DiT-Serve and DeepCoder: Enabling Video and Code Generation at Scale

Rachel Xin

Electrical Engineering and Computer Sciences
University of California, Berkeley

Technical Report No. UCB/EECS-2025-46
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2025/EECS-2025-46.html

May 9, 2025

DiT-Serve and DeepCoder: Enabling Video and Code Generation at Scale

by Rachel Xin

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

Committee:

Professor Joseph Gonzalez
Research Advisor

5/9/2025

(Date)

* * * * * * *

Professor Ion Stoica
Second Reader

5/9/2025

(Date)

DiT-Serve and DeepCoder: Enabling Video and Code Generation at Scale[1]

by

Rachel Xin

A thesis submitted in partial satisfaction of the

requirements for the degree of

Master of Science

in

Electrical Engineering and Computer Sciences

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Joseph Gonzalez, Advisor
Professor Ion Stoica

Spring 2025

[1]This thesis is adapted from **DiT-Serve: An Efficient Serving Engine for Diffusion Transformers**[13] and **DeepCoder: A Fully Open-Source 14B Coder at O3-mini Level**[8]. It is recommended to cite these papers over this report.

DiT-Serve and DeepCoder: Enabling Video and Code Generation at Scale[1]

[1]This thesis is adapted from **DiT-Serve: An Efficient Serving Engine for Diffusion Transformers**[13] and **DeepCoder: A Fully Open-Source 14B Coder at O3-mini Level**[8]. It is recommended to cite these papers over this report.

Abstract

DiT-Serve and DeepCoder: Enabling Video and Code Generation at Scale[2]

by

Rachel Xin

Master of Science in Electrical Engineering and Computer Sciences

University of California, Berkeley

Professor Joseph Gonzalez, Advisor

This thesis presents a unified approach to building scalable and intelligent generative systems by advancing two key areas: high-throughput inference for diffusion models and reinforcement learning-based training for large language models (LLMs). In the first part, we introduce **DiT-Serve**[13], a novel system architecture for video diffusion transformers designed to meet the high demand of generative workloads. We leverage denoising-level parallelisms to enable efficient batching and present a new attention algorithm. By addressing challenges in staggered request handling, multi-resolution support, and improvement of GPU utilization, **DiT-Serve**[13] contributes to the improvement of throughput and responsiveness of generative systems in production environments.

In the second part, we transition to explore how reinforcement learning (RL) can be used to enhance the reasoning capabilities of LLMs. We present a training methodology centered on curating high-quality, verifiable coding data and algorithmic and system optimizations. This integration of environment-based feedback and effective reward calculation represents the future of transforming small language models into powerful reasoning models. We introduce **DeepCoder**[8], a code reasoning model that matches the performance of much larger models, illustrating the potential of RL-based scaling.

By coupling efficient inference infrastructure with intelligent training strategies, this thesis contributes toward the democratization of generative AI—making high-performance systems more accessible, interpretable, and adaptable across domains.

[2]This thesis is adapted from **DiT-Serve: An Efficient Serving Engine for Diffusion Transformers**[13] and **DeepCoder: A Fully Open-Source 14B Coder at O3-mini Level**[8]. It is recommended to cite these papers over this report.

To my family, advisors, and research team.

I would like to extend my sincerest gratitude to UC Berkeley Sky Computing Lab, particularly the Video Serve and Agentica project teams for their intellectual contributions and enriching collaboration. I would like to thank Prof. Joseph Gonzalez and Prof. Ion Stoica for their thoughtful guidance. Most of all, I extend my heartfelt thanks to Dr. Michael Luo, whose support and mentorship have been instrumental to every stage of this journey.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The rapid progress in generative modeling has given rise to two critical trends: the emergence of diffusion transformer models as the state-of-the-art tools for image and video generation, and the evolution of large language models (LLMs) into systems capable of multi-step reasoning. Despite these breakthroughs, real-world deployment remains challenging. Diffusion transformer models suffer from high workloads, and their intensive computation leads to inefficient GPU utilization and poor scalability, especially for video generation. Meanwhile, LLMs often require a dramatic increase in model size and billions of additional parameters to yield marginal improvements in reasoning ability.

This thesis investigates building scalable and intelligent generative systems to address these limitations. Starting with the inference side for video generation, we focus on the bottlenecks in video generation pipelines. Due to the high variance in request lengths and resolutions, batching and GPU scheduling remains complicated. To address this, we present **DiT-Serve**[13], a Diffusion Transformer serving system designed to maximize throughput through efficient batching. The primary contributions are denoising level parallelism and a novel multi-GPU attention algorithm. Our work builds upon open-source video generation models, primarily OpenSora[17], CogVideoX[16], and Mochi[12], demonstrating gains in efficiency for heterogeneous workloads.

