RECURSIVE SCALING IN MASKED DIFFUSION MODELS

Anonymous authors
Paper under double-blind review

ABSTRACT

Masked diffusion models (MDMs) generate sequences by iteratively refining a partially masked state and committing tokens in parallel. We introduce recursion in MDMs and propose new Recursive Masked Diffusion Models (R-MDMs), which apply a shared denoising transformer L times within each denoising step, adding recursive depth as an additional compute axis without increasing parameter count. Across structured generation tasks, recursive depth improves quality at fixed parameter budget, matches substantially larger non-recursive models at matched FLOPs, and can reduce the number of denoising steps needed to reach a target quality. We interpret these gains with a dependence–fidelity decomposition of parallel decoding error: recursion refines model marginals at a fixed masked state, whereas denoising steps change that state by committing tokens. Building on this analysis, we propose to treat decoding as a two-axis decision (how many loops to run and which tokens to commit) and show that entropy-guided adaptive rules improve the quality–compute frontier over fixed schedules, transferring across various tasks on Sudoku, Countdown, RNA, and executable math generation. Together, these results establish recursive depth as a practical, complementary test-time scaling mechanism for MDMs.

1 INTRODUCTION

Diffusion models have emerged as a compelling paradigm for generative modeling of continuous data (Sohl-Dickstein et al., 2015; Ho et al., 2020; Song et al., 2021), while discrete diffusion models have more recently been developed for sequence generation, including masked diffusion (Austin et al., 2021; Sahoo et al., 2024; Nie et al., 2025). Among them, masked diffusion models (MDMs) generate sequences by repeatedly denoising a partially masked state, attending bidirectionally to the whole sequence, predicting every masked position at once, and committing some subset of them. Generation is therefore iterative, and the number of denoising steps T is a native test-time compute scaling axis: more steps mean fewer tokens are committed at a time, so fewer dependencies between them go unresolved and generation quality usually improves, but at proportionally higher cost.

Another scaling axis is model size, which recent MDM research has mostly pursued by following AR practice: adding more width, attention heads, and layers, so that compute grows with parameter count (Kaplan et al., 2020). Following this recipe, MDMs have grown from small models (Austin et al., 2021; Sahoo et al., 2024) to billions of parameters (Nie et al., 2025; Ye et al., 2025b; DiffusionGemma Team, 2026), reaching generation quality competitive with AR models.

Yet, model size and denoising steps are not the only axes of capability. A third, *recursive depth*, has recently been developed in AR models: applying the same transformation repeatedly to add computation without adding parameters (Dehghani et al., 2019; Giannou et al., 2023). In particular, a K-layer transformer looped L times can match a KL-layer model while using $L\times$ fewer parameters (Saunshi et al., 2025). We now bring recursion to MDMs, where it plays a different role. In AR models, looping deepens the computation behind a single next-token prediction. In contrast, bidirectional attention in MDMs lets each loop jointly refine all masked positions at a fixed partially masked state, while denoising steps commit tokens and advance generation.

This motivates our central question: how does recursive computation interact with the denoising process in MDMs? Although both provide additional computation, they operate at different levels: denoising steps change the partially masked sequence by committing tokens, whereas recursive loops refine predictions while holding the denoising state fixed. By making each parallel denoising

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.

update more accurate, recursion may reduce the number of denoising steps needed to reach a given quality. Thus, we ask whether additional recursive steps can substitute for denoising steps, whether the two axes address different failure modes, and how their computation should be allocated at inference time.

We study this question through *recursive masked diffusion models* (R-MDMs), which apply a denoising block L times within each denoising step, reusing the same weights at every application and emitting a prediction after each. Recursion is therefore a compute axis rather than a size axis: the parameter count does not depend on L. We show that increasing the recursive depth allows R-MDMs to match the performance of larger non-recursive models at equal FLOPs, and that it reduces predictive entropy, within-step conflicts, and early decoding failures, revealing how recursion improves the quality of parallel updates and explaining its stronger benefits at small denoising budgets. Building on this, we treat decoding as a two-axis policy and show that uncertainty-guided allocation between loops and denoising steps improves the quality–compute frontier across Sudoku, Countdown, RNA sequence generation, and executable math generation on TinyGSM.

Contributions. **(i) Model** (Section 3). We propose R-MDMs, a new, simple method for introducing recursive depth into MDMs, and ablate their main design choices; we empirically show that recursive depth improves generation quality at fixed parameter count and matches substantially larger non-recursive models at matched FLOPs. **(ii) Analysis** (Section 4). Through a careful decomposition of the parallel-decoding error we separate the roles of recursive steps and denoising steps (token commitment), identifying their complementary roles, and empirically establish why and when recursive refinement helps MDMs. **(iii) Decoding** (Section 5). We turn this analysis into entropy-guided rules for *when to stop refining* and *which tokens to commit*, and empirically validate their quality–compute benefits across generative tasks on Sudoku, Countdown, RNA infilling, and executable math generation.

2 Related Work

We review the lines of work most directly relevant to our study and defer a broader discussion of the related literature to Appendix B.

Masked diffusion language models. Masked (absorbing-state) discrete diffusion has become a competitive alternative to AR language modeling (Austin et al., 2021; Lou et al., 2024; Sahoo et al., 2024; Nie et al., 2025; DiffusionGemma Team, 2026). Because tokens are denoised in parallel from bidirectional context, the number of denoising steps T trades speed against the dependencies the sampler can resolve before committing. As a result, a large body of work asks *which* masked tokens to reveal and *how many*: MaskGIT (Chang et al., 2022) established confidence-based unmasking, Kim et al. (2025) show that adaptive ordering matters on structured problems, EB-Sampler (Ben-Hamu et al., 2025) uses an entropy bound to determine how many tokens may be safely unmasked per evaluation, and Cai & Li (2026) give iteration-complexity guarantees for entropy-sum unmasking. We take our commit rules from this line rather than proposing new ones.

Recursive and looped computation. Depth-wise weight sharing runs from Universal Transformers and ALBERT (Dehghani et al., 2019; Lan et al., 2020) to looped transformers, where a K-layer block applied L times can approach the effective depth of a KL-layer model with far fewer parameters (Giannou et al., 2023; Yang et al., 2024; Geiping et al., 2025; Yu et al., 2026; Xu & Sato, 2026); Saunshi et al. (2025) further interpret loops as latent reasoning steps. Among these variants we adopt the plainest construction (a single shared block applied repeatedly, with no architectural search) for simplicity of our exploration.

Positioning of this work. We study recursive computation inside each MDM denoising step as a complementary axis of inference-time scaling. Concurrent work introduces repeated computation into masked generation through different training mechanisms: LoopMDM (Lee et al., 2026b) loops a subset of transformer layers, and Fixed-Point Masked Generative Modeling (Miele et al., 2026) embeds a fixed-point solver in the denoiser. Rather than comparing recursive architectures for MDMs, we seek to determine whether recursion provides an effective scaling axis, when it is most beneficial, and how it should be used at inference time. We therefore keep our recursive architecture deliberately simple, enabling us to isolate the effects of recursion and study its interaction with MDM decoding.

3 Recursive Masked Diffusion Models

In this section, we introduce R-MDMs, describe their architecture, optimization procedure, and key design choices, and empirically show that recursive depth provides an effective scaling axis.

3.1 Definition

Preliminaries. For $x_0 \in \mathcal{V}^n$, the absorbing-state forward process masks each token independently, $q(x_t \mid x_0) = \prod_i \mathrm{Cat}(x_t^{(i)}; \alpha_t x_0^{(i)} + (1-\alpha_t)\mathbf{e}_{\texttt{[MASK]}})$ with α_t monotone from $\alpha_0{=}1$ to $\alpha_1{=}0$; the denoiser $p_\theta(x_0 \mid x_t)$ is a bidirectional transformer trained on the masked ELBO (Sahoo et al., 2024), and the sampler unmasks progressively over T steps. We write d for hidden width, K for the layers in the shared block, L for the loop budget, and $(K \otimes L)$ for a K-layer block applied L times, mirroring Saunshi et al. (2025).

Architecture. A standard MDM applies a KL-layer transformer once and decodes from its final hidden state. Our new R-MDM instead applies a K-layer block $f_\theta : \mathbb{R}^{n\times d} \to \mathbb{R}^{n\times d}$ with shared weights L times (Figure 1). Starting from $\mathbf{h}^{(0)} = \mathrm{Norm}(E_\theta(x_t) + P_\theta(x_t))$, the process computes

$$\mathbf{h}^{(\ell)} = f_\theta\big(\mathrm{Norm}(\mathbf{h}^{(\ell-1)} + \mathbf{v}^{(\ell)})\big), \qquad \hat{\mathbf{y}}^{(\ell)} = W_\theta\,\mathrm{Norm}\big(\mathbf{h}^{(\ell)}\big), \qquad \ell = 1, \dots, L, \tag{1}$$

with E_θ the token embedding, P_θ the positional encoding, Norm RMS normalization (Zhang & Sennrich, 2019), and W_θ a shared LM head producing logits after *every* loop.

Since f_θ and W_θ are identical across loops, the model has exactly the number of parameters of a single-pass $(K \otimes 1)$ baseline, and the only information carried between loops is $\mathbf{h}^{(\ell)}$; a $(KL \otimes 1)$ baseline instead applies KL *distinct* layers once, at identical FLOPs per forward pass and $L\times$ the parameters. Three design choices appear at *training* time when adding recursive depth to standard MDMs (Appendices D.2–D.4): **(i)** an added *step embedding* $\mathbf{v}^{(\ell)} = W_2\sigma(W_1 s_\ell + b_1)$ of the normalized loop progress $s_\ell = (\ell-1)/(L-1)$; **(ii)** the supervision scheme over the L logit sets produced; and **(iii)** how many loops L_t to run during training. We ablate these choices in Appendix D and, based on the results, use learned step embeddings, masked cross-entropy averaged across all loops (as in deep supervision (Lee et al., 2015)), and reverse-curriculum annealing or fixed training depth L_t. All other hyperparameters are given in Appendix D.

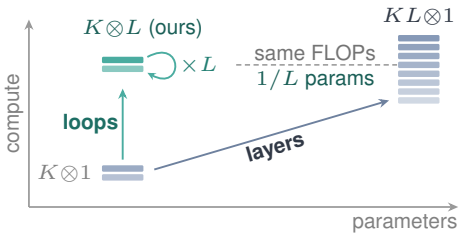


Figure 1: **Recursion is a second scaling axis.** Adding layers moves a model diagonally, buying compute with parameters. Looping a K-layer block L times moves it straight up: it matches the compute of a KL-layer model at the same FLOPs per evaluation, while holding $L\times$ fewer parameters.

> ***Contribution*** *R-MDMs reuse a shared K-layer block across L loops, increasing effective depth without increasing the parameter count. This introduces recursive depth as a distinct scaling axis for* MDMs.

