LATENT REASONING WITH NORMALIZING FLOWS

Anonymous authors
Paper under double-blind review

ABSTRACT

Large language models often improve reasoning by generating explicit chain-of-thought (CoT), demonstrating the importance of intermediate computation. However, textual CoT forces this computation through a discrete, serial, and communication-oriented token stream: each reasoning step must be verbalized before the model can proceed, even when the underlying update is semantic, uncertain, or only partially formed. Latent reasoning offers a higher-bandwidth alternative by performing intermediate computation in compact continuous states before committing to text. Yet existing latent-reasoning methods often sacrifice key advantages that make CoT effective in autoregressive language models, including native left-to-right generation, probabilistic sampling, compatibility with KV-cache decoding, and tractable likelihood estimation. We propose NF-CoT, a latent reasoning framework that preserves these advantages by modeling continuous thoughts with normalizing flows. NF-CoT instantiates a TarFlow-style normalizing flow inside the LLM backbone, defining a tractable probability model over compact continuous thoughts distilled from explicit CoT. Continuous-thought positions are generated by an NF head, while text positions are generated by the standard LM head within the same causal stream. This design provides exact likelihoods for latent thoughts, enables KV-cache-compatible probabilistic left-to-right decoding, and supports direct policy-gradient optimization in the latent reasoning space. On code-generation benchmarks, NF-CoT improves pass rates over explicit-CoT and prior latent-reasoning baselines while substantially reducing intermediate-reasoning cost, and a complementary study on six math benchmarks shows that NF-CoT transfers beyond code.

1 INTRODUCTION

Chain-of-thought (CoT) prompting has become a standard way to elicit reasoning in large language models (LLMs), improving performance by generating intermediate steps before the final answer (Wei et al., 2022; Kojima et al., 2022; Nye et al., 2021). One way to understand its effectiveness is that CoT introduces intermediate reasoning variables between the prompt and the answer, turning a direct input-output prediction into a conditioned prediction through a sampled reasoning path (Zelikman et al., 2022). In explicit CoT, these variables are represented as text tokens, making them naturally autoregressive, probabilistic, and likelihood-scored under the LLM. Yet text is a verbose, low-information-density medium for thought, making long reasoning costly and tying intermediate computation to surface forms (Hao et al., 2024).

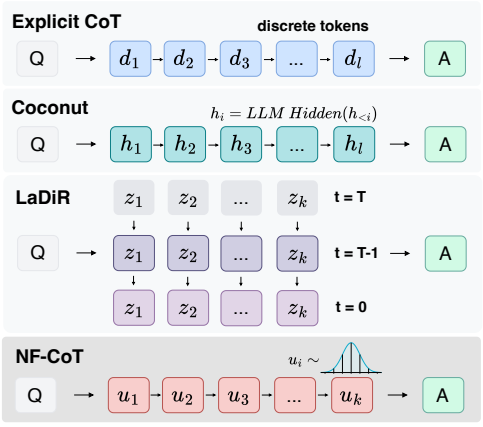

Figure 1: **Four chain-of-thought paradigms.**
Explicit CoT: *discrete* text tokens.
Coconut: *deterministic* hidden states.
LaDiR: diffusion-denoised latents.
NF-CoT: AR-sampled continuous latents.

This inefficiency has motivated latent CoT methods that replace textual reasoning traces with continuous or soft embedding states. Some methods keep reasoning close to the LLM by recycling hidden states or token-embedding mixtures as latent

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

thoughts (Hao et al., 2024; Zhang et al., 2026b; Tang et al., 2026); they are efficient, but remain local and tied to the model's token or embedding space. Others model stochastic continuous latents with diffusion models (Kang et al., 2026a;b), but require iterative denoising and lack native left-to-right likelihood modeling. The challenge is to learn latent CoT without giving up the native sampling, scoring, and decoding interface of language modeling in current LLMs.

In this work, we address this gap by proposing NF-CoT, a latent reasoning framework that gives continuous CoT the same modeling status as language tokens. NF-CoT places a scalable normalizing flow (Zhai et al., 2024; Gu et al., 2026a) inside the LLM's causal stream, using the LLM backbone as the shared autoregressive model for continuous thoughts and text answers. During training, NF-CoT turns explicit CoT supervision into continuous thoughts and optimizes an end-to-end likelihood objective over both latent reasoning and answers. The resulting tractable likelihood also enables policy-gradient optimization (Guo et al., 2025) directly in the continuous reasoning space. At inference, NF-CoT samples continuous thoughts left-to-right and decodes answers in the same causal stream. Together, this preserves the sampling, decoding, and likelihood interface of explicit CoT in a compact continuous space, at lower token cost.

We evaluate NF-CoT on MBPP, MBPP+, HumanEval, HumanEval+, and LiveCodeBench v6. NF-CoT improves pass rates over explicit-CoT and prior latent-reasoning baselines while substantially reducing intermediate-reasoning cost. Inference-time interventions show that the sampled thoughts carry task-specific content: zeroing them cuts HumanEval pass@1 from 84 to 34, and well-formed latents sampled for a different problem cost 7–9 points (Table 4). A complementary study on six math benchmarks shows that the same recipe transfers beyond code, improving over the reported base model on all six benchmarks (Section 4.3). We further study likelihood-based sampling and policy-gradient optimization in the continuous reasoning space. Our contributions are threefold:

- A likelihood-based formulation of continuous CoT that preserves the sampling, scoring, and decoding interface of explicit CoT.
- An architecture that jointly models latent thoughts and text answers with NF and LM heads over a shared LLM backbone.
- A unified optimization interface for continuous thoughts, enabling supervised likelihood training and policy-gradient refinement.

2 Preliminaries

2.1 Chain-of-thought reasoning

We consider supervised reasoning examples with a prompt q, a target answer sequence $x_{1:N}$, and, when available, an explicit chain-of-thought (CoT) trace $d_{1:L}$. A probabilistic view treats the trace as an intermediate variable:

$$p_\theta(x_{1:N} \mid q) = \sum_{d_{1:L}} p_\theta(d_{1:L} \mid q)\, p_\theta(x_{1:N} \mid q, d_{1:L}). \tag{2.1}$$

Since $d_{1:L}$ is a text sequence, it can be generated, sampled, and scored with the same autoregressive language-modeling interface as the answer. This is convenient but forces reasoning into verbose natural-language tokens at high token cost.

2.2 Continuous chain-of-thought

Continuous CoT replaces text rationales with continuous codes $e_{1:K} \in \mathbb{R}^{K \times D}$ of length K and dimension D, which play the role of the intermediate variable in Eq. (2.1).

Coconut (Hao et al., 2024) distills reasoning into hidden states fed back autoregressively, but the resulting thoughts are deterministic and do not define a distribution over reasoning paths. Another approach is to first learn a continuous code space with a VAE: an encoder maps $d_{1:L}$ together with K learnable queries to $q_\phi(e_{1:K} \mid d_{1:L})$, trained with

$$\mathcal{L}_{\text{VAE}} = \mathbb{E}_{q_\phi(e|d)}\left[-\log p_\psi(d \mid e)\right] + \beta\, \mathrm{KL}(q_\phi(e \mid d) \,\|\, \mathcal{N}(0, I)) \,. \tag{2.2}$$

After pretraining, a separate generative model (e.g., diffusion (Kang et al., 2026a)) is trained for $p_\theta(e_{1:K} \mid q)$. This gives a stochastic continuous representation but separates latent-thought generation from the native autoregressive likelihood interface used by explicit CoT.

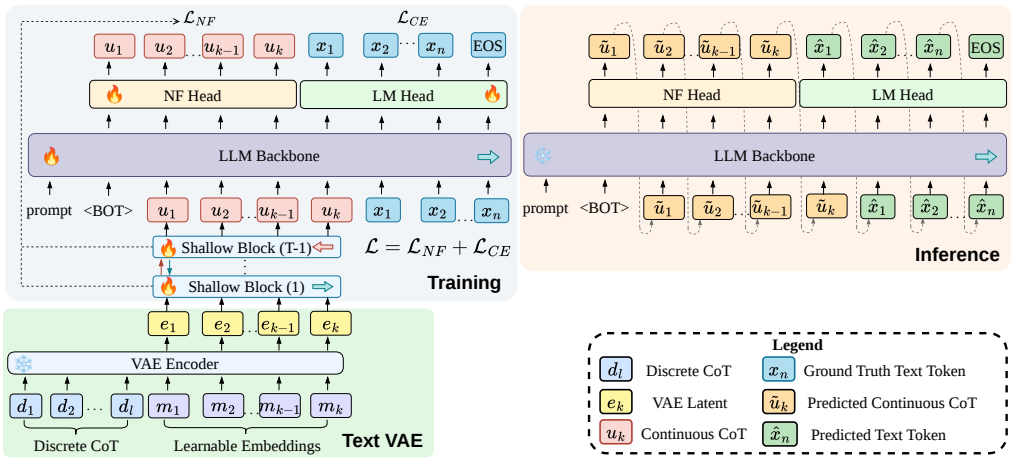

Figure 2: **Training and inference pipeline of NF-CoT.** Training: a frozen VAE encoder and shallow flow blocks map discrete CoT to continuous states $u_{1:k}$; one causal LLM pass over the prompt, <BOT>, $u_{1:k}$, and text $x_{1:n}$ optimizes the flow NLL and answer cross-entropy. Inference skips the VAE and shallow blocks: the LLM samples $\tilde{u}_{1:k}$ left-to-right and then decodes text in the same causal stream.

2.3 Scalable Normalizing Flows

Normalizing flows (Dinh et al., 2014; 2016) map data y to a simple base variable $z \sim \mathcal{N}(0, I)$ through an invertible network. For a conditional flow with context c,

$$z = f_\theta^{-1}(y; c), \qquad p_\theta(y \mid c) = p(z) \left| \det J_{f_\theta^{-1}}(y; c) \right|, \tag{2.3}$$

giving exact likelihood evaluation and direct sampling. Autoregressive flows (Kingma et al., 2016; Papamakarios et al., 2017) transform each position using only the context and previous positions, yielding a triangular Jacobian compatible with causal sequence modeling. Scalable variants such as TarFlow (Zhai et al., 2024) and STARFlow (Gu et al., 2026a) use Transformer blocks for high-dimensional structured latents. We build on this family to model continuous CoT with LLMs.

3 Method

3.1 Formulation

We start from the continuous CoT target $e_{1:K}$ defined in Sec. 2.2. Since this encoder-derived space is optimized for text compression rather than autoregressive generation, its prompt-conditioned distribution can be difficult to model directly. Prior work thus turns to diffusion models (Kang et al., 2026a), gaining expressive latent modeling at the cost of iterative sampling and a less direct likelihood interface.

NF-CoT takes a different route. We introduce an LLM-facing continuous thought space designed to make continuous thoughts easy to sample and score autoregressively. Following STARFlow (Gu et al., 2026a), we model continuous thought tokens $u_{1:K}$ with a causal Gaussian density:

$$p_\theta(u_{1:K} \mid q) = \prod_{i=1}^{K} \mathcal{N}\big(u_i;\, \mu_\theta(q, u_{<i}), \mathrm{diag}\big(\sigma_\theta^2(q, u_{<i})\big)\big), \tag{3.1}$$

where μ_θ and σ_θ are causal functions of the prompt and previous thoughts. This space supports natural left-to-right sampling with tractable likelihood.

The two spaces are connected through an invertible transformation $u_{1:K} = F_\theta(e_{1:K}; q)$, implemented as a stack of shallow autoregressive flow blocks following Gu et al. (2026a). Since F_θ is invertible, $u_{1:K}$ is information-equivalent to $e_{1:K}$ rather than a lossy compression. The full likelihood of the

original continuous CoT target can therefore be written through the reparameterized thought space:

$$\log p_\theta(e_{1:K} \mid q) = \log p_\theta(u_{1:K} \mid q) + \log \left|\det J_{F_\theta}(e_{1:K}; q)\right|. \tag{3.2}$$

This allows the shallow flow blocks and the autoregressive density over $u_{1:K}$ to be trained end-to-end under an exact likelihood. Thus, NF-CoT learns from the same continuous CoT targets as prior latent methods, but performs modeling and generation in an information-equivalent thought space that is easier to sample and score autoregressively.

3.2 Architecture

Architecturally, NF-CoT contains two flow components, as shown in Fig. 2: shallow flow blocks for reparameterizing e into u, followed by an LLM-parameterized deep autoregressive flow over u. The shallow blocks implement the invertible map $F_\theta : e_{1:K} \mapsto u_{1:K}$ and are trained from scratch with identity initialization ($F_\theta \approx I$). Because they are causal affine flows, their Jacobian is triangular and their log-determinant is tractable, contributing to the likelihood in Eq. (3.2).