On the other hand, we focus on enabling smaller LLMs to perform complex reasoning tasks without requiring large model architectures. Specifically, we target the domain of code generation, where reasoning capabilities are critical. Our approach involves constructuing a RL pipline that integrates high-quality, verifiable coding datasets, efficient training algorithms, and a sparse reward function. As a culmination of this effort, we introduce **DeepCoder**[8], a 14-billion-parameter code reasoning model fine-tuned from Deepseek-R1-Distilled-Qwen-14B [1] via distributed reinforcement learning. Evaluating on LiveCodeBench[4], DeepCoder achieves a 60.6% Pass@1 accuracy, which is an 8% improvement in performance.

Together, these contributions aim to democratize scalable generative systems by improving both the efficiency of diffusion-based inference and the accessibility of RL-based LLM training.

Chapter 2

Background

This thesis tackles two converging frontiers of generative AI: (1) accelerating inference for video diffusion models, and (2) training LLMs capable of structured reasoning. To ground these contributions, this section first introduces the foundations of diffusion models and transformers in generative modeling, followed by a survey of recent state-of-the-art models for video generation. Secondly, reinforcement learning (RL) has become a cornerstone for training large language models, demonstrating promising results for scaling and accelerating intelligence.

2.1 Diffusion Models

Diffusion models have emerged as the emphasis for generative modeling, particularly for both image and video generation. Diffusion models operate by simulating a Markovian forward process that progressively adds noise to the data, followed by denoising steps to reconstruct the original signal. Transformers have become the state of the art architecture to create these videos.

Diffusion Process

Diffusion models rely on two primary processes: a *forward* diffusion processes that slowly corrupts data by adding Gaussian noise over T timesteps, and a *reverse* denoising process that then reconstructs the data from noise. In the context of our serving system, the reverse denoising process is crucial in inference execution. We will optimize the reverse diffusion process for real-time serving, where our architecture will enable the system to handle denoising steps across multiple requests simultaneously.

Transformers

Traditional diffusion models use convolutional networks during the denoising process. However, transformers have recently become the new state of the art particularly due to their

scalability and ability to model long-range dependencies. The use of transformer backbones allows both for better performance as well as higher-resolution generation tasks. In replace of convolutional layers, self-attention blocks allow the model to learn richer contextual relationships across both spatial and temporal dimensions. For text-to-video tasks, a text prompt is tokenized using a language tokenizer, embedded, and cross-attended during the denoising steps.

Models

Several state-of-the-art models have emerged that demonstrate high quality video generation. OpenSora[17] is a large-scale open-source text-to-video diffusion model that is inspired by both the architecture of DiT and Sora. Mochi[12] is a diffusion-based video model emphasizing on fast decoding by leveraging compact latent space. CogVideoX[16] uses a cascaded architecture to generate these videos, focused on long-range video generation. For our system, we integrated all three of these video generation models, as well as asymmetric diffuser models for image generation such as Stable Diffusion Version 3 [2].

2.2 Training LLMs Using RL

Code Reasoning Models

Recent explorations in using reinforcement learning to train LLMs had demonstrated improvements in reasoning tasks, such as solving math problems. Among the leading reasoning models is DeepSeek-R1 [1], which achieves performance comparable to the o1 model on reasoning tasks. Deepseek-R1 is trained using a combination of supervised fine-tuning and reinforcement learning (RL) and builds on top of strong base models (e.g. Qwen-14B).

Distributed RL: Verl

To train LLMs at scale, we leverage verl[11], a distributed reinforcement learning library for training language models. Verl provides a high-level interface for defining reward functions, trajectory sampling policies, and gradient updates.

In this work, we use an extended version called verl-pipe, which includes various optimizations to improve throughput and training stability. In particular, one-off pipelining is introduced, where we sacrifice the first RL iteration for sampling only, and then use that batch to train the next iteration. Reward calculation is also interleaved with sampling, which reduces overhead. More detailed information regarding regarding verl optimizations can be found in the DeepCoder blog [8]. The verl-pipeline is used to train our **DeepCoder** model from Deepseek-R1-Distill-Qwen-14B model.

Chapter 3

DiT-Serve: An Efficient Serving Engine for Diffusion Transformers

3.1 Motivation

Serving video diffusion models efficiently remains a major challenge due to the substantial computational demands of their long denoising sequences. Video inference introduces both spatial and temporal complexities that hinder GPU utilization and throughput.