3.2 Recursive depth is a parameter-efficient scaling axis

We evaluate recursive depth as a scaling axis. We focus on two structured generation tasks that allow controlled comparisons across recursion depth and denoising budget: **Sudoku** (1.8M 9×9 boards and 100k synthetic 25×25 boards), evaluated by Valid Puzzle Rate (VPR) (He et al., 2026), and **Countdown**, with 100k examples for each operand count $k \in \{3, 4, 5\}$, evaluated by Reaches Target Rate (RTR) (Yao et al., 2023; Gandhi et al., 2024). Throughout this section, recursion matches the training setup: every denoising step runs $L = L_t = L_s$ loops, where L_s is the inference-time loop depth, T is fixed, and we commit the most confident tokens (alternative decoders are compared in Section 5). Dataset details, metrics, and hyperparameters are given in Appendices C and D.

Iso-parameter. At fixed parameter count, increasing L consistently improves quality, particularly when the denoising budget is small. This comparison adds compute but keeps the number of learned parameters fixed. For example, on Countdown-3 at $T{=}10$, $L{=}3$ reaches 92.4% RTR, compared with 59.4% for the single-pass model. The effect is larger on the more difficult Countdown-5 task: at $T{=}30$, $L{=}10$ reaches 56.4% RTR, compared with 15.6% for the baseline. On Sudoku, the same

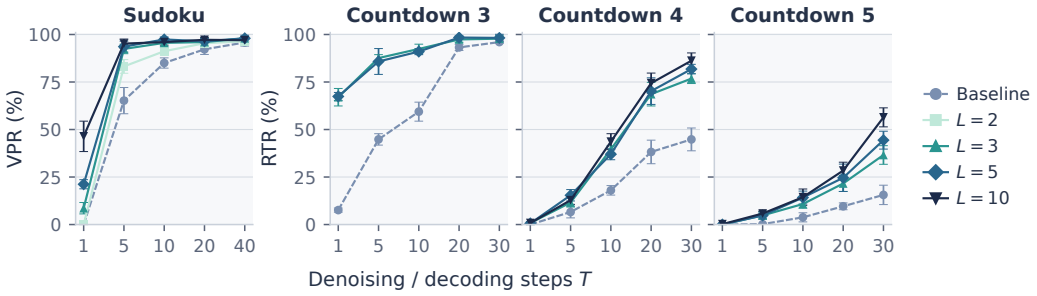


Figure 2: Effect of recursive refinement depth (L) on task performance for Sudoku 9×9 and Countdown across different target lengths (3, 4, and 5 digits). Increasing L consistently improves success rates, with the strongest gains for more difficult problems.

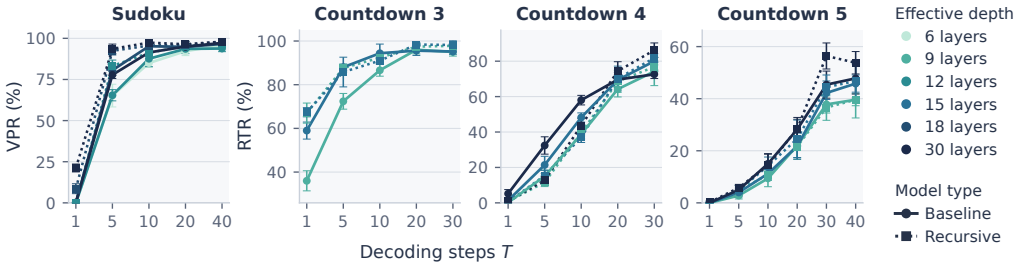


Figure 3: Effect of recursion at matched effective model depth (iso-FLOP) for Sudoku 9×9 and Countdown tasks of different target length (3, 4, and 5 digits). Recursion matches or exceeds non-recursive baselines of equal effective depth while using $L\times$ fewer parameters.

pattern is visible in the full sweep (Appendix G.1): at T=5, VPR increases from 65.2% for the baseline to 93.6% with L=5, while at T=40 it goes from 95.8% to 98.0%. By refining its hidden state before committing tokens, R-MDM can reach higher performance.

Iso-FLOP. To isolate the effect of weight sharing, we compare $(K \otimes L)$ R-MDM with a $(KL \otimes 1)$ baseline matched in effective depth and gradient updates; the recursive model uses $L\times$ fewer parameters (Appendix D; Figure 3). Recursion matches or outperforms the deeper baselines: on Sudoku at T=5, $(6 \otimes 5)$ achieves 93.6% VPR with 10.8M parameters versus 77.8% with 53.1M; on Countdown-4 at T=20, $(3 \otimes 5)$ and $(3 \otimes 10)$ reach 70.2% and 74.4% RTR, versus 69.4% and 69.6%, with 5–10$\times$ fewer parameters. Recursion also benefits beyond depth alone: at T=40, increasing Sudoku depth from 18 to 30 layers raises VPR only 96.6% $\rightarrow$ 97.2%, whereas increasing L from 1 to 5 at depth 6 gives 95.8% $\rightarrow$ 98.0%. This suggests that iterative refinement of the masked state provides a benefit beyond simply increasing depth.

Two further results reinforce this picture. On 25×25 Sudoku, recursion is most valuable with high masking: at 90% masking and T=100, VPR rises from 2.0% to 52.2% with L=3 (Appendix G.4). Training and sampling depths can differ: a model trained with L_t loops can be sampled with $L_s \neq L_t$ (Appendix G.3). Finally, Figure 15 fixes the total number of forward passes, and compares allocating compute to recursion or denoising. Recursion can be more effective and raise the performance ceiling; the two costs are equal but their effects are not. In Section 4 we explore why.

> ***Takeaway*** *These results establish recursion as a parameter-efficient scaling mechanism for* MDM*s, with the largest gains at small denoising budgets and on structured tasks with explicit multi-step dependencies.*

4 Benefits of Recursion at Test Time

The results in Section 3 validate that additional recursive steps can improve the quality of generations in an R-MDM. They have also suggested that while the cost of an additional denoising step and an additional recursive step are the same (effectively implying one more forward pass through the model), their effects can differ. In this Section, our goal is to better understand the interplay between the denoising and recursive steps and in doing so: **(i)** identify which types of errors each of these modes of computation can reduce (Section 4.1), **(ii)** investigate why recursion is most beneficial at small T (Section 4.2).

4.1 A DEPENDENCE–FIDELITY VIEW OF PARALLEL DECODING

A decoding run maintains a partially masked state $s_t = (x_{O_t}, M_t)$ with committed positions O_t and masked positions M_t. At step t it runs $\ell_t \le L$ loops while holding s_t fixed, reads the marginals $q_i^{(\ell)}(v \mid s_t) = \mathrm{softmax}(W_\theta \mathbf{h}_i^{(\ell)}(s_t))_v$ for $i \in M_t$, then chooses a commit set $U_t \subseteq M_t$ and samples those positions *independently*; we count both forms of computation by function evaluations $\mathrm{NFE} = \sum_t \ell_t$. An ideal step would draw x_U from the true conditional joint $p^\star(x_U \mid s) = \pi_{\mathrm{data}}(x_U \mid x_O)$; sampling from a product of the model's own marginals instead makes two separable bets: one about each position alone, one about the positions being unrelated. As shown by Ben-Hamu et al. (2025), the error decomposes along these two axes; in recursive decoding, this decomposition separates what loops and denoising steps can each reduce.

Lemma 4.1 (Dependence–fidelity decomposition)**.** *For any state s, commit set U, and strictly positive marginals q_i (as softmax outputs always are),*

$$D_{\mathrm{KL}}\Big(p^\star(x_U \mid s) \,\Big\|\, \prod_{i\in U} q_i\Big) \;=\; \underbrace{\mathrm{TC}\big(p^\star(x_U \mid s)\big)}_{D_{\mathrm{dep}}(U \mid s)} \;+\; \underbrace{\sum_{i\in U} D_{\mathrm{KL}}\big(p_i^\star \,\|\, q_i\big)}_{D_{\mathrm{fid}}(U \mid s,\, q)}, \tag{2}$$

where $\mathrm{TC}(p) = \sum_{i\in U} H(p_i) - H(p) \ge 0$ is the total correlation of the true joint distribution.

In simpler terms, D_{fid} measures how wrong the model is about each committed position considered alone, and D_{dep} quantifies what it forfeits by treating the positions it commits together as independent. The dependence term is not model-dependent: it is a property of the task and of *which* positions are committed together, and it vanishes whenever those positions are conditionally independent, in particular whenever only a single token is committed in each step (see proofs and details in Appendix A).

This decomposition isolates the role of denoising and recursive steps. **Extra denoising steps decrease D_{dep}.** Splitting one commit U into consecutive commits A and B lowers the dependence cost by exactly $\sum_{i\in B} I(x_i; x_A \mid s)$ (Corollary A.7). Thus, reducing D_{dep} can be achieved by committing fewer or less dependent positions at a time: a decision of the denoising schedule, not the loop. **At a fixed commit set, extra loops can only act on D_{fid}.** A loop leaves s and U untouched and changes only the marginals, so it cannot affect D_{dep}. However, when U itself is prediction-dependent, loops can also change U and hence affect D_{dep}; Section 4.2 fixes the commit order to isolate this effect. Altogether, the decomposition explains the complementary roles of recursive and denoising steps, and why the gains of Section 3.2 depend so strongly on T.

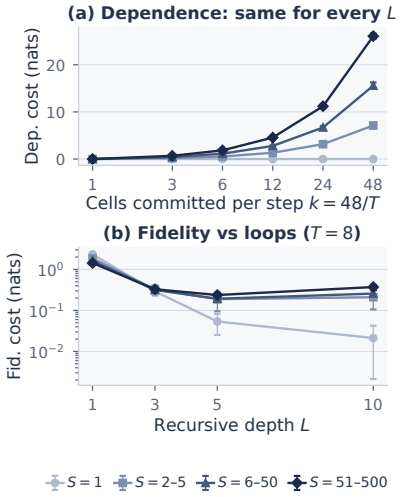


Figure 4: **Each term responds to one lever.** Exact costs along the oracle process. (a) $\sum_t D_{\mathrm{dep}}$ grows with S and commit size, and is independent of L by construction, since the commit order is fixed. (b) $\sum_t D_{\mathrm{fid}}$ decreases with L.

4.2 EMPIRICAL EVALUATION IN CONTROLLED SETTINGS

To empirically evaluate whether loops reduce D_{fid}, and how much error remains once they do, we require access to the true distribution $p^\star$. The Sudoku puzzles of Section 3.2 cannot answer it: each board has a unique solution, so $p^\star$ is a point mass, $D_{\mathrm{dep}} \equiv 0$, and only the fidelity error remains. We therefore construct boards with multiple valid completions, grouped by their number of solutions S (1, 2–5, 6–50, 51–500; 50 boards each). Since all solutions are known, $p^\star$, D_{dep}, and D_{fid} can be computed exactly. To stay within the assumptions of Lemma 4.1, we sample from the model's untruncated marginals (Remark A.2) and fix a random commit order before decoding, making the commit sets independent of L. Runs terminate at the first rule violation, either a *parallel* clash within a commit step or an *old-cell* clash with a previously committed cell (Appendix E).