The deep flow is implemented within the LLM backbone. Continuous thoughts $u_{1:K}$ are projected into the token embedding dimension and processed in the same causal stream as answer tokens, in the spirit of interleaved continuous-discrete generation (Shen et al., 2026). At continuous-thought positions, an NF head maps the hidden state to the parameters $\mu_\theta(q, u_{<i})$ and $\sigma_\theta(q, u_{<i})$ of the conditional density in Eq. (3.1). At answer positions, the standard LM head maps the hidden state to token logits. Thus, latent reasoning and answer generation share one causal backbone with separate heads for continuous and discrete variables. This unified design is the main architecture of NF-CoT; we report a dual-path variant, separating the latent-flow and answer-generation paths, as an ablation.

3.3 Learning

Supervision. We train NF-CoT from supervised reasoning examples with prompt q, explicit CoT trace $d_{1:L}$ when available, and answer $x_{1:N}$. As described in Sec. 2.2, we first pretrain a CoT encoder, freeze it, and use its posterior mean as the continuous CoT target $e_{1:K}$. The trainable model maps this target to the LLM-facing thought sequence $u_{1:K} = F_\theta(e_{1:K}; q)$ and learns to model both the latent reasoning path and the final answer.

Unified likelihood objective. We train latent reasoning and answer generation with a single supervised objective. The flow term assigns exact likelihood to the original continuous CoT target, $\mathcal{L}_{\text{flow}} = -\log p_\theta(e_{1:K} \mid q)$, computed through the reparameterized thought space using Eq. (3.2). The text term is the standard autoregressive likelihood conditioned on the continuous thoughts, $\mathcal{L}_{\text{text}} = -\sum_{j=1}^{N} \log p_\theta(x_j \mid q, u_{1:K}, x_{<j})$. The final objective is

$$\mathcal{L}_{\text{sup}} = \lambda_{\text{flow}}\mathcal{L}_{\text{flow}} + \lambda_{\text{text}}\mathcal{L}_{\text{text}}. \tag{3.3}$$

Both terms are likelihoods under the same causal LLM: the flow head scores continuous thoughts, while the LM head scores answer tokens.

Training curriculum. We use a two-stage curriculum. First, we freeze the LLM backbone and train only the shallow flow blocks and continuous-thought projection layers with the same supervised objective in Eq. (3.3), aligning the $e \to u$ reparameterization with the frozen LLM space. We then unfreeze all parameters and continue end-to-end training with the full objective.

Following the practice in TarFlow (Zhai et al., 2024), we add small Gaussian noise to $e_{1:K}$ before computing the flow loss. We find that this improves robustness to deterministic target codes.

3.4 Inference

At inference time, NF-CoT (Unified) does not run the shallow flow blocks F_θ, which are only needed during training (the dual-path ablation inverts them once after sampling; Appendix B.4). Instead, inference proceeds directly in u-space: given a prompt q, the model samples continuous thoughts left-to-right from the autoregressive density in Eq. (3.1). This sampling is stochastic, analogous to token sampling in a standard LLM, rather than a deterministic hidden-state update. After sampling $\tilde{u}_{1:K}$, the model switches to the LM head and continues answer generation in the same causal stream.

Because continuous thoughts and answer tokens share one causal backbone, the KV cache built during thought sampling can be carried into answer generation; our benchmark pipeline instead runs the two phases separately and re-encodes the prefix (Appendix B.4).

This inference procedure differs from prior latent-CoT methods in two ways. First, NF-CoT samples from an explicit likelihood model over continuous thoughts rather than deterministically recycling hidden states. Second, it avoids diffusion-style iterative latent generation: although training uses the invertible map between e and u, test-time reasoning happens directly in the information-equivalent u-space, which is shaped for autoregressive sampling and decoding.

3.5 Reinforcement learning

After supervised training, we further refine NF-CoT with reinforcement learning. During this stage, we freeze the shallow flow blocks and optimize the policy directly in the LLM-facing thought space u. For a sampled thought-answer trajectory $(\tilde{u}_{1:K}, \hat{x}_{1:\hat{N}})$, the policy likelihood decomposes as

$$\log \pi_\theta(\tilde{u}, \hat{x} \mid q) = \log p_\theta(\tilde{u}_{1:K} \mid q) + \log p_\theta(\hat{x}_{1:\hat{N}} \mid q, \tilde{u}_{1:K}). \tag{3.4}$$

The first term is the Gaussian autoregressive likelihood from the NF head, and the second term is the token likelihood from the LM head. This makes policy-gradient optimization directly applicable to both continuous thoughts and text answers.

In our experiments, we use a GRPO-style objective over groups of sampled trajectories. Given a reward $R(q, \hat{x})$, such as unit-test feedback for code generation, we compute a group-normalized advantage $A(q, \tilde{u}, \hat{x})$ and update the policy via the policy gradient

$$\nabla_\theta J_{\mathrm{RL}} = \mathbb{E}\left[A(q, \tilde{u}, \hat{x})\, \nabla_\theta \log \pi_\theta(\tilde{u}, \hat{x} \mid q)\right]. \tag{3.5}$$

Since the shallow blocks are frozen, this refinement is equivalent to applying policy gradients to the Gaussian likelihood over u together with the token likelihood over answers in a single update.

4 Experiments

We evaluate NF-CoT on code generation, a reasoning-intensive task with standard correctness metrics. We compare with strong baselines, including autoregressive CoT models, diffusion-based latent reasoners, and recent continuous-reasoning methods. All methods use the same setup with pass@1 and pass@k on multiple benchmarks. We additionally report a complementary math-reasoning study (Section 4.3) to test transfer beyond code.

4.1 Experimental setup

Datasets and evaluation. We fine-tune NF-CoT on Ling-Coder (Team et al., 2025), using 733K of its Python instruction-following samples. We evaluate on five Python code benchmarks: HumanEval (Chen et al., 2021), MBPP (Austin et al., 2021), their EvalPlus versions HumanEval+ and MBPP+ (Liu et al., 2023), and LiveCodeBench v6 (Jain et al., 2025). HumanEval+ and MBPP+ add more test cases for stricter evaluation. For each problem, we sample 16 solutions and report the average pass@1 to reduce variance. For the RL stage, we further train on data from AceCoder (Zeng et al., 2025) and KodCode (Xu et al., 2025). Decontamination checks are in Appendix C.1.

Baselines. We compare NF-CoT with strong open-source models and latent reasoning baselines. Autoregressive baselines include Qwen2.5-Coder (Hui et al., 2024), OpenCoder (Huang et al., 2025), OlympicCoder (Hugging Face, 2025), and Seed-Coder (Seed et al., 2025); diffusion baselines (Ho et al., 2020) include Dream (Ye et al., 2025), LLaDA (Nie et al., 2026), Diffu-Coder (Gong et al., 2026), Dream-Coder (Xie et al., 2025), and d1 (Zhao et al., 2026); we also include the looped latent reasoning model Ouro (Zhu et al., 2025). Under the same Qwen3-8B-Base backbone (Yang et al., 2025), we compare with Standard SFT, Soft Thinking (Zhang et al., 2026b), TaH+ (Fu et al., 2025), LaVAE (replaces our flow with an L_2-objective VAE) and LaDiR (Kang et al., 2026a) (replaces our flow with latent diffusion). We report two variants, NF-CoT (Dual-Path) and NF-CoT (Unified).

Experiment details. We use Qwen3-8B-Base with $N = 64$ latent slots and a flow stack of $K = 5$ shallow MetaBlocks (alternating Identity/Flip permutations) followed by the deep LLM block. Full training and inference details are in Appendix B; Qwen3-4B-Base results are in Appendix C.5.

Table 1: Pass@1 results on code generation benchmarks. Latent reasoning methods are highlighted in light yellow. Improvements over the base model are shown in green, and decreases in red. Best results across all reported methods are in **bold** and second-best results are underlined. The grey ± row gives the mean and standard deviation over three additional training seeds (43–45) evaluated under the same protocol. *Results are from the original papers.

Model / Method	Base Model	MBPP	MBPP+	HumanEval	HumanEval+	LCB v6	Avg.
Open-Sourced Models							
Autoregressive Coding Models							
Qwen 2.5 Coder*	Qwen2.5-7B	75.9	62.9	66.5	60.4	26.8	58.5
OpenCoder*	from scratch (8B)	79.9	70.4	66.5	63.4	29.6	62.0
OlympicCoder*	Qwen2.5-7B	80.0	66.4	82.1	76.9	37.3	68.5
Seed-Coder*	from scratch	82.0	69.0	77.4	68.3	28.4	65.0
Diffusion Language Models							
Dream*	Qwen2.5-7B	68.7	57.4	56.7	50.0	18.6	–
LLaDA*	from scratch (8B)	50.1	42.1	35.4	30.5	12.4	–
Diffu-Coder*	Qwen2.5-Coder-7B	75.1	61.9	72.0	65.2	24.5	59.7
Dream-Coder*	Qwen2.5-Coder-7B	75.9	61.6	66.5	60.4	21.4	57.2
d1*	LLaDA+RL	39.0	–	45.5	–	–	–
Looped Latent Reasoning Models							
Ouro*	from scratch (2.6B)	80.4	66.6	78.2	70.7	**38.7**	66.9
Method Comparison							
Non-RL Methods							
Base Model	Qwen3-8B-Base	60.5	53.8	78.2	68.6	17.7	55.8
Standard SFT	Qwen3-8B-Base	63.3+2.8	52.7-1.1	84.6+6.4	69.5+0.9	18.9+1.2	57.8+2.0
Soft Thinking	Qwen3-8B-Base	64.2+3.7	53.1-0.7	85.0+6.8	71.2+2.6	–	–
TaH+	Qwen3-8B-Base	65.6+5.1	56.5+2.7	85.8+7.6	74.3+5.7	–	–
LaVAE	Qwen3-8B-Base	42.0-18.5	30.2-23.6	47.8-30.4	32.8-35.8	10.8-6.9	32.7-23.1
LaDiR	Qwen3-8B-Base	66.8+6.3	59.5+5.7	**87.4**+9.2	73.2+4.6	21.3+3.6	61.6+5.8
NF-CoT (Dual-Path)	Qwen3-8B-Base	77.5+17.0	66.2+12.4	82.9+4.7	77.7+9.1	21.6+3.9	65.2+9.4
NF-CoT (Unified)	Qwen3-8B-Base	**83.9**+23.4	**72.1**+18.3	85.8+7.6	**78.3**+9.7	25.4+7.7	**69.1**+13.3
↪ 3 seeds (mean ± std)	Qwen3-8B-Base	84.2±0.8	72.1±0.5	82.6±0.4	77.2±0.9	25.2±0.7	68.3±0.5
↪ RL	Qwen3-8B-Base	85.4+24.9	73.3+19.5	86.7+8.5	80.2+11.6	26.1+8.4	70.3+14.5

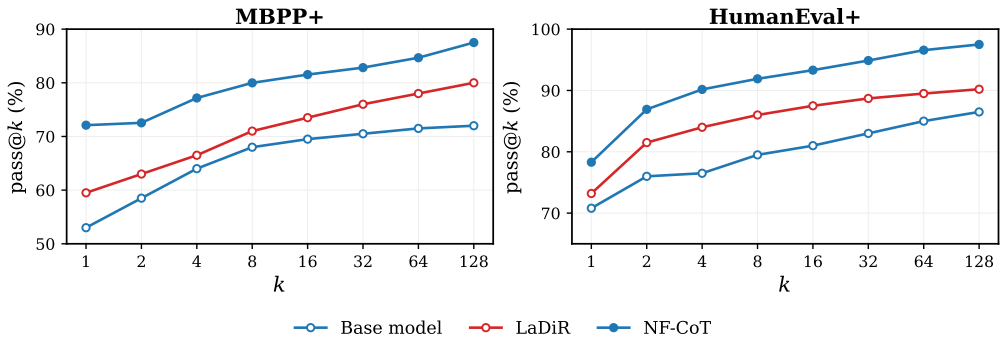

Figure 3: Pass@k scaling on MBPP+ and HumanEval+. NF-CoT outperforms the base model and LaDiR, and continues to improve with larger k.

4.2 Main Results

Pass@1 improvements on code generation. Table 1 reports pass@1 on five code benchmarks. NF-CoT (Unified) achieves the best average performance, improving Qwen3-8B-Base from 55.8 to 69.1 (+13.3%), while NF-CoT (Dual-Path) also improves the base by +9.4%. With the same backbone and training data, Standard SFT improves the five-benchmark average only from 55.8 to 57.8, while NF-CoT (Unified) reaches 69.1, a +11.3 gain over Standard SFT. Among prior latent reasoning methods, NF-CoT (Unified) outperforms the strongest baseline LaDiR with a +7.5% gain on average. NF-CoT (Unified) also surpasses the strongest open-source baselines OlympicCoder (+0.6%), Diffu-Coder (+9.4%), and the looped latent reasoning model Ouro (+2.2%) in average pass@1.

Pass@k scaling. We test whether latent reasoning scales with more samples by comparing NF-CoT with the base model and LaDiR on MBPP+ and HumanEval+ for k up to 128.

Table 2: Math reasoning results (pass@1, avg@16) with DeepSeek-R1-Distill-Qwen-7B (Guo et al., 2025) as the base model. NF-CoT uses a boxed-answer prompt and a 4,096-token budget (setup, decoding, and grading in Appendix B.6). Highlighting follows Table 1.

