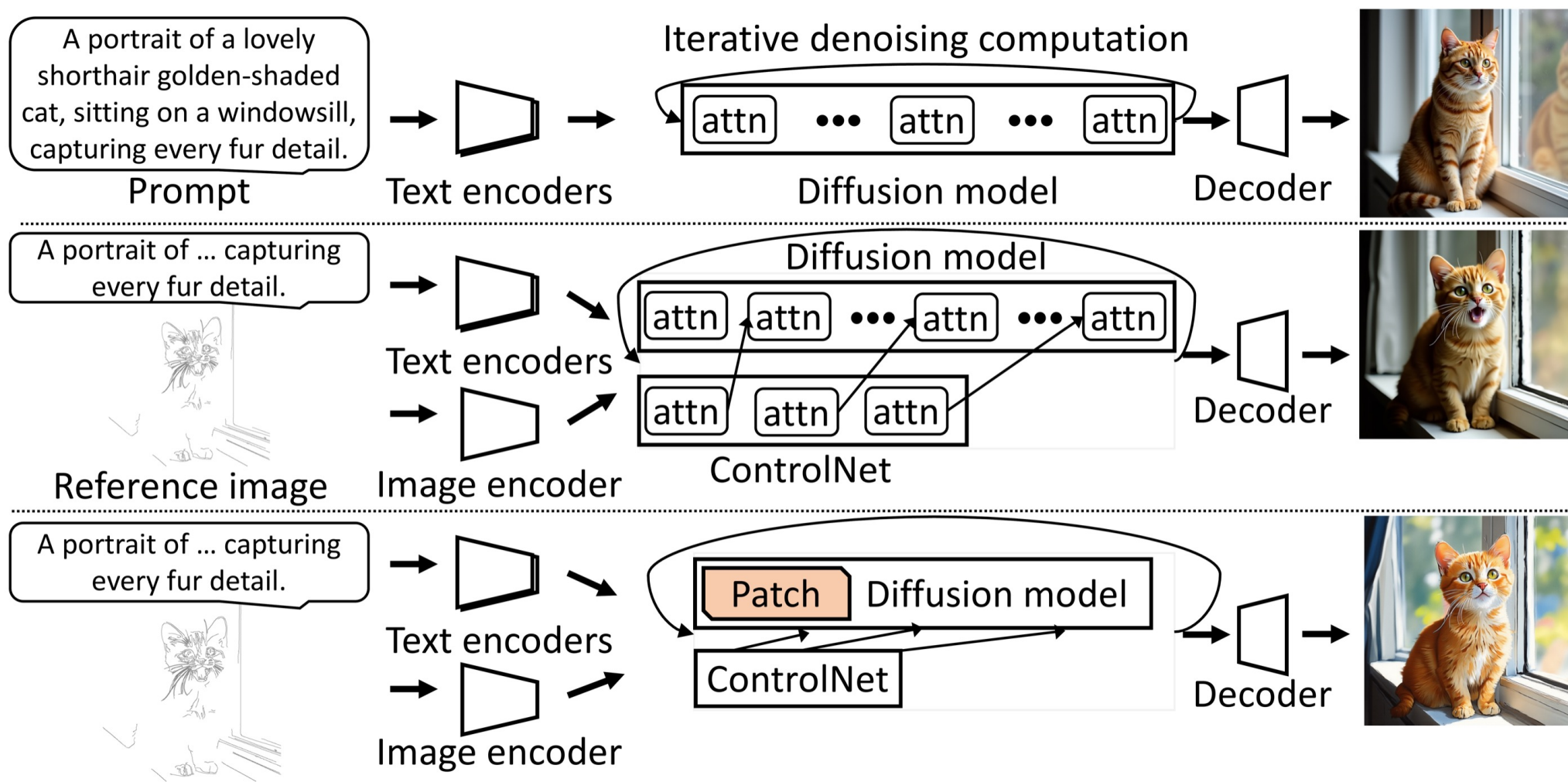
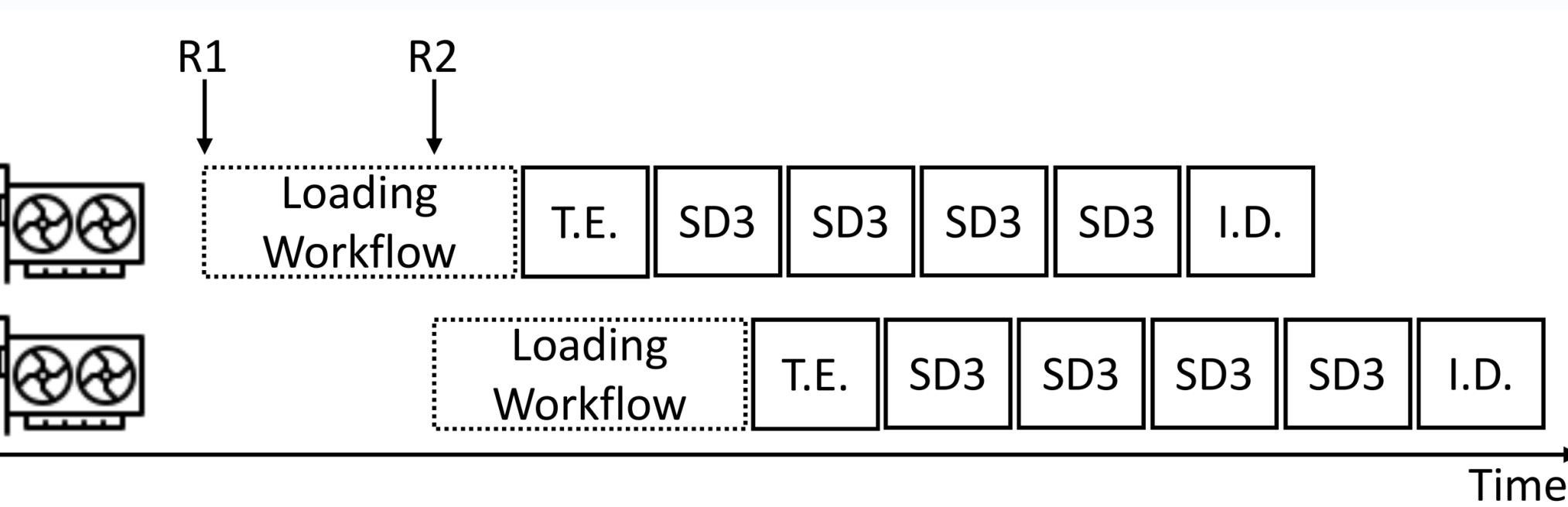


