RELADRIFT: RECURRENT LANGUAGE GENERATION VIA DRIFTING FOR TEST-TIME SCALING

Anonymous authors
Paper under double-blind review

ABSTRACT

The original Drifting Model generates a sample with a single function evaluation, which makes it attractive for fast parallel language generation, but its one-step formulation does not include iterative refinement. We propose Recurrent Language Generation via Drifting (ReLaDrift), which applies one Drifting Model recurrently: each round after the first corrects the draft of the previous round, and training applies the drifting loss to one randomly chosen round without backpropagation through time. On two reasoning tasks with ground-truth answers, Sudoku Hard and GSM8K, the accuracy of ReLaDrift increases with the number of function evaluations (NFE) in the few-step regime (NFE $\leq$ 32); a single round is already at least as accurate as a single-round Drifting Model trained in the same way, and a few more rounds make it far more accurate. At every NFE from 4 to 32, ReLaDrift outperforms diffusion and flow language models of comparable size trained on the same data, reaching 92.1% on Sudoku Hard at NFE=16 and 29.1% on GSM8K at NFE=4, versus 77.5% and 9.4% for the best of them.

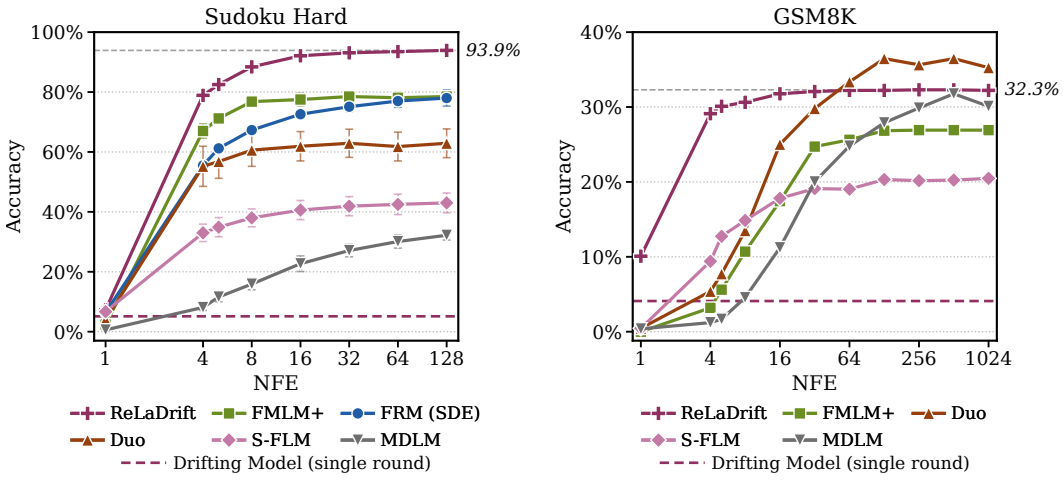

Figure 1: Accuracy versus NFE with a single trajectory and EMA weights. Left: Sudoku Hard at 20,000 updates (mean $\pm$ standard deviation over three seeds; FRM with the SDE sampler). Right: GSM8K test set at 240,000 updates (ReLaDrift with $K = 12$; S-FLM, Duo, and MDLM with temperature $T = 0.1$). K is the number of solutions given for each problem as positives in the training data. The dashed line is the single-round Drifting Model ($R_{\max} = 1$, the same training otherwise; $K = 8$ on GSM8K), which is evaluated at NFE=1.

1 INTRODUCTION

The original Drifting Model (Deng et al., 2026) is designed for one-step sampling and does not include an iterative refinement procedure. We call the ability to convert more test-time computation into better quality test-time scaling, following the use of the term for language models (Snell et al.,

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.

2025). Refinement matters for language generation, where outputs are long and every part must be consistent with the rest.

Diffusion and flow language models (Austin et al., 2021; Sahoo et al., 2024; 2025; Nie et al., 2025; Li et al., 2022; Lee et al., 2026; Deschenaux & Gulcehre, 2026; Hu et al., 2026) update token states in parallel but need many function evaluations (NFE), for example 1,024 on GSM8K and 180 on Sudoku for S-FLM, MDLM, and Duo (Deschenaux & Gulcehre, 2026), unless they are distilled (Deschenaux & Gulcehre, 2025; Lee et al., 2026) or refined over a few rounds (Agarwal et al., 2026; Helbling et al., 2026). The original Drifting Model starts from the other end: it is a one-step generator by design, and we seek to add a recurrent refinement procedure.

We propose Recurrent Language Generation via Drifting (ReLaDrift), which applies one Drifting Model recurrently in the continuous latent space of a fixed encoder. The first round generates a draft from noise, as a Drifting Model does, and each later round takes the current draft and returns a corrected one; the one-step generator is thus kept as the first round, and more rounds can be spent at test time. To train all rounds with one model at a low cost, each update samples one round, computes the earlier rounds without gradient, and applies the drifting loss only at the sampled round, without backpropagation through time.

We evaluate ReLaDrift on two reasoning tasks with ground-truth answers, on which the accuracy directly shows whether additional rounds fix errors, whereas generative perplexity on unconditional generation does not tell whether an output is correct (Deschenaux & Gulcehre, 2026). The two tasks are complementary: a Sudoku Hard puzzle has a unique answer that must satisfy global constraints, whereas a GSM8K (Cobbe et al., 2021) problem, which we solve by generating a Python program (Gao et al., 2023), has many correct programs, and on GSM8K parallel generators still lag far behind autoregressive models (Deschenaux & Gulcehre, 2026). We focus on the few-step regime, NFE ≤ 32, the range in which few-step language models are evaluated (Lee et al., 2026; Hu et al., 2026; Agarwal et al., 2026) (Section 5).

2 Background: Drifting Model

The Drifting Model (Deng et al., 2026) is a one-step generator: a network f maps a sample z of an initial distribution p_{init} on $\mathbb{R}^{D_{\text{in}}}$ to $\mathbb{R}^{D_{\text{out}}}$, and the generated distribution is the pushforward $q = f_{\#}p_{\text{init}}$. Like a GAN (Goodfellow et al., 2014), it generates in one step, but it is trained without an adversary: during training, the generated samples are moved along a vector field $V_{p,q}$ that points from q toward the target distribution p. The field is anti-symmetric, $V_{p,q} = -V_{q,p}$, so $V_{p,q} = 0$ when $q = p$, and matching the target distribution is an equilibrium.

The field is the difference between an attraction toward the target samples and a repulsion from the generated samples, with the kernel $k(x, y) = \exp(-\|x - y\|/\tau)$:

$$V_{p,q}(x) = V_p^+(x) - V_q^-(x),$$
$$V_p^+(x) = \frac{\mathbb{E}_{y\sim p}[k(x,y)\,(y-x)]}{\mathbb{E}_{y\sim p}[k(x,y)]}, \qquad V_q^-(x) = \frac{\mathbb{E}_{y\sim q}[k(x,y)\,(y-x)]}{\mathbb{E}_{y\sim q}[k(x,y)]}.$$

A sample is thus pulled toward nearby data and pushed away from nearby generated samples, and the two terms cancel when $q = p$.

Each update moves the generated samples a small step along the field, so that $q_\theta = (f_\theta)_{\#}p_{\text{init}}$ drifts toward p over training, with the stop-gradient regression loss $\mathcal{L}(\theta) = \mathbb{E}_{z\sim p_{\text{init}}}\big\|f_\theta(z) - \text{sg}\big(f_\theta(z) + V_{p,q_\theta}(f_\theta(z))\big)\big\|^2$, where sg denotes the stop-gradient operation. In practice, the expectations are estimated in each mini-batch from the target samples (positives) and the generated samples, which also serve as the negatives. The original paper also normalizes the distances and the field and evaluates the field at several kernel temperatures; we follow these choices except for the attraction (Appendix A.4).

3 Related Work

One-step and few-step generation. Diffusion and flow language models typically require multiple sampling steps, and generation quality can degrade as the number of steps is reduced (De-

schenaux & Gulcehre, 2025; Zheng et al., 2025; Agarwal et al., 2026). Recent work therefore compresses generation through distillation or directly learned one-step mappings (Deschenaux & Gulcehre, 2025; Lee et al., 2026; Song et al., 2023; Frans et al., 2025; Geng et al., 2025; Boffi et al., 2025). The Drifting Model (Deng et al., 2026) instead learns a one-step generator without an ODE, with later work connecting drifting to flow matching or using it to refine diffusion language models (Ma et al., 2026; Oba et al., 2026). ReLaDrift uses the drifting map itself as the language generator and obtains additional test-time computation by recurrently applying the same map to its own drafts.

Iterative refinement and recurrent computation. Iterative refinement is widely used in non-autoregressive generation and diffusion language models, where successive passes revise an existing prediction (Lee et al., 2018; Ghazvininejad et al., 2019; Gu et al., 2019; Savinov et al., 2022; Wang et al., 2025b; Agarwal et al., 2026). More generally, weight-tied recurrent models trade additional iterations for additional computation (Dehghani et al., 2019; Bai et al., 2019; Geiping et al., 2025). Closest to ReLaDrift, Helbling et al. (2026) train a discrete flow recurrently on its own rollouts without backpropagating through earlier iterations. ReLaDrift shares this recurrent, stop-gradient training structure, but each round is a one-step drifting map optimized with a drifting objective rather than a discrete flow trained with cross-entropy. A fuller comparison is provided in Appendix B.

4 METHODS

ReLaDrift applies one Drifting Model recurrently: the first round transports noise to a first draft of the output, as a standard Drifting Model does, and each later round feeds the current draft back into the same model and returns a corrected draft. The number of rounds at test time can then be chosen freely. Throughout, we consider conditional generation, where a problem statement is given and its solution is generated.

4.1 CONTINUOUS REPRESENTATION OF TOKEN SEQUENCES

Language is discrete, so we begin by embedding each token sequence in a continuous space where the Drifting Model can operate. A fixed encoder maps a token sequence $s = [s_1, \ldots, s_\ell]$ to one D_{emb}-dimensional vector per position, $\text{Encoder}(s) \in \mathbb{R}^{\ell \times D_{\text{emb}}}$. The encoder is not trained together with the model, as in latent generation with frozen pretrained encoders (Lovelace et al., 2023; Shabalin et al., 2025; Hu et al., 2026), and any map from token sequences to a continuous space can be used.

The problem statement is given to the model as the *prompt latent* P, the encoding of the problem statement alone. A solution in the training data is represented by a *positive* $y \in \mathbb{R}^{L \times D_{\text{emb}}}$, the encoding of the sequence that the model is trained to generate. Throughout, L is the length of the latent sequence that the model generates; which positions it covers depends on the task (Sections 5.1 and 5.2).

The output of the model is continuous, so a decoder maps it back to tokens; the decoder depends on the encoder. With a fixed code for each token, decoding is the $\arg\max$ of the inner products between the output and the codes, and no learned decoder is needed. With a pretrained contextual encoder, the outputs lie in a contextualized embedding space, so we train a separate decoder that predicts a token distribution at every position from the latent sequence (Shabalin et al., 2025); taking the $\arg\max$ independently at every position decodes all tokens in parallel (Appendix A.9). Our experiments use a one-hot code on Sudoku (Section 5.1) and a frozen pretrained encoder with a learned decoder on GSM8K (Section 5.2).

4.2 RECURRENT APPLICATION OF THE DRIFTING MODEL

ReLaDrift applies the network f_θ of the Drifting Model (Section 2), a bidirectional Transformer in our case, repeatedly: each application is a *round*, and its output is a *draft*. Round $r = 0$ produces the first draft $d_1 \in \mathbb{R}^{L \times D_{\text{emb}}}$, and each later round $r \geq 1$ takes the previous draft d_r and produces the next draft d_{r+1}. Thus N rounds are $r = 0, \ldots, N-1$, and round r returns d_{r+1}. Every round also receives the prompt latent P and the round index r, and all rounds share the parameters θ. Drafts are passed between rounds as continuous latents; they are not decoded into tokens and re-encoded.

We consider three ways to feed the draft back into the model: as the source of the transport map in place of the noise (source formulation), added to fresh noise (additive formulation), or appended to the input as extra tokens (conditional formulation). The source formulation is used in all main experiments, and the three formulations are compared on Sudoku (Section 5.1).

Source formulation. The draft becomes the source of the transport map: for a fixed prompt and round, $f_\theta(\,\cdot\,, P, r)$ maps a source $x \in \mathbb{R}^{L \times D_{\text{emb}}}$ to the same space ($D_{\text{in}} = D_{\text{out}} = LD_{\text{emb}}$ in the notation of Section 2), which is what allows the output of one round to be used as the source of the next. The first round transports noise, and each later round transports the previous draft; inside f_θ, the noise or the draft is projected to the model dimension and becomes the input u_r of the Transformer at the L generated positions:

$$\begin{aligned} d_1 &= f_\theta(\epsilon, P, 0), & u_0 &= \epsilon W_{\text{noise}}, & \epsilon \sim p_{\text{init}} = \mathcal{N}(0, I), \\ d_{r+1} &= f_\theta(d_r, P, r), & u_r &= d_r W_{\text{draft}} & (r \geq 1), \end{aligned}$$

where $W_{\text{noise}}, W_{\text{draft}} \in \mathbb{R}^{D_{\text{emb}} \times D_{\text{model}}}$. No new noise is drawn after the first round, so once the first draft is fixed, the refinement path is deterministic.

Additive and conditional formulations. Both keep the original form of the Drifting Model, which transports fresh noise at every round, and condition it on the previous draft, $d_{r+1} = f_\theta(\epsilon_r, P, d_r, r)$ with $\epsilon_r \sim p_{\text{init}}$ for $r \geq 0$, starting from a fixed initial draft d_0 (Appendix A.3); the refinement path thus remains stochastic. In the additive formulation, the projected draft is added to the projected noise at every generated position, $u_r = \epsilon_r W_{\text{noise}} + d_r W_{\text{draft}}$, so the draft enters the model in the same way as the noise. In the conditional formulation, the projected draft is instead appended to the input sequence as L separate tokens, so that $u_r = \epsilon_r W_{\text{noise}}$ and every position of the current generation can attend to every position of the draft without mixing the two, at the cost of a longer attention sequence.

Shared components. The three formulations differ only in how the noise and the draft enter the Transformer; the rest of the network is shared. In every formulation, the projected prompt latent PW_{prompt} and the projected round embedding $\phi(r)W_{\text{round}}$ are also given to the Transformer, and its output at the generated positions is projected linearly back to D_{emb}, with different output projections for the first round ($r = 0$) and the refinement rounds ($r \geq 1$). The round embedding $\phi(r)$, a fixed vector of features of r without learned parameters (an indicator of $r = 0$, $r/(1+r)$, and sinusoids), lets the model distinguish the rounds, whose input distributions differ (in the source formulation, Gaussian noise at the first round and model-produced drafts at the later rounds). Appendix A.3 gives the exact form, including how the prompt latent is placed in the input sequence for each task.

4.3 Training

Loss at one round. Each round is trained with the loss of the Drifting Model (Section 2), with the positives of a problem as samples of the target distribution p and the drafts that the round generates for the problem as samples of the generated distribution q. For the positives $y_1, \ldots, y_{N_+}$ and the n drafts $x_1, \ldots, x_n$ of a problem, which we call its *generated samples*, the field and the loss are

$$V_j = \lambda V_j^+ - V_j^-, \qquad \mathcal{L}(\theta) = \frac{1}{n} \sum_{j=1}^{n} \left\| x_j - \mathrm{sg}\big(x_j + V_j\big) \right\|^2,$$

where sg denotes the stop-gradient operation and λ is the attraction coefficient; the original Drifting Model uses $\lambda = 1$, for which the field is anti-symmetric (Section 2). The attractive drift V_j^+ is the kernel-weighted barycenter of the positives minus x_j, which is $y_1 - x_j$ with a single positive, and the repulsive drift V_j^- pushes x_j away from the other generated samples of the same problem, which serve as its negatives as in the original Drifting Model.

Training through rounds without backpropagation through time. Each update trains only one randomly chosen round and computes the earlier rounds without gradient, so that every round is trained without backpropagation through time. The same network is used at every round, so every round must be trained, but applying the loss at all rounds in each update would need a backward pass at every round. For each mini-batch, we sample the number of rounds

$$R \sim \mathrm{Uniform}\{1, \ldots, R_{\max}\},$$

where the *training depth* $R_{\max}$ is the largest number of rounds used in training, and use it for all problems in the mini-batch. The model is run for the rounds $r = 0, \dots, R-1$, and the last of them, round $R-1$, is the *terminal round*. In the source formulation, each generated sample x_j ($j = 1, \dots, n$) comes from its own noise $\epsilon_j \sim p_{\text{init}}$ through the rounds

$$d_1 = \text{sg}\big(f_\theta(\epsilon_j, P, 0)\big), \qquad d_{r+1} = \text{sg}\big(f_\theta(d_r, P, r)\big) \quad (r = 1, \dots, R-2),$$
$$x_j = f_\theta(d_{R-1}, P, R-1),$$

with $x_j = f_\theta(\epsilon_j, P, 0)$ when $R = 1$. The gradient flows only through the terminal round, and the drafts that enter it are treated as constants. Only the activations of the terminal round are stored, whatever R is, and the earlier rounds cost only forward passes (Appendix A.13). Because the terminal round is sampled, every round $r = 0, \dots, R_{\max} - 1$ is trained, each on the drafts that the current model itself produces at that round, as in training on model-generated inputs (Bengio et al., 2015; Savinov et al., 2022).