Video diffusion requests are highly heterogeneous. Each request may differ in the number of denoising steps required, the video resolution, as well as the output quality. This variation among requests makes batching particularly difficult. Variable denoising steps results in requests terminating at different time steps resulting in more complicated static batching strategies. Higher resolution videos also correspond to longer token sequences during transformer-based inference, which prevents naive batching because of mismatched context lengths. While padding is a potential solution, it is inefficient and results in wasted computation.

Temporal Inefficiencies

The variability in requests creates significant obstacles to efficient temporal scheduling on GPUs. A naive batching strategy of aggregating all available requests into a batch leads to "temporal bubbles" of inefficiency, as illustrated in (Fig. 3.1). Since different requests terminate at different timesteps, GPUs are left underutilized when they idle for shorter tasks after completion while waiting for longer requests to finish processing before starting the next batch. This results in the entire batch being bottlenecked by the longest-running requests, significantly reducing throughput.

Additionally, requests arrive asynchronously in real-world settings. Serving systems must handle a mix of bursty and sparse traffic patterns, where requests can arrive at unpredictable intervals. Naive batch scheduling struggles to respond to dynamic workloads. Continuous

batching improves temporal GPU utilization by dynamically grouping incoming requests as they arrive.

Spatial Inefficiencies

In addition to temporal issues, spatial inefficiencies arise from how multi-GPU inference is handled. Standard approaches split the computation across attention heads, with each GPU computing the attention output for a subset of heads. However, for video requests where context lengths can be extremely long, this becomes a bottleneck.

Context parallelism is more efficient for long sequences, where the computation is split across the context length with block-wise parallel transformers. For multi-GPU deployment, Ring Attention[6] enables each GPU to process each block in a circular fashion, ensuring balanced workload distribution and maximizing spatial efficiency. However, Ring Attention falls short when considering the need to pad shorter requests together with longer ones, leading to wasted computation. This spatial inefficiency allowed us to come up with a new attention mechanism called Brick Attention in order to maximize spatial efficiency.

3.2 Overall Architecture and Design

We present DiT-Serve's overall architecture and then explore the two key components: (1) denoising level request coordinator and (2) a distributed, multi-GPU attention algorithm for load balancing requests with diverse context lengths.

Overview

DiT-Serve is a online serving engine that processes text-to-video Diffusion Transformers. DiT-Serve focuses on two primary objectives: (1) improving the end-to-end latency of each video generation request and (2) maximizing GPU utilization for providers.

Architecture

The basic architecture is designed to efficiently serve video diffusion requests at scale. Each incoming video request undergoes multiple rounds of Transformer-based model execution, corresponding to the denoising steps in the diffusion process. The core of DiT-Serve is a model execution engine managed by a central scheduler. The high-level workflow proceeds as follows:

1. **Request Ingestion:** Video generation requests arrive through a FastAPI interface. Each request is handled by a coroutine running its own diffusion algorithm instance.
2. **Scheduling:** As denoising steps progress, model execution requests are submitted to a shared scheduler. These requests may come from multiple coroutines and are queued for processing.

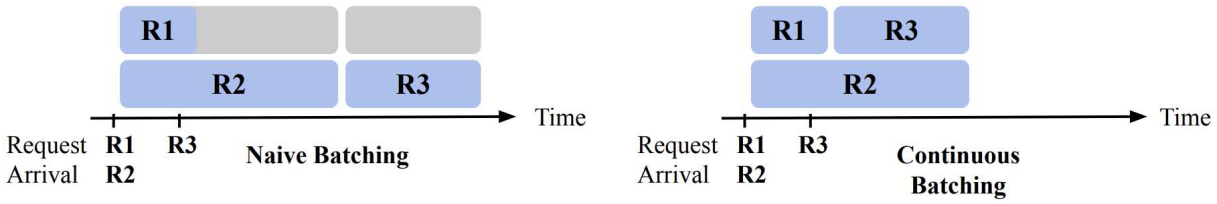

Figure 3.1: **Continuous Batching For Batch Size 2.** Naive batching suffers from poor GPU utilization. When request 1 is finished, the GPU must wait for the longer request 2 to finish processing, before it can process request 3. However, with continuous batching, temporal GPU utilization is improved as request 3 can be batched together with request 2 once request 1 is finished running.

3. **Execution Loop:** A persistent model loop polls the scheduler for available requests. When enough compatible requests are available, they are batched and executed by the engine.

4. **Response Handling:** Once a model call completes, results are returned to the corresponding coroutine, allowing it to proceed to the next denoising step or complete the request.