Method	AIME24	AIME25	AMC23	MATH500	Minerva	Olympiad	Avg.
Base Model	15.7	16.0	42.4	71.6	33.3	35.6	35.77
Soft Thinking	20.3+4.6	19.1+3.1	47.9+5.5	76.5+4.9	37.2+3.9	40.6+5.0	40.27+4.50
LaVAE	6.5-9.2	7.8-8.2	21.0-21.4	47.5-24.1	17.6-15.7	18.2-17.4	19.77-16.00
LaDiR	18.8+3.1	19.8+3.8	49.2+6.8	78.4+6.8	**38.9**+5.6	41.0+5.4	41.01+5.24
NF-CoT (Unified)	**24.8**+9.1	**22.3**+6.3	**58.3**+15.9	**81.9**+10.3	36.3+3.0	**44.1**+8.5	**44.62**+8.85

Figure 3 shows that NF-CoT dominates both baselines across the full k range in this separate scaling run. On MBPP+, NF-CoT's pass@1 (72.1) already matches the base model's pass@128 (72.0) and rises to 87.5 at $k = 128$. On HumanEval+, NF-CoT improves from 78.3 to 97.5 (+19.2), comparable to the base model's +16.5 despite starting from a much higher pass@1. LaDiR, in contrast, only improves from 73.2 to 90.2 (+17.0), with the curve flattening at larger budgets. This suggests that sampling the flow noise produces distinct continuous-CoT trajectories rather than collapsing onto a single solution mode; the decoded-latent examples in Appendix A illustrate this.

Execution-guided latent RL. We further apply GRPO to the supervised NF-CoT (Unified) using execution rewards on a randomly selected 20K-problem Python mixture for 150 training steps. As shown in Table 1, this lightweight RL stage improves NF-CoT on all five benchmarks, raising the average from 69.1 to 70.3. These modest gains show that the tractable likelihood of NF-CoT provides a direct interface for policy optimization in latent space, allowing execution rewards to update not only the answer decoder but also the distribution over latent reasoning paths (the likelihood itself is analyzed in Appendix C.3). RL setup details are in Appendix B.5.

4.3 Generalization to math reasoning

To test whether NF-CoT transfers beyond code, we apply the same recipe to math reasoning, with one additional training change in stage 2: the decoder is trained with scheduled sampling over the latent prefix rather than pure teacher forcing, to reduce train–test mismatch (Appendix B.6). We train on R1-distilled math rationales using DeepSeek-R1-Distill-Qwen-7B, the same backbone as LaDi-RL (Kang et al., 2026b), and evaluate pass@1 averaged over 16 samples on AIME24 (Veeraboina, 2023), AIME25 (Zhang & Math-AI, 2025), AMC23, MATH500 (Hendrycks et al., 2021), Minerva (Lewkowycz et al., 2022), and OlympiadBench (He et al., 2024). This setting uses a different teacher and a substantially longer trace distribution than our code corpus (rationales up to 4,096 tokens versus $\approx$385 on average), which we encode into 96 latent tokens (versus 64 for code); the math decoder also generates an explicit rationale before answering. As shown in Table 2, NF-CoT improves over the base model reported by LaDi-RL on all six benchmarks, reaching 44.6 average pass@1 versus 35.8, with gains of 15.9 percentage points on AMC23 and 3.0 on Minerva. NF-CoT also exceeds the reported LaDiR and Soft Thinking results on five of six benchmarks and on average (44.6 vs. 41.0 and 40.3), without LaDiR's iterative denoising. Minerva remains the exception: NF-CoT achieves 36.3, compared with 38.9 for LaDiR and 37.2 for Soft Thinking.

5 Analysis

5.1 Training and inference efficiency

Table 3 compares NF-CoT with LaDiR on HumanEval under matched settings (16 candidates per problem, Qwen3-8B backbone, vLLM decoding). Both methods represent reasoning using only 64 latent tokens, which on average encode a short CoT of 385 text tokens, corresponding to an effective compression rate of approximately $6.0\times$ compared to the explicit-CoT method. The key difference is in latent generation: LaDiR runs 30 denoising steps over the latent trajectory, while NF-CoT samples 64 continuous thoughts autoregressively in one left-to-right pass. This makes NF-CoT (Unified) $2.7\times$ faster at latent generation, $1.9\times$ faster overall, and $2.5\times$ cheaper in per-sample compute. Dual-Path, which keeps separate objective-specific paths, is slower but still $1.6\times$ faster and $2.3\times$ cheaper than

Table 3: Inference efficiency on HumanEval and stage-2 training throughput. LaDiR uses 30 denoising steps; NF-CoT samples its latents in one autoregressive pass. "Latent" and "Decode" report latent-generation and answer-decoding time; FLOPs are coarse estimates.

Method	Inference on HumanEval (16 candidates/problem)					Stage-2 training		
	Latent (s) ↓	Decode (s) ↓	Total (s) ↓	Samples /s ↑	FLOPs /sample ↓	Samples /s ↑	Tokens /s ↑	Total FLOPs ↓
NF-CoT (Unified)	**173.5**	152.1	**325.6**	**8.06**	**19.9T**	**18.4**	**5.88K**	**2.25e19**
NF-CoT (Dual-Path)	232.3	**147.7**	380.0	6.90	21.6T	12.2	3.91K	**2.25e19**
LaDiR	468.2	157.1	625.3	4.20	49.3T	6.45	1.03K	1.50e20

Table 4: Inference-time latent interventions (pass@1, avg@16). Above the rule only the latent prefix fed to the decoder changes; the pause row is a separately trained model. Protocol and pass@10 in Appendix C.4.

Latent prefix	Breaks	HumanEval	HumanEval+	MBPP	MBPP+
Flow sample (default)	–	**84.1**	**78.4**	**83.6**	**71.4**
Shuffled positions	order	79.8	74.5	82.3	70.6
Cross-question sample	relevance	76.9	71.6	74.4	62.2
Random Gaussian	form, relevance	73.7	68.5	68.5	55.6
Zeros	content	34.0	31.6	55.6	47.5
Pause tokens (separate model)	content	70.1	66.9	82.1	70.5

LaDiR. The same trend holds during training: NF-CoT (Unified) achieves $2.9\times$ sample and $5.7\times$ token throughput over LaDiR, with $6.7\times$ fewer total FLOPs (Appendix C.6). These results support our efficiency claim: replacing iterative denoising with an autoregressive flow speeds up latent generation, while exact-likelihood training avoids learning a multi-step denoising chain required by diffusion-based methods such as LaDiR.

5.2 Ablation Study

Do the latents carry task-specific content? We replace the flow-sampled prefix at inference with counterfactual prefixes, holding everything else fixed (Table 4). Zeroing it craters pass@1 (-50 HumanEval, -28 MBPP) but leaves pass@10 within three points: the prefix buys single-shot reliability, not capability the backbone lacks. Well-formed latents sampled for a *different* problem still lower pass@1 by 7.2 points on HumanEval and 9.2 on MBPP, so the content is task-specific. Random Gaussian prefixes, which also break form, lower it by 10.4 and 15.1, and shuffling the 64 positions by only 4.3 and 1.3, so latent order matters. A separately trained pause-token control (Goyal et al., 2024), evaluated under the same protocol, trails NF-CoT by 14.1 and 11.5 pass@1 percentage points on HumanEval and HumanEval+, respectively, with smaller gaps on MBPP and MBPP+.

Effect of the frozen-backbone warm-up stage. We ablate the two-stage training curriculum from Section 3. In the default setting, NF-CoT first trains the flow components with the LLM backbone frozen, then unfreezes all parameters for joint optimization. We compare this with a stage-2-only variant that skips the warm-up. Removing Stage 1 consistently hurts pass@1 (independent evaluation run): HumanEval drops from 84.4 to 81.5, HumanEval+ from 78.7 to 75.5, and LiveCodeBench v6 from 23.1 to 21.4. This is not due to worse final optimization: both runs reach nearly the same total loss and answer cross-entropy. The difference appears in the early training dynamics. With the warm-up, the flow components already provide a meaningful latent interface before the backbone is updated; without it, the backbone receives early gradients from randomly initialized flow components, which can disrupt the coding capability we aim to preserve. The warm-up thus acts as a curriculum that aligns the latent-flow interface with a frozen backbone before allowing the NF likelihood objective to update the full model. Appendix C.2 provides additional backbone-drift diagnostics.

Unified versus dual-path training. The dual-path variant trains the NF and CE objectives through separate latent interfaces (different prefix tokens, projections, and parameterizations), so the NF-

modeled distribution, the representation that conditions answer generation, and the trajectory used at inference do not coincide. The unified variant scores, conditions on, and decodes from the same continuous-CoT prefix in one causal sequence, which removes this mismatch and a second backbone pass; this explains its advantage in Table 1.

6 Related work

CoT and latent reasoning. Chain-of-thought (CoT) prompting elicits reasoning by generating intermediate steps before the answer (Wei et al., 2022; Kojima et al., 2022), with further gains from supervised or RL-based training (Zelikman et al., 2022; Shao et al., 2024) and test-time strategies that sample and score many chains (Wang et al., 2022; Snell et al., 2024; Yao et al., 2023). These approaches all reason in token space; others shorten the textual trace by compressing it (Xia et al., 2025; Ma et al., 2025; Han et al., 2025), internalizing it by distillation (Deng et al., 2023; 2024), or inserting pause tokens (Goyal et al., 2024). Latent reasoning instead feeds continuous thoughts back into the model (Hao et al., 2024; Zhang et al., 2026b; Tang et al., 2026; Su et al., 2025), but probing shows each collapses to a single greedy path, not a superposition (Wu et al., 2026). Recent work injects stochasticity to enable RL or model trajectory superpositions, including in the multimodal setting (Butt et al., 2026; Zhou et al., 2026; Deng et al., 2026; Wang et al., 2026), several via Gumbel-Softmax relaxations that turn token choices into soft embedding mixtures (Jang et al., 2016; Maddison et al., 2016), but none place a tractable density over the trajectories. Others run multiple latent channels or internal recurrences without modeling a trajectory distribution (Geiping et al., 2026; Zhu et al., 2025; Fu et al., 2025). The closest precedent denoises a VAE latent space, leaving the trajectory density implicit and intractable (Kang et al., 2026a;b). NF-CoT instead uses an autoregressive normalizing flow over VAE latents inside the LLM's causal stream, giving an exact log-likelihood over continuous-CoT trajectories and training latent reasoning and answer generation jointly.

Normalizing flows. Normalizing flows map a simple base distribution to data through invertible networks, and provide exact, tractable likelihoods (Rezende & Mohamed, 2015). Early work showed that flows scale to natural image generation (Dinh et al., 2014; 2016; Kingma & Dhariwal, 2018). A parallel line adapted flows to NLP by lifting discrete text into continuous latent spaces, with applications to language modeling and non-autoregressive translation (Ziegler & Rush, 2019; Ma et al., 2019). Autoregressive flows (Kingma et al., 2016; Papamakarios et al., 2017), whose triangular Jacobian aligns naturally with causal Transformers, have more recently been scaled to high-resolution images and video (Zhai et al., 2024; Gu et al., 2026a; 2025; 2026b) and, most recently, to text, either by unifying image and text generation or by modeling language as a continuous latent sequence (Shen et al., 2026; Zhang et al., 2026a). In contrast, NF-CoT uses autoregressive flows inside a pretrained LLM as a tool for *reasoning*, modeling a distribution over continuous CoT trajectories.

7 Discussion

We presented NF-CoT, a latent reasoning framework that gives continuous CoT the same modeling status as language tokens by running an autoregressive normalizing flow inside the LLM's causal stream. The resulting exact likelihood over reasoning trajectories supports both supervised training and policy-gradient refinement in the continuous reasoning space. Across code generation benchmarks, NF-CoT improves accuracy over SFT and prior latent-reasoning baselines and runs faster than LaDiR in training and inference, and a complementary math study suggests the recipe transfers beyond code.

Limitations. Our validation focuses on code generation, with a complementary math study (Section 4.3). NF-CoT currently learns its latent reasoning space from continuous representations derived from explicit-CoT supervision. Exploring more direct ways to learn and optimize latent reasoning representations remains an interesting direction. Our RL refinement relies on execution rewards, so extending it to domains without verifiers is open.

AI USE STATEMENT

In this work, we used generative AI tools (LLM-based coding assistants). The training, evaluation, and analysis pipelines were written by the authors; the assistants were used to debug that code, to draft parts of the evaluation, analysis, and job-orchestration scripts, and to write tests, and every AI-assisted change was reviewed and tested by the authors. We have not used generative AI tools to propose or refine hypotheses, to design or provide feedback on research methodology or experiments, to develop theoretical models or conceptual frameworks, to formulate or prove mathematical claims, or to interpret results; generating synthetic data sets, translation, cleaning or reformatting datasets, and qualitative or thematic data analysis are not applicable to this work. Additionally, we used generative AI tools to edit the paper for readability. We have reviewed all AI-assisted work: AI-assisted code was verified and tested by the authors, all reported numbers were produced by the authors' own evaluation runs, and all AI-edited text was checked by the authors. We take responsibility for the final content of this work, including text, claims, and artifacts produced with the aid of generative AI.