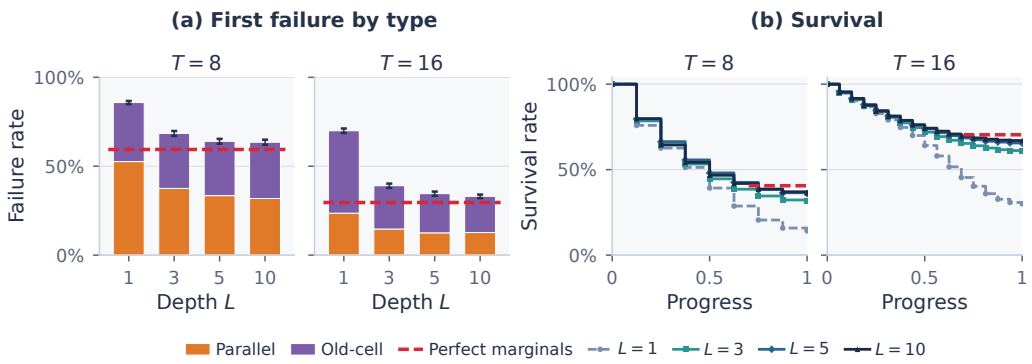


Figure 5: **Recursion removes failures down to the dependence floor.** Multi-solution boards (S=6–500). (a) First rule violation by type. (b) Survival against generation progress. Red: final failure (a) and survival (b) of a sampler with perfect marginals, $q_i{=}p_i^\star$, committing the same cells.

We first verify whether indeed Lemma 4.1 correctly predicts the dependence of D_{dep} on the number of denoising steps, and of D_{fid} on the number of recursive steps. We find that indeed, the dependence cost increases as we increase the set of tokens committed at each step, hence lowering the number of denoising steps (Figure 4a). This effect is particularly pronounced for Sudoku boards where multiple solutions exist ($S > 1$). In turn, additional recursive steps reduce the fidelity error D_{fid} by 6–43$\times$ from $L{=}1$ to $L{=}5$ (Figure 4b). Beyond $L{=}5$, the benefits of the extra loops seem to plateau, indicating that additional loops do not always lead to better results, and inspiring the design of our adaptive loop rules in Section 5.

This difference between denoising and recursive steps is also reflected in the error rates. On the multi-solution boards in Figure 5a, increasing L eventually brings the failure rate close to that obtained with perfect marginals under the same commit schedule (64% vs. 59% at $T{=}8$; 35% vs. 30% at $T{=}16$). At $L{=}5$, 83–88% of the remaining parallel clashes are dependence errors, involving values that are individually valid but jointly impossible. Thus, once this dependence floor is reached, reducing the commit size is more effective than adding loops; parallel-clash rate is consequently a useful proxy for D_{dep} when $p^\star$ is spread (Remark A.9). The lower error rate also translates into substantial gains in survival, the fraction of runs with no violation (Figure 5b), which more than doubles from 14% to 36% at $T{=}8$ and from 30% to 65% at $T{=}16$, approaching the perfect-marginal levels of 41% and 70%. We note that extra loops help particularly halfway through the denoising process: early on, few cells are committed and the true marginals are spread, leaving little for sharper predictions to fix, while late on predictions are already sharp.

Together, these results explain the behavior in Section 3.2. On unique puzzles, errors at small T are primarily fidelity errors amplified by large commits, so recursion substantially improves performance. With multiple valid completions, recursion eventually reaches a dependence-error floor; beyond that point, smaller commit steps are more effective, because they help to reduce the token dependencies. Section 5 turns this tradeoff into a practical decoder by adaptively choosing both how many recursive steps to take, and which tokens to commit once we stop looping.

> ***Takeaway*** *Recursion reduces fidelity error but cannot remove dependence error. Once the dependence floor is reached, smaller commit steps are more effective.*

5 Towards a Practical Recursive Decoder

Section 4 showed that additional recursive steps can reduce the fidelity error (D_{fid}) by improving the model marginals at a fixed masked state, but cannot reduce the dependence error (D_{dep}) introduced by committing tokens jointly. Their benefit therefore varies throughout the denoising process: additional loops are useful while the marginals continue to improve, but provide little benefit once these improvements saturate or dependence error becomes dominant. Using a fixed number of recursive steps at every masked state may therefore waste computation.

Prior work has studied adaptive MDM decoding by varying which and how many tokens are committed after each model evaluation (Ben-Hamu et al., 2025; Zhang et al., 2026a). Lee et al. (2026b)

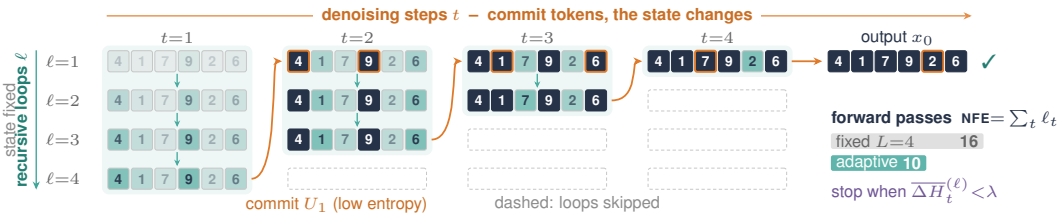


Figure 6: **Decoding an R-MDM has two axes of refinement.** Denoising steps run left to right and commit tokens, changing the masked state; recursive loops run top to bottom and sharpen the predictions while the state is held fixed. A decoder therefore makes two decisions per step: how many loops to spend, and which positions to commit.

further show that recursive depth can be chosen adaptively at a fixed denoising budget. To the best of our knowledge, however, no prior method jointly adapts recursive computation and token commitment. To combine these two adaptive axes, we cast an R-MDM decoder as a pair of rules (one determining the number of recursive steps and the other selecting the commit set) and use predictive entropy as a shared observable signal for both.

5.1 A TWO-AXIS DECODING FRAMEWORK

Decoding space. A decoder `[loop]_[commit]` pairs a loop-stopping rule, which sets $\ell_t \leq L$ at state s_t, with a commit rule, which selects $U_t \subseteq M_t$. At least one position is always committed per step so that decoding terminates, and compute is quantified as $\text{NFE} = \sum_t \ell_t$.

Entropy as a shared signal. Designing an adaptive decoder requires an observable signal for two decisions: whether to run another recursive step and which tokens to commit. We use predictive entropy, which summarizes the full marginal distribution over the vocabulary and, in R-MDM, can be computed for every masked position after each loop without additional model evaluations. Although recursive refinement acts on the fidelity term D_{fid}, this quantity depends on the unknown true marginals and is therefore unavailable at inference time. Instead, we monitor whether the model predictions continue to sharpen through the mean entropy over the masked positions and its reduction between consecutive loops:

$$\bar{H}(q^{(\ell)}) = \frac{1}{|M_t|} \sum_{i \in M_t} H\big(q_i^{(\ell)}\big), \qquad \overline{\Delta H}_t^{(\ell)} = \bar{H}(q^{(\ell-1)}) - \bar{H}(q^{(\ell)}). \tag{3}$$

A large positive entropy gain indicates that an additional loop is still sharpening the predicted distributions, whereas a small gain suggests diminishing returns.

Entropy also provides a principled signal for deciding which tokens to commit, as previously explored by Ben-Hamu et al. (2025) and Cai & Li (2026). Ben-Hamu et al. (2025) bound the dependence cost of a commit set by the sum of the true marginal entropies of its positions, and use the model's predictive entropies in their place as an approximation. We make this approximation explicit (Proposition A.3): substituting model entropies adds a correction term controlled by D_{fid} and vanishes as the fidelity error decreases. Recursive refinement therefore makes model entropy a more reliable guide, while selecting low-entropy positions limits the dependence cost of each commit.

Empirical validation. Using entropy for these two decisions requires two empirical properties: its reduction across loops should reflect continued refinement, and its value at each position should predict the likelihood of an error. Figure 7 tests both on Sudoku. At fixed masked states, mean entropy decreases with recursive depth. Pooled across mask fractions ($\rho_t = |M_t|/n$), the first additional loop accounts for 60% of the total reduction over ten loops, consistent with analyses of looped language models (Blayney et al., 2026). Later loops can still have a significant effect at intermediate states ($\rho_t \approx 0.50$), but little effect on early and late states, suggesting a stopping criterion. Because a loop may sharpen some positions while blurring others, $\overline{\Delta H}_t^{(\ell)}$ is a practical stopping signal rather than a convergence certificate. At the token level, entropy also predicts correctness: the pooled error rate increases from 7% near zero entropy to 85% at high entropy, with an AUROC of 0.86. These results support predictive entropy as a shared signal for loop stopping and token commitment.

Our decoder. We combine these two adaptive rules in the `entropy_entropy` decoder (Algorithm 2). At state s_t, its *loop rule* stops once the entropy gain of Equation 3 falls below

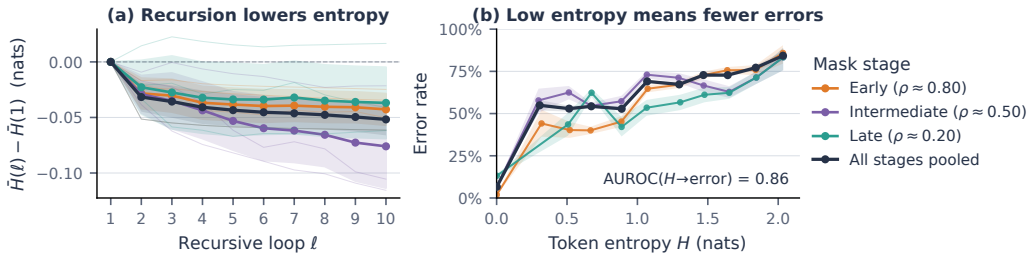


Figure 7: **Recursion sharpens predictions and reduces errors.** (a) Change in mean predictive entropy of masked cells relative to the first loop, entropy decreases monotonically. (b) Error rate versus predictive entropy. Low-entropy predictions are more reliable, with error increasing steadily with entropy (AUROC $= 0.86$).

a threshold, $\overline{\Delta H}_t^{(\ell)} < \lambda$ (or when the loops reach the upper limit L). In the spirit of EB-Sampler (Ben-Hamu et al., 2025), its *commit rule* greedily commits the lowest-entropy positions satisfying $\sum_{i \in U_t} H(q_i^{(\ell_t)}) \leq \delta$, which bounds D_{dep} by δ up to a correction term. Both rules use the same predictive entropies, and neither requires specifying T.

As baselines, we use the non-adaptive `fixed` loop rule ($\ell_t = L$) and the standard `uniform` commit rule, which commits the $|M_0|/T$ most confident positions. We also consider `sched` (linear decay from L to 1) and the `hidden` stopping rule, which stops when the relative hidden-state change $\|\mathbf{h}^{(\ell)} - \mathbf{h}^{(\ell-1)}\|/\|\mathbf{h}^{(\ell)}\|$ falls below ε (Lee et al., 2026b). The `hidden` rule achieves similar quality at higher NFE cost (Appendix G.6), motivating our use of entropy. The alternative `confidence` commit rule, which commits all positions with top-1 probability above c (Chang et al., 2022), is also evaluated. See further details in Appendix F and the complete evaluation in Appendix G.6.

> ***Contribution*** *We introduce a two-axis decoder for R-*MDM*s that adaptively decides* how long to refine *and* how much to commit*, using entropy as a common signal for both (Algorithm 2).*

5.2 Empirical verification across structured generation tasks

Tasks and metrics. We evaluate the `entropy_entropy` decoder across four domains. We retain Sudoku and Countdown from the previous sections, reporting Valid Puzzle Rate and Reaches Target Rate, respectively, and add RNA infilling and executable program generation on TinyGSM.