Continuous Batching

We introduce a scheduling strategy termed **Continuous Batching**, inspired by efficient memory management techniques recently proposed for serving large language models [5]. Since a single diffusion request relies on multiple calls to the same model in sequence, where the number of calls is a variable parameter, there is the opportunity to batch together distinct requests. Unlike traditional batching methods, continuous batching does not require all batched requests to complete simultaneously; instead, it dynamically manages video diffusion transformer requests by continuously adding new high-priority requests into the batch and removing those that have completed processing, resulting in better GPU utilization as shown in (Fig. 3.1). Specifically, at each timestep, we batch together the highest-priority video generation requests according to our scheduling policy, and execute their denoising step in one batch. All requests are then returned to the pool. This approach improves GPU utilization, reduces unnecessary idle time, and improves overall throughput and inference efficiency compared to conventional batching methods.

Scheduling Policy

The choice of scheduling policy significantly impacts average request latency. While **Shortest Job First (SJF)** is classically known to minimize average wait time, applying it effectively in the context of video diffusion inference—especially on multi-GPU clusters—requires a more nuanced definition of job length.

Instead, we schedule based on a **Shortest Remaining Processing Time First (SRPTF)** policy, tailored to video diffusion workloads. We define the length $L(\text{req}_t)$ of a video request as the product of the number of remaining denoising steps and the tokenized sequence length of the video. Let t be the current denoising timestep and T_{total} the total number of timesteps in the diffusion process. Then the number of steps remaining is $T_{\text{total}} - t$, and $S(\text{req}_t)$ denotes the tokenized sequence length for the request at timestep t. Together, the effective remaining cost of a request is:

$$L(\text{req}_t) = \big(T_{\text{total}} - t\big) \cdot S(\text{req}_t) \tag{3.1}$$

This cost formulation captures both time and spatial resource requirements, allowing the scheduler to prioritize shorter and less compute-intensive requests to improve overall system throughput and responsiveness.

Brick Attention

Ring Attention [6] is an attention mechanism designed to scale Transformer models to large context sizes by distributing computation across multiple GPUs arranged in a ring topology. Each GPU processes a slice of the input sequence and passes its computed key-value (KV) pairs to the next device in the ring. This approach eliminates the need to materialize the full attention matrix, significantly reducing memory overhead and latency during inference.

However, standard Ring Attention assumes that all input sequences share the same context length L, making it incompatible with real-world workloads where video requests often vary in resolution and number of frames—resulting in heterogeneous sequence lengths.

To address this, we introduce Brick Attention, a generalization of Ring Attention that supports batching across requests with varying context lengths. Rather than enforcing a uniform ring size, Brick Attention dynamically forms multiple independent rings of different sizes (e.g., 1, 2, 4, or 8 GPUs), and allows each GPU to participate in several rings concurrently. Each request is assigned to a dedicated ring based on its context size, and the KV communication pattern is carefully modified to isolate traffic across rings. This process is equivalent to calling ring attention on all requests.

A central Brick Coordinator schedules requests and assigns them to rings that best match their computational profile. During execution, each ring independently processes its assigned request using Ring Attention. Importantly, Brick Attention introduces no additional communication overhead and preserves the latency benefits of Ring Attention while enabling high GPU utilization across diverse workloads.

By supporting fine-grained batching and parallelism over heterogeneous requests, Brick Attention significantly improves both scalability and throughput for multi-GPU diffusion inference systems.

More detailed information on the specific implementation can be found in our paper.

3.3 Evaluation

We evaluate the performance by simulating realistic workloads representative of diverse scenarios encountered by state-of-the-art Video Diffusion models. Specifically, we generate synthetic requests using the following configurations and distributions.

Request Generation. Requests are generated using an exponential distribution, parameterized by varying lambda (λ) values to represent different request arrival rates. These request inter-arrival times effectively simulate realistic traffic conditions ranging from moderate to highly demanding workloads.

Request Specifications. Each generated request includes randomly selected parameters representative of realistic usage:

- **Prompts.** Requests randomly choose from a set of predefined textual prompts, such as "A beautiful waterfall," and "A Chinese Lunar New Year celebration video with a Chinese dragon."
- **Resolution.** Requests are randomly assigned resolutions from the set 240p, 360p, 480p, 720p, following a uniform categorical distribution.
- **Sampling Steps.** The number of denoising steps per request is chosen from the set 10, 20, 40, 80, according to a uniform distribution.
- **Frame Count.** Frame counts differ between models due to varying model sizes, configurations, and memory constraints. Specifically, we use 41 frames for CogVideo, 31 frames for Mochi, and 48 frames for Open-Sora.