The n generated samples of a problem branch at the first round in the source formulation and at the terminal round in the additive and conditional formulations, so that they are different drafts. Because the source formulation draws noise only at the first round, each of its n noises starts its own trajectory. In the additive and conditional formulations, the recurrence is the same with $f_\theta(d_r, P, r)$ replaced by $f_\theta(\epsilon_r, P, d_r, r)$, where a new noise $\epsilon_r \sim p_{\text{init}}$ is drawn at every round r and the trajectory starts from the initial draft d_0; one such trajectory is shared by the generated samples up to round $R-2$, and n independent noises at the terminal round give the n generated samples. Appendix A.6 describes one training step in detail and summarizes it in Algorithm 1.

Number of generated samples and attraction coefficient. To keep training affordable, we use only $n = 2$ generated samples per problem in the main experiments, far fewer than the original Drifting Model (Deng et al., 2026), because each generated sample costs a pass with gradient at the terminal round, its own trajectory through the earlier rounds in the source formulation, and, on GSM8K, a decoding and an execution by the verifier (Section 4.5). With $n = 2$, each sample x_j has a single negative, the other sample x_k, whose repulsive weight is normalized to 1, so the repulsion degenerates to $V_j^- = x_k - x_j$ and grows with the distance between the two samples; with $\lambda = 1$, it drives them apart at every update, whereas with $\lambda = 2$ the attraction cancels this growth (Appendix A.5). We therefore set $\lambda = 2$ with $n = 2$. Empirically, $\lambda = 1$ with $n = 2$ collapses to 0.0% on both tasks, whereas with $n = 4$ both fields train stably on Sudoku (Section 5.1 and Appendix A.16).

The field $2V^+ - V^-$ is not anti-symmetric (the original paper found that it makes training fail on ImageNet), and neither is our attraction (Appendix A.4), so matching the target distribution is no longer guaranteed to be an equilibrium; with a single positive, however, all generated samples coinciding with it still give $V_j = 0$ for any λ, although this does not guarantee a fixed point of the recurrent generation map.

4.4 Generation and Test-Time Scaling

At test time, the number of rounds is chosen freely; it is the NFE, which we denote by N, and it is independent of the number of rounds R and the training depth $R_{\max}$ used in training. Generation with NFE $= N$ runs the rounds $r = 0, \dots, N-1$, so it goes beyond the training depth when $N > R_{\max}$. The prompt is encoded once, the draft is updated N times with the same model f_θ, and the decoder is applied once to the final draft (Algorithm 3 in Appendix A.8).

Rounds beyond the training depth. Generation can run more rounds than the training depth: every refinement round ($r \geq 1$) is trained to correct the draft that it receives, and repeating this correction beyond $R_{\max}$ keeps improving the accuracy in our experiments (Section 5). The round embedding $\phi(r)$, however, is seen in training only for $r < R_{\max}$, and its term $r/(1+r)$ grows with r, so we clamp it to the last trained round,

$$\phi(r) \;\to\; \phi\big(\min(r,\, R_{\max} - 1)\big);$$

since all refinement rounds share every parameter of f_θ, each additional round then applies exactly the map trained at round $R_{\max} - 1$, and the number of rounds becomes a pure test-time choice. The clamp is part of the model definition; Appendix A.15 reports the results without it.

4.5 Multiple Positives, a Verifier, and a Positive Queue

This section extends the training to problems with many correct solutions, such as code generation on GSM8K; our main Sudoku experiments, whose answers are unique, do not use it (Section 5.1).

Why several positives and a verifier? Problems with many correct solutions need several positives each: the attraction pulls each generated sample toward its nearest positive (Appendix A.4), so a sample that decodes to a correct program not covered by the positives is pulled toward a different program, and in code generation there are infinitely many correct programs, which differ in variable names or in how the formulas are written. Datasets such as TinyGSM (Liu et al., 2023a), which we use for GSM8K (Section 5.2), provide only one solution per problem, so we give each problem K positives generated with Qwen3.5. Because even these cover only a limited set of programs, we also add generated programs that pass an execution-based check to the positives during training and keep them in a bounded first-in-first-out queue, the *positive queue*. The verifier and the queue are used only during training; generation (Section 4.4) is unchanged.

Verifier and positive queue. A generated sample is *verified* ($v_j = 1$) if the program decoded from it at the terminal round returns the pseudo-reference obtained by voting over the numerical answers of its K given positives, which is not a ground-truth answer, and $v_j = 0$ otherwise. The positive queue $\mathcal{Q}_c$ of a problem c keeps its K given positives permanently and has four additional first-in-first-out slots for verified programs, so a problem has $N_+ \in \{K, \dots, K+4\}$ positives; a verified program enters the queue before the loss of the same step is computed. Because a verified program has become a positive, we also remove the verified samples from the negatives of the other generated samples of the same problem (Section 4.3); the verified sample itself is still trained with the same loss. Appendix A.7 gives the matching rule, the treatment of tied votes, the time limit, and one optimizer step (Algorithm 2).

5 Experiments

We evaluate ReLaDrift on Sudoku Hard (Section 5.1) and GSM8K (Section 5.2), largely following the setups of S-FLM (Deschenaux & Gulcehre, 2026). We train every compared method ourselves on the same data and state where our setups differ from those of S-FLM.

Evaluation protocol. We compare the parallel generators at the same NFE, the number of forward passes of the network at test time, and focus on the *few-step regime*, NFE ≤ 32. For ReLaDrift, NFE is the number of rounds, that is, of forward passes of f_θ; the encoder and the decoder are applied once per problem (Appendix A.13). For the compared methods, NFE is the number of ODE or sampling steps (S-FLM, Duo, MDLM, and FRM) or of refinement rounds (FMLM+); the autoregressive model makes one forward pass per generated token (with a KV cache). ReLaDrift is trained with $R_{\max} = 5$, so NFE ≥ 6 applies the model beyond the training depth with the clamped round embedding $\phi(\min(r, 4))$ (Section 4.4; Appendix A.15 gives the results without the clamp). The main metric is the accuracy of a single trajectory, one candidate per puzzle on Sudoku and one program per problem on GSM8K, with the EMA weights. No execution-based selection or refinement is used during generation. On GSM8K, the final program is executed only to compute evaluation accuracy. The verifier is used only during training (Section 4.5).

5.1 Sudoku

Experimental setup. We use the Sudoku benchmark of S-FLM (Deschenaux & Gulcehre, 2026), generated with the generator of Alp (2024): 48,000 training puzzles and 2,000 validation puzzles with no overlap, each with a unique solution. Of its three difficulty levels (40, 35, and 30 of the 81 digits visible), we use the hard level, which we call *Sudoku Hard*, because it is the least saturated: the best accuracies across all methods in Deschenaux & Gulcehre (2026, Table 1) are 96.3%, 85.2%, and 58.4% on the easy, medium, and hard levels, respectively. We use the 180-token input format of S-FLM, which FMLM+ also uses: the first 91 tokens, which contain the puzzle, are given, and the 89 answer tokens are generated and scored by exact match (Appendix A.1).

As in S-FLM, every method is trained for 20,000 updates with batch size 256, and all methods, including ReLaDrift, see the same training puzzles in the same order and use the optimization settings of the official implementation of S-FLM (Appendix A.12). We compare with the autoregressive model (AR), MDLM (Sahoo et al., 2024), Duo (Sahoo et al., 2025), and S-FLM, all trained with the official repository of S-FLM; FMLM+ (Agarwal et al., 2026), trained with its official repository; and FRM (Helbling et al., 2026), which we reimplement following its paper and evaluate with its three samplers (Appendix A.12). All of them have about 28.6M parameters (25.2M for AR), and ReLaDrift has 28.4M. ReLaDrift encodes each token by its one-hot code ($D_{\text{emb}} = 12$), decodes by the $\arg\max$ over the 12 codes, and generates the 89 answer positions ($L = 89$) after the prompt latent of the first 91 tokens (Appendices A.1 and A.3). Unless noted, ReLaDrift is the source formulation with $2V^+ - V^-$, $n = 2$ generated samples, $R_{\max} = 5$, and no verifier. Because the answer is unique, the attraction uses a single positive; a verifier would add no new positive, and its only effect would be to remove the repulsion from verified samples (Appendix A.14).

Table 1: Sudoku Hard accuracy (%) at 20,000 updates: exact match of the 89 answer tokens on the 2,000 validation puzzles, one candidate per puzzle, EMA weights, mean ± standard deviation over three training seeds. ReLaDrift settings as in Section 5.1; the lower blocks vary the drift field, n, and $R_{\max}$ of the source formulation, with the round embedding clamped to $\phi(\min(r, R_{\max} - 1))$. The autoregressive model (89 sequential forward passes): greedy 0.5±0.1, sampling 0.5±0.2. Bold: best in each column among the methods above the ablation blocks.

Method	NFE=1	4	5	8	16	32	64	128
S-FLM	6.7±1.1	33.0±2.9	34.9±3.2	38.0±3.0	40.6±3.2	41.9±3.2	42.5±3.4	43.0±3.3
FMLM+	2.4±0.4	67.0±2.4	71.2±1.5	76.8±1.4	77.5±2.1	78.5±1.4	78.1±2.0	78.5±1.5
Duo	4.0±2.1	55.2±6.7	56.8±5.6	60.6±5.4	61.9±4.9	62.9±4.7	61.8±4.8	62.9±4.8
MDLM	0.6±0.2	8.1±1.0	11.6±1.6	15.9±1.9	22.7±2.6	27.1±2.1	30.1±2.2	32.2±1.6
FRM (Euler)	6.4±0.9	54.9±1.0	60.6±0.8	66.7±1.1	71.8±1.5	74.2±1.6	75.2±1.7	75.8±1.7
FRM (SDE)	6.4±0.9	55.5±0.6	61.2±1.1	67.3±0.9	72.6±1.3	75.1±1.6	77.0±2.1	78.0±2.7
FRM (held-time)	6.4±0.9	61.0±2.5	65.3±1.4	70.6±1.5	73.5±1.7	74.4±1.5	74.5±1.5	74.5±1.5
ReLaDrift, Source	7.3±1.0	**78.9±0.6**	**82.5±0.8**	**88.4±0.4**	**92.1±0.4**	**93.1±0.6**	**93.5±0.7**	**93.9±0.7**
ReLaDrift, Additive	7.1±2.2	77.6±2.6	81.5±3.1	86.2±3.6	89.5±3.1	90.5±3.0	91.0±2.8	91.6±2.5
ReLaDrift, Conditional	**7.4±0.9**	77.6±1.7	81.1±1.1	86.1±1.2	89.6±1.8	90.7±1.7	91.2±2.0	91.6±2.3
ReLaDrift, Source: drift field and number of generated samples								
$2V^+ - V^-$, $n = 2$ (Source above)	7.3±1.0	78.9±0.6	82.5±0.8	88.4±0.4	92.1±0.4	93.1±0.6	93.5±0.7	93.9±0.7
$2V^+ - V^-$, $n = 4$	7.6±2.2	78.1±1.4	81.7±1.3	87.4±1.5	91.1±1.2	92.1±1.1	92.6±1.2	92.7±1.2
$V^+ - V^-$, $n = 2$	0.0±0.0	0.0±0.0	0.0±0.0	0.0±0.0	0.0±0.0	0.0±0.0	0.0±0.0	0.0±0.0
$V^+ - V^-$, $n = 4$	6.5±1.7	76.0±2.6	80.7±2.2	88.0±1.2	93.5±0.6	95.9±0.4	97.1±0.3	98.0±0.2
ReLaDrift, Source: training depth $R_{\max}$								
$R_{\max} = 3$	5.7±2.7	66.2±3.4	68.5±2.9	71.4±2.1	72.6±2.0	72.9±1.9	73.1±1.9	73.1±2.0
$R_{\max} = 5$ (Source above)	7.3±1.0	78.9±0.6	82.5±0.8	88.4±0.4	92.1±0.4	93.1±0.6	93.5±0.7	93.9±0.7
$R_{\max} = 8$	6.4±2.0	83.4±1.7	86.9±1.7	91.2±1.7	94.9±1.3	96.0±1.0	96.6±1.0	96.8±0.9
$R_{\max} = 16$	4.2±2.1	85.2±2.9	88.8±2.1	93.0±1.7	96.1±0.8	97.2±0.5	97.4±0.4	97.5±0.4

Results. ReLaDrift is more accurate than every compared method at every NFE from 4 to 128 (Table 1 and Figure 1, left), reaching 92.1% at NFE=16 versus 77.5% for FMLM+, the strongest compared method. At NFE=1 (7.3%), it is at the same level as the best compared methods (S-FLM 6.7% and FRM 6.4%), but it reaches 78.9% at NFE=4 and 93.5% at NFE=64 (93.9% versus 78.5% for FMLM+ at NFE=128), so its advantage holds both in the few-step regime (NFE $\leq$ 32) and beyond it. Its accuracy increases monotonically with NFE for every seed, and it keeps improving at NFE 8–128, which apply the model beyond the training depth. A single-round Drifting Model trained under the same conditions ($R_{\max} = 1$) reaches 5.1±0.8% (Appendix A.14), about the same as ReLaDrift at NFE=1; the gain on Sudoku therefore comes from the recurrent correction. ReLaDrift obtains these results without any of the additions used on GSM8K (a pretrained encoder, several positives, and the verifier) and with the same data and number of updates as the compared methods; only its update costs more computation, about 3.3 times an update of S-FLM, MDLM, or Duo, because it computes two generated samples per puzzle and the rounds before the terminal round (Appendix A.13).

The autoregressive model achieves only 0.5% accuracy, consistent with the low accuracies reported by Deschenaux & Gulcehre (2026, Table 1). Our runs of S-FLM, Duo, MDLM, and FMLM+ are within about 4.5 points of the corresponding published results (Appendix A.12).

The three formulations are close, and the source formulation is the most accurate and the most stable across seeds (standard deviation at most 1.0 at NFE $\geq$ 4), while the additive and conditional formulations are 2–3 points lower at NFE $\geq$ 8 with a larger seed variance.

Drift field and number of generated samples. With $n = 2$ generated samples, only $2V^+ - V^-$ trains, whereas with $n = 4$ both $2V^+ - V^-$ and $V^+ - V^-$ work (middle block of Table 1). With $n = 2$, $V^+ - V^-$ collapses to 0.0% at every NFE for all three seeds, as expected from Section 4.3. With $n = 4$, $2V^+ - V^-$ is close to its $n = 2$ result (91.1% versus 92.1% at NFE=16), and $V^+ - V^-$ is slightly lower at NFE ≤ 8 but higher at large NFE (93.5% and 97.1% at NFE=16 and 64, versus 92.1% and 93.5% for the default). Since $n = 2$ about halves the training cost of the source formulation compared with $n = 4$ (Appendix A.13), we use $2V^+ - V^-$ with $n = 2$ as the default on both tasks.

Training depth. A larger training depth raises the accuracy at every NFE ≥ 4 (bottom block of Table 1), at the cost of more forward passes per update. With $R_{\max} = 16$, ReLaDrift reaches 85.2% at NFE=4, 96.1% at NFE=16, and 97.4% at NFE=64, whereas $R_{\max} = 3$ saturates at about 73%; the accuracy at NFE=1 stays between 4 and 8% for every depth. The test-time scaling curve itself therefore improves with the training depth, while the source formulation with $n = 2$ runs $R_{\max} + 1$ forward passes per update on average, of which only two carry gradients (Appendix A.13).

5.2 GSM8K

Experimental setup. As in S-FLM, we train on TinyGSM (Liu et al., 2023a), synthetic math word problems generated with GPT-3.5 from GSM8K-based prompts and paired with Python programs (11,846,109 examples in the release we use; Appendix A.10), and evaluate on the 1,319 GSM8K test problems by executing one generated program per problem; a program is correct if its return value matches the answer (Appendix A.12). Because TinyGSM is derived from GSM8K training questions, we report results on the test set.