RNA infilling is a natural test of the mechanisms of Section 4 because base pairing creates long-range dependencies between nucleotide positions, challenging decoders that commit multiple tokens in parallel. We randomly mask 40% of held-out RNA sequences of length $n \in \{76, 110, 254\}$ from RNAcentral (The RNAcentral Consortium, 2026), using the original complete sequence as a reference. Because raw folding energy depends on sequence length and composition, we evaluate each completion using its ViennaRNA (Lorenz et al., 2011) minimum-free-energy Z-score, standardized against length- and composition-matched shuffles. We report the paired gap ($\Delta Z = Z(\text{infilled}) - Z(\text{native})$), which controls for sequence-specific difficulty; $\Delta Z = 0$ matches the reference folding score, and lower is better (see Appendix C.3 for further details on metrics).

TinyGSM, our largest setting (161M parameters, 512 tokens, 49k vocabulary), requires executable Python solutions to grade-school math problems (Liu et al., 2023), whose correctness is decided only by running the program. We report GSM8K execution accuracy (Cobbe et al., 2021) and further details in Appendix C.4.

Because `entropy_entropy` varies the number of recursive steps at inference time, we use checkpoints trained with variable loop depths: (*Schedule* $10 \to 1$) for Sudoku and Countdown, and (*Schedule* $5 \to 1$) for RNA and TinyGSM (see ablation in Appendix D.2). The non-recursive baseline (L=1) is a separate model with the same architecture, trained without recursion.

Results. Figure 8 compares four decoders on the quality–NFE plane: `fixed_uniform`, the default of Section 3; `fixed_entropy` and `entropy_uniform`, each of which makes only one axis adaptive; and our `entropy_entropy` (we consider alternative decoders in Appendix G.6). The best recursive result exceeds the non-recursive baseline in every domain: Sudoku from 80.7% to

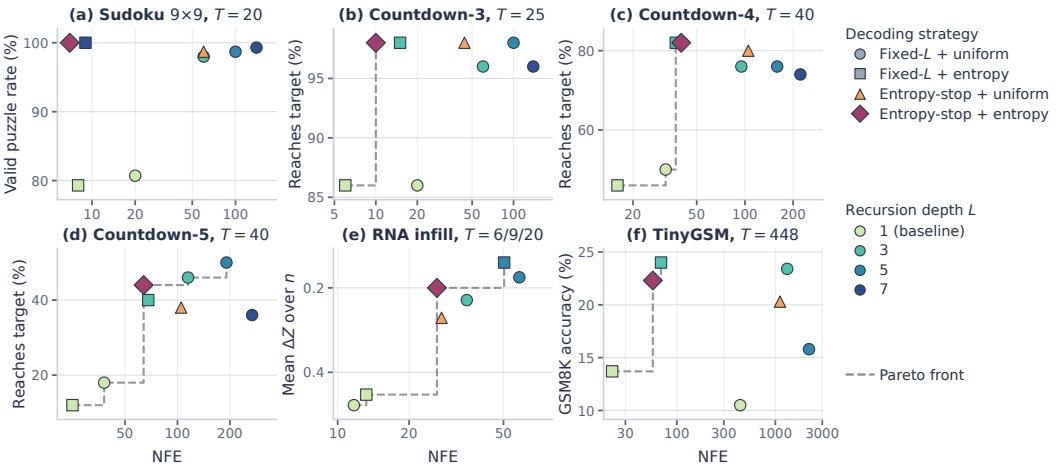


Figure 8: **Decoding results across tasks.** Quality vs. realized NFE per decoder (marker) and loop budget (color); `entropy_entropy` (diamonds) lies on or near the Pareto front on every task. Metrics: VPR (Sudoku), RTR (Countdown), ΔZ (RNA; lower is better), GSM8K accuracy (TinyGSM).

100% VPR, Countdown-4 from 50.0% to 82.0% RTR, RNA from $\Delta Z = 0.453$ to 0.140 (closing 69% of the gap to the native fold), and TinyGSM from 13.7% to 24.0% accuracy.

Under `fixed` stopping, NFE grows linearly with L even once extra loops no longer change the prediction; schemes which make one part of the decoding process adaptive already help to lower the computational cost, at small costs to the quality. However, our doubly-adaptive method lies on or near the Pareto front across all datasets, enabling even further efficiency gains.

We further observe that on long sequences, making the commit rule adaptive explains most of the efficiency gains: on TinyGSM, `fixed_entropy` at L=3 matches the best `fixed_uniform` accuracy (24.0% vs. 24.2%) with 69 instead of 768 evaluations, while loop stopping trims only half a loop per step (Appendix G.8). However, when allowing for large L, loop stopping is necessary for efficiency: on Countdown-4, the cost of `fixed_entropy` grows from 37 to 98 evaluations between L=3 and $L = 7$, while `entropy_entropy` stays near 40.

> ***Takeaway*** *Entropy effectively guides both decoding axes:* `entropy_entropy` *pays for loops only while predictions still change, keeps cost flat as L grows, and matches the best quality on Sudoku and Countdown-3/4 at a fraction of the compute.*

6 Conclusions, Limitations, and Future Work

In this work, we studied recursion as a scaling mechanism for masked diffusion models. Applying the same transformer block L times within each denoising step increases effective depth without increasing parameter count. Across Sudoku and Countdown, recursive depth improves quality at fixed parameter count, matches substantially larger non-recursive models at matched FLOPs, and can reduce the number of denoising steps required to reach a given quality.

Our analysis interprets these gains through the different roles of recursive loops and denoising steps. Recursion refines model marginals at a fixed masked state, while denoising steps change that state by committing tokens; accordingly, the two axes affect different components of parallel-decoding error. Recursive refinement reduces predictive entropy and conflicts among simultaneously decoded tokens, especially when each denoising step must resolve many dependencies, but it cannot remove the dependence error among tokens committed together. These observations motivate adaptive decoding, where loop stopping and token commitment are selected from state-dependent signals. The resulting methods improve the quality–compute trade-off across all four domains.

This work establishes the viability of recursive MDMs on structured tasks and provides a framework for scaling their effective depth without increasing parameter count. Future work can build on this framework by scaling the paradigm to larger models, and test whether the gains offered by our recursive framework extend to less structured settings such as open-ended language generation.

AI USE STATEMENT

In this work, we used generative AI tools to help formulate mathematical claims, to provide guidelines for proving those claims and assist in the writing of proofs, to provide feedback on our experiments, and to implement methods in code. We have not used generative AI tools to help develop theoretical models or conceptual frameworks, to propose or refine hypotheses, to design our research methodology, to clean and reformat datasets, to support qualitative and thematic data analysis, to suggest a structure for the paper, to generate synthetic datasets, or to interpret results, and assistance with translation are not applicable to this work. Additionally, we used generative AI tools to create and edit software code, and to edit the paper to improve readability. We have reviewed all AI-assisted work: all mathematical claims and their proofs were checked in full by the authors, AI-assisted code was verified and tested for correctness by two authors, and all AI-edited text was reviewed by all the authors to confirm that it accurately reflects our intended meaning and claims. We take responsibility for the final content of this work, including text, claims or artifacts produced with the aid of generative AI.

ETHICS STATEMENT

The primary objective of this paper is to advance discrete generative models with a more parameter-efficient architecture. By reducing the model size and number of denoising steps required to achieve high-quality generation, recursive masked diffusion models could lower the computational cost of deploying sequence generation systems, with potential benefits in resource-constrained settings. While more capable and accessible generative models carry the general dual-use concerns associated with language generation research, we do not foresee any immediate societal concerns specific to the methodology proposed here.

REPRODUCIBILITY STATEMENT

To support reproducibility, Section 3.1 describes the model and training objective, while Section 5.1 and Algorithm 2 specify the decoding rules. Appendix C documents dataset sources, construction and preprocessing, together with the evaluation protocol and metrics for each task. Appendix D reports the training procedure (Algorithm 1), the architecture and optimization settings for every dataset (Table 2), the computational resources, and the ablations behind our training choices (training loop count, step embedding, and supervision). Appendix E describes how the multi-solution Sudoku boards of Section 4.2 are built and how the dependence and fidelity terms are computed exactly on them. Appendix F gives the complete decoding-rule definitions and ablations, the tuning grids and selected values of every decoding threshold, and the sampling temperature and top-k used for each dataset (Table 6). The proofs of the theoretical results, with their assumptions and scope, are provided in Appendix A. Full experimental results, with the number of sampling runs or evaluation examples and the uncertainty behind each number, are reported in Appendix G. Sample generations are available in Appendix H.

REFERENCES

Koenraad M R Audenaert. A sharp continuity estimate for the von Neumann entropy. *Journal of Physics A: Mathematical and Theoretical*, 40(28):8127, 2007. URL https://doi.org/10.1088/1751-8113/40/28/S18. 19

Jacob Austin, Daniel D. Johnson, Jonathan Ho, Daniel Tarlow, and Rianne van den Berg. Structured Denoising Diffusion Models in Discrete State-Spaces. *Advances in Neural Information Processing Systems (NeurIPS)*, 2021. URL https://arxiv.org/abs/2107.03006. 1, 2, 21

Heli Ben-Hamu, Itai Gat, Daniel Severo, Niklas Nolte, and Brian Karrer. Accelerated Sampling from Masked Diffusion Models via Entropy Bounded Unmasking. *Advances in Neural Information Processing Systems (NeurIPS)*, 2025. URL https://arxiv.org/abs/2505.24857. 2, 5, 6, 7, 8, 17, 18, 19, 20, 22, 38

Hugh Blayney, Álvaro Arroyo, Johan Obando-Ceron, Pablo Samuel Castro, Aaron Courville, Michael M. Bronstein, and Xiaowen Dong. A mechanistic analysis of looped reasoning lan-

guage models. *arXiv preprint arXiv:2604.11791*, 2026. URL https://arxiv.org/abs/2604.11791. 7

Changxiao Cai and Gen Li. Confidence-Based Decoding is Provably Efficient for Diffusion Language Models. *arXiv preprint arXiv:2603.22248*, 2026. URL https://arxiv.org/abs/2603.22248. 2, 7, 19, 22

Huiwen Chang, Han Zhang, Lu Jiang, Ce Liu, and William T. Freeman. MaskGIT: Masked Generative Image Transformer. *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2022. URL https://arxiv.org/abs/2202.04200. 2, 8, 21, 37

Chen-Hao Chao, Wei-Fang Sun, Hanwen Liang, Chun-Yi Lee, and Rahul G. Krishnan. Beyond Masked and Unmasked: Discrete Diffusion Models via Partial Masking. *Advances in Neural Information Processing Systems (NeurIPS)*, 2025. URL https://arxiv.org/abs/2505.18495. 23

Sitan Chen, Kevin Cong, and Jerry Li. Optimal Inference Schedules for Masked Diffusion Models. *Conference on Learning Theory (COLT)*, 2026. URL https://arxiv.org/abs/2511.04647. 18, 22

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. URL https://arxiv.org/abs/2110.14168. 8

Mostafa Dehghani, Stephan Gouws, Oriol Vinyals, Jakob Uszkoreit, and Łukasz Kaiser. Universal Transformers. *International Conference on Learning Representations (ICLR)*, 2019. URL https://arxiv.org/abs/1807.03819. 1, 2, 23, 30