Baselines. To rigorously evaluate our proposed system, we benchmark its performance against four distinct baselines. Each baseline leverages the same maximum batch size constraint to guarantee a fair comparative analysis. The baselines are described in detail as follows:

- **Naive + Ring Attention.** This baseline does not utilize batching; instead, each incoming request is individually assigned to GPUs based on its computational demand, determined by resolution and frame count (requiring 1, 2, 4, or 8 GPUs). Requests are processed sequentially according to a First-Come First-Served (FCFS) policy. A significant limitation of this method is head-of-line (HoL) blocking, where longer or more resource-intensive requests delay subsequent shorter tasks. To distribute computational load efficiently across multiple GPUs for single requests, ring attention is employed, allowing parallel processing within GPU clusters.
- **Naive Batching + Ring Attention.** This method extends the naive approach by introducing batching to mitigate head-of-line blocking. Requests are grouped into

batches according to identical configurations, such as matching resolutions and frame counts, with each batch having up to a maximum size of 5. While this approach still processes batches in an FCFS manner, batching improves GPU utilization and throughput by concurrently processing similar requests. Ring attention is applied to these batches, facilitating effective distribution of the workload across multiple GPUs.

- **Continuous Batching + Brick Attention + FIFO.** This baseline incorporates our proposed continuous batching strategy, which dynamically adjusts batch composition in real-time. Incoming high-priority requests are continuously integrated into existing batches, while completed requests are promptly removed, enhancing overall system responsiveness. Brick Attention, a method optimized for varying context lengths and resource demands, replaces ring attention to better handle diverse request types within multiple GPU rings. Scheduling is performed using a First-In-First-Out (FIFO) policy, ensuring equitable treatment of incoming tasks while dynamically maintaining efficient GPU utilization.

- **Continuous Batching + Brick Attention + SRPTF.** Building upon the previous continuous batching approach, this baseline differentiates itself through the adoption of the Shortest Remaining Processing Time First (SRPTF) scheduling policy. Unlike FIFO, SRPTF explicitly prioritizes requests with fewer remaining computational demands, calculated based on the product of their remaining denoising steps and sequence length. This prioritization strategy minimizes overall waiting times, particularly benefiting smaller or near-completion tasks, and significantly enhances throughput and turnaround times for diverse workloads.

Performance

Figure 3.2 illustrates the comparative performance in terms of normalized latency and job completion times across various scheduling and batching strategies for three Video Diffusion models: Open-Sora, Mochi, and CogVideo. The results clearly indicate that Continuous Batching combined with Brick Attention and SRPTF scheduling consistently outperforms other methods, achieving the lowest latency and job completion times. This performance advantage becomes increasingly pronounced under moderate to high request rates, emphasizing superior scalability and throughput. In contrast, Naive FIFO and Naive Batching strategies demonstrate significantly higher latencies due to inefficient GPU utilization and pronounced head-of-line (HoL) blocking, which exacerbates delays in processing subsequent requests.

Specifically, for Open-Sora, the Continuous Batching strategy integrated with Brick Attention and SRPTF achieves latency reductions of approximately 2-3× compared to naive approaches at moderate to high request rates. Similar significant improvements are observed with Mochi and CogVideo models, underlining the general applicability and effectiveness of our proposed scheduling methodology in various practical video diffusion contexts.