ETHICS STATEMENT

This work uses only public code and math benchmarks and public training corpora; no human-subject data or personally identifiable information is involved. As with any code LLM, generated programs may be incorrect or insecure and should be executed in a sandbox and reviewed before use. Decoded latent CoTs are qualitative probes, not faithful explanations of the model's internal reasoning, and should not be presented as such. We see no ethical concerns beyond those of general-purpose code-generation models.

REPRODUCIBILITY STATEMENT

We describe the experimental setup for the code-generation results in Appendix B: the two-stage training procedure (Appendix B.3, Table 6), the inference and decoding configuration (Appendix B.4), and the RL refinement objective and settings (Appendix B.5); the math setting, its decoding and grading protocol, and the provenance of the quoted math baselines are given in Appendix B.6. Evaluation follows the public EvalPlus and LiveCodeBench protocols, and all baselines are either re-evaluated under the same protocol or marked or noted as reported numbers in Tables 1 and 2. Code, configuration files, and evaluation scripts will be released upon publication.

REFERENCES

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021.

Natasha Butt, Ariel Kwiatkowski, Ismail Labiad, Julia Kempe, and Yann Ollivier. Soft tokens, hard truths. In *International Conference on Learning Representations*, volume 2026, pp. 114650–114675, 2026.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

Jingcheng Deng, Liang Pang, Zihao Wei, Shicheng Xu, Zenghao Duan, Kun Xu, Yang Song, Huawei Shen, and Xueqi Cheng. LLM latent reasoning as chain of superposition. *arXiv preprint arXiv:2510.15522*, 2026.

Yuntian Deng, Kiran Prasad, Roland Fernandez, Paul Smolensky, Vishrav Chaudhary, and Stuart Shieber. Implicit chain of thought reasoning via knowledge distillation. *arXiv preprint arXiv:2311.01460*, 2023.

Yuntian Deng, Yejin Choi, and Stuart Shieber. From explicit cot to implicit cot: Learning to internalize cot step by step. *arXiv preprint arXiv:2405.14838*, 2024.

Laurent Dinh, David Krueger, and Yoshua Bengio. Nice: Non-linear independent components estimation. *arXiv preprint arXiv:1410.8516*, 2014.

Laurent Dinh, Jascha Sohl-Dickstein, and Samy Bengio. Density estimation using real nvp. *arXiv preprint arXiv:1605.08803*, 2016.

Tianyu Fu, Yichen You, Zekai Chen, Guohao Dai, Huazhong Yang, and Yu Wang. Think-at-hard: Selective latent iterations to improve reasoning language models. *arXiv preprint arXiv:2511.08577*, 2025.

Jonas Geiping, Sean McLeish, Neel Jain, John Kirchenbauer, Siddharth Singh, Brian Bartoldson, Bhavya Kailkhura, Abhinav Bhatele, and Tom Goldstein. Scaling up test-time compute with latent reasoning: A recurrent depth approach. *Advances in Neural Information Processing Systems*, 38: 41340–41391, 2026.

Shansan Gong, Ruixiang Zhang, Huangjie Zheng, Jiatao Gu, Navdeep Jaitly, Lingpeng Kong, and Yizhe Zhang. Diffucoder: Understanding and improving masked diffusion models for code generation. In *International Conference on Learning Representations*, volume 2026, pp. 15723–15750, 2026.

Sachin Goyal, Ziwei Ji, Ankit Singh Rawat, Aditya Krishna Menon, Sanjiv Kumar, and Vaishnavh Nagarajan. Think before you speak: Training language models with pause tokens. In *International Conference on Learning Representations*, volume 2024, pp. 27896–27923, 2024.

Jiatao Gu, Ying Shen, Tianrong Chen, Laurent Dinh, Yuyang Wang, Miguel Angel Bautista, David Berthelot, Josh Susskind, and Shuangfei Zhai. Starflow-v: End-to-end video generative modeling with normalizing flows. *arXiv preprint arXiv:2511.20462*, 2025.

Jiatao Gu, Tianrong Chen, David Berthelot, Huangjie Zheng, Yuyang Wang, Ruixiang Zhang, Laurent Dinh, Miguel Angel Bautista, Joshua Susskind, and Shuangfei Zhai. Starflow: Scaling latent normalizing flows for high-resolution image synthesis. *Advances in Neural Information Processing Systems*, 38:120986–121022, 2026a.

Jiatao Gu, Tianrong Chen, Ying Shen, David Berthelot, Shuangfei Zhai, and Josh Susskind. Normalizing trajectory models. *arXiv preprint arXiv:2605.08078*, 2026b.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Peiyi Wang, Qihao Zhu, Runxin Xu, Ruoyu Zhang, Shirong Ma, Xiao Bi, et al. Deepseek-r1 incentivizes reasoning in llms through reinforcement learning. *Nature*, 645(8081):633–638, 2025.

Tingxu Han, Zhenting Wang, Chunrong Fang, Shiyu Zhao, Shiqing Ma, and Zhenyu Chen. Token-budget-aware llm reasoning. In *Findings of the Association for Computational Linguistics: ACL 2025*, pp. 24842–24855, 2025.

Shibo Hao, Sainbayar Sukhbaatar, DiJia Su, Xian Li, Zhiting Hu, Jason Weston, and Yuandong Tian. Training large language models to reason in a continuous latent space. *arXiv preprint arXiv:2412.06769*, 2024.

Chaoqun He, Renjie Luo, Yuzhuo Bai, Shengding Hu, Zhen Thai, Junhao Shen, Jinyi Hu, Xu Han, Yujie Huang, Yuxiang Zhang, et al. Olympiadbench: A challenging benchmark for promoting agi with olympiad-level bilingual multimodal scientific problems. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 3828–3850, 2024.

Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt. Measuring mathematical problem solving with the MATH dataset. *arXiv preprint arXiv:2103.03874*, 2021.

Jonathan Ho, Ajay Jain, and Pieter Abbeel. Denoising diffusion probabilistic models. *Advances in neural information processing systems*, 33:6840–6851, 2020.

Siming Huang, Tianhao Cheng, Jason Klein Liu, Weidi Xu, Jiaran Hao, Liuyihan Song, Yang Xu, Jian Yang, Jiaheng Liu, Chenchen Zhang, et al. Opencoder: The open cookbook for top-tier code large language models. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 33167–33193, 2025.

Hugging Face. Open r1: A fully open reproduction of deepseek-r1, January 2025. URL https://github.com/huggingface/open-r1.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, et al. Qwen2.5-Coder technical report. *arXiv preprint arXiv:2409.12186*, 2024.

Naman Jain, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. Livecodebench: Holistic and contamination free evaluation of large language models for code. In *International Conference on Learning Representations*, volume 2025, pp. 58791–58831, 2025.

Eric Jang, Shixiang Gu, and Ben Poole. Categorical reparameterization with gumbel-softmax. *arXiv preprint arXiv:1611.01144*, 2016.

Haoqiang Kang, Yizhe Zhang, Nikki Kuang, Nicklas Majamaki, Navdeep Jaitly, Yian Ma, and Lianhui Qin. Ladir: Latent diffusion enhances llms for text reasoning. In *International Conference on Learning Representations*, volume 2026, pp. 81705–81733, 2026a.

Haoqiang Kang, Yizhe Zhang, Nikki Lijing Kuang, Yi-An Ma, and Lianhui Qin. LaDi-RL: Latent diffusion reasoning prevents entropy collapse in reinforcement learning. *arXiv preprint arXiv:2602.01705*, 2026b.

Durk P Kingma and Prafulla Dhariwal. Glow: Generative flow with invertible 1x1 convolutions. *Advances in neural information processing systems*, 31, 2018.

Durk P Kingma, Tim Salimans, Rafal Jozefowicz, Xi Chen, Ilya Sutskever, and Max Welling. Improved variational inference with inverse autoregressive flow. *Advances in neural information processing systems*, 29, 2016.

Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. Large language models are zero-shot reasoners. *Advances in neural information processing systems*, 35: 22199–22213, 2022.

Aitor Lewkowycz, Anders Andreassen, David Dohan, Ethan Dyer, Henryk Michalewski, Vinay Ramasesh, Ambrose Slone, Cem Anil, Imanol Schlag, Theo Gutman-Solo, Yuhuai Wu, Behnam Neyshabur, Guy Gur-Ari, and Vedant Misra. Solving quantitative reasoning problems with language models. In *Advances in Neural Information Processing Systems*, volume 35, pp. 3843–3857, 2022.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *Advances in neural information processing systems*, 36:21558–21572, 2023.

Xinyin Ma, Guangnian Wan, Runpeng Yu, Gongfan Fang, and Xinchao Wang. Cot-valve: Length-compressible chain-of-thought tuning. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 6025–6035, 2025.

Xuezhe Ma, Chunting Zhou, Xian Li, Graham Neubig, and Eduard Hovy. Flowseq: Non-autoregressive conditional sequence generation with generative flow. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pp. 4282–4292, 2019.

Chris J Maddison, Andriy Mnih, and Yee Whye Teh. The concrete distribution: A continuous relaxation of discrete random variables. *arXiv preprint arXiv:1611.00712*, 2016.

Sathwik Tejaswi Madhusudhan, Shruthan Radhakrishna, Jash Mehta, and Toby Liang. Millions scale dataset distilled from r1-32b. https://huggingface.co/datasets/ServiceNow-AI/R1-Distill-SFT, 2025.

Shen Nie, Fengqi Zhu, Zebin You, Xiaolu Zhang, Jingyang Ou, Jun Hu, Jun Zhou, Yankai Lin, Ji-Rong Wen, and Chongxuan Li. Large language diffusion models. *Advances in Neural Information Processing Systems*, 38:50608–50646, 2026.

Maxwell Nye, Anders Johan Andreassen, Guy Gur-Ari, Henryk Michalewski, Jacob Austin, David Bieber, David Dohan, Aitor Lewkowycz, Maarten Bosma, David Luan, et al. Show your work: Scratchpads for intermediate computation with language models. *arXiv preprint arXiv:2112.00114*, 2021.

George Papamakarios, Theo Pavlakou, and Iain Murray. Masked autoregressive flow for density estimation. *Advances in neural information processing systems*, 30, 2017.

Danilo Rezende and Shakir Mohamed. Variational inference with normalizing flows. In *International conference on machine learning*, pp. 1530–1538. PMLR, 2015.

ByteDance Seed, Yuyu Zhang, Jing Su, Yifan Sun, Chenguang Xi, Xia Xiao, Shen Zheng, Anxiang Zhang, Kaibo Liu, Daoguang Zan, et al. Seed-coder: Let the code model curate data for itself. *arXiv preprint arXiv:2506.03524*, 2025.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, YK Li, Yang Wu, et al. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*, 2024.

Ying Shen, Tianrong Chen, Yuan Gao, Yizhe Zhang, Yuyang Wang, Miguel Ángel Bautista, Shuangfei Zhai, Joshua M Susskind, and Jiatao Gu. Starflow2: Bridging language models and normalizing flows for unified multimodal generation. *arXiv preprint arXiv:2605.08029*, 2026.

Charlie Snell, Jaehoon Lee, Kelvin Xu, and Aviral Kumar. Scaling llm test-time compute optimally can be more effective than scaling model parameters. *arXiv preprint arXiv:2408.03314*, 2024.

DiJia Su, Hanlin Zhu, Yingchen Xu, Jiantao Jiao, Yuandong Tian, and Qinqing Zheng. Token assorted: Mixing latent and text tokens for improved language model reasoning. *arXiv preprint arXiv:2502.03275*, 2025.

Yao Tang, Li Dong, Yaru Hao, Qingxiu Dong, Furu Wei, and Jiatao Gu. Multiplex thinking: Reasoning via token-wise branch-and-merge. *arXiv preprint arXiv:2601.08808*, 2026.

Ling Team, Wenting Cai, Yuchen Cao, Chaoyu Chen, Chen Chen, Siba Chen, Qing Cui, Peng Di, Junpeng Fang, Zi Gong, et al. Every sample matters: Leveraging mixture-of-experts and high-quality data for efficient and accurate code llm. *arXiv preprint arXiv:2503.17793*, 2025.

Hemish Veeraboina. AIME problem set 1983–2024. Kaggle, 2023. URL https://www.kaggle.com/datasets/hemishveeraboina/aime-problem-set-1983-2024.

Qixun Wang, Yang Shi, Yifei Wang, Yuanxing Zhang, Pengfei Wan, Kun Gai, Xianghua Ying, and Yisen Wang. Monet: Reasoning in latent visual space beyond image and language. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 12030–12040, 2026.

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. Self-consistency improves chain of thought reasoning in language models. *arXiv preprint arXiv:2203.11171*, 2022.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837, 2022.