Training data. We train all methods on 1,588,781 TinyGSM problems with 12 solutions each, generated by Qwen3.5-9B (Qwen Team, 2026) and not filtered by correctness (Appendix A.10), because the Drifting Model needs several positives per problem (Section 4.5), whereas S-FLM trains on the single solution of every TinyGSM problem. ReLaDrift uses the first $K \in \{1, 4, 8, 12\}$ of the 12 solutions of each problem as its given positives, and the compared methods are trained on all 12, expanded into 19,065,372 (problem, solution) pairs and shuffled.

Compared methods. We train AR, MDLM, Duo, and S-FLM with the official repository of S-FLM and FMLM+ with its official repository on these pairs, following the GSM8K setup of S-FLM (the SmolLM-135M tokenizer (Ben Allal et al., 2024), 160.5–168.0M parameters, batch size 512, and its optimization settings), except that we train for 240,000 updates instead of 250,000 (Appendix A.12). None of them uses a verifier. AR, S-FLM, Duo, and MDLM sample at $T = 0.1$, which is more accurate than $T = 1$ for all four (Appendix A.16); FMLM+ uses $\arg\max$.

ReLaDrift. ReLaDrift generates in the latent space of the frozen CodeT5-small encoder (Wang et al., 2021) (Appendix A.1); f_θ is a 12-layer Transformer with width 768 and 12 heads (89.8M parameters), and a separately trained 42.2M-parameter decoder (Appendix A.9) maps its output back to tokens by the $\arg\max$ at every position, for 167.3M parameters in total, including the encoder, within the range of the compared methods (Appendix A.12). ReLaDrift is the source formulation with $2V^+ - V^-$, $n = 2$, $R_{\max} = 5$, the verifier, and the positive queue. It is trained with the same number of updates (240,000), batch size (512), and learning-rate schedule as the compared methods, but its Transformer and optimizer settings are based on the generator of the Drifting Model (Deng et al., 2026) rather than on S-FLM (Appendix A.12).

Results. *Test-time scaling in the few-step regime.* Up to NFE=32, the accuracy of ReLaDrift increases monotonically with NFE for every K (Table 2): for $K = 12$, it rises from 10.1% at NFE=1 to 29.1% at NFE=4, 31.8% at NFE=16, and 32.1% at NFE=32. Most of the gain is obtained by NFE=4, and the accuracy still increases slowly beyond the training depth (NFE 8–32). With more rounds, the accuracy neither improves nor degrades: from NFE=64 to NFE=1,024 it stays within 32.2–32.3% for $K = 12$. Appendix A.18 shows a program that is corrected over the rounds.

Comparison with existing methods. In the few-step regime, ReLaDrift with $K \geq 8$ is more accurate than every compared diffusion and flow language model at the same NFE. The compared methods are trained on the same data without a verifier, and our runs of them are comparable to the values reported by S-FLM (Appendix A.12). At every NFE ≤ 16, ReLaDrift with $K \geq 4$

Table 2: GSM8K test accuracy (%; 1,319 problems) at 240,000 updates, EMA weights, one trajectory per problem. K is the number of given positives per problem (Section 4.5); ReLaDrift settings as in Section 5.2. AR, S-FLM, Duo, and MDLM sample with temperature $T = 0.1$; FMLM+ commits tokens by argmax. AR generates up to 512 tokens sequentially, including the prompt. The columns to the right of the vertical line (NFE $\geq$ 64) lie outside the few-step regime. Bold: best parallel generator in each column.

Method	NFE=1	4	5	8	16	32	64	128	256	512	1024
AR ($T = 0.1$)	71.7 (sequential)										
S-FLM ($T = 0.1$)	0.5	9.4	12.7	14.9	17.8	19.1	19.0	20.3	20.2	20.2	20.5
FMLM+	0.0	3.2	5.6	10.7	17.4	24.7	25.6	26.8	26.9	26.9	26.9
Duo ($T = 0.1$)	0.5	5.4	7.7	13.5	25.0	29.8	**33.4**	**36.5**	**35.6**	**36.5**	**35.3**
MDLM ($T = 0.1$)	0.4	1.2	1.7	4.5	11.2	20.0	24.8	27.9	29.9	31.8	30.1
ReLaDrift, $K = 1$	0.5	10.5	11.5	12.9	15.3	15.6	15.8	15.8	15.8	15.6	15.7
ReLaDrift, $K = 4$	8.0	26.0	26.5	27.4	28.6	29.1	29.5	29.5	29.3	29.5	29.5
ReLaDrift, $K = 8$	8.5	27.6	29.2	29.9	31.2	31.6	31.8	31.8	31.8	31.8	31.8
ReLaDrift, $K = 12$	**10.1**	**29.1**	**30.1**	**30.6**	**31.8**	**32.1**	32.2	32.2	32.3	32.3	32.2

outperforms S-FLM, FMLM+, Duo, and MDLM at the same NFE, and at NFE=32 ReLaDrift with $K \geq 8$ (31.6–32.1%) outperforms all of them (Duo, 29.8%). ReLaDrift with $K = 4$ and $K = 8$ is given only 4 and 8 of the 12 solutions of each problem, whereas the compared methods are trained on all 12. At NFE=4, ReLaDrift (26.0–29.1%) is already close to the best compared method at NFE=32 (Duo, 29.8%), and at NFE=16 ReLaDrift with $K \geq 8$ (31.2–31.8%) exceeds every compared method at any NFE ≤ 32. With 64 or more steps, Duo becomes more accurate than ReLaDrift (33.4% at NFE=64 and 35.3% at NFE=1,024, versus 32.2% for $K = 12$), whereas S-FLM, FMLM+, and MDLM stay below ReLaDrift with $K = 12$ up to NFE=1,024 (20.5, 26.9, and 30.1% at NFE=1,024). The autoregressive model is much stronger (71.7%).

Ablations. Every ablation that trains keeps the increase of accuracy with NFE, while more positives, the verifier, and the contextualized embeddings each raise the accuracy. Giving more positives raises the accuracy: 15.3% with $K = 1$, 28.6% with $K = 4$, 31.2% with $K = 8$, and 31.8% with $K = 12$ at NFE=16. The largest gain comes from going from one to four positives, which supports the need for several positives in code generation (Section 4.5); the gain from 8 to 12 positives is small. The other ablations use $K = 8$ (Appendix A.16). A single-round Drifting Model trained in the same way ($R_{\max} = 1$) reaches 4.1% at NFE=1, whereas ReLaDrift with $K = 8$ reaches 8.5% at NFE=1 and 27.6% at NFE=4. Without the verifier and the positive queue, the accuracy at NFE=16 drops from 31.2% to 20.6%; this model is still more accurate than every compared parallel generator at NFE ≤ 8, but falls below Duo at NFE=16 and below Duo and FMLM+ at NFE=32. Replacing the contextualized CodeT5 embeddings with a context-free embedding table lowers the accuracy at NFE=16 to 18.1% (CodeT5 embedding table) and 11.3% (random table).

6 Conclusion

We proposed ReLaDrift, which applies a Drifting Model recurrently so that each round after the first corrects the draft of the previous round. ReLaDrift works in a continuous embedding space and is trained by applying the drifting loss to one randomly chosen round without backpropagating through the earlier rounds. With this training, its accuracy increases with NFE in the few-step regime without losing one-step quality. On Sudoku Hard, ReLaDrift outperforms the compared diffusion and flow language models at every NFE from 4 to 128. On GSM8K, it outperforms them only in the few-step regime and remains far below an autoregressive model.

Limitations. On GSM8K, our advantage over parallel generators at NFE=16 and 32 depends on a verifier and pretrained CodeT5 embeddings unavailable to the compared methods; the verifier also requires executable outputs and may select an incorrect pseudo-reference (Section 5.2; Appendix A.16). Training is about 3.3× more expensive per update than a one-pass method, and more with verification (Appendix A.13). GSM8K results use a single seed, with Qwen3.5-9B solutions covering only 13.4% of TinyGSM. Scaling up, handling multiple valid targets without a verifier, and narrowing the gap to autoregressive models remain future work.

AI USE STATEMENT

We used generative AI tools, including ChatGPT, Claude Code, and Codex, to assist with translation, language editing and readability, literature search and summarization, discussion of experimental methodology, interpretation of experimental results, and software implementation and debugging. We also used Qwen3.5-9B to generate candidate Python solutions for TinyGSM training problems, as described in Section 5.2 and Appendix A.10.

The authors reviewed the AI-assisted writing and analyses, checked the technical claims and references, and tested the software implementation. The generation and filtering procedures for the synthetic training solutions are documented in Appendix A.10. The authors made all final research decisions and take full responsibility for the paper and its associated code and artifacts, including material produced with generative AI assistance.

REFERENCES

Manan Agarwal, Sheel Shah, Chanhyuk Lee, Jaehoon Yoo, Jerry Huang, Seunghoon Hong, Aditi Raghunathan, Jinwoo Kim, and Nicholas M. Boffi. Posterior refinement: Fast language generation via any-order flow maps. *arXiv preprint arXiv:2606.24773*, 2026.

Ali Alp. Sudoku puzzle generator. https://github.com/alicommit-malp/sudoku, 2024.

Jacob Austin, Daniel D. Johnson, Jonathan Ho, Daniel Tarlow, and Rianne van den Berg. Structured Denoising Diffusion Models in Discrete State-Spaces. In *Advances in Neural Information Processing Systems*, volume 34, 2021.

Shaojie Bai, J. Zico Kolter, and Vladlen Koltun. Deep Equilibrium Models. In *Advances in Neural Information Processing Systems*, volume 32, 2019.

Arpit Bansal, Avi Schwarzschild, Eitan Borgnia, Zeyad Emam, Furong Huang, Micah Goldblum, and Tom Goldstein. End-to-end Algorithm Synthesis with Recurrent Networks: Extrapolation without Overthinking. In *Advances in Neural Information Processing Systems*, volume 35, 2022.

Loubna Ben Allal, Anton Lozhkov, Elie Bakouch, Leandro von Werra, and Thomas Wolf. SmolLM - blazingly fast and remarkably powerful. https://huggingface.co/HuggingFaceTB/SmolLM-135M, 2024. Model card.

Samy Bengio, Oriol Vinyals, Navdeep Jaitly, and Noam Shazeer. Scheduled Sampling for Sequence Prediction with Recurrent Neural Networks. In *Advances in Neural Information Processing Systems*, volume 28, 2015.

Nicholas M. Boffi, Michael S. Albergo, and Eric Vanden-Eijnden. Flow map matching with stochastic interpolants: A mathematical framework for consistency models. *Transactions on Machine Learning Research*, 2025.

Ting Chen, Ruixiang Zhang, and Geoffrey Hinton. Analog Bits: Generating Discrete Data using Diffusion Models with Self-Conditioning. In *International Conference on Learning Representations*, 2023.

Yuxin Chen, Chumeng Liang, Hangke Sui, Ruihan Guo, Chaoran Cheng, Jiaxuan You, and Ge Liu. LangFlow: Continuous Diffusion Rivals Discrete in Language Modeling. *arXiv preprint arXiv:2604.11748*, 2026.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*, 2021.

Mostafa Dehghani, Stephan Gouws, Oriol Vinyals, Jakob Uszkoreit, and Łukasz Kaiser. Universal Transformers. In *International Conference on Learning Representations*, 2019.

Mingyang Deng, He Li, Tianhong Li, Yilun Du, and Kaiming He. Generative modeling via drifting. *arXiv preprint arXiv:2602.04770*, 2026.

Justin Deschenaux and Caglar Gulcehre. Beyond Autoregression: Fast LLMs via Self-Distillation Through Time. In *International Conference on Learning Representations*, 2025.

Justin Deschenaux and Caglar Gulcehre. Language modeling with hyperspherical flows. *arXiv preprint arXiv:2605.11125*, 2026.

Sander Dieleman, Laurent Sartran, Arman Roshannai, Nikolay Savinov, Yaroslav Ganin, Pierre H. Richemond, Arnaud Doucet, Robin Strudel, Chris Dyer, Conor Durkan, Curtis Hawthorne, Rémi Leblond, Will Grathwohl, and Jonas Adler. Continuous diffusion for categorical data. *arXiv preprint arXiv:2211.15089*, 2022.

Gintare Karolina Dziugaite, Daniel M. Roy, and Zoubin Ghahramani. Training Generative Neural Networks via Maximum Mean Discrepancy Optimization. In *Proceedings of the Thirty-First Conference on Uncertainty in Artificial Intelligence*, 2015.

Kevin Frans, Danijar Hafner, Sergey Levine, and Pieter Abbeel. One Step Diffusion via Shortcut Models. In *International Conference on Learning Representations*, 2025.

Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, and Graham Neubig. PAL: Program-aided Language Models. In *Proceedings of the 40th International Conference on Machine Learning*, 2023.

Jonas Geiping, Sean McLeish, Neel Jain, John Kirchenbauer, Siddharth Singh, Brian R. Bartoldson, Bhavya Kailkhura, Abhinav Bhatele, and Tom Goldstein. Scaling up Test-Time Compute with Latent Reasoning: A Recurrent Depth Approach. In *Advances in Neural Information Processing Systems*, 2025.

Zhengyang Geng, Mingyang Deng, Xingjian Bai, J. Zico Kolter, and Kaiming He. Mean Flows for One-step Generative Modeling. In *Advances in Neural Information Processing Systems*, 2025.

Marjan Ghazvininejad, Omer Levy, Yinhan Liu, and Luke Zettlemoyer. Mask-Predict: Parallel Decoding of Conditional Masked Language Models. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing*, 2019.

Shansan Gong, Mukai Li, Jiangtao Feng, Zhiyong Wu, and Lingpeng Kong. DiffuSeq: Sequence to Sequence Text Generation with Diffusion Models. In *International Conference on Learning Representations*, 2023.

Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative Adversarial Nets. In *Advances in Neural Information Processing Systems*, volume 27, 2014.

Jiatao Gu, Changhan Wang, and Jake Zhao. Levenshtein Transformer. In *Advances in Neural Information Processing Systems*, volume 32, 2019.

Ishaan Gulrajani and Tatsunori B. Hashimoto. Likelihood-Based Diffusion Language Models. In *Advances in Neural Information Processing Systems*, volume 36, 2023.

Alec Helbling, Andrey Bryutkin, Mauro Martino, Duen Horng Chau, Nima Dehmamy, and Hendrik Strobelt. Flow reasoning models: Turning flows into efficient recurrent reasoners. *arXiv preprint arXiv:2606.29150*, 2026.

Keya Hu, Linlu Qiu, Yiyang Lu, Hanhong Zhao, Tianhong Li, Yoon Kim, Jacob Andreas, and Kaiming He. ELF: Embedded Language Flows. *arXiv preprint arXiv:2605.10938*, 2026.

Benhao Huang, Zhengyang Geng, and Zico Kolter. Equilibrium Reasoners: Learning Attractors Enables Scalable Reasoning. In *International Conference on Machine Learning*, 2026. URL https://arxiv.org/abs/2605.21488.

Alexia Jolicoeur-Martineau. Less is More: Recursive Reasoning with Tiny Networks. *arXiv preprint arXiv:2510.04871*, 2025.

Chanhyuk Lee, Jaehoon Yoo, Manan Agarwal, Sheel Shah, Jerry Huang, Aditi Raghunathan, Seunghoon Hong, Nicholas M. Boffi, and Jinwoo Kim. Flow map language models: One-step language modeling via continuous denoising. *arXiv preprint arXiv:2602.16813*, 2026.

Jason Lee, Elman Mansimov, and Kyunghyun Cho. Deterministic Non-Autoregressive Neural Sequence Modeling by Iterative Refinement. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, 2018.

Xiang Lisa Li, John Thickstun, Ishaan Gulrajani, Percy Liang, and Tatsunori B. Hashimoto. Diffusion-LM Improves Controllable Text Generation. In *Advances in Neural Information Processing Systems*, volume 35, 2022.

Yujia Li, Kevin Swersky, and Richard Zemel. Generative Moment Matching Networks. In *Proceedings of the 32nd International Conference on Machine Learning*, 2015.

Yaron Lipman, Ricky T. Q. Chen, Heli Ben-Hamu, Maximilian Nickel, and Matt Le. Flow Matching for Generative Modeling. In *International Conference on Learning Representations*, 2023.

Bingbin Liu, Sebastien Bubeck, Ronen Eldan, Janardhan Kulkarni, Yuanzhi Li, Anh Nguyen, Rachel Ward, and Yi Zhang. TinyGSM: achieving >80% on GSM8k with small language models. *arXiv preprint arXiv:2312.09241*, 2023a.

Xingchao Liu, Chengyue Gong, and Qiang Liu. Flow Straight and Fast: Learning to Generate and Transfer Data with Rectified Flow. In *International Conference on Learning Representations*, 2023b.

Aaron Lou, Chenlin Meng, and Stefano Ermon. Discrete Diffusion Modeling by Estimating the Ratios of the Data Distribution. In *Proceedings of the 41st International Conference on Machine Learning*, 2024.