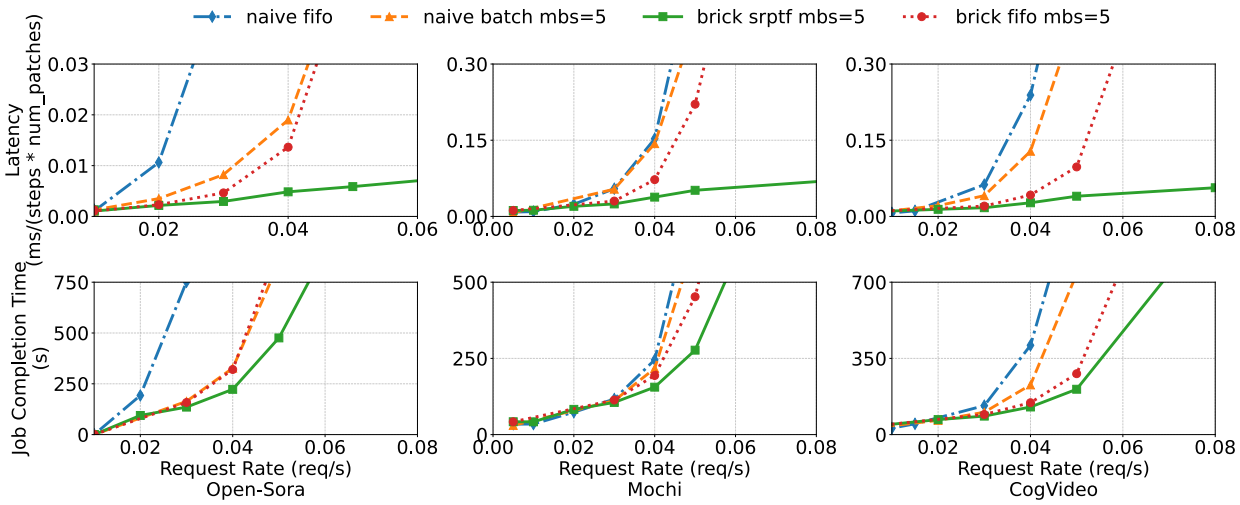

Figure 3.2: **Main Results.** Average latency and average job completion time for different Video Diffusion models.

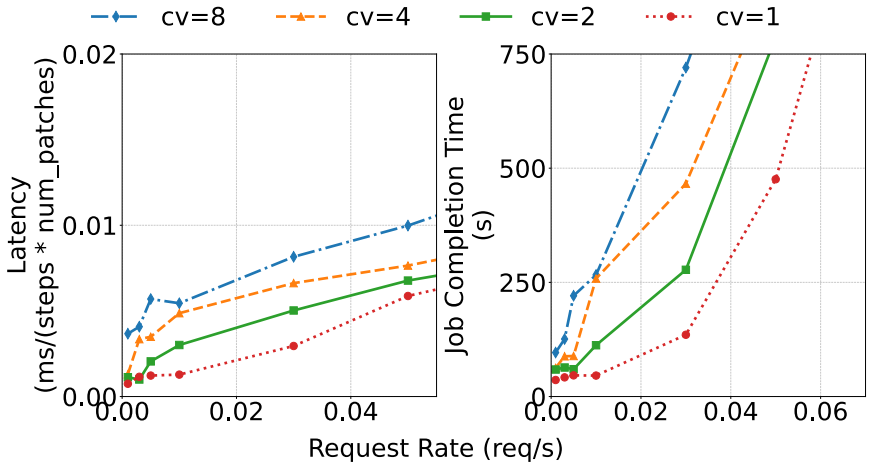

Figure 3.3: **Impact of Requests Burstiness.** DiT-Serve maintains robust performance across varying levels of request burstiness. Ablation experiments run Open-Sora on synthetic workload.

Workload Ablation

We ablate the effect of arrival burstiness by varying the coefficient of variation of request inter-arrival times, drawing arrivals from a Gamma Distribution. This models a range from regular (CV = 1) to highly bursty workloads (CV = 8) to capture variability observed in real-world inference systems. Figure 3.3 shows the effect on normalized latency and average job completion time. We observe that across all CVs, both latency and JCT remain

relatively low with lower request rates, indicating the system absorbs moderate levels of burstiness. As arrival rate increases, performance degrades with CV but the system maintains its scaling. These results demonstrate our baseline is robust across a range of realistic arrival distributions, with predictable scaling behavior and no sharp performance cliffs.

Extension to Image Models

While our current implementation targets video diffusion workloads, the underlying components are general and extensible, particularly with text-to-image generation models. We integrated asymmetric diffuser models such as Stable Diffusion, a latent diffusion model introduced to generate high-resolution images. By extending our logic, we envision support for both video and image generation with the same infrastructure, enabling inference across various visual modalities.

Chapter 4

DeepCoder: A Code Reasoning Model Finetuned Using RL

4.1 Motivation

Overview

The past year has seen rapid advancements in scaling large language models (LLMs) for reasoning-heavy domains, such as mathematics. Reinforcement learning (RL) has played a pivotal role in this progress, such as AReaL[10] and Light-R1[14], where targeted feedback and trajectory-level optimization yields models that outperform baselines in complex reasoning tasks.

Despite these gains, the coding domain has lagged behind. While LLMs have shown promise in competitive programming benchmarks, the lack of scalable, high-quality reward signals has made it difficult to apply RL effectively. Unlike math problems, where correctness is more clearly defined through proofs and evaluations, code generation tasks require more complexity, particularly through execution-based verification and environmental control, all of which pose additional challenges for reward modeling and sample efficiency.

Deepcoder explores how reinforcement learning can push the boundaries of code generation. Our goal is to develop scalable RL pipelines that close the performance gap in coding tasks and enable improved reasoning in smaller language models.