Junhong Wu, Jinliang Lu, Zixuan Ren, Gangqiang Hu, Zhi Wu, Dai Dai, et al. Llms are single-threaded reasoners: Demystifying the working mechanism of soft thinking. In *International Conference on Learning Representations*, volume 2026, pp. 4533–4548, 2026.

Heming Xia, Chak Tou Leong, Wenjie Wang, Yongqi Li, and Wenjie Li. Tokenskip: Controllable chain-of-thought compression in llms. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pp. 3351–3363, 2025.

Zhihui Xie, Jiacheng Ye, Lin Zheng, Jiahui Gao, Jingwei Dong, Zirui Wu, Xueliang Zhao, Shansan Gong, Xin Jiang, Zhenguo Li, et al. Dream-coder 7b: An open diffusion language model for code. *arXiv preprint arXiv:2509.01142*, 2025.

Zhangchen Xu, Yang Liu, Yueqin Yin, Mingyuan Zhou, and Radha Poovendran. Kodcode: A diverse, challenging, and verifiable synthetic dataset for coding. In *Findings of the Association for Computational Linguistics: ACL 2025*, pp. 6980–7008, 2025.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025.

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Tom Griffiths, Yuan Cao, and Karthik Narasimhan. Tree of thoughts: Deliberate problem solving with large language models. *Advances in neural information processing systems*, 36:11809–11822, 2023.

Jiacheng Ye, Zhihui Xie, Lin Zheng, Jiahui Gao, Zirui Wu, Xin Jiang, Zhenguo Li, and Lingpeng Kong. Dream 7b: Diffusion large language models, 2025. URL https://arxiv.org/abs/2508.15487.

Eric Zelikman, Yuhuai Wu, Jesse Mu, and Noah Goodman. Star: Bootstrapping reasoning with reasoning. *Advances in Neural Information Processing Systems*, 35:15476–15488, 2022.

Huaye Zeng, Dongfu Jiang, Haozhe Wang, Ping Nie, Xiaotong Chen, and Wenhu Chen. Acecoder: Acing coder rl via automated test-case synthesis. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 12023–12040, 2025.

Shuangfei Zhai, Ruixiang Zhang, Preetum Nakkiran, David Berthelot, Jiatao Gu, Huangjie Zheng, Tianrong Chen, Miguel Angel Bautista, Navdeep Jaitly, and Josh Susskind. Normalizing flows are capable generative models. *arXiv preprint arXiv:2412.06329*, 2024.

Ruixiang Zhang, Shuangfei Zhai, Jiatao Gu, Yizhe Zhang, Huangjie Zheng, Tianrong Chen, Miguel Angel Bautista, Joshua Susskind, and Navdeep Jaitly. Flexible language modeling in continuous space with transformer-based autoregressive flows. *Advances in Neural Information Processing Systems*, 38:173273–173323, 2026a.

Yifan Zhang and Team Math-AI. American invitational mathematics examination (aime) 2025, 2025.

Zhen Zhang, Xuehai He, Weixiang Yan, Ao Shen, Chenyang Zhao, and Xin Wang. Soft thinking: Unlocking the reasoning potential of llms in continuous concept space. *Advances in Neural Information Processing Systems*, 38:168990–169012, 2026b.

Siyan Zhao, Devaansh Gupta, Qinqing Zheng, and Aditya Grover. d1: Scaling reasoning in diffusion large language models via reinforcement learning. *Advances in Neural Information Processing Systems*, 38:56729–56762, 2026.

Yuyan Zhou, Jiarui Yu, Hande Dong, Zhezheng Hao, Hong Wang, Jianqing Zhang, and Qiang Lin. Lepo: Latent reasoning policy optimization for large language models. In *Findings of the Association for Computational Linguistics: ACL 2026*, pp. 14416–14427, 2026.

Rui-Jie Zhu, Zixuan Wang, Kai Hua, Tianyu Zhang, Ziniu Li, Haoran Que, Boyi Wei, Zixin Wen, Fan Yin, He Xing, et al. Scaling latent reasoning via looped language models. *arXiv preprint arXiv:2510.25741*, 2025.

Zachary Ziegler and Alexander Rush. Latent normalizing flows for discrete sequences. In *International Conference on Machine Learning*, pp. 7673–7682. PMLR, 2019.

A QUALITATIVE ANALYSIS OF DECODED LATENT COTS

In this appendix, we provide qualitative examples of decoded latent CoTs on HumanEval. Each example contains the raw text decoded from a sampled latent CoT vector and the program generated from the same latent prefix. The decoded latent text is not edited for correctness or fluency. Because this text is reconstructed through the frozen VAE decoder, we do not interpret it as a faithful natural-language explanation of the model's internal computation. Instead, we use it as a qualitative probe of whether sampled latent trajectories align with coarse algorithmic choices in the final program. These examples show that NF-CoT's latent variables often correspond to high-level algorithmic choices: different latent samples for the same problem can lead to different implementation strategies while still producing functionally correct programs.

Experimental setup. We use the NF-CoT unified checkpoint with 64 latent tokens. For each task, we sample latent trajectories with guidance scale 1.0, NF temperature 0.9, and answer temperature 0.6. The decoded latent CoT is obtained by converting the unified latent back to the VAE-latent space and greedily decoding it with the VAE decoder. The benchmark subset is from HumanEval and is not part of the training data.

Table 5: Pass counts for the HumanEval tasks considered in the latent-CoT showcase. We provide detailed decoded examples for a representative subset below.

Task	Entry point	Passed / 64
HumanEval/75	`is_multiply_prime`	55 / 64
HumanEval/124	`valid_date`	64 / 64
HumanEval/154	`cycpattern_check`	61 / 64
HumanEval/63	`fibfib`	64 / 64
HumanEval/44	`change_base`	64 / 64
HumanEval/38	`decode_cyclic`	64 / 64
HumanEval/123	`get_odd_collatz`	58 / 64
HumanEval/89	`encrypt`	64 / 64

A.1 HUMANEVAL/63: FIBFIB

Problem

```
def fibfib(n: int):
    """The FibFib number sequence is a sequence similar to the Fibbonacci
    sequnece that's defined as follows:
    fibfib(0) == 0
    fibfib(1) == 0
    fibfib(2) == 1
    fibfib(n) == fibfib(n-1) + fibfib(n-2) + fibfib(n-3).
    Please write a function to efficiently compute the n-th element of the
    fibfib number sequence.
    >>> fibfib(1)
    0
    >>> fibfib(5)
    4
    >>> fibfib(8)
    24
    """
```

This task is a strong showcase because the same problem produces three passing answers with genuinely different implementation strategies: rolling-state dynamic programming, explicit tabulation, and recursive memoization.

Decoded Latent CoT: sample_idx=49, rolling three-state DP

Why useful: The decoded latent CoT discusses compressing the recurrence into fixed state, and the generated answer uses only `a`, `b`, and `c`.

```
To calculate the Fibonacci number sequence up to the \( n \)th number, we need
    to follow these steps:

### Plan

1. **Understanding the Fibonacci Sequence**: The Fibonacci sequence is defined
    as:
   - \( F(1) = 0 \)
   - \( F(2) = 1 \)
   - \( F(n) = F(n-1) + F(n-2) \)
2. **Compressing the Formula for Efficiency**: We use a fixed amount of space
    to store the last three Fibonacci numbers and calculate the next. This
    optimizes, since we don't need to use \( n \) variables.
3. **Recursive Fibonacci Term**: Calculate the \( n \)th Fibonacci number:
   - \( F(n) = F(n-1) + F(n-2) \)
4. **Flow**: This calculation is done in constant \( O(1) \) extra space by
    using three variables.
```

Generated Answer: sample_idx=49, passed

```
def fibfib(n: int) -> int:
    """Compute the n-th element of the fibfib sequence."""
    # Base cases
    if n == 0:
        return 0
    elif n == 1:
        return 0
    elif n == 2:
        return 1

    # Initialize the first three elements of the sequence
    a, b, c = 0, 0, 1

    # Compute the sequence iteratively up to the n-th element
    for _ in range(3, n + 1):
        # Calculate the next element in the sequence
        next_value = a + b + c
        # Update the previous three elements for the next iteration
        a, b, c = b, c, next_value

    return c
```

Decoded Latent CoT: sample_idx=3, explicit tabulation list

Why useful: The decoded latent CoT discusses tabulation/list storage, and the generated answer appends every FibFib value to `fibfib_values`.

```
To solve the Fibonacci number sequence using a tabular approach, we can
    efficiently compute the nth Fibonacci number and compute the sequence for
    the first n Fibonacci numbers. The Fibonacci number is defined by the
    formula:

- `fib(n) = fib(n-1) + fib(n-2)`

Here's a step-by-step breakdown of how we can implement this:

### Step-by-Step Explanation
```

```
1. **Initialization:**
   - We use a stack (list) to store Fibonacci numbers, starting with the base
     cases `n = 0` and `n = 1`.
   - This prevents the exponential number of computations and ensures the
     sequence is computed efficiently.
2. **Base Case:**
   - The first two Fibonacci numbers are known: `fib(0) = 0` and `fib(1) = 1`.
3. **Compute the Sequence:**
   - For a given `n`, compute the value of `fib(n)` by summing the previous two
     Fibonacci numbers (last and nth).
   - Compute the rest of the sequence numbers: `fib(n-1)`, `fib(n-2)`, etc.,
     from 3 to `n`.
4. **Return Computed Values:**
   - Return the array of computed Fibonacci numbers.
5. **Implementation**: The function computes the necessary Fibonacci number for
     `n`, and indeed utilizes efficient tabulation to avoid redundant
     computations.
```

Generated Answer: sample_idx=3, passed

```python
def fibfib(n: int) -> int:
    """Compute the n-th element of the FibFib sequence."""
    # Base cases
    if n == 0:
        return 0
    if n == 1:
        return 0
    if n == 2:
        return 1

    # Initialize the first three values of the sequence
    fibfib_values = [0, 0, 1]

    # Compute the sequence iteratively up to n
    for i in range(3, n + 1):
        # Calculate the next value as the sum of the last three values
        next_value = fibfib_values[-1] + fibfib_values[-2] + fibfib_values[-3]
        # Append the new value to the list
        fibfib_values.append(next_value)

    # Return the n-th value in the sequence
    return fibfib_values[n]
```

Decoded Latent CoT: sample_idx=35, recursive memoization

Why useful: The decoded latent CoT discusses cache and recursion, and the generated answer uses a nested memoized helper.

```
To solve this problem, we can use a cache to efficiently compute the nth
    Fibonacci number. We'll use a recursive function to compute the Fibonacci
    numbers, but we'll optimize the recursion by storing the results of
    previous Fibonacci numbers. This method avoids the exponential number of
    calculations and efficiently fills the cache, leading to a significant
    efficiency improvement. Here's the detailed version:

## Efficient Fibonacci Sequence

### Step-by-Step Explanation

1. **Define the Function:**
```

```
   - Use a cache (dictionary) to store the value of Fibonacci numbers we have
     computed.
2. **Recursive Computation:**
   - We compute the Fibonacci number by first calculating the function
     recursively and store it in the cache.
   - Use base cases to reduce complexity.
```

Generated Answer: sample_idx=35, passed

```
def fibfib(n: int) -> int:
    """Compute the n-th element of the FibFib sequence."""

    # Initialize a dictionary to store the computed values
    memo = {0: 0, 1: 0, 2: 1}

    def helper(x):
        """Recursive helper function with memoization."""
        # Check if the value is already computed
        if x in memo:
            return memo[x]

        # Compute the value recursively and store it in the memo dictionary
        memo[x] = helper(x - 1) + helper(x - 2) + helper(x - 3)
        return memo[x]

    # Call the helper function with the input n
    return helper(n)
```

A.2 HUMANEVAL/154: CYCPATTERN_CHECK

Problem

```
def cycpattern_check(a , b):
    """You are given 2 words. You need to return True if the second word or any
     of its rotations is a substring in the first word
    cycpattern_check("abcd","abd") => False
    cycpattern_check("hello","ell") => True
    cycpattern_check("whassup","psus") => False
    cycpattern_check("abab","baa") => True
    cycpattern_check("efef","eeff") => False
    cycpattern_check("himenss","simen") => True

    """
```

This example shows that different latent samples can instantiate the same rotation-substring idea with different surface forms.