Justin Lovelace, Varsha Kishore, Chao Wan, Eliot Shekhtman, and Kilian Q. Weinberger. Latent Diffusion for Language Generation. In *Advances in Neural Information Processing Systems*, volume 36, 2023.

Chenrui Ma, Xi Xiao, Lin Zhao, Tianyang Wang, Ferdinando Fioretto, and Yanning Shen. Drift Flow Matching. *arXiv preprint arXiv:2605.17244*, 2026.

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. In *Advances in Neural Information Processing Systems*, 2025.

Daisuke Oba, Hiroki Furuta, and Naoaki Okazaki. Drifting Objectives for Refining Discrete Diffusion Language Models. *arXiv preprint arXiv:2605.19470*, 2026.

William Peebles and Saining Xie. Scalable Diffusion Models with Transformers. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, 2023.

Patrick Pynadath, Jiaxin Shi, and Ruqi Zhang. CANDI: Hybrid discrete-continuous diffusion models. *arXiv preprint arXiv:2510.22510*, 2025.

Qwen Team. Qwen3.5-9B. https://huggingface.co/Qwen/Qwen3.5-9B, 2026. Model card.

Subham Sekhar Sahoo, Marianne Arriola, Yair Schiff, Aaron Gokaslan, Edgar Marroquin, Justin T. Chiu, Alexander Rush, and Volodymyr Kuleshov. Simple and effective masked diffusion language models. In *Advances in Neural Information Processing Systems*, volume 37, 2024.

Subham Sekhar Sahoo, Justin Deschenaux, Aaron Gokaslan, Guanghan Wang, Justin Chiu, and Volodymyr Kuleshov. The diffusion duality. In *Proceedings of the 42nd International Conference on Machine Learning*, 2025.

Nikolay Savinov, Junyoung Chung, Mikolaj Binkowski, Erich Elsen, and Aaron van den Oord. Step-unrolled Denoising Autoencoders for Text Generation. In *International Conference on Learning Representations*, 2022.

Alexander Shabalin, Viacheslav Meshchaninov, Egor Chimbulatov, Vladislav Lapikov, Roman Kim, Grigory Bartosh, Dmitry Molchanov, Sergey Markov, and Dmitry Vetrov. TEncDM: Understanding the Properties of the Diffusion Model in the Space of Language Model Encodings. *Proceedings of the AAAI Conference on Artificial Intelligence*, 39(23):25110–25118, 2025. doi: 10.1609/aaai.v39i23.34696. URL https://ojs.aaai.org/index.php/AAAI/article/view/34696.

Junzhe Shen, Jieru Zhao, Ziwei He, and Zhouhan Lin. CoDAR: Continuous Diffusion Language Models are More Powerful Than You Think. *arXiv preprint arXiv:2603.02547*, 2026.

Jiaxin Shi, Kehang Han, Zhe Wang, Arnaud Doucet, and Michalis K. Titsias. Simplified and Generalized Masked Diffusion for Discrete Data. In *Advances in Neural Information Processing Systems*, volume 37, 2024.

Charlie Snell, Jaehoon Lee, Kelvin Xu, and Aviral Kumar. Scaling LLM Test-Time Compute Optimally Can be More Effective than Scaling Parameters for Reasoning. In *International Conference on Learning Representations*, 2025.

Yang Song, Prafulla Dhariwal, Mark Chen, and Ilya Sutskever. Consistency Models. In *Proceedings of the 40th International Conference on Machine Learning*, 2023.

Robin Strudel, Corentin Tallec, Florent Altché, Yilun Du, Yaroslav Ganin, Arthur Mensch, Will Grathwohl, Nikolay Savinov, Sander Dieleman, Laurent Sifre, and Rémi Leblond. Self-conditioned Embedding Diffusion for Text Generation. *arXiv preprint arXiv:2211.04236*, 2022.

Jianlin Su, Yu Lu, Shengfeng Pan, Ahmed Murtadha, Bo Wen, and Yunfeng Liu. RoFormer: Enhanced transformer with rotary position embedding. *arXiv preprint arXiv:2104.09864*, 2021.

Guan Wang, Jin Li, Yuhao Sun, Xing Chen, Changling Liu, Yue Wu, Meng Lu, Sen Song, and Yasin Abbasi Yadkori. Hierarchical Reasoning Model. *arXiv preprint arXiv:2506.21734*, 2025a.

Guanghan Wang, Yair Schiff, Subham Sekhar Sahoo, and Volodymyr Kuleshov. Remasking Discrete Diffusion Models with Inference-Time Scaling. In *Advances in Neural Information Processing Systems*, 2025b.

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. In *International Conference on Learning Representations*, 2023. URL https://openreview.net/forum?id=1PL1NIMMrw.

Yue Wang, Weishi Wang, Shafiq Joty, and Steven C. H. Hoi. CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pp. 8696–8708, 2021.

Jiacheng Ye, Zhihui Xie, Lin Zheng, Jiahui Gao, Zirui Wu, Xin Jiang, Zhenguo Li, and Lingpeng Kong. Dream 7B: Diffusion Large Language Models. *arXiv preprint arXiv:2508.15487*, 2025.

Jaehoon Yoo, Wonjung Kim, Floor Eijkelboom, Chanhyuk Lee, Nicholas M. Boffi, Seunghoon Hong, and Jinwoo Kim. Self-conditioned Flow Map Language Models via Fixed-point Flows. *arXiv preprint arXiv:2607.00714*, 2026.

Zheng Yuan, Hongyi Yuan, Chengpeng Li, Guanting Dong, Keming Lu, Chuanqi Tan, Chang Zhou, and Jingren Zhou. Scaling Relationship on Learning Mathematical Reasoning with Large Language Models. *arXiv preprint arXiv:2308.01825*, 2023. URL https://arxiv.org/abs/2308.01825.

Eric Zelikman, Yuhuai Wu, Jesse Mu, and Noah Goodman. STaR: Bootstrapping Reasoning With Reasoning. In *Advances in Neural Information Processing Systems*, volume 35, pp. 15476–15488, 2022. URL https://proceedings.neurips.cc/paper_files/paper/2022/hash/639a9a172c044fbb64175b5fad42e9a5-Abstract-Conference.html.

Kaiwen Zheng, Yongxin Chen, Hanzi Mao, Ming-Yu Liu, Jun Zhu, and Qinsheng Zhang. Masked Diffusion Models are Secretly Time-Agnostic Masked Models and Exploit Inaccurate Categorical Sampling. In *International Conference on Learning Representations*, 2025. URL https://proceedings.iclr.cc/paper_files/paper/2025/hash/9e3b203e72c4e058de26d02a92a81844-Abstract-Conference.html.

A Appendix

A.1 Encoding of the Prompt and the Target

The prompt and the target are encoded in two different ways. The target latent is obtained by encoding the whole sequence, that is, the problem statement followed by the solution, as one sequence. With a contextual encoder, the latent at every solution position is thus contextualized on the problem statement; with a per-token code such as the one-hot code of the Sudoku experiments, the two encodings coincide position by position. The prompt latent P is obtained by encoding the problem statement alone. The two tasks lay out the prompt and the solution differently.

GSM8K. We use the frozen encoder of CodeT5-small (Wang et al., 2021) (35.3M parameters), whose contextualized embeddings have 512 dimensions, as the T5-small encoder of ELF (Hu et al., 2026) does. We use CodeT5 because its tokenizer is designed for source code; Wang et al. (2021, Section 4.2) report that the default T5 tokenizer maps code symbols such as { and } to unknown tokens. In a separate encode–decode check of a pinned T5-small tokenizer, we also observed loss of newlines and indentation, whereas the CodeT5 tokenizer used in our experiments preserved the same examples exactly (Appendix A.2). Each example is tokenized with the CodeT5 tokenizer as BOS, the problem statement, a newline, the program, and EOS, and padded to $L = 576$ tokens; the prompt consists of BOS, the problem statement, and the newline. The target latent is the output of the frozen CodeT5 encoder for the whole sequence, in which the padding tokens are hidden from the attention keys but still receive an output, so it covers all 576 positions, including the prompt positions and the padding. The prompt latent $P \in \mathbb{R}^{576\times512}$ is the output of the encoder for the prompt alone, placed at the prompt positions, with zeros at all later positions. The model generates all 576 positions, but the drift field and the loss use only the positions after the prompt (the program, EOS, and the padding), so the outputs at the prompt positions are never compared with a target; they enter the next round only as part of the draft. The model is not given the length of the prompt: the prompt positions are marked only by the nonzero prompt latent. At test time, the problem statement is encoded in the same way, and the generated program is read from the positions after the prompt up to the first EOS.

Both latents are standardized with fixed scalar constants before they enter the model:

$$z \leftarrow \frac{z-\mu}{\sigma}, \qquad \mu = 0, \quad \sigma = 0.3.$$

The same constants are used for the prompt latent, the target latent, and the continuous output of the model.

Sudoku. The sequence is the 180-token format of S-FLM: BOS, the 89-token puzzle (81 cells, where an empty cell is the blank token, and 8 row separators), BOS, and the 89-token answer, which is the solution grid with the same separators. Each of the 12 tokens is encoded by a row of the 12×12 identity matrix ($D_{\text{emb}} = 12$), without standardization. The model generates only the 89 answer positions ($L = 89$): the target latent is the code of the 89 answer tokens, and the prompt latent $P \in \mathbb{R}^{91\times12}$ is the code of the first 91 tokens. Because the code of a token does not depend on its context, the puzzle does not need to be part of the target. The drift field and the loss use all 89 answer positions, including the row separators. All 89 answer tokens, including the given digits and the row separators, are generated, and the metric is their exact match; the given digits are not overwritten, and decoding is not restricted to digits.

We also experiment with context-free embedding tables on GSM8K (the CodeT5 input embedding table and a random Gaussian table); these results are reported in Appendix A.16.

A.2 Tokenizer Round-Trip Check

We checked the round-trip preservation of code with `google-t5/t5-small` at revision `df1b051c49625cf57a3d0d8d3863ed4d13564fe4` and the tokenizer used in our experiments, `Salesforce/codet5-small` at revision `b1ee9570c289f21b5922b9c768a1ce12957bf968`. This is an empirical check of these snapshots, separate from the discussion of code-specific tokenization by Wang et al. (2021). With Python 3.12.3, Transformers 4.45.0, and Tokenizers 0.20.3, we loaded both with `AutoTokenizer` and `use_fast=True`; the returned classes were `T5TokenizerFast` and `RobertaTokenizerFast`, respectively. We encoded without added special tokens and decoded with `clean_up_tokenization_spaces=False`, both keeping and skipping special tokens.

For T5-small, the input `a\n  b\n  c\n`, where `\n` denotes an actual newline, encoded to $[3, 9, 3, 115, 3, 75]$ and decoded to `a b c` under both settings. A newline alone and four spaces alone each encoded to an empty token list. Each of {, }, and < encoded to $[3, 2]$, where token 2 is `<unk>`; decoding produced `<unk>` when special tokens were kept and an empty string when they were skipped. The following valid Python input therefore did not survive a round trip:

```
def f(x):
    if x < 2:
        return {"x": x}
    return {}
```

Keeping special tokens, T5-small decoded it as

```
def f(x): if x <unk> 2: return <unk>"x": x<unk> return <unk>
```

CodeT5-small preserved these inputs exactly, including the final newline, with no unknown tokens under either decoding setting. These observations concern the tested tokenizer snapshots and examples; they are not a claim about all T5-derived tokenizers.

A.3 Model Architecture and Input Layout

Input of the source formulation. Inside f_θ, the inputs are projected to the model dimension and summed into the initial hidden representation

$$h_r = \begin{cases} \epsilon_0 W_{\text{noise}} + P W_{\text{prompt}} + \mathbf{1}_L \phi(0) W_{\text{round}}, & r = 0, \\ d_r W_{\text{draft}} + P W_{\text{prompt}} + \mathbf{1}_L \phi(r) W_{\text{round}}, & r \geq 1. \end{cases}$$

The projection matrices $W_{\text{noise}}, W_{\text{prompt}}, W_{\text{draft}} \in \mathbb{R}^{D_{\text{emb}} \times D_{\text{model}}}$ and $W_{\text{round}} \in \mathbb{R}^{10 \times D_{\text{model}}}$ have no bias, and $\mathbf{1}_L$ is a column vector of length L whose entries are all 1, so that the round term is added at every position. In the GSM8K experiments, P, ϵ_0, and d_r all lie in $\mathbb{R}^{576 \times 512}$, and the terms are added position by position: the prompt positions receive the prompt latent together with the noise or the draft, and the later positions receive only the noise or the draft, because P is zero there; thus $h_r \in \mathbb{R}^{576 \times D_{\text{model}}}$. In the Sudoku experiments, the noise and the draft cover only the 89 answer positions, so the prompt latent is placed before them instead of being added to them:

$$h_r = \text{cat}_{\text{seq}}\big(P W_{\text{prompt}},\ x_r\big) + \mathbf{1}_{180}\, \phi(r) W_{\text{round}}, \qquad x_r = \begin{cases} \epsilon_0 W_{\text{noise}}, & r = 0, \\ d_r W_{\text{draft}}, & r \geq 1, \end{cases}$$

with $P \in \mathbb{R}^{91 \times 12}$ and $\epsilon_0, d_r \in \mathbb{R}^{89 \times 12}$, so that $h_r \in \mathbb{R}^{180 \times D_{\text{model}}}$; f_θ returns the outputs at the last 89 positions.

Round embedding. To provide the model with the current round, we use the round embedding $\phi(r)$, where r is the round index starting from 0. Specifically, we use

$$\phi(r) = \big(\mathbf{1}[r = 0],\ \frac{r}{1 + r},\ \sin(\pi r),\ \sin(\pi r/2),\ \sin(\pi r/4),\ \sin(\pi r/8),$$
$$\cos(\pi r),\ \cos(\pi r/2),\ \cos(\pi r/4),\ \cos(\pi r/8)\big).$$

$\mathbf{1}[r = 0]$ is 1 if and only if $r = 0$, and 0 otherwise. Since r is an integer, $\sin(\pi r) = 0$ for every round; we keep this constant coordinate so that the sine and cosine terms use the same four frequencies.

Backbone. We feed h_r into g_θ, a stack of bidirectional pre-normalization Transformer blocks followed by a final normalization layer. No normalization is applied to h_r itself before the first block. In the TinyGSM experiments, each block applies RMSNorm before the attention and before the feed-forward network, the feed-forward network is a SwiGLU, and the final layer is an RMSNorm. In the Sudoku experiments, each block uses a scale-only LayerNorm and a GELU feed-forward network, following the compared methods. Positional information is provided by RoPE (Su et al., 2021) applied to the queries and keys inside every attention layer of g_θ. We do not add a positional embedding to h_r. No attention mask is used; on GSM8K, the padding positions take part in the attention like the other positions. The sizes of g_θ are given in Appendix A.12.

Output projection. Finally, we apply the output projection to the normalized output of g_θ:

$$f_\theta(\cdot, P, r) = g_\theta(h_r)\,(W_{\text{out}} + W_r) + b_0.$$

Here,

$$W_r = \begin{cases} W_{\text{init}}, & r = 0, \\ W_{\text{refine}}, & r \geq 1. \end{cases}$$

Thus, we use different projections for the initial generation at $r = 0$ and the refinement rounds at $r \geq 1$. The input projections W_{noise}, W_{prompt}, W_{draft}, and W_{round} have no bias. Among the output projections, only W_{out} has the bias b_0; W_{init} and W_{refine} are bias-free.

Initialization. The round projection W_{round} and the output projections W_{init} and W_{refine} are initialized to zero, and W_{out} is drawn from $\mathcal{N}(0, 0.02^2)$ with a zero bias. On GSM8K, W_{noise} and W_{prompt} are drawn independently from $\mathcal{N}(0, 0.02^2)$, W_{draft} is initialized to zero, and the linear layers of the Transformer blocks are drawn from $\mathcal{N}(0, 0.02^2)$, except that the output projections of the attention and feed-forward layers are drawn from $\mathcal{N}(0, 0.02^2/(2 \cdot 12))$ for the 12 blocks. On Sudoku, W_{noise}, W_{prompt}, and W_{draft} start from the same matrix drawn from $\mathcal{N}(0, 0.02^2)$, and the Transformer blocks use the default initialization of PyTorch.

Additive formulation. The projected draft is added to the projected noise,

$$h_r = \epsilon_r W_{\text{noise}} + P W_{\text{prompt}} + d_r W_{\text{draft}} + \mathbf{1}_L \phi(r) W_{\text{round}},$$

with the same layout for each task as above (on Sudoku, $\epsilon_r W_{\text{noise}} + d_r W_{\text{draft}}$ occupies the 89 answer positions after $P W_{\text{prompt}}$); the rest of f_θ is the same as in the source formulation. The initial draft d_0 is the all-zero latent on GSM8K and the code of the blank token at every answer position on Sudoku.