4.2 Dataset Curation

In our preliminary experiments, we evaluated a range of existing coding datasets commonly used for training and benchmarking code generation models. However, there were certain limitations. Datasets like KodCode[15] and LeetCode were too easy for our models, resulting in reward saturation and limiting opportunities for policy improvement. Additionally, other datasets like APPS[3] and CodeContests suffered from noise and inconsistency, resulting in

missed test cases and flawed ground-truth solutions. These issues led to misleading and null reward signals that undermined reinforcement learning training.

Dataset. To address these limitations, we curated a high-quality training set optimized for reliable, reward-based learning. Our dataset consists of:

- TACO Verified problems.
- Verified problems from PrimeIntellect's SYNTHETIC-1 dataset.
- LiveCodeBench problems submitted between May 1, 2023 and July 31, 2024.

Filtering Pipeline. To ensure that the data was of high quality, our filtering pipeline consists of:

- **Programmatic Verification.** We filter our datasets to include only those problems whose official solutions pass all unit tests.
- **Test Filtering.** Each problem must include at least 5 unit tests. Problems with fewer tests led to memorization where the model recognized common test cases.
- **Deduplication.** We remove duplicate problems across different datasets to avoid contamination.

After filtering, we are left with 24K high-quality coding problems that are used for our RL training, with 7.5K problems from TACO Verified, 16K problems from PrimeIntellect's SYNTHETIC-1, and 600 from LiveCodeBench.

Reward Calculation

Our reward function is based on a sparse **Outcome Reward Model (ORM)**, designed to encourage robust and generalizable code generation while avoiding common pitfalls such as reward hacking. We intentionally do not assign partial credit (such as proportional rewards like K/N for passing K out of N tests). Such heuristics often lead models to overfit to public test cases or exploit shortcuts that do not generalize well, which negatively impacts reinforcement learning convergence. Thus, our reward function is as follows:

- **Reward = 1.** The generated code must successfully pass all the sampled unit tests. For efficiency, we sample the 15 most challenging tests for each problem, identified by the length of their input strings.
- **Reward = 0.** The model receives no reward if it fails any test case or if the output format is invalid. (ie. missing `python [CODE])`. Each test case is also assigned a timeout of 6-12 seconds.

More detailed information on the specific training recipe can be found in our blog.

4.3 Evaluation

Table 4.1: **Model Performance Comparison Across Benchmarks**

Model	**Codeforces***		**LCB**	**HumanEval+**	**AIME**
	Rating	Percentile	(8/1/24–2/1/25)	Pass@1	2024
DeepCoder-14B-Preview	**1936**	**95.3**	**60.6**	**92.6**	**73.8**
DeepSeek-R1-Distill-Qwen-14B	1791	92.7	53.0	92.0	69.7
O1-2024-12-17 (Low)	**1991**	**96.1**	59.5	90.8	**74.4**
O3-Mini-2025-1-31 (Low)	1918	94.9	**60.9**	92.6	60.0
O1-Preview	1658	88.5	42.7	89.0	40.0
DeepSeek-R1	1948	95.4	62.8	92.6	79.8
Llama-4-Behemoth†	–	–	49.4	–	–

*Codeforces metrics estimated from contest performance.
†No available data for HumanEval+ or AIME benchmarks.

We evaluate DeepCoder-14B-Preview [8] across a diverse set of coding and reasoning benchmarks, including LiveCodeBench (LCB) [4], Codeforces, HumanEval+ [7], and AIME 2024.

Despite its relatively modest scale (14 billion parameters), the model exhibits strong performance across all evaluated benchmarks. It achieves **60.6%** on LiveCodeBench and a Codeforces Rating of 1936, which places it on par with competitive baselines such as o3-mini and o1. Notably, although the model was not explicitly trained on mathematical tasks, its coding-derived reasoning capabilities generalize well to math, as reflected in a **73.8%** score on AIME 2024, which improved by **4.1%** on the base model.

These results highlight the model's robust generalization across both code synthesis and mathematical reasoning, reinforcing the effectiveness of RL-based training on verifiable coding datasets.

4.4 Future Steps

Moving forward, we aim to explore how reinforcement learning can enhance LLM-based agents by extending DeepCoder's capabilities toward interactive tool use and real-world software engineering workflows. In particular, we plan to integrate environments such as SWE-Gym[9], which simulate practical development tasks including bug fixing, code refactoring, and test generation. This direction will also allow us to investigate the nuances of effective tool usage, bringing DeepCoder closer to deployment in realistic engineering settings.

Chapter 5

Conclusion

This thesis presents a unified vision for building open, efficient, and intelligent generative systems—advancing the state of the art in both high-throughput inference infrastructure and reinforcement learning-based training for language models.