Decoded Latent CoT: sample_idx=0, direct rotation loop

Why useful: The decoded latent CoT says to inspect all rotations, and the generated answer loops over rotations with early return.

```
## solve the problem, we need to check for a pattern in a string rotated a
    certain number of times. The first step is to rotate the string and the
    second step is to inspect all rotations for the given pattern.

## Step-by-Step Explanation

1. **Understanding Rotations**:
```

```
   - A rotation of a string is a substring of the original string.
   - For example, if `s = "abcdef"`, then the rotations are: `a., b., c., d., e
     ., f.` and `b., c., d., e., f., a.`.
2. **Checking Rotations**:
   - We can generate all possible rotations of a string.
   - For each rotation, we check if it matches the pattern.
3. **Rotating and Sorting**:
   - This approach can be somewhat complex for all rotations.
```

Generated Answer: sample_idx=0, passed

```python
def cycpattern_check(a, b):
    """Check if any rotation of the second word is a substring in the first
    word."""
    # Get the length of the second word.
    len_b = len(b)

    # Check each rotation of the second word.
    for i in range(len_b):
        # Create a rotation by slicing the string.
        rotation = b[i:] + b[:i]

        # Check if this rotation is a substring in the first word.
        if rotation in a:
            return True

    # If no rotation is a substring, return False.
    return False
```

Decoded Latent CoT: sample_idx=16, precompute rotations plus `any`

Why useful: The decoded latent CoT says to generate cyclic permutations, and the generated answer materializes rotations and calls `any`.

```
To solve the problem of checking if a word is a substring of any rotation, we
    need to consider the given function and introduce some checks to check the
    rotations. Here's how we can implement the function:

### Step-by-Step Explanation

1. **Check the String Rotation:**
   - For each rotation, check if the `check_string` is a substring. Rotations
    can be generated by checking if the string is a substring of any of its
    cyclic permutations.
2. **Understanding Rotations:**
   - **Rotations** are permutations of the string. For any given rotation, if
    you concatenate the string with itself, the original string becomes a
    substring.
   - Example: If `s = "abcd"`, the rotations are `["abcd", "bcda", "cdab", "
    dabc"]`.
3. **Check Substring Check:**
   - If `check_string` is a substring of any rotation, it can be checked if it
    is a substring of `s + s` (the concatenated version of `s`).
4. **Check for No Rotation:**
   - We need to check if the `check_string` is also a substring of the first
    rotation. A simple check would be:
     - Check if `check_string` is a substring of `a`, `b`, `c`, or `d` for each
     rotation.
5. **Implement a Check for All Rotations:**
   - Check all rotations by rotating the string and checking the substring
    condition.
```

```
   - If any rotation contains ‘check_string‘, return ‘True‘.
```

Generated Answer: sample_idx=16, passed

```
def cycpattern_check(a, b):
    if not b:
        return True
    if len(b) > len(a):
        return False

    # Generate all rotations of b
    rotations = [b[i:] + b[:i] for i in range(len(b))]

    # Check if any rotation of b is a substring of a
    return any(rot in a for rot in rotations)
```

A.3 HUMANEVAL/89: ENCRYPT

Problem

```
def encrypt(s):
    """Create a function encrypt that takes a string as an argument and
    returns a string encrypted with the alphabet being rotated.
    The alphabet should be rotated in a manner such that the letters
    shift down by two multiplied to two places.
    For example:
    encrypt('hi') returns 'lm'
    encrypt('asdfghjkl') returns 'ewhjklnop'
    encrypt('gf') returns 'kj'
    encrypt('et') returns 'ix'
    """
```

This task illustrates another form of latent-controlled implementation diversity: the same Caesar-style cipher is solved either through alphabet-index lookup or ASCII/modulo arithmetic.

Decoded Latent CoT: sample_idx=1, alphabet-index lookup

Why useful: The decoded latent CoT describes alphabet rotation, and the generated answer uses an explicit alphabet string and `.index()`.

```
To solve the problem of encrypting a string using a Caesar cipher, we'll
    implement a function that handles the rotation of characters. Here's a step
    -by-step explanation of how we can achieve this:

### Step-by-Step Explanation

1. **Handle Case**:
   - Convert the string to handle cases (uppercase and lowercase). This will
    handle the case where the user strings are partially lowercased.
2. **Handle Edge Cases**:
   - Handle edge cases such as empty strings or the handle of digits properly.
3. **Encoding and Rotation**:
   - Each character in the string should be shifted by the two-digit number
    provided.
   - If the shift handles the letters beyond 'z' and 'Z', wrap around to the
    beginning of the alphabet.
4. **Handle Edge Handling**:
   - Repeat the shifted character to handle each character individually.
```

Generated Answer: sample_idx=1, passed

```
def encrypt(s):
    """Encrypt the string by shifting the alphabet by 4 places."""
    # Define the alphabet
    alphabet = 'abcdefghijklmnopqrstuvwxyz'
    # Calculate the shift amount
    shift = 4
    # Initialize an empty result string
    encrypted = ''

    # Iterate over each character in the input string
    for char in s:
        # Check if the character is a lowercase letter
        if char in alphabet:
            # Find the original position of the character in the alphabet
            original_position = alphabet.index(char)
            # Calculate the new position after shifting
            new_position = (original_position + shift) % 26
            # Append the encrypted character to the result
            encrypted += alphabet[new_position]
        else:
            # If it's not a letter, keep the character as is
            encrypted += char

    return encrypted
```

Decoded Latent CoT: sample_idx=23, ASCII modulo arithmetic

Why useful: The decoded latent CoT describes Caesar rotation and wraparound, and the generated answer uses ord/chr modulo arithmetic.

```
To solve the problem of encrypting a string using a Caesar cipher, we need to
    follow these steps:

1. **Function Definition**: Create the function `rotate_encrypt` that takes two
     parameters: the `string` and the `shift`.
2. **Initialize the Result**: Use a variable to shift each character by the
    specified number. If the character is shifted by 26, it should wrap around
    the alphabet.
3. **Iterate Through the String**: Iterate through each character in the string
     and apply the rotation:
   - Shift each letter by the ASCII value of the letter in the opposite
    alphabet.
   - Use modulo 26 to ensure the shifts are shifted.
4. **Encrypt the Character**:
   - Initialize the function to add a shifted character to the result.
5. **Return the Encrypted String**: Return the encrypted string.
```

Generated Answer: sample_idx=23, passed

```
def encrypt(s):
    """Encrypts the input string by rotating the alphabet by 4 positions."""
    encrypted_string = []

    for char in s:
        if 'a' <= char <= 'z':  # Check if the character is a lowercase letter
            # Rotate the character by 4 positions
            new_char = chr((ord(char) - ord('a') + 4) % 26 + ord('a'))
            encrypted_string.append(new_char)
```

```
        elif 'A' <= char <= 'Z':  # Check if the character is an uppercase
    letter
            # Rotate the character by 4 positions
            new_char = chr((ord(char) - ord('A') + 4) % 26 + ord('A'))
            encrypted_string.append(new_char)
        else:
            # Non-alphabet characters are not changed
            encrypted_string.append(char)

    return ''.join(encrypted_string)
```

Takeaway. Across these examples, decoded latent CoTs are imperfect at the token level but consistently reflect the coarse algorithmic route followed by the final program. The `fibfib` examples are especially illustrative: the same problem yields rolling-state dynamic programming, explicit table-based dynamic programming, and recursive memoization. This supports the view that NF-CoT does not merely store a single hidden solution trace. Instead, its latent space captures a distribution over plausible reasoning trajectories, allowing different samples to steer generation toward distinct but correct implementations.

B Experimental setup

We evaluate two variants of our latent-reasoning normalizing flow (NF): the *dual-path* model, in which the backbone LLM is invoked twice per training step (once for flow-density estimation and once for cross-entropy answer prediction), and the *unified-path* model, in which a single causal forward computes both losses. Both variants share the same backbone, latent geometry, dataset, two-stage training schedule, and decoding pipeline, and differ only in the conditioning sequence and which components are trainable in Stage 1.

B.1 Dual-path model

The dual-path model forwards the backbone twice per gradient step. The two paths use *distinct* latent–to–hidden projectors (`flow_projector` for the flow path, `vae.latent_to_decoder` for the CE path) and *distinct* boundary tokens (`BOT` and `AE`). Optional diffusion-forcing on the CE path is supported but disabled in our reported runs.

B.2 Unified-path model

The unified model fuses the two paths into a single causal sequence

$$[\,\text{prompt};\ \texttt{BOT};\ \text{flow_proj}(\tilde{x});\ \text{answer}\,],$$

where $\tilde{x} = \pi_{\text{deep}}\big(\text{shallow}(x+\epsilon)\big)$ is the latent in post-shallow, post-permutation space and $\epsilon \sim \mathcal{N}(0, \sigma_{\text{dq}}^2 I)$ is dequantization noise (the *same* noise drives NF and CE). A single backbone forward yields the hidden states used to compute the NF NLL on positions $[T_p, T_p+N)$ via `output_head` and the CE loss on positions $[T_p+N, T_p+N+T_a)$ via the LM head. Because the shallow MetaBlocks are invertible, conditioning the CE path on $\tilde{x}$ is representationally equivalent to conditioning on raw latents while eliminating the second backbone forward; the unified model therefore needs *only one* latent–to–hidden projector and *one* prefix marker.

B.3 Training hyperparameters

Backbone and latent geometry. Both variants use `Qwen3-8B-Base` as the backbone. The latent sequence has $N = 64$ slots of dimension $D = 2560$, the hidden size of the Qwen3-4B-Base VAE encoder; an MLP projector maps each slot to the backbone's 4,096-dimensional hidden size. Latents are produced by a frozen transformer-encoder VAE with encoder `Qwen3-4B-Base`; the VAE encoder is never updated. Dequantization noise has $\sigma_{\text{dq}} = 0.3$. The flow-density head consists of an MLP projector ($\text{LayerNorm} \to \text{Linear} \to \text{GELU} \to \text{Linear}$), five MetaBlocks (Gu et al., 2026a) with

Table 6: Hyperparameters for the dual-path and unified-path NF models. Shared settings are shown once across both model columns.

Setting	Dual-path	Unified-path
Backbone LLM	Qwen3-8B-Base	
Frozen latent encoder	TransEncoderVAE with Qwen3-4B-Base encoder	
Latent geometry	$N = 64$ slots, latent dimension $D = 2560$	
Shallow MetaBlocks	5 blocks, frozen in Stage 1, trained in Stage 2	5 blocks, trained in Stages 1 and 2
Shallow block width	2048 channels, 2 layers per block, head dimension 64	
Deep block	Causal left-to-right LLM block	
Soft-clip / dequantization	Soft-clip 4.0; dequantization noise $\sigma_{\text{dq}} = 0.3$	
Loss weights	$\lambda_{\text{NF}} = 1.0$, $\lambda_{\text{CE}} = 1.0$	
Prompt dropout	$p_{\text{drop}} = 0.1$ at batch level	$p_{\text{drop}} = 0.1$ per sample
Latent-to-hidden projector	`flow_projector` plus W_{dec}	Shared `flow_projector`
Prefix marker(s)	`BOT`, `AE`	`BOT`
Stage-1 frozen modules	Backbone and shallow blocks	Backbone only
Stage-2 trainable modules	All model parameters except the VAE encoder	
Optimizer	AdamW, $(\beta_1, \beta_2) = (0.9, 0.95)$, weight decay 0.01, gradient clip 1.0	
Learning rate / warmup	Stage 1: 1×10^{-4} with 100 warmup steps; Stage 2: 5×10^{-5} with 200 warmup steps	
Schedule	Cosine decay	
Batch configuration	Per-device batch 2 / 1 and gradient accumulation 4 / 8 for Stage 1 / Stage 2	
Hardware	4 / 8 NVIDIA B200 GPUs for Stage 1 / Stage 2	
Epochs / Stage-1 budget	Stage 1: 1 epoch over 100,000 samples; Stage 2: 2 epochs	
Precision / sharding	bfloat16; FSDP `hybrid_shard` with activation checkpointing	
Random seed	HuggingFace Trainer default when unset in the configuration	

channel width 2048, two layers per block, head dim 64, alternating FLIP/IDENTITY permutations (so the top, deep, block is IDENTITY and matches the LLM's L2R causal mask), and a zero-init `NFOutputHead` that emits affine parameters (σ, μ) with $\mathrm{softplus}$-positivized scale and $\tanh$ soft-clip at ± 4.0.

Loss. The training objective is $\mathcal{L} = \lambda_{\text{NF}}\,\mathcal{L}_{\text{NF}} + \lambda_{\text{CE}}\,\mathcal{L}_{\text{CE}}$ with $\lambda_{\text{NF}} = \lambda_{\text{CE}} = 1$. The flow-NLL has the standard form $\mathcal{L}_{\text{NF}} = \frac{1}{2}\mathbb{E}\|z\|_2^2 - \mathbb{E}\sum_k \log\det J_k$, and the CE term is teacher-forcing cross-entropy on answer tokens with PAD positions masked out. Classifier-free guidance is supported via a learned `null_embed`; the prompt is dropped to `null_embed` with probability $p_{\text{drop}} = 0.1$ (batch-level for the dual-path model, per-sample for the unified-path model).