Chunyuan Deng, Yizhe Zhang, Rui-Jie Zhu, Yuanyuan Xu, Jiarui Liu, T. S. Eugene Ng, and Hanjie Chen. LT2: Linear-Time Looped Transformers. *arXiv preprint arXiv:2605.20670*, 2026. URL https://arxiv.org/abs/2605.20670. 23

DiffusionGemma Team. DiffusionGemma Technical Report. *arXiv preprint arXiv:2608.00146*, 2026. URL https://arxiv.org/abs/2608.00146. 1, 2, 21

Ying Fan, Yilun Du, Kannan Ramchandran, and Kangwook Lee. Looped Transformers for Length Generalization. *International Conference on Learning Representations (ICLR)*, 2025. URL https://arxiv.org/abs/2409.15647. 30

David Fox, Sam Bowyer, Song Liu, Laurence Aitchison, Raul Santos-Rodriguez, and Mengyue Yang. Learning Generation Orders for Masked Discrete Diffusion Models via Variational Inference. *arXiv preprint arXiv:2602.23968*, 2026. URL https://arxiv.org/abs/2602.23968. 23

Kanishk Gandhi, Denise Lee, Gabriel Grand, Muxin Liu, Winson Cheng, Archit Sharma, and Noah D. Goodman. Stream of Search (SoS): Learning to Search in Language. *Conference on Language Modeling (COLM)*, 2024. URL https://arxiv.org/abs/2404.03683. 3, 25

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. *Advances in Neural Information Processing Systems (NeurIPS)*, 2025. URL https://arxiv.org/abs/2502.05171. 2, 23, 30

Angeliki Giannou, Shashank Rajput, Jy-yong Sohn, Kangwook Lee, Jason D. Lee, and Dimitris Papailiopoulos. Looped Transformers as Programmable Computers. *International Conference on Machine Learning (ICML)*, 2023. URL https://arxiv.org/abs/2301.13196. 1, 2, 23

Andre He, Sean Welleck, and Daniel Fried. Reasoning with Latent Tokens in Diffusion Language Models. *arXiv preprint arXiv:2602.03769*, 2026. URL https://arxiv.org/abs/2602.03769. 3, 21, 25

Jonathan Ho, Ajay Jain, and Pieter Abbeel. Denoising Diffusion Probabilistic Models. *Advances in Neural Information Processing Systems (NeurIPS)*, 2020. URL https://arxiv.org/abs/2006.11239. 1, 21

Feng Hong, Geng Yu, Yushi Ye, Haicheng Huang, Huangjie Zheng, Ya Zhang, Yanfeng Wang, and Jiangchao Yao. Wide-In, Narrow-Out: Revokable Decoding for Efficient and Effective DLLMs. *International Conference on Learning Representations (ICLR)*, 2026. URL https://arxiv.org/abs/2507.18578. 22

Zachary Horvitz, Raghav Singhal, Hao Zou, Carles Domingo-Enrich, Zhou Yu, Rajesh Ranganath, and Kathleen McKeown. Rethinking Reasoning with MDLMs: Early Exits, Post-hoc Reasoning, and Beyond. *Conference on Language Modeling (COLM)*, 2026. URL https://arxiv.org/abs/2510.19990. 23

Hanzhang Hu, Debadeepta Dey, Martial Hebert, and J. Andrew Bagnell. Learning Anytime Predictions in Neural Networks via Adaptive Loss Balancing. *Association for the Advancement of Artificial Intelligence (AAAI)*, 2019. URL https://arxiv.org/abs/1708.06832. 32

Jiahe Huang, Sihan Xu, Sharvaree Vadgama, and Rose Yu. Recursive Flow Matching. *arXiv preprint arXiv:2605.26535*, 2026a. URL https://arxiv.org/abs/2605.26535. 23

Pengcheng Huang, Tianming Liu, Zhenghao Liu, Yukun Yan, Shuo Wang, Tong Xiao, Zulong Chen, and Maosong Sun. Empirical Analysis of Decoding Biases in Masked Diffusion Models. *Association for Computational Linguistics (ACL)*, 2026b. URL https://arxiv.org/abs/2508.13021. 22

Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling Laws for Neural Language Models. *arXiv preprint arXiv:2001.08361*, 2020. URL https://arxiv.org/abs/2001.08361. 1

Jaeyeon Kim, Kulin Shah, Vasilis Kontonis, Sham Kakade, and Sitan Chen. Train for the Worst, Plan for the Best: Understanding Token Ordering in Masked Diffusions. *International Conference on Machine Learning (ICML)*, 2025. URL https://openreview.net/forum?id=DjJmre5IkP. 2, 22

Zhenzhong Lan, Mingda Chen, Sebastian Goodman, Kevin Gimpel, Piyush Sharma, and Radu Soricut. ALBERT: A lite BERT for self-supervised learning of language representations. *International Conference on Learning Representations (ICLR)*, 2020. URL https://arxiv.org/abs/1909.11942. 2, 23

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*, 2026a. URL https://arxiv.org/abs/2602.16813. 21

Chen-Yu Lee, Saining Xie, Patrick Gallagher, Zhengyou Zhang, and Zhuowen Tu. Deeply-Supervised Nets. *International Conference on Artificial Intelligence and Statistics (AISTATS)*, 2015. URL https://arxiv.org/abs/1409.5185. 3, 32

Sanghyun Lee, Chunsan Hong, Seungryong Kim, Jonghyun Lee, Jongho Park, and Dongmin Park. Looped Diffusion Language Models. *arXiv preprint arXiv:2605.26106*, 2026b. URL https://arxiv.org/abs/2605.26106. 2, 6, 8, 23

Gen Li and Changxiao Cai. Breaking AR's Sampling Bottleneck: Provable Acceleration via Diffusion Language Models. *Advances in Neural Information Processing Systems (NeurIPS)*, 2025. URL https://arxiv.org/abs/2505.21400. 18, 22

Shuzhen Li, Yifan Zhang, Jiacheng Guo, Quanquan Guo, and Mengdi Wang. DeepLoop: Depth Scaling for Looped Transformers. *arXiv preprint arXiv:2607.13491*, 2026. URL https://arxiv.org/abs/2607.13491. 23

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*, 2023. URL https://arxiv.org/abs/2312.09241. 8, 28

Sulin Liu, Juno Nam, Andrew Campbell, Hannes Stärk, Yilun Xu, Tommi Jaakkola, and Rafael Gómez-Bombarelli. Think While You Generate: Discrete Diffusion with Planned Denoising. *International Conference on Learning Representations (ICLR)*, 2025. URL https://arxiv.org/abs/2410.06264. 22

Ronny Lorenz, Stephan H. Bernhart, Christian Höner zu Siederdissen, Hakim Tafer, Christoph Flamm, Peter F. Stadler, and Ivo L. Hofacker. ViennaRNA Package 2.0. *Algorithms for Molecular Biology*, 6(1):26, 2011. 8

Aaron Lou, Chenlin Meng, and Stefano Ermon. Discrete Diffusion Modeling by Estimating the Ratios of the Data Distribution. *International Conference on Machine Learning (ICML)*, 2024. URL https://arxiv.org/abs/2310.16834. 2, 21

Andrea Miele, Yiming Qin, Alba Carballo-Castro, Justin Deschenaux, and Pascal Frossard. Fixed-Point Masked Generative Modeling. *arXiv preprint arXiv:2605.31215*, 2026. URL https://arxiv.org/abs/2605.31215. 2, 23

Sajad Movahed, Vera Milovanović, Shlomo Libo Feigin, Alexander Theus, Thomas Hofmann, Valentina Boeva, T. Konstantin Rusch, and Antonio Orvieto. Fixed-Point Reasoners: Stable and Adaptive Deep Looped Transformers. *arXiv preprint arXiv:2606.18206*, 2026. URL https://arxiv.org/abs/2606.18206. 23

Shen Nie, Fengqi Zhu, Zebin You, Xiaolu Zhang, Jingyang Ou, Jun Hu, Jun Zhou, Yankai Lin, Ji-Rong Wen, and Chongxuan Li. Large Language Diffusion Models. *Advances in Neural Information Processing Systems (NeurIPS)*, 2025. URL https://arxiv.org/abs/2502.09992. 1, 2, 21

Yong-Hyun Park, Chieh-Hsin Lai, Satoshi Hayakawa, Yuhta Takida, and Yuki Mitsufuji. Jump Your Steps: Optimizing Sampling Schedule of Discrete Diffusion Models. *International Conference on Learning Representations (ICLR)*, 2025. URL https://arxiv.org/abs/2410.07761. 22

Fred Zhangzhi Peng, Zachary Bezemek, Sawan Patel, Jarrid Rector-Brooks, Sherwood Yao, Avishek Joey Bose, Alexander Tong, and Pranam Chatterjee. Path Planning for Masked Diffusion Model Sampling. *arXiv preprint arXiv:2502.03540*, 2025. URL https://arxiv.org/abs/2502.03540. 22

Rafael Josip Penić, Tin Vlašić, Roland G. Huber, Yue Wan, and Mile Šikić. RiNALMo: General-Purpose RNA Language Models Can Generalize Well on Structure Prediction Tasks. *Nature Communications*, 16:5671, 2025. URL https://www.nature.com/articles/s41467-025-60872-5. 27, 29

Yinuo Ren, Haoxuan Chen, Yuchen Zhu, Wei Guo, Yongxin Chen, Grant M. Rotskoff, Molei Tao, and Lexing Ying. Fast Solvers for Discrete Diffusion Models: Theory and Applications of High-Order Algorithms. *Advances in Neural Information Processing Systems (NeurIPS)*, 2025. URL https://arxiv.org/abs/2502.00234. 23

Yusuf Sahin, Ahmed Rockey Saikia, Volkan Cevher, and Paolo Favaro. Attention-Discounted Adaptive Sampler for Masked Diffusion Language Models. *arXiv preprint arXiv:2606.10829*, 2026. URL https://arxiv.org/abs/2606.10829. 22

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. *Advances in Neural Information Processing Systems (NeurIPS)*, 2024. URL https://arxiv.org/abs/2406.07524. 1, 2, 3, 21, 28, 29

Subham Sekhar Sahoo, Jean-Marie Lemercier, Zhihan Yang, Justin Deschenaux, Jingyu Liu, John Thickstun, and Ante Jukic. Scaling Beyond Masked Diffusion Language Models. *International Conference on Machine Learning (ICML)*, 2026. URL https://arxiv.org/abs/2602.15014. 21

Shreshth Saini, Avinab Saha, Balu Adsumilli, Neil Birkbeck, Yilin Wang, and Alan C. Bovik. TABES: Trajectory-Aware Backward-on-Entropy Steering for Masked Diffusion Models. *arXiv preprint arXiv:2602.00250*, 2026. URL https://arxiv.org/abs/2602.00250. 23

Nikunj Saunshi, Nishanth Dikkala, Zhiyuan Li, Sanjiv Kumar, and Sashank J. Reddi. Reasoning with Latent Thoughts: On the Power of Looped Transformers. *International Conference on Learning Representations (ICLR)*, 2025. URL https://arxiv.org/abs/2502.17416. 1, 2, 3, 23