In the first part, we introduced **DiT-Serve**, a scalable efficient system architecture for video diffusion transformers. By leveraging denoising-level parallelism, continuous batching, and novel attention mechanisms, DiT-Serve achieves significant improvements in GPU utilization and responsiveness by targeting real-world challenges in multi-resolution heterogeneous generative workloads.

In the second part, we introduced **DeepCoder**, a reinforcement learning-trained LLM focused on code generation tasks. Through meticulous dataset curation, sparse reward design, and other system-level training optimizations, DeepCoder achieves 60.6% Pass@1 on LiveCodeBench—matching or surpassing much larger baselines like o3-mini. More importantly, its learned reasoning capabilities generalize beyond code, as demonstrated by strong performance on math benchmarks such as AIME 2024. This progress demonstrates that RL can scale and improve the performance of smaller models.

Together, these two components—**DiT-Serve** for scalable inference and **DeepCoder** for LLM training—embody a holistic approach to building next-generation generative systems. We believe that the future of AI lies not only in advancing capabiltity, but also in ensuring that these capabilities are accessible. Through **DiT-Serve**, we democratize high-throughput video generation; through **DeepCoder**, we democratize RL for LLMs. Together, they represent a step toward building open-source, intelligent, and scalable generative AI.

Bibliography

[1] DeepSeek-AI et al. *DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning*. 2025. arXiv: 2501.12948 [cs.CL]. URL: https://arxiv.org/abs/2501.12948.

[2] Patrick Esser et al. *Scaling Rectified Flow Transformers for High-Resolution Image Synthesis*. 2024. arXiv: 2403.03206 [cs.CV]. URL: https://arxiv.org/abs/2403.03206.

[3] Dan Hendrycks et al. "Measuring Coding Challenge Competence With APPS". In: *NeurIPS* (2021).

[4] Naman Jain et al. "LiveCodeBench: Holistic and Contamination Free Evaluation of Large Language Models for Code". In: *arXiv preprint arXiv:2403.07974* (2024).

[5] Woosuk Kwon et al. *Efficient Memory Management for Large Language Model Serving with PagedAttention*. 2023. arXiv: 2309.06180 [cs.LG]. URL: https://arxiv.org/abs/2309.06180.

[6] Hao Liu, Matei Zaharia, and Pieter Abbeel. *Ring Attention with Blockwise Transformers for Near-Infinite Context*. 2023. arXiv: 2310.01889 [cs.CL]. URL: https://arxiv.org/abs/2310.01889.

[7] Jiawei Liu et al. "Is Your Code Generated by ChatGPT Really Correct? Rigorous Evaluation of Large Language Models for Code Generation". In: *Thirty-seventh Conference on Neural Information Processing Systems*. 2023. URL: https://openreview.net/forum?id=1qvx610Cu7.

[8] Michael Luo et al. *DeepCoder: A Fully Open-Source 14B Coder at O3-mini Level*. Notion Blog, https://pretty-radio-b75.notion.site/DeepCoder-A-Fully-Open-Source-14B-Coder-at-O3-mini-Level-1cf81902c14680b3bee5eb349a512a51. 2025.

[9] Jiayi Pan et al. *Training Software Engineering Agents and Verifiers with SWE-Gym*. 2024. arXiv: 2412.21139 [cs.SE]. URL: https://arxiv.org/abs/2412.21139.

[10] Ant Research RL Lab. *AReaL: Ant Reasoning RL*. https://github.com/inclusionAI/AReaL. 2025.

[11] Guangming Sheng et al. "HybridFlow: A Flexible and Efficient RLHF Framework". In: *arXiv preprint arXiv: 2409.19256* (2024).

[12] Genmo Team. *Mochi 1*. https://github.com/genmoai/models. 2024.

[13] VideoServe Team. "DiT-Serve: An Efficient Serving Engine for Diffusion Transformers". Manuscript in preparation. 2025.

[14] Liang Wen et al. *Light-R1: Curriculum SFT, DPO and RL for Long COT from Scratch and Beyond*. 2025. arXiv: 2503.10460 [cs.CL]. URL: https://arxiv.org/abs/2503.10460.

[15] Zhangchen Xu et al. "KodCode: A Diverse, Challenging, and Verifiable Synthetic Dataset for Coding". In: (2025). arXiv: 2503.02951 [cs.LG]. URL: https://arxiv.org/abs/2503.02951.

[16] Zhuoyi Yang et al. "CogVideoX: Text-to-Video Diffusion Models with An Expert Transformer". In: *arXiv preprint arXiv:2408.06072* (2024).

[17] Zangwei Zheng et al. "Open-sora: Democratizing efficient video production for all". In: *arXiv preprint arXiv:2412.20404* (2024).