Two-stage curriculum. We follow a frozen-backbone → joint-finetune curriculum:

- **Stage 1 (latent-projector warm-up).** The backbone is frozen. In the unified model the shallow blocks are trained in Stage 1 together with the projector, output head, and prefix embeddings; in the dual-path model the shallow blocks stay frozen in Stage 1 and only the projectors, output head, and prefix embeddings are updated (Table 6). Stage 1 uses 100 000 samples, 1 epoch, per-device batch size 2, gradient accumulation 4 (global batch 32 on 4 GPUs), peak learning rate 1×10^{-4} with 100 linear-warmup steps.
- **Stage 2 (joint finetuning).** The backbone and shallow blocks are unfrozen and the full Python subset of the dataset is used for 2 epochs. We use per-device batch size 1, gradient accumulation 8 (global batch 64 on 8 GPUs), peak learning rate 5×10^{-5}, with 200 linear-warmup steps.

For both stages we use AdamW with $(\beta_1, \beta_2) = (0.9, 0.95)$, weight decay 0.01, gradient clip 1.0, cosine learning-rate decay, `bfloat16` mixed precision, and FSDP `hybrid_shard` with auto-wrap on `Qwen3DecoderLayer`, backward/forward prefetching, CPU-RAM-efficient loading, and activation checkpointing. Training is run on NVIDIA DGX B200 nodes (4 GPUs in Stage 1, 8 GPUs in Stage 2). The backbone weights for Stage 2 come directly from the pre-trained `Qwen3-8B-Base` checkpoint. Training uses the HuggingFace Trainer default seed when no explicit seed is specified in the configuration. The total number of optimizer steps is determined by the epoch budgets, global batch sizes, and filtered training split rather than by a fixed `max_steps` override.

Hyperparameter table. Table 6 summarizes the hyperparameters of both variants.

B.4 INFERENCE CONFIGURATION

For the main benchmark evaluations, inference is split into two phases per evaluation example. Phase 1 generates latent thoughts via the NF reverse pass; Phase 2 produces an answer by feeding the latent prefix to the backbone as soft-token `input_embeds`. The KV cache built in Phase 1 is *not* carried into Phase 2: the vLLM engine re-encodes the full prefix (prompt, marker, and projected latents) before sampling answer tokens, so the KV-cache compatibility discussed in Section 3.4 is an architectural property of the shared causal backbone rather than a measured speed-up, and the "Decode" column of Table 3 includes this prefix re-encoding. The speed ablation in Table 3 uses vLLM answer decoding as stated there; this subsection describes the main benchmark decoding pipeline.

Phase 1: latent generation (PyTorch). We sample $z \sim \mathcal{N}(0, T_z^2 I) \in \mathbb{R}^{N \times D}$, prefix the prompt embeddings with the model's boundary token (`BOT` for both variants; the dual-path model adds `AE` only on the CE path), and run an autoregressive NF reverse pass through the LLM with KV-cache reuse: at each step t, the LLM hidden at the latest position is mapped through `output_head` to obtain (σ_t, μ_t), the next latent is computed as $x_t = \sigma_t z_t + \mu_t$, the projected latent $\mathrm{flow_proj}(x_t)$ is appended, and the LLM is advanced by one position over the cached keys/values. After $N = 64$ steps, the deep permutation is inverted; for the dual-path model we additionally invert the five shallow MetaBlocks to recover raw latents in encoder space, whereas the unified model keeps the latent in $\tilde{x}$ space (post-shallow, post-permute) since the LLM was trained to read it there. Classifier-free guidance is applied in (σ, μ) space following STARFlow (Gu et al., 2026a).

Phase 2: answer decoding (vLLM). The full backbone weights are extracted from the trained checkpoint and loaded into an offline `vllm.LLM` engine with `enable_prompt_embeds=True` (required for embedding inputs). For each problem we feed the soft prefix $[\,\mathrm{prompt_embeds};\ \texttt{marker};\ \mathrm{proj}(\mathrm{latents})\,]$ through the engine's `prompt_embeds` interface and sample answer tokens. The marker and projector are `AE`/W_{dec} for the dual-path model and `BOT`/$\mathrm{flow_projector}$ for the unified model.

Decoding hyperparameters. For each benchmark problem, we draw 16 independent samples and report pass@1 as the average single-sample correctness over the 16 candidates. Each sample uses an independent NF noise draw (no fixed seed at inference). For Table 1, we use this standard $n = 16$ sampling configuration unless otherwise specified. For Figure 3, we run a separate pass@k scaling evaluation with up to 128 sampled candidates per problem. The pass@1 endpoints in that scaling run are reported from the same candidate pool used for the pass@k curves and are not used to compute Table 1. Decoding uses temperature $T_a = 0.6$, top-$p = 1.0$, and a maximum of 1024 new tokens; NF sampling temperature and CFG strength are *variant-specific* and were tuned on a held-out validation slice:

- **Dual-path:** $T_z = 1.0$, $w_{\text{CFG}} = 3.5$.
- **Unified-path:** $T_z = 0.9$, $w_{\text{CFG}} = 1.0$; a moderately under-temperatured NF sampler is preferred for this model.

B.5 EXECUTION-GUIDED RL DETAILS

Overview. We further apply execution-guided RL to the supervised NF-CoT (Unified) checkpoint. During RL, the VAE encoder/decoder shell, shallow flow blocks, flow projector, affine output head, and auxiliary latent-control embeddings are kept frozen. We update only the shared backbone LLM, which is used both as the autoregressive block of the normalizing flow and as the answer decoder. Thus, the same trainable parameters receive reward-driven updates from both the latent-trajectory policy and the answer-token policy.

Policy factorization. Each rollout consists of a latent trajectory $\tilde{x} = (\tilde{x}_1, \ldots, \tilde{x}_N)$ with $N = 64$, followed by an answer token sequence $y = (y_1, \ldots, y_L)$. We view both the continuous latent trajectory and the answer tokens as actions. The latent density is computed by inverting the affine transformation predicted by the flow block at each latent position. For the realized latent $\tilde{x}_t$, the

model predicts affine parameters (x_a^t, x_b^t) and recovers the corresponding Gaussian noise as

$$z_t = \frac{\tilde{x}_t - x_b^t}{x_a^t}. \tag{B.1}$$

The latent log-probability is then

$$\log p_\theta(\tilde{x}_t \mid p, \tilde{x}_{<t}) = -\frac{1}{2}\|z_t\|_2^2 - \sum_{d=1}^{D} \log |x_{a,d}^t|, \tag{B.2}$$

up to the constant Gaussian normalization term, which cancels in the PPO ratio. Since the shallow flow blocks are frozen and the action is defined in post-shallow latent space, their log-determinants are constant with respect to the trainable parameters and are omitted from the RL objective.

RL objective. For each prompt, we sample a group of $n = 8$ rollouts and compute GRPO advantages from their execution rewards:

$$\hat{A}_i = \frac{R_i - \mu}{\sigma + \varepsilon_\sigma}, \tag{B.3}$$

where μ and σ are the group mean and standard deviation. The final RL loss combines a standard token-level PPO objective with a sequence-level latent PPO objective:

$$\mathcal{L}_\theta = \mathcal{L}_\theta^{\text{tok}} + \mathcal{L}_\theta^{\text{lat}}. \tag{B.4}$$

The token term is computed on answer tokens only. The latent term applies the same clipped PPO surrogate to the likelihood ratio of the complete latent trajectory. Specifically, we define the per-step latent log-ratio as

$$\begin{aligned} \delta_{i,t}^{\text{lat}} &= \log p_\theta(\tilde{x}_{i,t} \mid p_i, \tilde{x}_{i,<t}) \\ &\quad - \log p_{\theta_{\text{old}}}(\tilde{x}_{i,t} \mid p_i, \tilde{x}_{i,<t}), \end{aligned} \tag{B.5}$$

$$\Delta_i^{\text{lat}} = \sum_{t=1}^{N} \delta_{i,t}^{\text{lat}}, \tag{B.6}$$

$$r_i^{\text{lat}} = \exp\big(\text{clip}(\Delta_i^{\text{lat}}, -20, 20)\big). \tag{B.7}$$

Then the latent PPO term is

$$\begin{aligned} \mathcal{L}_\theta^{\text{lat}} &= -\mathbb{E}_i\Big[\min\Big(r_i^{\text{lat}}\hat{A}_i,\, \bar{r}_i^{\text{lat}}\hat{A}_i\Big)\Big] \\ &\quad + \beta_{\text{KL}}\mathcal{K}_i^{\text{lat}}, \end{aligned} \tag{B.8}$$

where

$$\bar{r}_i^{\text{lat}} = \text{clip}(r_i^{\text{lat}}, 1-\varepsilon, 1+\varepsilon). \tag{B.9}$$

Here $\varepsilon = 0.2$, and $\mathcal{K}_i^{\text{lat}}$ is a sampled log-ratio penalty between the actor and the frozen reference policy on the same realized latent trajectory. We use the same KL coefficient for the token and latent terms.

Optimization and evaluation. We train for 150 RL steps with AdamW using learning rate 3×10^{-6}, batch size 64 prompts, and 8 rollouts per prompt. The KL coefficient is $\beta_{\text{KL}} = 10^{-3}$, and the PPO clip range is 0.2. During RL rollout, we use answer temperature 0.9 and latent temperature 1.0.

After RL, we evaluate using the same decoding protocol as the supervised NF-CoT models: answer temperature 0.6, NF temperature 0.9, classifier-free guidance scale 1.0, and 16 samples per problem for HumanEval, HumanEval+, MBPP, MBPP+, and LiveCodeBench v6.

B.6 Math setup and disclosures

Model and data. The math model uses DeepSeek-R1-Distill-Qwen-7B as the backbone with 96 latent tokens of dimension 2,560. The VAE encoder is a fine-tuned Qwen3-4B-Base and the (frozen) decoder is Qwen3-8B-Base. Training data are the 698,922 NuminaMath-CoT examples

from R1-Distill-SFT (Madhusudhan et al., 2025) whose <think>-formatted training target fits in 4,090 backbone tokens (rationales of up to ≈4,096 tokens); the training target is the full <think> rationale followed by the answer, so at inference the model emits an explicit rationale after the 96 latents before answering. In the earlier bare-prompt evaluation, this occurred in 100% of completions; mean output length was 3,409 tokens on AIME24, 1,805 on MATH500, and 1,760 on Minerva, and 48%, 7%, and 4% of samples, respectively, reached the 4,096-token budget.

Training. Stage 1 (100K samples, one epoch, learning rate 10^{-4}) is shared with an earlier run. Stage 2 trains for two epochs (21,842 steps, global batch 64, learning rate 5×10^{-5} with cosine decay and 200 warm-up steps, AdamW with $\beta = (0.9, 0.95)$, weight decay 0.01, gradient clipping 1.0, bf16, on 8 GPUs) with prompt dropout 0.1 (applied only to the teacher-forced examples described next). Unlike the code models, stage 2 uses scheduled sampling over the latent prefix: for each example the latents fed to the decoder are the teacher-forced VAE latents with probability 0.3, latents sampled from the stage-1 flow ($\tau_z = 0.3$, guidance 2.0) with probability 0.6, or zeros with probability 0.1; the flow loss is computed only on teacher-forced examples. Results are from a single training run.

Decoding and grading. For Table 2, the prompt requests that the final answer be placed in a \boxed expression. We sample 16 completions per problem with $\tau_z = 0.3$, guidance scale 2.0, answer temperature 0.6, top-p 1.0, and a 4,096-token output budget, without a fixed sampling seed. The latent temperature and guidance scale were selected by a 3×3 grid ($\tau_z \in \{0.3, 0.6, 1.0\}$, guidance $\in \{1.0, 2.0, 3.5\}$; 4 samples per problem) on a 200-problem slice of the MATH training set using the final checkpoint of an earlier scheduled-sampling run, and reused unchanged. Answers are graded with `math_verify` on the text after </think>, taking the last \boxed expression when present. Table 2 reports this strict grading with the boxed-answer prompt. Its six-benchmark average and average gain are computed from the reported per-benchmark scores. For reference, a more lenient grading variant applied to the earlier bare-prompt outputs gives 21.0 / 22.7 / 57.5 / 80.7 / 32.3 / 44.3 (average 43.08). For completions with no \boxed, that variant extracts candidate answers from the final 400 characters (the expression after an "answer is"/"=" marker, or the last number) and accepts them if `math_verify` matches the reference. These diagnostic scores and the output-length statistics above are from the earlier bare-prompt run.

Baselines. The Base, Soft Thinking, LaVAE, and LaDiR rows of Table 2 are the numbers reported by LaDi-RL (Kang et al., 2026b) for the same backbone and benchmark suite under LaDi-RL's reported configuration (its Table 6 lists a 768-token maximum text length, temperature 1.0, and top-p 0.98 for its own model; decoding settings for its baseline rows are not stated separately).

C Additional analysis

C.1 Data decontamination

We perform a full decontamination analysis between every training corpus and every evaluation benchmark. Following standard n-gram decontamination practice, we normalize text by lowercasing, removing code fences, collapsing whitespace, and word-tokenizing. We then measure three forms of overlap: (i) corpus-level word 13-gram coverage, (ii) the maximum overlap with any single training example, and (iii) normalized exact-substring matches. We apply the analysis independently to both benchmark problem statements and reference solutions.