Kulin Shah, Nishanth Dikkala, Xin Wang, and Rina Panigrahy. Causal Language Modeling Can Elicit Search and Reasoning Capabilities on Logic Puzzles. *Advances in Neural Information Processing Systems (NeurIPS)*, 2024. URL https://arxiv.org/abs/2409.10502. 24

Jascha Sohl-Dickstein, Eric A. Weiss, Niru Maheswaranathan, and Surya Ganguli. Deep Unsupervised Learning using Nonequilibrium Thermodynamics. *International Conference on Machine Learning (ICML)*, 2015. URL http://arxiv.org/abs/1503.03585. 1, 21

Yang Song, Jascha Sohl-Dickstein, Diederik P. Kingma, Abhishek Kumar, Stefano Ermon, and Ben Poole. Score-Based Generative Modeling through Stochastic Differential Equations. *International Conference on Learning Representations (ICLR)*, 2021. URL https://arxiv.org/abs/2011.13456. 1, 21

Jianlin Su, Yu Lu, Shengfeng Pan, Ahmed Murtadha, Bo Wen, and Yunfeng Liu. RoFormer: Enhanced Transformer with Rotary Position Embedding. *Neurocomputing*, 568, 2023. URL https://arxiv.org/abs/2104.09864. 29

The RNAcentral Consortium. RNAcentral in 2026: genes and literature integration. *Nucleic Acids Research*, 54, 2026. doi: 10.1093/nar/gkaf1329. URL https://academic.oup.com/nar/article/54/D1/D303/8382359. 8, 26

Guanghan Wang, Yair Schiff, Subham Sekhar Sahoo, and Volodymyr Kuleshov. Remasking Discrete Diffusion Models with Inference-Time Scaling. *Advances in Neural Information Processing Systems (NeurIPS)*, 2025. URL https://arxiv.org/abs/2503.00307. 22

Chengyue Wu, Hao Zhang, Shuchen Xue, Zhijian Liu, Shizhe Diao, Ligeng Zhu, Ping Luo, Song Han, and Enze Xie. Fast-dLLM: Training-Free Acceleration of Diffusion LLM by Enabling KV Cache and Parallel Decoding. *International Conference on Learning Representations (ICLR)*, 2026. URL https://arxiv.org/abs/2505.22618. 23

Kevin Xu and Issei Sato. A Formal Comparison Between Chain of Thought and Latent Thought. *International Conference on Machine Learning (ICML)*, 2026. URL https://arxiv.org/abs/2509.25239. 2, 23

Ruihan Xu, Yuting Gao, Lan Wang, Jianing Li, Weihao Chen, Qingpei Guo, Ming Yang, and Shiliang Zhang. Looping Back to Move Forward: Recursive Transformers for Efficient and Flexible Large Multimodal Models. *arXiv preprint arXiv:2602.09080*, 2026. URL https://arxiv.org/abs/2602.09080. 30

Liu Yang, Kangwook Lee, Robert Nowak, and Dimitris Papailiopoulos. Looped Transformers are Better at Learning Learning Algorithms. *International Conference on Learning Representations (ICLR)*, 2024. URL https://arxiv.org/abs/2311.12424. 2, 23, 30, 33

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Thomas L. Griffiths, Yuan Cao, and Karthik Narasimhan. Tree of Thoughts: Deliberate Problem Solving with Large Language Models. *Advances in Neural Information Processing Systems (NeurIPS)*, 2023. URL https://arxiv.org/abs/2305.10601. 3, 25

Jiacheng Ye, Jiahui Gao, Shansan Gong, Lin Zheng, Xin Jiang, Zhenguo Li, and Lingpeng Kong. Beyond autoregression: Discrete diffusion for complex reasoning and planning. *International Conference on Learning Representations (ICLR)*, 2025a. URL https://arxiv.org/abs/2410.14157. 21

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*, 2025b. URL https://arxiv.org/abs/2508.15487. 1

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. URL https://arxiv.org/abs/2607.00714. 23

Qifan Yu, Zhenyu He, Sijie Li, Xun Zhou, Jun Zhang, Jingjing Xu, and Di He. Enhancing Autoregressive Chain-of-Thought through Loop-Aligned Reasoning. *Association for Computational Linguistics (ACL)*, 2026. URL https://arxiv.org/abs/2502.08482. 2, 23, 32

Biao Zhang and Rico Sennrich. Root Mean Square Layer Normalization. *Advances in Neural Information Processing Systems (NeurIPS)*, 2019. URL https://arxiv.org/abs/1910.07467. 3

Leo Zhang and Saifuddin Syed. The Cosine Schedule is Fisher-Rao-Optimal for Masked Discrete Diffusion Models. *arXiv preprint arXiv:2508.04884*, 2025. URL https://arxiv.org/abs/2508.04884. 22

Shaorong Zhang, Longxuan Yu, Rob Brekelmans, Luhan Tang, Salman Asif, and Greg Ver Steeg. Generation Order and Parallel Decoding in Masked Diffusion Models: An Information-Theoretic Perspective. *arXiv preprint arXiv:2602.00286*, 2026a. URL https://arxiv.org/abs/2602.00286. 6, 17, 18, 20, 21

Tong Zhang, Junhao Hu, Yun Peng, and Tao Xie. When Does Recurrence Become an Algorithm? Convergence Selection in Weight-Tied Looped Transformers. *arXiv preprint arXiv:2607.20594*, 2026b. URL https://arxiv.org/abs/2607.20594. 23

Huangjie Zheng, Shansan Gong, Ruixiang Zhang, Tianrong Chen, Jiatao Gu, Mingyuan Zhou, Navdeep Jaitly, and Yizhe Zhang. Continuously augmented discrete diffusion model for categorical generative modeling. *International Conference on Learning Representations (ICLR)*, 2026. URL https://arxiv.org/abs/2510.01329. 21

Lin Zheng, Jianbo Yuan, Lei Yu, and Lingpeng Kong. A Reparameterized Discrete Diffusion Model for Text Generation. *Conference on Language Modeling (COLM)*, 2024. URL https://arxiv.org/abs/2302.05737. 22

Cai Zhou, Chenxiao Yang, Yi Hu, Chenyu Wang, Chubin Zhang, Muhan Zhang, Lester Mackey, Tommi Jaakkola, Stephen Bates, and Dinghuai Zhang. Coevolutionary Continuous Discrete Diffusion: Make Your Diffusion Language Model a Latent Reasoner. *arXiv preprint arXiv:2510.03206*, 2025. URL https://arxiv.org/abs/2510.03206. 21

Appendix

Table of Contents

A Proofs 17
- A.1 Notation and setting . . . 17
- A.2 Proof of Lemma 4.1 . . . 17
- A.3 The entropy certificate . . . 18
- A.4 Trajectory decomposition and the price of a finer schedule . . . 19
- A.5 Scope of the results . . . 20

B Extended Related Work 21
- B.1 The Landscape of Discrete Diffusion for Language . . . 21
- B.2 Decoding Strategies for Masked Diffusion . . . 21
- B.3 Recursive and Fixed-Point Computation . . . 23

C Dataset and Metrics 24
- C.1 Sudoku . . . 24
- C.2 Countdown . . . 25
- C.3 RNA . . . 26
- C.4 TinyGSM . . . 28

D Training Algorithm and Design Choices 28
- D.1 Training procedure . . . 28
- D.2 Choosing the training loop count L_t . . . 29
- D.3 Step embedding ablation . . . 30
- D.4 Training objectives . . . 31

E Exact Dependence–Fidelity Measurement 33

F The Decoding Zoo 35
- F.1 Zoo overview . . . 35
- F.2 Loop-stopping rules (how many inner loops at a state) . . . 36
- F.3 Commit rules (which masked positions to reveal) . . . 37
- F.4 Ablation on the decoding mechanisms . . . 37
- F.5 Hyperparameter tuning . . . 38

G Extended Results 40
- G.1 Sudoku 9×9: Full Tables . . . 40
- G.2 Sudoku 9×9: Depth Scaling Without Recursion . . . 42
- G.3 Sudoku 9×9: Cross-Recursion Trade-off and Efficient Compute . . . 42
- G.4 Sudoku 25×25: Effect of Recursion and Masking . . . 44
- G.5 Countdown: Extended Results . . . 45
- G.6 Decoding algorithms comparison . . . 48
- G.7 RNA Infilling: Extended Results . . . 50
- G.8 TinyGSM: Extended Results . . . 51

H Samples 52
- H.1 Sudoku 9×9 . . . 52
- H.2 Countdown-5 . . . 53
- H.3 TinyGSM . . . 54
- H.4 RNA . . . 55

A PROOFS

This appendix proves the results of Section 4.1: Lemma 4.1, Proposition A.3 and Theorem A.5, and states their scope (Appendix A.5). Lemma 4.1 and Theorem A.5 follow the decomposition of Ben-Hamu et al. (2025), extended to policies that also choose the loop count.

A.1 NOTATION AND SETTING

Fix a partially masked state $s = (x_O, M)$ with committed positions O and masked positions M, as in Section 3.1. Throughout, states satisfy $\pi_{\text{data}}(x_O) > 0$, so that the conditional below is defined; this holds at every state visited by the oracle process P below, but not after a wrong commit by the sampler. Let π_{data} be the data distribution and let

$$p^\star(x_U \mid s) \;=\; \pi_{\text{data}}\big(x_U \mid x_O\big), \qquad U \subseteq M, \tag{4}$$

denote the true conditional joint over the commit set U given the committed context, with position-wise marginals $p_i^\star(\cdot \mid s)$. All entropies are Shannon entropies in nats over the finite vocabulary $\mathcal{V}$, $m = |\mathcal{V}|$, and TC denotes total correlation (multi-information),

$$\mathrm{TC}(p) \;=\; \sum_{i\in U} H(p_i) - H(p) \;=\; D_{\mathrm{KL}}\Big(p \,\Big\|\, \prod_{i\in U} p_i\Big) \;\geq\; 0, \tag{5}$$

which is zero if and only if p factorizes across U.

The two processes. A *decoding policy* is a pair of maps $s \mapsto \ell(s) \in \{1, \dots, L\}$ and $s \mapsto U(s) \subseteq M(s)$ with $U(s) \neq \emptyset$. Both may depend on the model's predictions at s, since θ is fixed and those predictions are themselves functions of s. More generally, both may depend on the step index, on the history $(s_0, \dots, s_t)$, and on randomness ξ drawn before decoding, such as a random commit order; reading s_t as that history and conditioning on ξ keeps both chains below Markov, and every result holds unchanged. This covers every decoder of Section 5, including the step-indexed `uniform` and `sched` rules, as well as the split schedule of Corollary A.7 and the fixed commit order of Section 4.2. Given x_O and a policy we compare two chains started at $s_0 = (x_O, M)$:

- the **oracle process** P, which at state s_t sets $U_t = U(s_t)$ and draws $x_{U_t} \sim p^\star(\cdot \mid s_t)$;
- the **sampler** Q, which at s_t sets $U_t = U(s_t)$, $\ell_t = \ell(s_t)$ and draws $x_{U_t} \sim \prod_{i\in U_t} q_i^{(\ell_t)}(\cdot \mid s_t)$.

Both chains halt when $M_t = \emptyset$, after a number of steps $\tau \leq |M|$ determined by the state sequence; terminal states contribute nothing to any sum below.