Diffusion applications are workflows

Text-to-image applications combine a diffusion backbone with text encoders, a decoder, and optional adapters such as ControlNet and LoRA. Each request executes a workflow whose components differ in size, latency, and reuse.



Monolithic serving imposes four cluster-level limitations



Full-workflow replication

Scaling a bottleneck duplicates every stage, adding loading time and GPU memory.

Replica isolation

Compatible requests cannot batch on replicas already resident in other workflows.

Static runtime configuration

Fixed GPU assignments cannot match per-model demand or resource availability.

Coupled workflow boundary

Updates and failures expand validation and recovery to the entire pipeline.

Production workloads favor model reuse



Popular backbones and adapters create shared demand across workflow identities. Model-level scheduling can reuse resident replicas and scale only the nodes that need more capacity.

Micro-serving exposes this reuse without asking workflow developers to abandon ordinary Python composition.

What micro-serving unlocks

Model-level execution creates a common scheduling boundary across workflow identities. DiFlow can batch compatible nodes on resident replicas, isolate bottlenecks, and avoid reloading components that still have capacity.

This model boundary lets the runtime share hot replicas across workflows and add capacity only where the critical path needs it.

DiFlow preserves the
Python workflow interface
while micro-serving every
model invocation as an independently schedulable node,
enabling cross-workflow model reuse
and adaptive GPU allocation.



Explore DiFlow on GitHub



github.com/diflow-project/diflow

We welcome contributions of all kinds.

Programming model

DiFlow separates model implementation from workflow composition while keeping both interfaces in Python.