Conditional formulation. We concatenate the latent sequence h_r^{draft}, which represents the draft from the previous round, to the latent sequence h_r^{cur}, which is used for the current generation:

$$h_r^{\text{cur}} = \epsilon_r W_{\text{noise}} + P W_{\text{prompt}} + \mathbf{1}\phi(r) W_{\text{round}} + \mathbf{1} e_{\text{cur}},$$
$$h_r^{\text{draft}} = d_r W_{\text{draft}} + \mathbf{1}\phi(r) W_{\text{round}} + \mathbf{1} e_{\text{draft}},$$

where h_r^{cur} has the layout of the source formulation for each task, e_{cur} and e_{draft} are learnable embeddings that distinguish the two sequences, and $\mathbf{1}$ is a column vector of ones of the corresponding length. The two sequences are concatenated along the sequence dimension, $H_r = \text{cat}_{\text{seq}}(h_r^{\text{cur}}, h_r^{\text{draft}})$, and fed into g_θ. Each draft position receives the same RoPE position as the corresponding position of the current generation. The next draft is obtained by applying the output projection of the source formulation to the hidden states of $g_\theta(H_r)$ at the generated positions of h_r^{cur}. We report the conditional formulation only on Sudoku. There, h_r^{cur} consists of the 91 prompt positions followed by the 89 answer positions, which carry $\epsilon_r W_{\text{noise}}$, and e_{cur} is added to all 180 positions; h_r^{draft} has the 89 answer positions, which receive the RoPE positions of the answer positions of h_r^{cur}. Attention therefore runs over 269 positions instead of the 180 positions of the source and additive formulations.

A.4 Drift Field and Loss

This section gives the exact form of the drift field and the loss of Section 4.3. Let $x_1, \ldots, x_n \in \mathbb{R}^{L \times D_{\text{emb}}}$ be the generated samples of one problem at the terminal round, and let the positives be encoded as in Appendix A.1; with a single positive, we write it as y. Norms of latent sequences are taken over all their entries.

Scaled coordinates. Following the official implementation of the Drifting Model, the drift field is computed in scaled coordinates, and the repulsive term is evaluated at several kernel temperatures rather than with a single bandwidth τ. Let G denote the set of coordinates at the positions that enter the drift field and the loss, that is, the positions after the prompt (all 89 answer positions on Sudoku; Appendix A.1); let $\|\cdot\|_G$ denote the Euclidean norm restricted to these coordinates, and let $|G|$ be their number, averaged over the mini-batch. Let $z_1, \dots, z_{n+1}$ be the n generated samples followed by the target, that is, $z_k = x_k$ for $k \le n$ and $z_{n+1} = y$, and let

$$\Delta_{jk} = \|x_j - z_k\|_G, \qquad j = 1, \dots, n,\ k = 1, \dots, n+1,$$

be the distances from each generated sample to every reference, including the self distance Δ_{jj}. Let m be the mean of all Δ_{jk} over the mini-batch, and define the scale

$$\kappa = \max\left(\frac{m}{\sqrt{|G|}},\ \delta\right), \qquad \tilde{x}_j = \frac{x_j}{\kappa}, \qquad \tilde{y} = \frac{y}{\kappa},$$

where $\delta = 10^{-3}$ prevents division by a vanishing scale. The scale κ and every quantity below except $\tilde{x}_j$ are treated as constants (stop-gradient).

Attractive drift. In general, the attractive drift is the kernel-weighted barycenter of the targets minus the sample. When N_+ targets (positives) $y_1, \dots, y_{N_+}$ are available for the same prompt,

$$V_j^+ = \sum_{i=1}^{N_+} w_{ji}^+ \tilde{y}_i - \tilde{x}_j, \qquad w_{ji}^+ = \frac{\exp\left(-\|x_j - y_i\|_G/\tau^+\right)}{\sum_{i'=1}^{N_+} \exp\left(-\|x_j - y_{i'}\|_G/\tau^+\right)}.$$

With several positives, the references z_k of the scaled coordinates (and of the repulsive affinity below) are the n generated samples followed by all N_+ positives. Unlike the original Drifting Model, which normalizes one kernel matrix over the positives and the generated samples together, we normalize the attractive weights over the positives only. With the joint normalization, the attractive weights of the positives of one problem became almost uniform in our experiments, so the attraction pointed to the centroid of the positives, which rarely decodes to a valid program. The repulsive term keeps the joint normalization of the official implementation and is then restricted to the generated samples (see below). The bandwidth τ^+ is 0.01 times the median distance between two distinct positives of the same prompt over the mini-batch; pairs whose distance is at most 10^{-4} (identical programs) are excluded from the median. We use the median of the positive set rather than a fixed value because the positive queue (Section 4.5) adds positives during training and the spread of the positive set changes; the factor 0.01 was chosen empirically. With such a small bandwidth, the attractive weight concentrates on the nearest positive. In the TinyGSM experiments, N_+ is the number of positives in the queue of the prompt, between K and $K + 4$ (Section 4.5). In the Sudoku experiments, the answer is unique and only one target y is available for each prompt ($N_+ = 1$). Then the softmax has a single entry, $w_{j1}^+ = 1$, and the attractive drift reduces to the difference between the target and the sample:

$$V_j^+ = \frac{\exp\left(-\|x_j - y\|_G/\tau^+\right)}{\exp\left(-\|x_j - y\|_G/\tau^+\right)}\,\tilde{y} - \tilde{x}_j = \tilde{y} - \tilde{x}_j.$$

Repulsive drift with multiple temperatures. The repulsive drift uses the kernel weights of the official implementation, which are computed at three temperatures. Let $\bar{\Delta}_{jk} = \Delta_{jk}/m$ be the normalized distances, where a large constant is added to the self entries $\bar{\Delta}_{jj}$ so that a sample does not repel itself. For each temperature $\tau \in \{0.02,\ 0.05,\ 0.2\}$, we form the logits $\ell_{jk} = -\bar{\Delta}_{jk}/\tau$ and the doubly normalized affinity

$$a_{jk}^{(\tau)} = \sqrt{\mathrm{softmax}_k(\ell_{j\cdot})_k \ \mathrm{softmax}_j(\ell_{\cdot k})_j},$$

where the first softmax is over the $n+1$ references of sample j and the second is over the n generated samples of reference k. Restricting to the other generated samples and renormalizing gives the weights

$$w_{jk}^{(\tau)} = \frac{a_{jk}^{(\tau)}}{\sum_{\ell \le n,\, \ell \ne j} a_{j\ell}^{(\tau)}}, \qquad k \le n,\ k \ne j,$$

and the repulsive drift at temperature τ is

$$V_j^{-,(\tau)} = \sum_{k \neq j} w_{jk}^{(\tau)} \left(\tilde{x}_k - \tilde{x}_j\right).$$

A small temperature makes the weights concentrate on the nearest generated sample, while a large temperature spreads them over all generated samples; using the three temperatures together captures both the local and the global structure of the generated samples.

Drift field and loss. For each temperature, the drift field is the difference of the two terms with an attraction coefficient λ,

$$V_j^{(\tau)} = \lambda V_j^{+} - V_j^{-,(\tau)}.$$

The original Drifting Model uses $\lambda = 1$ ($V^+ - V^-$). We use $\lambda = 2$ ($2V^+ - V^-$) in all main experiments (Tables 1 and 2). The reason is the number of generated samples: for computational reasons we use only $n = 2$ generated samples per prompt, so each sample j has a single negative k, whose weight $w_{jk}^{(\tau)}$ is normalized to 1, and the repulsion degenerates to $V_j^{-,(\tau)} = \tilde{x}_k - \tilde{x}_j$ at every temperature: the farther the other sample, the stronger the push, which breaks the local repulsion that the kernel is meant to give. With $\lambda = 1$, this drives the two samples apart at every update, whereas with $\lambda = 2$ the attraction cancels this growth; Appendix A.5 derives this. Empirically, $\lambda = 1$ with $n = 2$ collapses to 0% on both Sudoku and GSM8K, whereas $\lambda = 2$ trains stably (Table 1 and Appendix A.16). Let ρ_τ be the root mean square of $V^{(\tau)}$ over all generated samples in the mini-batch and all coordinates in G. The final drift is the sum of the normalized fields over the three temperatures,

$$V_j = \sum_{\tau \in \{0.02,\, 0.05,\, 0.2\}} \frac{V_j^{(\tau)}}{\rho_\tau}.$$

We then construct the stop-gradient target as

$$x_j^{\mathrm{tgt}} = \mathrm{sg}\left(\tilde{x}_j + V_j\right).$$

With a single target, this target has a simple form. Consider first $\lambda = 1$ and one temperature without the RMS normalization, $V_j = V_j^+ - V_j^-$. Substituting $V_j^+ = \tilde{y} - \tilde{x}_j$, the sample cancels and only the target and the repulsive term remain:

$$\tilde{x}_j + V_j = \tilde{x}_j + (\tilde{y} - \tilde{x}_j) - V_j^- = \tilde{y} - V_j^-, \qquad x_j^{\mathrm{tgt}} = \mathrm{sg}\left(\tilde{y} - V_j^-\right).$$

That is, the loss pulls each sample toward the target $\tilde{y}$ shifted by the repulsion from the other samples, which is exactly the stop-gradient target of the original Drifting Model. With the three RMS-normalized temperatures and a general λ, the same substitution gives

$$\begin{aligned} x_j^{\mathrm{tgt}} &= \mathrm{sg}\left(\tilde{x}_j + \sum_\tau \frac{\lambda(\tilde{y} - \tilde{x}_j) - V_j^{-,(\tau)}}{\rho_\tau}\right) \\ &= \mathrm{sg}\left((1 - \lambda\gamma)\,\tilde{x}_j + \lambda\gamma\,\tilde{y} - \sum_\tau \frac{V_j^{-,(\tau)}}{\rho_\tau}\right), \qquad \gamma = \sum_\tau \frac{1}{\rho_\tau}, \end{aligned}$$

so the target is a weighted combination of the sample and $\tilde{y}$ minus the normalized repulsive terms, and it reduces to $\mathrm{sg}(\tilde{y} - V_j^-)$ when $\lambda\gamma = 1$.

For one training example, the loss is

$$\mathcal{L}(\theta) = \frac{1}{nLD_{\mathrm{emb}}} \sum_{j=1}^{n} \left\| \tilde{x}_j - x_j^{\mathrm{tgt}} \right\|_G^2,$$

where the gradient flows only through $\tilde{x}_j = x_j / \mathrm{sg}(\kappa)$.

The loss is applied only at the sampled terminal round. Appendix A.6 describes the full procedure over a mini-batch step by step and summarizes it in Algorithm 1.

A.5 Why $2V^+ - V^-$ Is Needed with Two Generated Samples

This appendix explains why, with $n = 2$ generated samples, the field $V^+ - V^-$ collapses on Sudoku whereas $2V^+ - V^-$ trains stably (Table 1). In the Drifting Model, the repulsive weights are a kernel of the distance, so a sample is pushed away mainly by the generated samples that are close to it. With $n = 2$ generated samples, however, each sample j has a single negative, the other generated sample k, and the normalization over the other generated samples degenerates: for every temperature τ,

$$w_{jk}^{(\tau)} = \frac{a_{jk}^{(\tau)}}{a_{jk}^{(\tau)}} = 1, \qquad V_j^{-,(\tau)} = \tilde{x}_k - \tilde{x}_j,$$

whatever the distance between the two samples (the RMS normalization only rescales the field). The repulsion therefore grows linearly with the distance: the farther the other sample, the stronger the push, which is the opposite of the local repulsion that the kernel is meant to give. With a single positive, $V_j^+ = \tilde{y} - \tilde{x}_j$, and the field

$$V_j^{(\tau)} = \lambda(\tilde{y} - \tilde{x}_j) - (\tilde{x}_k - \tilde{x}_j)$$

is the same at every temperature, so $V_j = \gamma\, V_j^{(\tau)}$ with $\gamma = \sum_\tau 1/\rho_\tau$ (Appendix A.4). For the targets $x_j^{\mathrm{tgt}} = \mathrm{sg}(\tilde{x}_j + V_j)$ and $x_k^{\mathrm{tgt}} = \mathrm{sg}(\tilde{x}_k + V_k)$ of the two samples, this gives

$$x_j^{\mathrm{tgt}} - x_k^{\mathrm{tgt}} = \big(1 + \gamma\,(2 - \lambda)\big)\,(\tilde{x}_j - \tilde{x}_k).$$

With $\lambda < 2$, the factor $1 + \gamma(2 - \lambda)$ exceeds 1: every update asks the two samples to move further apart, and nothing bounds their separation; with $\lambda = 1$, the factor is $1 + \gamma$, which is consistent with the collapse to 0.0% in Table 1. With $\lambda = 2$, the factor is exactly 1: the target keeps the current separation, so this growth of the separation disappears. Thus $\lambda = 2$ is the smallest attraction coefficient that prevents this growth. On GSM8K, the attraction concentrates on the nearest positive (Appendix A.4), so the same relation holds approximately when the two samples share their nearest positive. With $n = 4$, the weights over the three other samples depend on the distance again, and both fields train stably in our experiments (Table 1).

A.6 Training Procedure of the Recurrent Model

This section describes one optimizer step of the source formulation (Section 4.3) over a mini-batch of B problems in words; Algorithm 1 summarizes it. The additive and conditional formulations differ in the recurrence, which draws fresh noise at every round and starts from the initial draft d_0, and in where the generated samples branch (Section 4.3), and the changes for the verifier and the positive queue are given in Section 4.5 and Appendix A.7.

Step 1: sampling the number of rounds. For each mini-batch, the number of rounds $R \sim \mathrm{Uniform}\{1, \ldots, R_{\max}\}$ is drawn and shared by all B problems; the terminal round is round $R - 1$. Sharing R aligns the rounds of all examples, so every intermediate round is one batched forward pass over the mini-batch.

Step 2: encoding the prompt and the positives. The prompt latent P_b and the positives $y_{b,i}$ are obtained as in Appendix A.1; on GSM8K, both are standardized with the fixed constants (μ, σ). The encoder is frozen, so these latents are computed once and cached; in the TinyGSM experiments the latents of the K given positives are precomputed and only the programs added to the positive queue are encoded during training.

Step 3: intermediate rounds without gradient. For each problem b, n independent noises $\epsilon'_{b,j} \sim \mathcal{N}(0, I)$ are drawn at all L positions, one per trajectory, and the rounds $r = 0, \ldots, R - 2$ of each trajectory are computed under stop-gradient: the first round transports the noise, $d_1 = \mathrm{sg}(f_\theta(\epsilon'_{b,j}, P_b, 0))$, and each later round is applied to the previous output with the round embedding $\phi(r)$, $d_{r+1} = \mathrm{sg}(f_\theta(d_r, P_b, r))$. No new noise is drawn after the first round, no decoding or re-encoding takes place between rounds, and no activations are stored, so these passes cost only forward computation. In the additive and conditional formulations, one intermediate trajectory is started from the fixed initial draft d_0 with fresh noise at every round, shared by the generated samples, and the branching happens at the terminal round. When $R = 1$, this step is empty and the terminal round is the first round.

Step 4: terminal round with gradient. At round $R-1$, each trajectory gives its generated sample $x_{b,j} = f_\theta(d_{R-1}, P_b, R-1)$ (or $x_{b,j} = f_\theta(\epsilon'_{b,j}, P_b, 0)$ when $R = 1$), computed with gradient. With the verifier, each $x_{b,j}$ is decoded and executed at this point, and verified programs are pushed into the positive queue before the drift is computed.

Step 5: distances and scale. For each generated sample, the Euclidean distances to every reference (all generated samples of the same problem, including itself, and its positives) are computed on the coordinates in G only. Their mean m over the whole mini-batch gives the scale $\kappa = \max(m/\sqrt{|G|}, \delta)$; all samples and positives are divided by κ, and κ is treated as a constant.

Step 6: attractive drift. The attractive drift of a generated sample is the kernel-weighted barycenter of the positives of its problem minus the sample, with weights $\mathrm{softmax}(-\|x_{b,j} - y_{b,i}\|_G/\tau^+)$ over the positives only. The bandwidth τ^+ is 0.01 times the median distance between two distinct positives of the same problem over the mini-batch; with a single positive, the drift is simply $\tilde{y}_b - \tilde{x}_{b,j}$.

Step 7: repulsive drift at three temperatures. For each temperature $\tau \in \{0.02, 0.05, 0.2\}$, the normalized distances $\Delta_{b,jk}/m$ (with the self distance masked out) are turned into the doubly normalized affinity of the official implementation, restricted to the other generated samples of the same problem, and renormalized. The repulsive drift at temperature τ is the affinity-weighted sum of the differences $\tilde{x}_{b,k} - \tilde{x}_{b,j}$. With the verifier, the weights of verified samples are set to zero after the normalization so that accepted samples do not repel the others.