Remark A.1 ($D_{\mathrm{KL}}(P\|Q)$ is a statement about outputs). Under P, $\prod_t p^\star(x_{U_t} \mid s_t) = \pi_{\text{data}}(x_M \mid x_O)$ for any adapted commit order, and the trajectory is a deterministic function of x_M and vice versa, so $D_{\mathrm{KL}}(P\|Q)$ equals the KL between output laws (Ben-Hamu et al., 2025). It measures coverage of $\pi_{\text{data}}(\cdot \mid x_O)$ and is less sensitive to off-support samples (Zhang et al., 2026a), which Section 4.2 measures via violation rates.

A.2 PROOF OF LEMMA 4.1

Proof. Write $p = p^\star(\cdot \mid s)$ and $q_U = \prod_{i\in U} q_i$. The hypothesis is that each $p_i^\star$ is absolutely continuous with respect to q_i, *i.e.*, $q_i(v) = 0 \Rightarrow p_i^\star(v) = 0$; this is what makes $D_{\mathrm{KL}}(p_i^\star\|q_i)$, and hence every term below, finite. Expanding,

$$\begin{aligned}
D_{\mathrm{KL}}(p\|q_U) &= \sum_{x_U} p(x_U) \log \frac{p(x_U)}{\prod_{i\in U} q_i(x_i)} \\
&= -H(p) - \sum_{i\in U} \sum_{x_i} p_i(x_i) \log q_i(x_i) && \text{(marginalizing all } j \neq i\text{)} \\
&= -H(p) + \sum_{i\in U} \Big[H(p_i) + D_{\mathrm{KL}}(p_i\|q_i)\Big] && (\mathrm{CE} = H + D_{\mathrm{KL}}) \\
&= \underbrace{\Big[\textstyle\sum_{i\in U} H(p_i) - H(p)\Big]}_{=\mathrm{TC}(p) = D_{\mathrm{dep}}(U|s)} + \underbrace{\sum_{i\in U} D_{\mathrm{KL}}(p_i\|q_i)}_{=D_{\mathrm{fid}}(U|s,q)},
\end{aligned} \tag{6}$$

which is Equation 2. Both terms are non-negative: the first by Equation 5 and the second as a sum of KL divergences. □

Remark A.2 (Which distribution the analysis refers to). Throughout, q_i denotes the *denoiser marginal*, the softmax over the vocabulary in Equation 1. Since softmax outputs are strictly positive, $p_i^\star \ll q_i$ holds for any $p^\star$ and the hypothesis of Lemma 4.1 is never binding for q_i. The predictive entropies used by the commit and stopping rules are also computed from q_i.

This is not the distribution the experiments sample from. At sampling time we apply temperature scaling and top-k truncation (Table 6), giving a deployed law $\tilde{q}_i \neq q_i$ whose support is a strict subset of $\mathcal{V}$ when $k < m$; for Countdown and TinyGSM, $k = 1$. Whenever $p_i^\star$ places mass outside that support, $D_{\mathrm{KL}}(p_i^\star \| \tilde{q}_i) = \infty$, so the finite-KL statements below are results about the untruncated sampler that draws from q_i, not certificates for every decoding configuration we run. We keep the truncated samplers in the experiments because they are what one deploys, and read the analysis as describing the denoiser marginals they are built from.

Two immediate consequences are used in the main text. First, $D_{\mathrm{dep}}(U \mid s)$ depends on $p^\star$, s and U only, so any operation that holds s and U fixed and modifies only the deployed marginals (in particular running additional recursive loops, which by construction leave the state and the commit set unchanged) changes the step error only through D_{fid}. Second, if $|U| = 1$ then $\mathrm{TC} = 0$, so a single-position commit incurs no dependence cost at all; this is the formal version of the standard observation that the speed–quality trade-off of parallel MDM decoding is governed by inter-token dependence (Ben-Hamu et al., 2025; Zhang et al., 2026a; Li & Cai, 2025; Chen et al., 2026).

A.3 The entropy certificate

Section 5.1 uses the following bound to justify committing low-entropy positions; we state it here in full, with the slack term that the main text describes only qualitatively.

Proposition A.3 (Entropy bounds the dependence cost)**.** *Fix a state s, a non-empty commit set U, a vocabulary of size $m \geq 2$, and denoiser marginals $q = q^{(\ell)}$; write $H_i = H(q_i)$ for the decoder's own predictive entropies. Then*

$$D_{\mathrm{dep}}(U \mid s) \;\leq\; \sum_{i \in U} H_i \;-\; \max_{i \in U} H_i \;+\; \varepsilon, \qquad \varepsilon \;\coloneqq\; \sum_{i \in U} \big| H(p_i^\star) - H_i \big|, \tag{7}$$

where ε is the total error in those entropies. If in addition $D_{\mathrm{KL}}(p_i^\star \| q_i) \leq 2(1 - 1/m)^2$ for every $i \in U$, with $m = |\mathcal{V}|$, then $\varepsilon \leq |U|\, \omega\big(D_{\mathrm{fid}}(U \mid s, q)/|U|\big)$ for an increasing function ω with $\omega(0) = 0$, given explicitly below.

The function referred to in the statement is

$$\omega(x) \;=\; f\big(\sqrt{x/2}\big), \qquad f(\eta) \;=\; \eta \log(m-1) + h(\eta), \tag{8}$$

with h the binary entropy. We show below that ω is concave and increasing on $[0, 2(1 - 1/m)^2]$ with $\omega(0) = 0$.

Proof. **Step 1: the bound in the true marginals.** Joint entropy dominates any marginal entropy: for $j \in U$, $H(x_U) = H(x_j) + H(x_{U \setminus j} \mid x_j) \geq H(x_j)$, so $H(p^\star(x_U \mid s)) \geq \max_{i \in U} H(p_i^\star)$. Substituting into Equation 5,

$$D_{\mathrm{dep}}(U \mid s) \;\leq\; \sum_{i \in U} H(p_i^\star) - \max_{i \in U} H(p_i^\star). \tag{9}$$

This is the bound of Ben-Hamu et al. (2025).

Step 2: transfer to the model's entropies. Write $a_i = H(p_i^\star)$ and $b_i = H_i = H(q_i)$, so that $\varepsilon = \sum_{i \in U} |a_i - b_i|$. The functional $F(z) = \sum_{i \in U} z_i - \max_{i \in U} z_i$ is 1-Lipschitz with respect to $\|\cdot\|_1$: each subgradient has entries in $[0, 1]$ and at least one zero entry. Therefore $F(a) \leq F(b) + \|a - b\|_1 = F(b) + \varepsilon$, i.e.

$$\sum_{i \in U} a_i - \max_{i \in U} a_i \;\leq\; \sum_{i \in U} b_i - \max_{i \in U} b_i + \varepsilon \;=\; \sum_{i \in U} H_i - \max_{i \in U} H_i + \varepsilon, \tag{10}$$

which combined with Equation 9 is Equation 7. Note that Steps 1 and 2 are unconditional.

Step 3: ε is controlled by D_{fid}. We assume the *coordinate-wise* condition $D_i \le 2(1-1/m)^2$ for each $i \in U$, rather than the cumulative condition $D_{\mathrm{fid}} \le 2(1-1/m)^2$. The latter would force the average per-position error below $2/|U|$ nats and so would void the certificate precisely when commits are large, which is the regime of interest. Let $D_i = D_{\mathrm{KL}}(p_i^\star \| q_i)$ and $\eta_i = \mathrm{TV}(p_i^\star, q_i)$, so $D_{\mathrm{fid}} = \sum_{i\in U} D_i$. The sharpened Fannes inequality (Audenaert, 2007) states that for distributions on an alphabet of size m with total variation $\eta \le 1 - 1/m$, $|H(p) - H(q)| \le f(\eta)$. Pinsker's inequality gives $\eta_i \le \sqrt{D_i/2}$, and the hypothesis $D_i \le 2(1-1/m)^2$ gives $\sqrt{D_i/2} \le 1 - 1/m$ directly, so the inequality applies at each position. It applies in the useful direction because f is non-decreasing on $[0, 1-1/m]$: $f'(\eta) = \log(m-1) + \log\frac{1-\eta}{\eta}$ vanishes exactly at $\eta = 1-1/m$ and is positive to its left. (Note that h alone is not monotone on this interval when $m > 2$; it is the sum that is.) Hence $|H(p_i^\star) - H_i| \le f(\sqrt{D_i/2}) = \omega(D_i)$, and

$$\varepsilon \;=\; \sum_{i\in U} \big|H(p_i^\star) - H_i\big| \;\le\; \sum_{i\in U} \omega(D_i) \;\le\; |U|\,\omega\Big(\tfrac{1}{|U|}\textstyle\sum_{i\in U} D_i\Big) \;=\; |U|\,\omega\Big(\tfrac{D_{\mathrm{fid}}}{|U|}\Big), \tag{11}$$

the second inequality by Jensen. This is legitimate because ω is concave, being an increasing concave function (f) of a concave one ($x \mapsto \sqrt{x/2}$); it is increasing for the same reason, and $\omega(0) = f(0) = 0$. The argument $D_{\mathrm{fid}}/|U| = \frac{1}{|U|}\sum_i D_i \le \max_i D_i$ lies in $[0, 2(1-1/m)^2]$ by hypothesis, so Jensen applies on the interval where ω is concave. □

The bound $\sum_i H_i - \max_i H_i$ is the criterion of EB-Sampler (Ben-Hamu et al., 2025), who use model entropies in place of true ones as an approximation valid when model error is negligible. The budget $\sum_{i\in U} H_i \le \delta$, which our `entropy` commit rule uses and Cai & Li (2026) analyse, is a weaker bound on D_{dep} and hence a stricter commit criterion; it follows since $\max_i H_i \ge 0$. What Steps 2 and 3 add is to make that approximation explicit: substituting model entropies costs at most ε, which vanishes with the fidelity error at a rate set by the *average* fidelity error per committed position, $D_{\mathrm{fid}}/|U|$.

Remark A.4 (Where the certificate is weakest). The slack $|U|\,\omega(D_{\mathrm{fid}}/|U|)$ increases in both arguments, so it is largest where commits are large and the model is poor, a regime that often corresponds to small T. The per-position hypothesis $D_{\mathrm{KL}}(p_i^\star \| q_i) \le 2(1-1/m)^2$ is likewise most likely to fail there, since it must hold at every committed position. This is a limitation of the certificate rather than of the decomposition: Equation 2 holds unconditionally, and so do Steps 1 and 2 above. A forced commit, the single position revealed when no candidate meets the budget (Appendix F.3), has $|U| = 1$ and therefore $D_{\mathrm{dep}} = 0$ identically, so it never violates the certificate.

A.4 Trajectory decomposition and the price of a finer schedule

Theorem A.5 (Trajectory decomposition and schedule-dependent lower bound)**.**

$$D_{\mathrm{KL}}(P\|Q) \;=\; \mathcal{D}_{\mathrm{dep}}(\mathcal{U}) \;+\; \sum_t \mathbb{E}_{s_t\sim P}\big[D_{\mathrm{fid}}\big(U_t \mid s_t, q^{(\ell_t)}\big)\big] \;\ge\; \mathcal{D}_{\mathrm{dep}}(\mathcal{U}), \tag{12}$$

where $\mathcal{D}_{\mathrm{dep}}(\mathcal{U}) = \sum_t \mathbb{E}_{s_t\sim P}[D_{\mathrm{dep}}(U_t \mid s_t)]$. For a fixed schedule $\mathcal{U}$, this term is determined by $p^\star$ and $\mathcal{U}$ and contains no learned marginals q. If the schedule is selected adaptively from model predictions, q can affect it indirectly through $\mathcal{U}$.