A benchmark example is flagged if its 13-gram corpus-level coverage is ≥ 0.8 or if an exact-substring match is found. We additionally run an 8-gram scan as a secondary check, including for benchmark prompts that are too short to contain a 13-gram. As a positive control, injecting benchmark examples into the training corpus recovers all injected examples with coverage 1.0, confirming that the pipeline detects direct contamination.

Across approximately 1.42M training records (the full Ling-Coder-SFT release plus KodCode and AceCoder, a superset of our training data) and 1,597 benchmark problems, we observe zero prompt-side flags and zero exact-substring matches. Table 7 summarizes the results. The largest remaining overlaps correspond to generic programming boilerplate rather than problem-specific content.

Table 7: Problem-statement decontamination results. Each cell reports the number of flagged benchmark problems, with the maximum observed 13-gram corpus-level coverage in parentheses. A problem is flagged when its 13-gram coverage is ≥ 0.8 or an exact-substring match is found.

Training corpus	**HumanEval(+)**	**MBPP(+)**	**LiveCodeBench v6**
Ling-Coder-SFT (1.4M)	0 (0.246)	0 (0.500)	0 (0.456)
KodCode (9.8K)	0 (0.331)	0 (0.054)	0 (0.044)
AceCoder (10K)	0 (0.095)	0 (0.167)	0 (0.015)

Table 8: Backbone drift relative to the original Qwen3-8B-Base checkpoint. The overall drift difference is small, but the stage-2-only run shows substantially larger drift in the final transformer layer.

Metric	Stage 1+2	Stage-2-only	Diff.
Overall rel-L2	0.1574	0.1599	$1.016\times$
Overall $1-\cos$	0.01237	0.01278	$1.033\times$
Layer 0 rel-L2	0.142	0.144	$+1\%$
Layer 18 rel-L2	0.144	0.144	0%
Layer 35 rel-L2	0.121	0.143	$+18\%$
Embed/head/norm rel-L2	0.261	0.261	0%

We also perform the same analysis on reference solutions. Ling-Coder produces four thresholded near-matches involving HumanEval/157 and MBPP/67, MBPP/608, and MBPP/752. Manual inspection shows that these are independent implementations of standard textbook algorithms, with no corresponding problem-statement overlap or exact-substring match. KodCode and AceCoder produce no such solution-side matches.

C.2 Backbone drift diagnostic

To further diagnose the effect of the frozen-backbone warm-up stage, we compare the default two-stage curriculum against a stage-2-only variant that directly starts full-parameter joint training from randomly initialized flow components. We analyze both early training dynamics and the final backbone drift relative to the original Qwen3-8B-Base checkpoint.

Early training dynamics. The two runs reach similar final optimization losses, but they differ substantially at the beginning of joint training. In the default curriculum, the flow-side components have already been trained while the backbone is frozen. At the start of stage 2, the model has $L_{\mathrm{NF}} \approx -0.42$ and log-determinant approximately -0.92, indicating that the shallow flow blocks have moved away from an identity map and already provide a nontrivial density model over the latent targets.

By contrast, the stage-2-only run starts joint training with $L_{\mathrm{NF}} \approx 0.47$ and log-determinant close to zero. This is close to a random Gaussian-NLL baseline and suggests that the shallow blocks initially behave almost like an identity map. Its initial gradient norm is also substantially larger than that of the warm-started run, 1.96 versus 0.96. These early dynamics suggest that, without the warm-up stage, the pretrained backbone is immediately exposed to large gradients from a poorly calibrated NF branch.

Final backbone drift. Table 8 shows that the stage-2-only model does not simply drift uniformly farther from the pretrained model. Its global relative L2 distance is only 1.6% larger than the default curriculum, and the global cosine drift increases by only 3.3%. However, the layerwise pattern is much more concentrated: the final transformer layer has 18% larger relative L2 drift, while the embedding layer, the middle layer, and the embedding/head/normalization parameters are nearly unchanged.

The final transformer layer is the shared representation that directly feeds both the LM head and the NF output head. In the stage-2-only run, this layer is exposed to early NF gradients before the flow projector, shallow blocks, and output head have learned a meaningful latent-density interface.

As a result, the largest additional movement appears exactly at the layer most directly coupled to the randomly initialized NF branch. The warm-up stage mitigates this effect by first stabilizing the flow-side components while the backbone is frozen, so that the subsequent joint-finetuning stage sends a more calibrated NF learning signal into the pretrained backbone.

C.3 Latent likelihood is not a correctness signal

We test whether the exact latent likelihood $\log p_\theta(\tilde{u} \mid q)$ can select correct programs. For NF-CoT (Unified, ckpt-22902) on HumanEval we draw K=32 samples per problem (5,248 in total; T_z=0.9, w_{CFG}=1.0, T_a=0.6) and select one of K by score. Table 9 compares the NF density with random selection, the mean per-token answer log-likelihood (ans-LL), and an oracle that picks any passing sample. Selecting by NF density stays within 2.3 points of random at every K and is below it at $K \in \{2, 4, 16\}$, whereas ans-LL gives a small lift and the oracle reaches 95.7 at K=32. On the 67 problems with both passing and failing samples, the density ranks a passing sample above a failing one with probability 0.546 (mean per-problem AUC; mean per-problem Spearman $\rho = 0.02$; standardized gap $+0.18\sigma$). Maximum-likelihood training aligns the density with the teacher's reasoning distribution, not with correctness: the likelihood is a training and policy-gradient interface, not a reranker (see Limitations).

Table 9: HumanEval pass@1 after selecting one of K samples by score.

K	2	4	8	16	32
Random	83.2	84.6	84.2	83.8	84.4
NF density	82.3	82.3	84.1	82.9	85.4
ans-LL	81.7	85.4	89.0	86.6	87.2
Oracle	88.4	92.7	93.3	93.9	95.7

C.4 Inference-time latent interventions

To test whether the sampled prefix carries task-specific content, we keep the trained NF-CoT (Unified, checkpoint 22902) model, prompt, decoder, and decoding protocol of Table 1 fixed (n=16, T_z=0.9, w_{CFG}=1.0, T_a=0.6, top-p=1.0, 1024 new tokens, vLLM answer decoding) and replace only the 64-slot latent prefix fed to the decoder: **flow** is the default NF sample; **shuffled** permutes the 64 positions of each sample (breaks order); **cross-question** feeds problem j the flow sample generated for problem j+1 (well-formed latents, wrong problem: breaks relevance); **random** draws $\mathcal{N}(0, T_z^2 I)$ in latent space without running the flow (breaks form and relevance); **zeros** feeds an all-zero prefix (removes content). Intervention seed 0 is used for the shuffled and random arms. The flow row is an independent re-evaluation of the Table 1 checkpoint and differs slightly from the published row (85.8 / 78.3 / 83.9 / 72.1).

Table 10 reports pass@1 and pass@10. Zeroing the prefix removes 50 pass@1 points on HumanEval and 28 on MBPP, yet pass@10 stays within a few points of the default (91.7 vs. 94.5 and 94.7 vs. 93.1): the reachable solution set at k=10 is essentially unchanged, so the latent content is what concentrates probability mass on correct programs at k=1. Cross-question latents cost 7.2 and 9.2 points despite being well-formed flow samples, which shows the content is problem-specific rather than a generic "think longer" signal; random prefixes cost more (10.4/15.1) because they break form as well; shuffling costs least (4.3/1.3), so order matters least. The ordering flow > shuffled > cross-question > random > zeros holds on all four benchmarks. All arms are single-seed. The last row is a separate pause-token model (Goyal et al., 2024) rather than an intervention: it replaces the 64 flow-sampled positions with 64 learned constant embeddings and is trained on the same data with the same two-stage curriculum, so it matches NF-CoT in parameters, sequence length and training compute and differs only in whether the prefix carries sampled content. Evaluated under the same protocol, it trails NF-CoT by 14.1 and 11.5 pass@1 points on HumanEval and HumanEval+, with smaller gaps of 1.5 and 0.9 on MBPP and MBPP+. The zero arm is decoded without a fixed vLLM sampler seed: with the fixed per-request seed used elsewhere, identical all-zero prefixes produce near-duplicate candidates (2.3 and 1.3 distinct completions per problem out of 16 on HumanEval and MBPP), which biases pass@10 toward pass@1; the superseded fixed-seed run is shown in grey

Table 10: Inference-time latent interventions on NF-CoT (Unified, checkpoint 22902): pass@1 and pass@10 under the Table 1 protocol (n=16, T_z=0.9, w_{CFG}=1.0, T_a=0.6, vLLM decoding). Above the first rule only the latent prefix changes; the pause row is a separately trained model. The zero arm is decoded without a fixed sampler seed (see text); the grey row is the superseded fixed-seed zero run.

		pass@1				pass@10			
Latent prefix	Breaks	HE	HE+	MBPP	MBPP+	HE	HE+	MBPP	MBPP+
Flow sample (default)	–	**84.1**	**78.4**	**83.6**	**71.4**	94.5	90.5	93.1	82.7
Shuffled positions	order	79.8	74.5	82.3	70.6	95.5	92.0	95.1	84.8
Cross-question sample	relevance	76.9	71.6	74.4	62.2	93.8	90.2	92.3	82.3
Random Gaussian	form, relevance	73.7	68.5	68.5	55.6	95.2	92.4	94.0	82.5
Zeros	content	34.0	31.6	55.6	47.5	91.7	88.5	94.7	84.4
Pause tokens (separate model)	content	70.1	66.9	82.1	70.5	93.9	90.3	92.4	82.4
Zeros (fixed seed)	content	27.2	25.1	50.3	44.4	33.1	30.7	51.8	45.8

Table 11: Generalization across backbone scales. NF-CoT improves both Qwen3-8B-Base and Qwen3-4B-Base across all five code-generation benchmarks.

Model / Method	**Backbone**	**MBPP**	**MBPP+**	**HumanEval**	**HumanEval+**	**LCB v6**	**Avg.**
Base Model	Qwen3-8B-Base	60.5	53.8	78.2	68.6	17.7	55.8
NF-CoT (Unified)	Qwen3-8B-Base	**83.9**	**72.1**	**85.8**	**78.3**	**25.4**	**69.1**
Base Model	Qwen3-4B-Base	55.5	50.2	66.3	60.2	13.5	49.1
NF-CoT (Unified)	Qwen3-4B-Base	**68.4**	**61.0**	**69.7**	**67.2**	**16.4**	**56.5**

for reference. The flow, shuffled, cross-question, and random arms are unaffected (12–15 distinct completions per problem).

C.5 Generalization across backbone scales

To test whether the gains of NF-CoT depend on the scale of the backbone model, we additionally train the unified model with Qwen3-4B-Base using the same overall training recipe and evaluate it on the same five code-generation benchmarks. Table 11 compares the 4B results with the corresponding 8B setting.

On Qwen3-4B-Base, NF-CoT improves performance on every benchmark, raising the five-benchmark average from 49.1 to 56.5, a gain of $+7.4$ points. Together with the improvements obtained using the 8B backbone, these results indicate that the benefit of latent reasoning with normalizing flows is not specific to a single backbone scale.

C.6 Parameter, compute, and memory overhead

We provide additional accounting of the parameter, compute, and memory overhead of NF-CoT relative to explicit-CoT supervised fine-tuning and LaDiR. Table 12 reports matched measurements on the Ling-Coder Python training setting.

Although NF-CoT introduces 620M additional trainable parameters, corresponding to approximately 7.6% of the 8B backbone, 582M of these parameters belong to flow components used only during training. The deployed model therefore adds only 38M parameters, or approximately 0.46% of the backbone size.

The flow-density head and the LM head each require approximately 1.24 GFLOPs when evaluated at a single sequence position. However, the flow head is evaluated at only 64 latent positions out of roughly 750 total positions in a typical training sequence. Consequently, the additional head computation contributes less than 1% of the total per-sample training FLOPs; most of the additional training cost comes from the flow-modeling procedure rather than from evaluating the output head itself.

Table 12: Training and deployment overhead under the matched Ling-Coder Python setting. Training-only flow parameters are not required by the deployed NF-CoT model.

Setting	Explicit CoT	LaDiR	NF-CoT
Epochs to converge	2	10	2
Throughput (samples/s)	27.2	6.45	18.4
Training GPU-hours	120	$\approx$2,500	**182**
Extra trainable parameters	–	+132M	+620M
Training-only parameters	–	–	582M
Inference-added parameters	–	+132M	**+38M**

The same pattern holds for memory. Peak memory consumption is dominated by the shared backbone, activations, and optimizer state rather than by the additional output head. On the 180 GB B200 GPUs used in our experiments, the marginal optimizer-state footprint of the head accounts for less than 1% of device memory. In practice, training throughput rather than memory capacity is therefore the binding resource.

Overall, NF-CoT converges in the same two epochs as explicit-CoT SFT, requiring approximately $1.5\times$ its GPU-hours (182 versus 120), while using approximately $14\times$ fewer GPU-hours than LaDiR (182 versus $\approx$2,500). This is consistent with the substantial training-FLOPs reduction reported in Table 3.