Step 8: drift field, normalization, and target. The field at each temperature is $\lambda V^+ - V^{-,(\tau)}$ with $\lambda = 2$. Each field is divided by its root mean square ρ_τ over all generated samples in the mini-batch and all coordinates in G, and the three normalized fields are summed. The stop-gradient target is $x^{\mathrm{tgt}}_{b,j} = \mathrm{sg}(\tilde{x}_{b,j} + V_{b,j})$.

Step 9: loss and update. The loss is the mean squared distance between $\tilde{x}_{b,j}$ and $x^{\mathrm{tgt}}_{b,j}$ on the coordinates in G, divided by $nBLD_{\mathrm{emb}}$. Its gradient flows only through the terminal-round outputs $\tilde{x}_{b,j} = x_{b,j}/\,\mathrm{sg}(\kappa)$; the intermediate rounds and every other quantity are constants. The gradient norm is clipped to 1, one AdamW step is taken, and the EMA weights are updated.

A.7 Training Step with the Verifier and the Positive Queue

Algorithm 2 shows one optimizer step of the source formulation with the verifier and the positive queue (Section 4.5). It extends Algorithm 1 by decoding and executing the terminal samples, updating the positive queues, and removing the repulsion from verified samples.

In the TinyGSM experiments, we use $n = 2$ generated samples (independent trajectories) per problem and $\lambda = 2$. Learning from generated reasoning selected using known answers has precedents in STaR (Zelikman et al., 2022) and rejection sampling fine-tuning (Yuan et al., 2023); here, accepted programs enter the drifting positive set, and their corresponding generated samples are removed from the negative set. Deng et al. (2026, Appendix A.8) cache real samples in queues; our per-problem queue stores generated programs accepted by an execution-based check.

Pseudo-reference. The verifier uses a voted pseudo-reference from the numerical answers of the K given positives, not a ground-truth answer. The vote selects the unique largest answer group (a plurality, without requiring more than half the votes). Values are sorted and grouped when their distance from the group's minimum is at most 10^{-3}; the reference is the value from the earliest selected positive in the winning group. If the largest group is tied, or if the tolerance relation is nontransitive, the verifier is disabled for that problem, which is then trained with its K given positives only; no generated sample is accepted or masked. The original TinyGSM solution is not used to construct this reference. Self-consistency aggregates sampled reasoning paths to select an answer at inference time (Wang et al., 2023); here, voting supplies a training-time pseudo-reference. Agreement with it does not establish ground-truth correctness.

Execution. At the sampled terminal round, each generated sample is decoded into tokens with the decoder (Appendix A.9), the program after the prompt (up to the first EOS) is executed in a sandbox

Algorithm 1 One training step of ReLaDrift (source formulation)

Require: Mini-batch of B problems with prompts P_b and positives $\{y_{b,i}\}$, model f_θ, maximum number of rounds $R_{\max}$, number of generated samples n, attraction coefficient λ, temperatures $\mathcal{T} = \{0.02, 0.05, 0.2\}$

Sample the number of rounds $R \sim \mathrm{Uniform}\{1, \dots, R_{\max}\}$ for the mini-batch
for $b = 1, \dots, B$ and $j = 1, \dots, n$ **do**
  Sample $\epsilon'_{b,j} \sim \mathcal{N}(0, I)$ and set $d \leftarrow \epsilon'_{b,j}$ ▷ one trajectory per generated sample
  for $r = 0, \dots, R-2$ **do** ▷ intermediate rounds, no gradient
    $d \leftarrow \mathrm{sg}\big(f_\theta(d, P_b, r)\big)$ ▷ round 0 transports the noise; later rounds refine the previous output
  end for
  $x_{b,j} \leftarrow f_\theta(d, P_b, R-1)$ ▷ terminal round, with gradient
end for
Compute the mini-batch scale κ and the scaled coordinates $\tilde{x}_{b,j} \leftarrow x_{b,j}/\kappa$, $\tilde{y}_{b,i} \leftarrow y_{b,i}/\kappa$ (Appendix A.4)
for each generated sample (b, j) **do**
  $V^{+}_{b,j} \leftarrow$ kernel-weighted barycenter of the positives of problem b minus $\tilde{x}_{b,j}$
  $V^{-,(\tau)}_{b,j} \leftarrow$ affinity-weighted repulsion from the other generated samples of problem b, for each $\tau \in \mathcal{T}$
  $V^{(\tau)}_{b,j} \leftarrow \lambda V^{+}_{b,j} - V^{-,(\tau)}_{b,j}$
end for
$V_{b,j} \leftarrow \sum_{\tau \in \mathcal{T}} V^{(\tau)}_{b,j} / \rho_\tau$, where ρ_τ is the root mean square of $V^{(\tau)}$ over the mini-batch
$x^{\mathrm{tgt}}_{b,j} \leftarrow \mathrm{sg}\big(\tilde{x}_{b,j} + V_{b,j}\big)$
$\mathcal{L}(\theta) \leftarrow \frac{1}{nBLD_{\mathrm{emb}}} \sum_{b,j} \|\tilde{x}_{b,j} - x^{\mathrm{tgt}}_{b,j}\|^2_G$, and update θ with its gradient, which flows only through $\tilde{x}_{b,j}$

with a time limit of 15 ms, and the sample is verified if its return value matches the reference answer within 10^{-3}.

Positive queue. A verified program is written back as the prompt followed by the program and EOS and is encoded with the same encoder in the same way as the given positives (Appendix A.1). It is pushed into the four additional slots only if it is not already in the queue, and only the programs added during training are pushed out; the K given positives are never pushed out, even when the queue is full.

Repulsion mask. With the weights $w^{(\tau)}_{jk}$ of Appendix A.4, the repulsive drift becomes

$$V^{-,(\tau)}_{j} = \sum_{k \neq j} w^{(\tau)}_{jk} \, (1 - v_k) \, (\tilde{x}_k - \tilde{x}_j) \, ,$$

that is, the weights of verified samples are set to zero after the normalization, without renormalizing the remaining weights. The root mean square ρ_τ is computed from the field before this masking. The loss itself is applied to all generated samples, including the verified ones.

Sudoku. On Sudoku, the verifier checks the rules of Sudoku and the agreement with the clues. Because the answer is unique, a verified answer is always the given positive and the queue never grows (we observed $N_+ = 1$ throughout training); there the verifier acts only through the repulsion mask.

A.8 Generation Procedure

Algorithm 3 gives the generation procedure of Section 4.4: the prompt is encoded once, the draft is updated N times with the same model f_θ, with the round embedding clamped beyond the training depth, and the decoder is applied once to the final draft.

Algorithm 2 One training step of the source formulation with the verifier and the positive queue

Require: Problems $\{c_b\}_{b=1}^{B}$, model f_θ, decoder, positive queues $\{\mathcal{Q}_c\}$, pseudo-references $\{a_c\}$ (votes of the K given positives), number of generated samples n, maximum number of rounds $R_{\max}$, attraction coefficient λ

Encode the prompts P_b (zeros after the prompt) and set G_b to the positions after the prompt of problem c_b
Sample the number of rounds $R \sim \mathrm{Uniform}\{1, \dots, R_{\max}\}$ for the whole mini-batch
for $b = 1, \dots, B$ and $j = 1, \dots, n$ **do**
    Sample $\epsilon_{b,j} \sim \mathcal{N}(0, I)$ and run the source recurrence for R rounds; rounds $0, \dots, R-2$ are computed without gradient
    $x_{b,j} \leftarrow$ output of the terminal round $R-1$ (with gradient)
end for
for $b = 1, \dots, B$ and $j = 1, \dots, n$ **do**
    Decode $x_{b,j}$, execute the program, and set $v_{b,j} \leftarrow \mathbf{1}[|\text{output} - a_{c_b}| \le 10^{-3}]$ ($v_{b,j} \leftarrow 0$ if execution fails or c_b has no valid pseudo-reference)
    if $v_{b,j} = 1$ **then**
        Push the program into $\mathcal{Q}_{c_b}$ (the K given positives are never pushed out)
    end if
end for
Encode the programs newly pushed into $\mathcal{Q}_{c_b}$ (the latents of the K given positives are cached) and collect the positives $y_{b,1}, \dots, y_{b,N_{+,b}}$ of $\mathcal{Q}_{c_b}$
Compute the drift $V_{b,j}$ with these positives, the coefficient λ, and the repulsion mask v, and set $x_{b,j}^{\mathrm{tgt}} \leftarrow \mathrm{sg}(\tilde{x}_{b,j} + V_{b,j})$ (Appendix A.4 and this section)
$\mathcal{L} \leftarrow \frac{1}{nBLD_{\mathrm{emb}}} \sum_{b,j} \|\tilde{x}_{b,j} - x_{b,j}^{\mathrm{tgt}}\|_{G_b}^2$; backpropagate, clip the gradient norm to 1, and take an AdamW step
Update the EMA weights

Algorithm 3 Generation with N function evaluations.
Red: source formulation; blue: additive and conditional formulations.

Require: Problem statement, model f_θ, decoder, number of rounds N (the NFE), training depth $R_{\max}$; initial draft d_0 (additive and conditional formulations only)
Ensure: Generated token sequence

$P \leftarrow$ prompt latent of the problem statement (Section 4.1)
Sample $\epsilon \sim \mathcal{N}(0, I)$
if source formulation **then**
    $d \leftarrow f_\theta(\epsilon, P, 0)$ ▷ first round transports the noise
else
    $d \leftarrow f_\theta(\epsilon, P, d_0, 0)$ ▷ first round: noise and the initial draft
end if
for $r = 1, \dots, N-1$ **do**
    $\bar{r} \leftarrow \min(r, R_{\max} - 1)$ ▷ clamp the round embedding beyond the training depth
    if source formulation **then**
        $d \leftarrow f_\theta(d, P, \bar{r})$ ▷ refine the previous output; no new noise
    else
        Sample $\epsilon \sim \mathcal{N}(0, I)$
        $d \leftarrow f_\theta(\epsilon, P, d, \bar{r})$ ▷ fresh noise; the previous output is the draft
    end if
end for
return the tokens decoded from d (Section 4.1)

A.9 DECODER

This appendix describes the learned decoder used with the pretrained contextual encoder (CodeT5) in the GSM8K experiments; on Sudoku, decoding is the inner product with the one-hot codes followed by the argmax (Section 5.1). The decoder is a separate bidirectional Transformer (8 layers, width 512, 8 heads; 42.2M parameters) that takes a sequence of latents as input and predicts a token distribution at every position (Section 5.2). It is trained to recover the token sequence from perturbed latents with the corruption that ELF uses to train decoding (Hu et al., 2026, Appendix C.1): at every position, the clean latent z is replaced by $\alpha z + (1-\alpha)\epsilon$ with $\epsilon \sim \mathcal{N}(0, I)$ and $\alpha = \mathrm{sigmoid}(0.8 + 0.8\,\xi)$, $\xi \sim \mathcal{N}(0, 1)$, drawn independently for each position. The training data are the CodeT5 latents of one Qwen3.5 solution for each of 100,000 training problems; we train for 20,000 updates with batch size 64, AdamW (learning rate 3×10^{-4} after 500 warm-up updates, decayed to 3×10^{-5}, weight decay 0.01), and use the checkpoint with the best validation score, at 18,000 updates. The decoder is trained on latents standardized with its own statistics $(\mu_{\mathrm{dec}}, \sigma_{\mathrm{dec}})$, which differ from the constants (μ, σ) used by the Drifting Model (Appendix A.1). Therefore, a generated latent z is re-standardized by the affine map

$$z_{\mathrm{dec}} = \frac{z\,\sigma + \mu - \mu_{\mathrm{dec}}}{\sigma_{\mathrm{dec}}}$$

before it is given to the decoder. The same decoder is used by the verifier during training (Section 4.5).

A.10 GENERATING THE POSITIVES

We use the `train` split of `TinyGSM/TinyGSM`, revision `5049a7840fa546985eaae4c698d5daf6fdf7b47d`, containing 11,846,109 question–program pairs. This is the size of the released dataset used here; the original paper (Liu et al., 2023a) reports 12.3M pairs. We retain the 11,811,180 pairs that fit within 576 CodeT5 tokens, then use `datasets.Dataset.train_test_split` with a validation fraction of 0.01 and seed 42 to obtain 11,693,068 training pairs and 118,112 validation pairs. We generate alternatives for the first 2,483,048 problems of this shuffled training split (21.0% of the unfiltered release).

We sampled 32 Python solutions per problem using `Qwen/Qwen3.5-9B` (Qwen Team, 2026) at revision `c202236235762e1c871ad0ccb60c8ee5ba337b9a`, with the thinking mode off, temperature 0.85, top-p 0.95, top-k 20, min-p 0, no presence, frequency, or repetition penalty, and at most 600 generated tokens, using the prompt below, which follows the TinyGSM paper. A solution is kept if it is executable, which we check with a restricted interpreter, fits within 576 tokens of the CodeT5 tokenizer used by ReLaDrift and 512 tokens of the SmolLM tokenizer used by the compared methods, and differs from the other kept solutions of the same problem after removing comments, docstrings, and whitespace. In the kept solutions, we remove the comments but keep the docstrings. The training set consists of the 1,588,781 problems (13.4% of TinyGSM) with at least 12 kept solutions, with 12 solutions each; all 12 return the same value for 62.7% of the problems. Except for the comments, the generated programs are similar in style to the TinyGSM solutions (Appendix A.11). In the prompt, {{ `question` }} is replaced by the problem statement.

```
You are given a grade-school math word problem.
Generate a correct Python solution that satisfies all of the following
requirements:
1. Define exactly one function with this signature:

def simple_math_problem():

2. Immediately after the function definition, include a triple-quoted
   docstring containing the given problem statement exactly as provided.
3. Solve the problem using ordinary Python arithmetic.
4. Use meaningful variables for quantities appearing in the problem.
5. Solve the problem step by step using intermediate variables when natural.
6. The function must return the final numerical answer.
7. The program must be executable and must return the mathematically
   correct answer.
8. Do not print the answer.
```

```
9. Do not use Markdown code fences.
10. Do not include any explanation or text outside the Python function.
11. Keep the solution natural and similar in style to real TinyGSM
    solutions. Do not use unnecessarily sophisticated Python features,
    helper functions, classes, loops, comprehensions, symbolic algebra, or
    external libraries unless they are genuinely necessary for the problem.
12. Keep comments minimal. Do not perform calculations in comments, and do
    not include numerical values in comments.
13. The numerical computation must be derived from the quantities in the
    problem. Do not simply return the known final answer directly.

### Question

{{ question }}

### Program
```

A.11 Examples of the Positives

The following shows the TinyGSM solution of one problem and two of the solutions generated by Qwen3.5-9B (before removing comments; long docstrings are shortened here).

```
# TinyGSM solution
def simple_math_problem() -> int:
    '''
    David's property tax rate is 2.5%.
    His house is currently worth $500,000.
    ... (problem statement)
    '''
    tax_rate = 0.025
    current_value = 500000
    increase_percentage = 0.20
    max_property_tax = 20000
    # Calculate the new value of the house after the increase
    new_value = current_value * (1 + increase_percentage)
    # Calculate the maximum allowed property tax
    max_property_tax_allowed = new_value * tax_rate
    # Calculate the maximum value of improvements David can make
    max_improvement_value = max_property_tax_allowed - max_property_tax
    result = max_improvement_value
    return result

# Qwen3.5-9B, example 1
def simple_math_problem():
    """
    David's property tax rate is 2.5%. ... (problem statement)
    """
    tax_rate = 0.025
    current_value = 500000.0
    value_increase_percent = 0.20
    max_tax_affordable = 20000.0

    increased_value = current_value * (1 + value_increase_percent)
    current_tax = current_value * tax_rate
    new_tax = increased_value * tax_rate

    tax_increase_allowed = max_tax_affordable - current_tax
    improvements_possible = tax_increase_allowed / tax_rate

    return improvements_possible

# Qwen3.5-9B, example 2
def simple_math_problem():
    """
```

```
    David's property tax rate is 2.5%. ... (problem statement)
    """
    tax_rate = 0.025
    current_value = 500000.0
    value_increase_factor = 1.20
    max_tax_budget = 20000.0

    projected_value = current_value * value_increase_factor
    max_affordable_value = max_tax_budget / tax_rate
    additional_value_capacity = max_affordable_value - projected_value

    return additional_value_capacity
```

The three programs return different values ($-5{,}000$, 300,000, and 200,000): neither the Qwen3.5 solutions nor the original TinyGSM solutions are guaranteed to be correct, and we do not filter the solutions by correctness (Section 5.2).