MODEL DEVELOPER

setup_io() defines typed inputs and outputs

load() and **execute()** define the model lifecycle

add_patch() and **rm_patch()** manage adapters

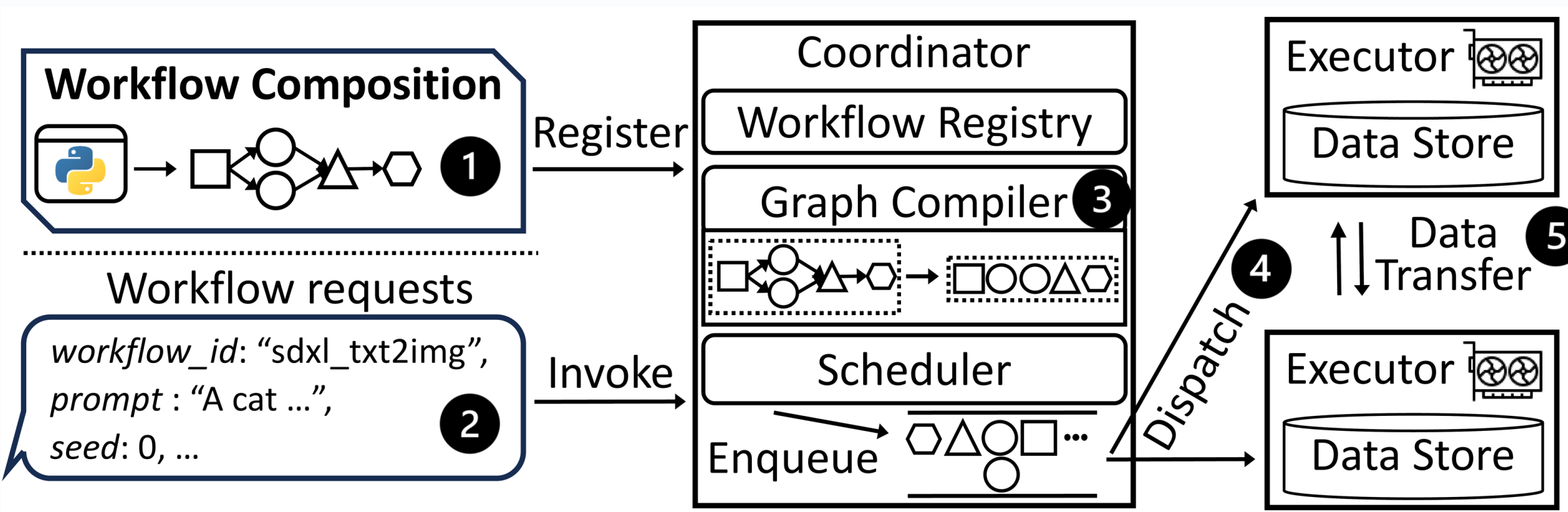
WORKFLOW DEVELOPER

add_input() and **add_output()** define workflow I/O

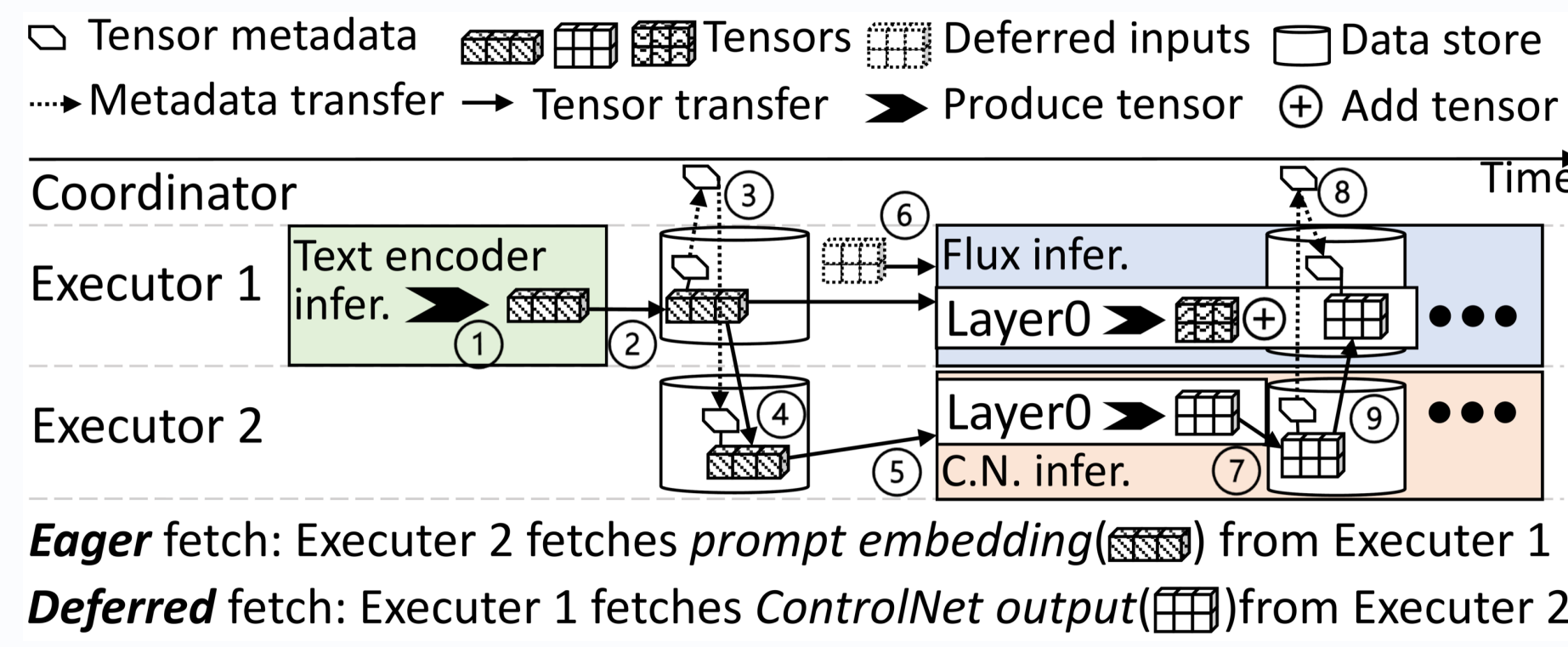
Ordinary model calls and Python control flow compose the workflow

The compiler infers dependencies and fetch modes

Runtime design



The compiler creates model-execution nodes. The coordinator tracks readiness and placement, while GPU executors keep model state and tensors local.