A.12 Experimental Settings

Tables 3 and 4 summarize the settings of ReLaDrift and the compared methods on Sudoku (Section 5.1) and GSM8K (Section 5.2). On both tasks, AR, MDLM, Duo, and S-FLM are built from the official repository of S-FLM (https://github.com/jdeschena/s-flm, commit `30def79`), which also implements AR, MDLM, and Duo, and FMLM+ from its official repository (https://github.com/MananAg007/posterior-refinement, commit `73652f9`); neither repository is modified. On GSM8K, we run their official training pipelines. On Sudoku, all methods share our single-GPU training loop, which calls the official model, loss, optimizer, learning-rate schedule, and EMA of each repository, so that all methods see the same puzzles in the same order, with the incomplete last batch of each epoch dropped. For FRM, we use our own implementation following its paper (Helbling et al., 2026). On Sudoku, the S-FLM setting is the one of the official Sudoku script of S-FLM (the DiT on the sphere and the truncated, adaptive noise schedule), and the S-FLM paper also uses the standard DiT, not the S-arch, on Sudoku. On GSM8K, it is the one of the official TinyGSM script (the S-arch), run without the run-time patch that we use on Sudoku (Table 3); with the official data loader, the adaptive schedule is first refitted at update 37,251.

On GSM8K, the compared methods pad the sequences to 512 tokens and compute the loss on the padding, as in S-FLM. On GSM8K, a generated program is correct if its return value matches the answer exactly for integers and within 10^{-3} otherwise, with a time limit of 5 s. The feed-forward width of f_θ (2,136) was chosen so that the total number of parameters of ReLaDrift, including the frozen encoder and the decoder, lies within the range of the compared methods (160.5–168.0M). On GSM8K, the Transformer of ReLaDrift uses RMSNorm and SwiGLU, and its AdamW uses $\beta = (0.9, 0.95)$ and weight decay 0.01, as in the generator of the Drifting Model (Deng et al., 2026); unlike that generator, it has no QK-Norm and no adaLN (the round embedding is added to the input), and the normalization gains and biases are excluded from weight decay. The compared methods instead use LayerNorm (normalized activations in the S-arch of S-FLM) with GELU, $\beta = (0.9, 0.999)$, and no weight decay; all backbones, including ours, use RoPE. On GSM8K, ReLaDrift uses no dropout and a constant EMA decay of 0.9999, whereas the compared methods use dropout 0.1 and the warm-up-corrected EMA decay $\min(0.9999, (u+1)/(u+10))$ at update u of the official implementation. The corrected decay reaches 0.9999 after about 90,000 updates, so at 240,000 updates the weights from before that point contribute a fraction $0.9999^{150{,}000} \approx 3 \times 10^{-7}$ to either EMA, and the two averages are practically the same at the final checkpoint; at earlier checkpoints they differ (Table 9).

Sudoku benchmark. The 2,000 validation puzzles of the benchmark of S-FLM are used only for evaluation; we verified that each of them has a unique solution and that they share no solution grid with the training puzzles. We use the hard level because the other levels are nearly saturated (Section 5.1); the Sudoku-Extreme benchmark (Wang et al., 2025a) is also nearly saturated (99.5% for FRM; Helbling et al., 2026).

Accuracy of our runs of the compared methods. On Sudoku, our runs of S-FLM and MDLM are within 2 points of the published accuracies and our Duo runs are 4.5 points above them, whereas our FMLM+ runs are about 4 points below the published results: at NFE=128, our S-FLM, Duo, and MDLM reach 43.0, 62.9, and 32.2%, compared with 45.0, 58.4, and 30.2% reported with 180 steps by Deschenaux & Gulcehre (2026, Table 1). Our FMLM+ reaches 67.0 and 77.5% at NFE=4 and 16, compared with 71.2 and 81.4% reported by Agarwal et al. (2026, Appendix D, Table 13). Among the compared parallel generators, FMLM+ is the strongest at every NFE from 4 to 128, FRM comes close at NFE $\geq$ 64 with the SDE sampler (77.0% and 78.0% at NFE=64 and 128), Duo reaches 61.9% at NFE=16, and S-FLM and MDLM reach at most 43.0%. The autoregressive model reaches 0.5% with greedy decoding and with sampling (Table 1), in agreement with the 1.0% (greedy) and 0.6% (sampling) reported on the hard level by Deschenaux & Gulcehre (2026, Table 1). On GSM8K, at NFE=1,024, our S-FLM, Duo, and MDLM runs are comparable to the values reported at $T = 0.1$ with the original TinyGSM solutions (about 18, 36, and 33%; Deschenaux & Gulcehre, 2026, Section 4.2 and Figure 3).

A.13 Training and Generation Cost

Training. ReLaDrift runs several forward passes of f_θ per update, whereas AR, S-FLM, Duo, and MDLM run one pass and FMLM+ about two (self-distillation); FRM propagates four states per puzzle and, in Stage B, additionally runs a rollout without gradient for fixed-point forcing. With $R \sim \mathrm{Uniform}\{1, \ldots, 5\}$, the expected depth is $\bar{R} = 3$. The source formulation propagates $n = 2$ independent trajectories, so it runs $2\bar{R} = 6$ forward passes per update, of which only the two terminal passes carry gradients. The additive formulation shares one intermediate trajectory and branches into two candidates at the terminal round, so it runs $\bar{R} - 1 + 2 = 4$ forward passes, again with two gradient passes. The conditional formulation runs the same passes as the additive one, but on Sudoku its attention covers 269 instead of 180 positions, because the 89 draft tokens are appended. Counting a backward pass as twice a forward pass, one update of the source formulation costs about $(4 + 2 \times 3)/3 \approx 3.3$ times an update of a one-pass method, and the additive formulation about $(2 + 2 \times 3)/3 \approx 2.7$ times; the intermediate passes run without gradient and store no activations. Measured on one H200 for Sudoku (batch 256 puzzles, $n = 2$), one update takes 0.22 s for Source, so 20,000 updates take about 1.2 h; Additive takes 0.17 s per update (0.9 h), Conditional 0.26 s (1.4 h), and the single-round model 0.12 s (0.7 h). With $n = 4$, Source takes 0.42 s per update (2.3 h). On GSM8K, the verifier additionally decodes the two terminal samples with the decoder, executes the programs (time limit 15 ms each), and encodes the new positives of the queue with the CodeT5 encoder; these steps run without gradient. On 8 H200 GPUs, one update including these steps takes 0.70, 0.75, and 0.80 s for $K = 4$, 8, and 12, so 240,000 updates take about 47, 50, and 53 h. The decoder (Appendix A.9) is trained once, before ReLaDrift, for 20,000 updates on one GPU in less than one hour, which is less than 0.3% of the GPU time of training ReLaDrift with $K = 12$ (8 GPUs for about 53 h).

Generation. Generation costs NFE forward passes of f_θ and at most one pass each of the encoder and the decoder, which are cheaper than a pass of f_θ and do not depend on NFE. On GSM8K, the frozen CodeT5 encoder (35.3M parameters) encodes the prompt once per problem, which can be cached, and the decoder (42.2M parameters) decodes the final draft; both are smaller than the 89.8M-parameter f_θ. On Sudoku, the codes are fixed and decoding is an $\arg\max$ over 12 codes, so no extra pass is needed. On one H200, evaluating all 2,000 Sudoku puzzles at every NFE up to 16 (one shared 16-round trajectory, 16 passes per puzzle) takes about 3 s.

A.14 Additional Sudoku Results

Table 5 lists all eleven ReLaDrift conditions of the unified Sudoku Hard study (three seeds each, 20,000 updates, EMA, clamped round embedding) at every evaluated NFE, including the single-round model ($R_{\max} = 1$), the source formulation with the verifier, and the training depths $R_{\max} \in \{3, 8, 16\}$. On Sudoku, the verifier can only remove the repulsion from verified samples, because the answer is unique and the queue never grows; with $n = 2$ it makes little difference: it is 1.1–1.7 points more accurate at NFE=2–5 and within 0.3 points at NFE=1 and NFE $\geq$ 8 (92.0% versus 92.1% at NFE=16). The training depths are discussed in Section 5.1; this table adds NFE=2.

Table 3: Sudoku settings of ReLaDrift and the compared methods. AR, Duo, and MDLM are trained with the official implementation of S-FLM and its settings (the same 8-layer, width-512 DiT (Peebles & Xie, 2023), 28.6M parameters; the AR model is causal and has no adaLN, 25.2M). Duo and MDLM use ancestral sampling at temperature 1, and AR is evaluated with greedy decoding and with sampling at temperature 1. S-FLM uses its truncated, adaptive noise schedule; all methods drop the last, incomplete batch of each epoch (128 of the 48,000 puzzles), whereas the official loaders keep it; with this batching, the official code never refits the adaptive schedule, so we added a patch at run time, without modifying the repository, so that the schedule is refitted (first refit at update 1,051). All methods use dropout 0.1 and bf16 mixed precision.

Item	ReLaDrift (ours)	S-FLM	FMLM+	FRM (our implementation)
Sequence, vocabulary	180 tokens, vocabulary 12, fixed prefix 91	same	same	same
State	continuous vector (12-dim one-hot code)	learned embedding on the sphere	one-hot + Gaussian noise	one-hot + carry
Encoder / decoder	one-hot code / inner product with the codes + argmax	learned embedding on the sphere / $\arg\max$ of the output layer	one-hot / argmax	one-hot / argmax
Transformer	9 layers, width 512, 8 heads, FFN 2,048, RoPE, dropout 0.1, round embedding $\phi(r)$ added to the input	DiT on the sphere, 8 layers, width 512, 8 heads	DiT, 8 layers, width 512, 8 heads, two-time adaLN	DiT, 8 layers, width 512, 8 heads, adaLN
Parameters	28.39M	28.56M	28.61M	28.57M
Optimizer	AdamW 3e-4 (warmup 2,500, then constant), $\beta = (0.9, 0.999)$, weight decay 0, clip 1	same (official)	same (official)	same; optimizer and EMA restarted at Stage B
EMA	decay $\min(0.9999, (u+1)/(u+10))$	same	same	same
Training budget	20,000 updates, batch 256	20,000 updates, batch 256	20,000 updates, batch 256	Stage A 10,000 + Stage B 10,000, 256 puzzles $\times$ 4 states
Forward passes in training	gradient only at the terminal round; $n = 2$ independent trajectories	1	2 (self-distillation)	FPF (rollout of depth 16)
Generation	recurrent, NFE 1–128, $\phi(\min(r, 4))$	ODE with the exact velocity, $\arg\max$ at the last step, NFE 1–128	Posterior Refinement, threshold 0.999, 1–128 rounds	Euler / SDE / held-time, NFE 1–128
Scoring	exact match on the 89 answer tokens, no overwriting	same	same	same

Table 4: GSM8K settings. All methods use the same 1,588,781 problems, batch size 512, learning rate 3e-4 after 2,500 warm-up updates (then constant), and 240,000 updates (the S-FLM paper trains for 250,000). The EMA decay is 0.9999 for ReLaDrift and $\min(0.9999, (u+1)/(u+10))$ for the compared methods (official implementation). Duo (167.9M) and MDLM (167.9M) are trained with the same code base and settings as S-FLM; S-FLM, Duo, and MDLM sample with 1,024 steps in the S-FLM paper (Deschenaux & Gulcehre, 2026). All methods use bf16 mixed precision. Only ReLaDrift uses a verifier during training.

Item	ReLaDrift (ours)	AR	S-FLM	FMLM+
Positives	first K of the 12 solutions (fixed) + up to 4 verified samples (positive queue)	12 (problem, solution) pairs per problem, shuffled	same	same
Tokenizer, length	CodeT5 (vocabulary 32,100), 576	SmolLM-135M (vocabulary 49,153), 512	same	same
Latent	CodeT5 contextualized embedding, 512 dim (frozen)	discrete tokens	learned embedding on the sphere	one-hot + Gaussian noise
Transformer	12 layers, width 768, 12 heads, FFN 2,136, RoPE, round embedding $\phi(r)$	DiT, 12 layers, width 768, 12 heads (causal)	S-arch, 12 layers, width 768, 12 heads	DiT, 12 layers, width 768, 12 heads, two-time adaLN
Parameters	core 89.8M (+ frozen encoder 35.3M, decoder 42.2M = 167.3M)	160.5M	163.0M	168.0M
Optimizer	AdamW $\beta = (0.9, 0.95)$, weight decay 0.01 (weight matrices), clip 1	AdamW $\beta = (0.9, 0.999)$, weight decay 0 (= Adam), clip 1 (official)	same	same
Dropout	0	0.1	0.1	0.1
Objective	Source, $2V^+ - V^-$, temperatures 0.02/0.05/0.2, $\tau^+ = 0.01\times$median, verifier and positive queue	next-token cross-entropy	official	official
Verifier in training	yes (execution of the 2 terminal samples)	no	no	no
Forward passes in training	2 trajectories of $R \sim U\{1..5\}$ rounds, stop-gradient on intermediate rounds; decoding and execution of the 2 terminal samples	1	1	2 (self-distillation)
Generation (official setting)	recurrent, NFE 1–1,024, one trajectory	sequential sampling (temperature 1, top-p 1; we report $T = 0.1$), up to 512 tokens including the prompt	1,024 ODE steps, exact velocity	Posterior Refinement, 32 rounds (NFE 32), threshold 0.999
Decoding	42.2M decoder, argmax per position	–	argmax	argmax

Table 5: Sudoku Hard accuracy (%) of all ReLaDrift conditions at 20,000 updates (2,000 puzzles, one candidate, EMA, mean ± standard deviation over three seeds, round embedding clamped to $\phi(\min(r, R_{\max} - 1))$). Unless noted: source formulation, $2V^+ - V^-$, $n = 2$, $R_{\max} = 5$, no verifier. The single-round model is evaluated at NFE=1 only.

Condition	NFE=1	2	4	5	8	16	32	64	128
Source (main)	7.3±1.0	54.7±0.3	78.9±0.6	82.5±0.8	88.4±0.4	92.1±0.4	93.1±0.6	93.5±0.7	93.9±0.7
Additive	7.1±2.2	54.8±3.9	77.6±2.6	81.5±3.1	86.2±3.6	89.5±3.1	90.5±3.0	91.0±2.8	91.6±2.5
Conditional	7.4±0.9	54.3±3.3	77.6±1.7	81.1±1.1	86.1±1.2	89.6±1.8	90.7±1.7	91.2±2.0	91.6±2.3
Single round ($R_{\max} = 1$)	5.1±0.8	–	–	–	–	–	–	–	–
Source, w/ verifier	7.0±1.0	56.4±0.6	80.0±1.0	83.7±0.7	88.4±0.7	92.0±0.6	93.0±0.6	93.4±0.8	93.6±0.8
Source, $V^+ - V^-$	0.0±0.0	0.0±0.0	0.0±0.0	0.0±0.0	0.0±0.0	0.0±0.0	0.0±0.0	0.0±0.0	0.0±0.0
Source, $V^+ - V^-$, $n = 4$	6.5±1.7	50.7±4.2	76.0±2.6	80.7±2.2	88.0±1.2	93.5±0.6	95.9±0.4	97.1±0.3	98.0±0.2
Source, $n = 4$	7.6±2.2	55.1±3.0	78.1±1.4	81.7±1.3	87.4±1.5	91.1±1.2	92.1±1.1	92.6±1.2	92.7±1.2
Source, $R_{\max} = 3$	5.7±2.7	45.5±5.8	66.2±3.4	68.5±2.9	71.4±2.1	72.6±2.0	72.9±1.9	73.1±1.9	73.1±2.0
Source, $R_{\max} = 8$	6.4±2.0	59.1±4.3	83.4±1.7	86.9±1.7	91.2±1.7	94.9±1.3	96.0±1.0	96.6±1.0	96.8±0.9
Source, $R_{\max} = 16$	4.2±2.1	56.3±6.1	85.2±2.9	88.8±2.1	93.0±1.7	96.1±0.8	97.2±0.5	97.4±0.4	97.5±0.4

A.15 Results without the Clamp of the Round Embedding

The model uses the round embedding clamped to $\phi(\min(r, R_{\max} - 1))$, so that it never receives a round embedding that it did not see in training (Section 4.4). This appendix reports, on the same checkpoints, the accuracy when $\phi(r)$ is instead extrapolated to the rounds $r \geq R_{\max}$; for NFE $\leq R_{\max}$ the two are identical (Tables 6 and 7). The clamp only keeps the round embedding within the values seen in training; it does not make the drafts after more than $R_{\max}$ rounds follow the distribution of the drafts seen in training.

On Sudoku, some runs collapse without the clamp. For the source formulation, one seed falls to 3.0% at NFE=8 and to at most 0.1% at NFE $\geq$ 16; of the other two, one falls to 32.5% at NFE=8 and the other to 59.3% at NFE=16, whereas with the clamp all three seeds stay at 88–93% at NFE=8 and 16. The additive and conditional formulations also have seeds that fall at NFE=8 or 16. With $R_{\max} = 3$, all three seeds fall to 0.0% at every NFE ≥ 8, and with $n = 4$, every seed falls below 10% at NFE ≥ 16. The same runs collapse on 2,000 additional puzzles generated with a different seed. With $R_{\max} = 8$ and $R_{\max} = 16$, the extrapolation does not collapse. With $R_{\max} = 8$, it is up to 1.8 points below the clamp in the mean over seeds (at most 3.2 points for one seed, at NFE=16); with $R_{\max} = 16$, where the two differ only at NFE ≥ 17, it is 0.1–0.5 points above the clamp at NFE=32–128 (97.3, 97.8, and 98.0% versus 97.2, 97.4, and 97.5%).

On GSM8K, the accuracy without the clamp is higher for every K at NFE=8, 16, and 32, by 0.2–2.8 points (0.2–0.7 points for $K = 12$; Table 7), and the difference generally grows with NFE. Among the other runs with $K = 8$ in Appendix A.16 (without the verifier and the queue, $V^+ - V^-$, and the two embedding tables), evaluated in the same way, the run with a random embedding table collapses without the clamp (2.6% and 1.8% versus 10.6% and 11.3% at NFE=8 and 16), whereas the others change by at most 1 point at NFE=8 and 16. The GSM8K numbers without the clamp at every checkpoint are listed in Appendix A.17. Overall, the clamp trades a small loss on GSM8K for stability on Sudoku.

Table 6: Sudoku Hard accuracy (%) at 20,000 updates with the round embedding extrapolated or clamped for NFE ≥ 8 (same checkpoints; mean ± standard deviation over three seeds).

Condition	Round embedding	NFE=8	16	32	64
Source	extrapolated	39.4±40.2	46.4±41.4	54.4±47.2	57.1±49.5
	clamped	88.4±0.4	92.1±0.4	93.1±0.6	93.5±0.7
Additive	extrapolated	38.6±40.3	33.5±19.0	49.0±26.8	55.6±29.6
	clamped	86.2±3.6	89.5±3.1	90.5±3.0	91.0±2.8
Conditional	extrapolated	74.7±18.5	87.6±1.5	89.5±2.5	90.1±2.7
	clamped	86.1±1.2	89.6±1.8	90.7±1.7	91.2±2.0
Source, $n = 4$	extrapolated	11.8±20.4	1.2±0.9	1.8±1.7	2.6±2.6
	clamped	87.4±1.5	91.1±1.2	92.1±1.1	92.6±1.2
Source, $V^+ - V^-$, $n = 4$	extrapolated	0.0±0.0	1.9±2.2	2.9±2.3	4.0±3.6
	clamped	88.0±1.2	93.5±0.6	95.9±0.4	97.1±0.3
Source, $R_{\max} = 8$	extrapolated	91.2±1.7	93.1±1.9	95.5±1.0	96.3±0.9
	clamped	91.2±1.7	94.9±1.3	96.0±1.0	96.6±1.0

Table 7: GSM8K test accuracy (%) of ReLaDrift at 240,000 updates with the round embedding extrapolated or clamped for NFE ≥ 8 (EMA, single).

	Round embedding	NFE=8	16	32
$K = 1$	extrapolated	14.3	17.3	18.4
	clamped	12.9	15.3	15.6
$K = 4$	extrapolated	27.8	29.9	30.6
	clamped	27.4	28.6	29.1
$K = 8$	extrapolated	30.9	32.6	33.6
	clamped	29.9	31.2	31.6
$K = 12$	extrapolated	30.8	32.0	32.8
	clamped	30.6	31.8	32.1

A.16 GSM8K Ablations and Training Progress

Table 8 ablates ReLaDrift ($K = 8$) at 240,000 updates with the clamped round embedding: the verifier and the positive queue, the attraction coefficient, the embedding space, the formulation, and the number of given positives K. Without the verifier and the positive queue, the accuracy at NFE=16 drops from 31.2% to 20.6%; with $V^+ - V^-$, training collapses to 0.0% at every checkpoint and every NFE. Replacing the contextualized CodeT5 embeddings with a context-free embedding table (the CodeT5 input embedding of each token, or a random Gaussian table) lowers the accuracy to 18.1% and 11.3% at NFE=16, but the accuracy still increases with NFE. The additive formulation, trained in the same way from scratch, is 0.8–3.0 points below the source formulation at every NFE (28.8% versus 31.2% at NFE=16), as on Sudoku (Section 5.1), and its accuracy also increases monotonically with NFE up to NFE=32.

The compared methods were also evaluated at sampling temperature $T = 1$ (single, 240k; NFE=32 for S-FLM, Duo, and MDLM, and sequential generation for AR): AR 64.6, S-FLM 13.2, Duo 15.5, and MDLM 7.3, versus 71.7, 19.1, 29.8, and 20.0 at $T = 0.1$; the main table therefore reports $T = 0.1$. Table 9 shows the accuracy over training with the generation settings of the compared methods (AR sequential and S-FLM with 32 steps, both at $T = 1$; FMLM+ with 32 rounds and argmax) and ReLaDrift at NFE=16 without the clamp.

Table 8: GSM8K test ablations at 240,000 updates (%; EMA, single; round embedding clamped to $\phi(\min(r, 4))$). Unless noted, the source formulation with $K = 8$, $2V^+ - V^-$, the verifier, and the positive queue. The single-round model ($R_{\max} = 1$, the same training otherwise) is evaluated at NFE=1 only. The columns to the right of the vertical line (NFE ≥ 64) lie outside the few-step regime.

Setting	NFE=1	4	5	8	16	32	64	128	256	512	1024
$2V^+ - V^-$, verifier, $K = 8$ (default)	8.5	27.6	29.2	29.9	31.2	31.6	31.8	31.8	31.8	31.8	31.8
without verifier and queue	2.4	16.5	17.6	19.3	20.6	21.0	21.0	21.0	21.1	21.1	20.9
$V^+ - V^-$, verifier	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
CodeT5 embedding table	3.4	14.1	15.1	17.1	18.1	18.7	18.7	18.8	18.8	18.7	18.7
random embedding table	0.5	8.2	8.7	10.6	11.3	11.8	11.9	12.0	12.0	12.0	12.0
additive formulation	7.7	24.9	26.2	28.3	28.8	29.9	30.3	30.1	30.2	30.3	30.4
single round ($R_{\max} = 1$)	4.1	–	–	–	–	–	–	–	–	–	–
$K = 1$	0.5	10.5	11.5	12.9	15.3	15.6	15.8	15.8	15.8	15.6	15.7
$K = 4$	8.0	26.0	26.5	27.4	28.6	29.1	29.5	29.5	29.3	29.5	29.5
$K = 12$	10.1	29.1	30.1	30.6	31.8	32.1	32.2	32.2	32.3	32.3	32.2

A.17 All GSM8K Results of ReLaDrift

Table 10 lists the GSM8K test accuracy of ReLaDrift at every evaluated checkpoint and NFE without the clamp, that is, with the round embedding $\phi(r)$ extrapolated for NFE ≥ 6 (Appendix A.15); for NFE ≤ 5, the clamp does not change the round embedding.

Table 9: GSM8K test accuracy (%) over training (EMA, single). AR and S-FLM sample with temperature $T = 1$ (AR sequential, S-FLM 32 steps), and FMLM+ uses 32 rounds with argmax; ReLaDrift is at NFE=16 without the clamp (extrapolated $\phi(r)$), because the clamped model was not evaluated at every checkpoint. "w/o ver." is $2V^+ - V^-$ without the verifier and the queue, and "$\lambda = 1$" is $V^+ - V^-$ with the verifier (both $K = 8$).

				ReLaDrift, NFE=16				
Updates	AR	S-FLM	FMLM+	$K = 4$	$K = 8$	$K = 12$	w/o ver.	$\lambda = 1$
20k	48.1	0.7	3.1	0.5	0.8	1.5	0.8	0.0
40k	56.7	2.4	6.7	9.5	15.1	17.1	9.7	0.0
60k	59.8	4.7	9.0	15.6	20.8	22.1	14.3	0.0
80k	62.5	5.9	11.2	19.2	23.4	25.4	16.8	0.0
100k	62.5	8.0	16.9	22.1	26.5	26.1	17.4	0.0
120k	63.5	8.9	18.0	24.4	27.7	27.1	19.0	0.0
140k	62.5	10.0	18.3	26.5	29.6	29.0	20.5	0.0
160k	63.8	12.6	20.8	27.1	30.0	30.2	20.5	0.0
180k	65.2	11.4	19.5	28.1	30.0	30.8	21.0	0.0
200k	66.0	13.2	21.3	28.5	31.3	32.2	20.3	0.0
220k	65.1	13.5	23.7	29.3	32.3	32.0	22.7	0.0
240k	64.6	13.2	24.7	29.9	32.6	32.0	21.6	0.0

Table 10: GSM8K test accuracy (%) of ReLaDrift (Source, $2V^+ - V^-$, verifier and positive queue) at every checkpoint and NFE without the clamp (round embedding extrapolated for NFE ≥ 6; EMA, single).

K	Updates	NFE=1	2	4	5	8	12	16
4	20k	0.0	0.3	0.3	0.3	0.5	0.5	0.5
	40k	0.5	2.4	6.4	7.3	8.6	8.9	9.5
	60k	0.7	6.0	11.8	12.5	14.4	15.2	15.6
	80k	1.4	7.9	14.7	16.3	17.7	18.7	19.2
	100k	2.4	12.2	17.4	18.2	19.6	21.5	22.1
	120k	3.6	13.0	19.5	20.2	22.1	23.9	24.4
	140k	4.4	15.0	21.0	22.0	24.0	26.2	26.5
	160k	5.7	16.1	22.1	23.4	25.5	26.2	27.1
	180k	6.3	16.8	23.1	24.1	26.2	27.7	28.1
	200k	6.7	17.5	23.7	24.1	26.2	27.9	28.5
	220k	7.9	18.0	24.9	25.9	27.5	28.1	29.3
	240k	8.0	18.3	26.0	26.5	27.8	28.7	29.9
8	20k	0.0	0.4	0.5	0.8	0.8	0.9	0.8
	40k	0.5	5.5	10.6	11.5	13.3	14.4	15.1
	60k	1.9	9.9	16.8	17.6	19.6	20.6	20.8
	80k	3.3	12.5	19.6	20.2	22.2	22.4	23.4
	100k	4.2	14.7	21.7	22.7	24.0	25.3	26.5
	120k	5.4	16.1	22.8	23.9	25.7	27.2	27.7
	140k	5.6	17.6	23.6	24.9	27.0	28.8	29.6
	160k	6.6	18.7	25.2	25.9	28.0	29.4	30.0
	180k	7.4	20.3	25.5	26.3	28.4	29.3	30.0
	200k	8.7	20.0	26.5	27.4	29.5	30.3	31.3
	220k	8.1	20.8	26.5	27.3	29.6	31.6	32.3
	240k	8.5	21.5	27.6	29.2	30.9	32.1	32.6
12	20k	0.2	0.4	0.7	0.9	1.3	1.3	1.5
	40k	0.8	5.5	11.9	12.9	15.2	16.9	17.1
	60k	2.4	11.1	17.6	18.7	20.3	21.7	22.1
	80k	3.9	14.7	20.5	21.6	23.5	24.9	25.4
	100k	4.9	16.8	22.5	23.3	24.6	25.8	26.1
	120k	6.2	17.8	23.5	24.9	25.9	26.4	27.1
	140k	7.4	18.7	24.0	24.9	27.0	28.4	29.0
	160k	8.6	21.1	26.3	27.2	28.8	29.6	30.2
	180k	9.0	21.5	27.4	27.8	29.6	30.6	30.8
	200k	8.6	22.4	28.4	29.0	30.8	31.2	32.2
	220k	9.7	21.2	28.1	28.7	30.2	31.4	32.0
	240k	10.1	22.4	29.1	30.1	30.8	31.6	32.0

A.18 Generated Samples

The following programs were generated by ReLaDrift ($K = 12$) at 80,000 updates on GSM8K test problems. The docstring, which repeats the problem statement inside the program (Appendix A.10), is generated like the rest of the program.

```
# Problem 1 (correct at NFE=1)
def simple_math_problem():
    """
    A robe takes 2 bolts of blue fiber and half that much white fiber
    it. How many bolts in total does it take?
    """
    blue_bolts = 2
    white_bolts = blue_bolts / 2
    total_bolts = blue_bolts + white_bolts
    return total_bolts

# Problem 2, NFE=1 (wrong)
def simple_math_problem():
    """
    James decides to run 3 sprints 3 times a week. By He runs 60 meters
    each sprint. How How many total meters does he run a week?
    """
    num_ssns = 3
    speaks_per_week = 3
    distance_per_sprint = 60

    total_ss = num_ssns * sabs_per_week
    total_distance = total_ss * distance_per_sprint

    return total_distance

# Problem 2, NFE=5 (correct; identical at NFE=16)
def simple_math_problem():
    """
    James decides to run 3 sprints 3 times a week. By He runs 60 meters
    each sprint. By How many total meters does he run a week?
    """
    num_sprints = 3
    sprint_per_week = 3
    distance_per_sprint = 60

    total_sprint = num_sprints * sprint_per_week
    total_distance = total_sprint * distance_per_sprint

    return total_distance
```

B Additional Related Work

ReLaDrift differs from prior work in two ways: its generator is a one-step drifting map rather than a diffusion or flow model, and it spends additional test-time computation by applying this map recurrently to its own drafts.

Diffusion and flow language models. Many diffusion and flow language models update token states in parallel, although some latent-space approaches use autoregressive decoders (Shen et al., 2026). Reducing the number of sampling steps can lower generation quality (Deschenaux & Gulcehre, 2025; Zheng et al., 2025; Agarwal et al., 2026). Discrete diffusion models corrupt tokens by masking or by uniform noise and learn to reverse the corruption (Austin et al., 2021; Lou et al., 2024; Sahoo et al., 2024; Shi et al., 2024; Sahoo et al., 2025), and they have been scaled to large language models (Nie et al., 2025; Ye et al., 2025). Continuous diffusion and flow models (Li et al., 2022; Dieleman et al., 2022; Gong et al., 2023; Gulrajani & Hashimoto, 2023; Lee et al., 2026; Chen et al., 2026; Deschenaux & Gulcehre, 2026; Pynadath et al., 2025) generate in an embedding

space, on one-hot vectors, or on the hypersphere by integrating the ODE or SDE of a diffusion or flow model (Lipman et al., 2023; Liu et al., 2023b), and several of them generate in the space of pretrained embeddings or of a frozen encoder (Strudel et al., 2022; Lovelace et al., 2023; Shabalin et al., 2025; Shen et al., 2026; Hu et al., 2026). ReLaDrift also generates in the latent space of a frozen encoder, but it produces a complete draft in one step and uses additional steps only to correct it.

Few-step and one-step generation. Few-step and one-step generators reduce sampling steps through distillation or directly learned maps. Self-distillation compresses the sampling process of discrete diffusion language models (Deschenaux & Gulcehre, 2025), while continuous diffusion and flow methods learn one-step or few-step maps (Lee et al., 2026; Song et al., 2023; Frans et al., 2025; Geng et al., 2025; Boffi et al., 2025). The Drifting Model (Deng et al., 2026) learns a one-step generator without an ODE. Ma et al. (2026) connect drifting with flow matching so that the generation can be refined with more inference steps, and Oba et al. (2026) use a drifting objective in a frozen semantic space to refine discrete diffusion language models. ReLaDrift instead uses the drifting map itself as the language generator and obtains additional steps by applying the same map to its own output.

Iterative refinement and recurrent computation. Non-autoregressive generation has long refined a draft over several passes (Lee et al., 2018; Ghazvininejad et al., 2019; Gu et al., 2019; Savinov et al., 2022), and diffusion and flow language models revise their outputs by self-conditioning, remasking, or regenerating tokens with low confidence (Chen et al., 2023; Wang et al., 2025b; Agarwal et al., 2026). Yoo et al. (2026) interpret self-conditioning as fixed-point iteration and distill both the flow process and the fixed-point iterations; ReLaDrift directly trains recurrent applications of a drifting map. Weight-tied recurrence is also used to spend more computation on an input, sometimes for more iterations than in training (Dehghani et al., 2019; Bai et al., 2019; Bansal et al., 2022; Geiping et al., 2025; Wang et al., 2025a; Jolicoeur-Martineau, 2025). Equilibrium Reasoners (Huang et al., 2026) learn iterative latent dynamics toward task-aligned attractors, use truncated gradients, and scale inference through additional iterations and stochastic trajectories. ReLaDrift instead trains each round with a drifting objective over generated and target latent distributions. Closest to our work, Helbling et al. (2026) turn a discrete flow into a recurrent reasoner by self-conditioning and train it on its own rollouts without backpropagating through them. ReLaDrift is likewise trained on its own drafts without backpropagation through the earlier rounds, but each round is a one-step drifting map trained with the drifting loss rather than a flow trained with a cross-entropy.