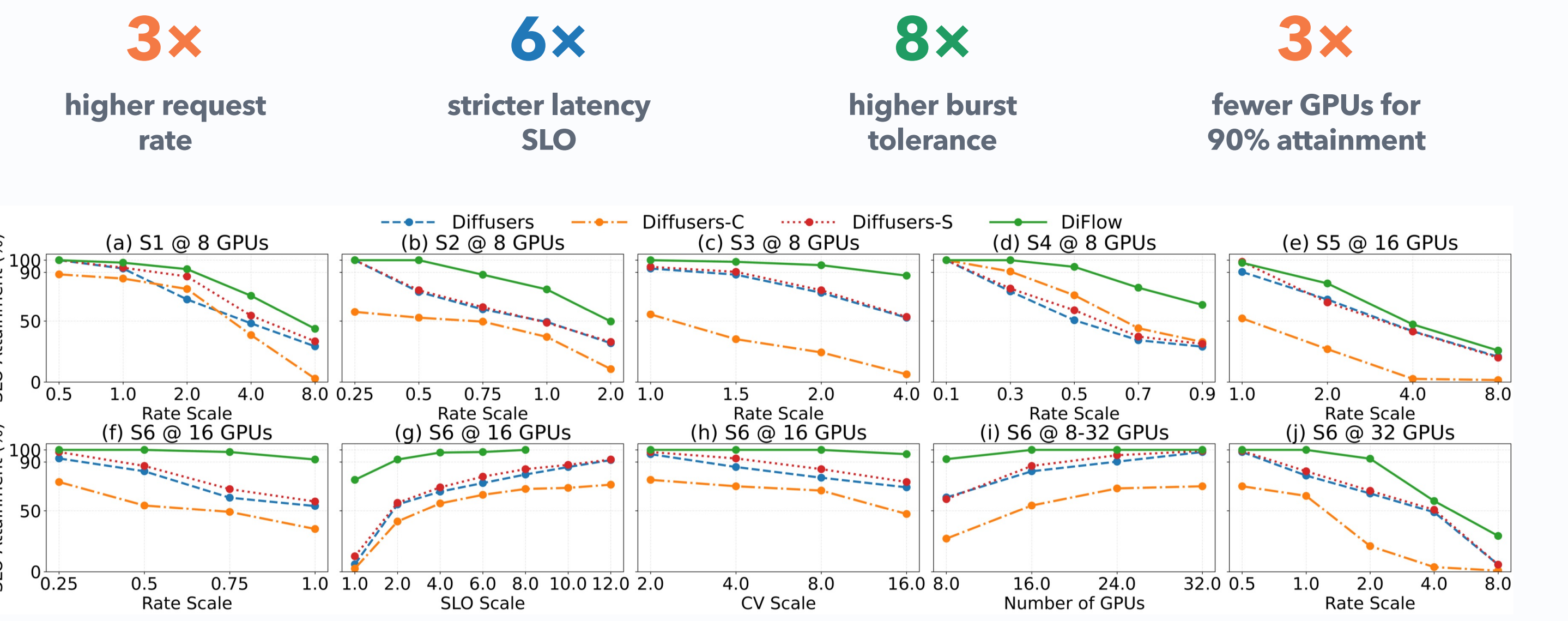


CUDA tensors move directly over NVLink or RDMA. The coordinator carries metadata rather than tensor payloads.

- Model sharing** Batch compatible nodes across workflows on resident model replicas.
- Adaptive execution** Use extra GPUs when parallel execution improves deadline feasibility.
- Admission control** Abort requests whose remaining critical path already exceeds the SLO.

Evaluation

Six deployment settings use SD3 and Flux models on 8-32 NVIDIA H800 GPUs. Requests follow a production trace; the default SLO is 2x solo latency.



At the 90% attainment target, DiFlow sustains higher load, tighter deadlines, and burstier arrivals while using fewer GPUs.

¹ S. Li et al., Katz: Efficient Workflow Serving for Diffusion Models with Many Adapters, USENIX ATC 2025.