Trajectory decomposition and the floor. Both P and Q are Markov chains on the finite space of partial assignments, started at the same s_0, and at every state they fill the *same* set of positions $U(s_t)$, differing only in the conditional law used to fill it. The map $(s_t, x_{U_t}) \mapsto s_{t+1}$ is injective for fixed s_t, so the divergence between the one-step transition kernels is the divergence between the laws of x_{U_t}. The chain rule for KL divergence between two Markov chains with a common initial state gives

$$\begin{aligned} D_{\mathrm{KL}}(P\|Q) &= \sum_t \mathbb{E}_{s_t\sim P}\Big[D_{\mathrm{KL}}\big(P(\cdot \mid s_t)\,\big\|\,Q(\cdot \mid s_t)\big)\Big] \\ &= \sum_t \mathbb{E}_{s_t\sim P}\Big[D_{\mathrm{KL}}\Big(p^\star(x_{U_t} \mid s_t)\,\Big\|\,\prod_{i\in U_t} q_i^{(\ell_t)}\Big)\Big]. \end{aligned} \tag{13}$$

This is the trajectory form of Ben-Hamu et al. (2025), whose reference process q_ϕ is our P; with $q_i^{(\ell_t)} = p_i^\star$ it reduces to the dependence-only decomposition of Zhang et al. (2026a). Adaptivity is not an obstacle to the identity: $U(\cdot)$ and $\ell(\cdot)$ are deterministic functions of the state, so conditionally on s_t both chains commit at the same locations with the same loop count, and no policy-action distribution enters. Applying Lemma 4.1 to each summand (legitimate by Remark A.2) splits it into a dependence and a fidelity part; collecting the former into $\mathcal{D}_{\mathrm{dep}}(\mathcal{U})$ gives Equation 12. Dropping the non-negative fidelity terms gives the lower bound. The qualification in the theorem matters when comparing different loop budgets: a prediction-dependent commit policy may produce different schedules at different L, even if $|U_t|$ is fixed. □

Lemma A.6 (Splitting a commit)**.** *For any state s and any partition $U = A \sqcup B$ into non-empty blocks,*

$$D_{\mathrm{dep}}(U \mid s) \;=\; D_{\mathrm{dep}}(A \mid s) \;+\; \mathbb{E}_{x_A}\big[D_{\mathrm{dep}}(B \mid s_A)\big] \;+\; \sum_{i\in B} I\big(x_i; x_A \mid s\big), \tag{14}$$

where s_A is the state after committing $x_A \sim p^\star(\cdot \mid s)$ and all information quantities are taken under $p^\star(\cdot \mid s)$. Every term on the right is non-negative, so splitting a commit never increases the dependence cost.

Proof. Fix $s = (x_O, M)$ and $U = A \sqcup B$ with $A, B \neq \emptyset$. All entropies below are under $p^\star(\cdot \mid s) = \pi_{\mathrm{data}}(\cdot \mid x_O)$, and s_A denotes the state after committing x_A, so that $p^\star(\cdot \mid s_A) = \pi_{\mathrm{data}}(\cdot \mid x_O, x_A)$. Averaging the definition of D_{dep} at s_A over $x_A \sim p^\star(\cdot \mid s)$ turns the entropies into conditional entropies,

$$\mathbb{E}_{x_A}\big[D_{\mathrm{dep}}(B \mid s_A)\big] = \sum_{i\in B} H(x_i \mid x_A) - H(x_B \mid x_A). \tag{15}$$

Adding $D_{\mathrm{dep}}(A \mid s) = \sum_{i\in A} H(x_i) - H(x_A)$ and using the chain rule $H(x_A) + H(x_B \mid x_A) = H(x_U)$,

$$D_{\mathrm{dep}}(A \mid s) + \mathbb{E}_{x_A}\big[D_{\mathrm{dep}}(B \mid s_A)\big] = \sum_{i\in A} H(x_i) + \sum_{i\in B} H(x_i \mid x_A) - H(x_U). \tag{16}$$

Subtracting this from $D_{\mathrm{dep}}(U \mid s) = \sum_{i\in A} H(x_i) + \sum_{i\in B} H(x_i) - H(x_U)$ leaves $\sum_{i\in B}\big[H(x_i) - H(x_i \mid x_A)\big] = \sum_{i\in B} I(x_i; x_A)$, which is Equation 14. Non-negativity of the three terms is that of TC and of mutual information. □

Corollary A.7 (Refining a schedule lowers the floor)**.** *Let $\mathcal{U}$ be a commit schedule and let $\mathcal{U}'$ be obtained from it by replacing U_t with the two consecutive commits A and B, $A \sqcup B = U_t$, leaving the rest unchanged. Then*

$$\mathcal{D}_{\mathrm{dep}}(\mathcal{U}) - \mathcal{D}_{\mathrm{dep}}(\mathcal{U}') \;=\; \mathbb{E}_{s_t\sim P}\Big[\sum_{i\in B} I\big(x_i; x_A \mid s_t\big)\Big] \;\geq\; 0. \tag{17}$$

Consequently $\mathcal{D}_{\mathrm{dep}}$ is non-increasing under refinement; it equals $\mathrm{TC}\big(\pi_{\mathrm{data}}(x_M \mid x_O)\big)$ for one-shot decoding ($T{=}1$, $U_1 = M$) and 0 when every commit is a singleton.

Proof. Under P, drawing $x_U \sim p^\star(\cdot \mid s_t)$ and drawing $x_A \sim p^\star(\cdot \mid s_t)$ followed by $x_B \sim p^\star(\cdot \mid s_A)$ induce the same law on x_U, hence the same law on s_{t+1}. The steps before t and, since the remaining rules are functions of the state, the steps after t therefore contribute identically to $\mathcal{D}_{\mathrm{dep}}$ under $\mathcal{U}$ and $\mathcal{U}'$, and the difference is the step-t difference, which is $\mathbb{E}_{s_t\sim P}[\sum_{i\in B} I(x_i; x_A \mid s_t)]$ by Equation 14. Non-negativity is that of mutual information. The endpoint values follow from TC evaluated at $U_1 = M$ and, for singletons, from $\mathrm{TC} = 0$ when $|U| = 1$. □

A.5 Scope of the results

Remark A.8 (What is proved and what is not). Theorem A.5 is an identity together with the bound obtained by discarding one non-negative term; it is not an optimality result. It does not prove that additional recursive loops monotonically reduce D_{fid} (that is an empirical question about the trained model, tested in Section 4.2, where D_{fid} falls up to $L{=}5$ but plateaus beyond it on multi-solution boards) and it does not identify an optimal allocation of a fixed NFE budget between T and L, which

would require modeling how D_{fid} varies with ℓ. What it does establish is the conditional asymmetry the paper uses: at a fixed state and commit set, recursion cannot change the dependence term, while a refinement of the commit schedule lowers that term by the amount made explicit in Equation 14.

Remark A.9 (KL versus task metrics). By Remark A.1, $D_{\mathrm{KL}}(P\|Q)$ is the divergence between the sampler's output law and $\pi_{\mathrm{data}}(\cdot \mid x_O)$. The metrics we report (VPR, RTR, execution accuracy, structural gap ΔZ) are task functionals and are not monotone in that divergence (in particular, forward KL can remain moderate while factorized sampling produces frequent jointly impossible samples (Zhang et al., 2026a)). A decoder can raise validity while losing diversity, and the entropy budget of Section 5.1 trades exactly along that axis. We therefore use task metrics to test the qualitative predictions of Section 4.1 rather than as estimators of either error term. The parallel-clash rate of Section 4.2 is the closest task-level proxy for D_{dep}, since it counts inconsistencies attributable specifically to simultaneous commitment. When $p^\star$ is spread and loops have reduced D_{fid}, it is a useful proxy (83–88% of parallel clashes at $L{=}5$ are dependence errors), but it is not an estimator: fidelity error also produces parallel clashes, and produces all of them on unique boards, where $D_{\mathrm{dep}} \equiv 0$.

B Extended Related Work

Section 2 covers the threads most directly relevant to our contribution. This appendix broadens the context in three parts: the wider discrete-diffusion landscape (Appendix B.1); decoding strategies for masked diffusion, including token ordering, adaptive denoising schedules, and remasking and iterative refinement (Appendix B.2); and looped or fixed-point computation (Appendix B.3).

B.1 The Landscape of Discrete Diffusion for Language

The discrete diffusion family has grown rapidly. The D3PM framework (Austin et al., 2021) provided the absorbing-state (masking), uniform, and nearest-neighbor transition variants inspired from continuous settings (Sohl-Dickstein et al., 2015; Ho et al., 2020; Song et al., 2021), while SEDD (Lou et al., 2024) proposed score-based training for discrete spaces. Within this broader class, masked (absorbing-state) discrete diffusion (MDMs) have received particular attention. In contrast to AR models, which generate tokens in a fixed left-to-right order, MDMs iteratively denoise a partially masked sequence using bidirectional context. This flexible generation order may benefit tasks requiring global or non-sequential reasoning (Ye et al., 2025a; He et al., 2026), while the ability to update multiple tokens per step creates opportunities for faster inference. It also introduces an inference-time scaling axis absent from standard autoregressive decoding: increasing the number of denoising steps gives the model additional opportunities to resolve dependencies among tokens, at the cost of greater computation. In this work, we investigate recursive computation within each MDM denoising step as a complementary axis of inference-time scaling.

Among masked approaches, MDLM (Sahoo et al., 2024) demonstrated strong language modeling performance with a simplified objective, and LLaDA (Nie et al., 2025) scaled masked diffusion to billions of parameters. Finally, more recent work including DiffusionGemma (DiffusionGemma Team, 2026) further demonstrates the practical appeal of block-parallel masked diffusion for high-throughput generation. Subsequent scaling law studies (Sahoo et al., 2026) challenge the assumption that masked diffusion is the uniquely best discrete noise type, finding that uniform diffusion may be more parameter-efficient at scale. Continuous–discrete hybrids have also been proposed (Zheng et al., 2026; Lee et al., 2026a; Zhou et al., 2025), but tend to suffer from trainability issues in the continuous space. This supports our decision to remain in the masked discrete setting, where training is stable and the parallel denoising structure makes bidirectional recursion most natural.

B.2 Decoding Strategies for Masked Diffusion

A central characteristic of MDMs is that training does not prescribe a unique inference order. Consequently, a substantial literature studies how to choose which masked tokens to reveal and how many to reveal at each iteration.

Confidence- and entropy-based token ordering. MaskGIT (Chang et al., 2022) is an early and influential example of iterative masked generation in which token confidence determines which po-

sitions are progressively revealed. More recent work makes this idea explicit for MDMs. Kim et al. (2025) theoretically and empirically study token ordering in masked diffusion and show that adaptive ordering can substantially improve performance on structured tasks. EB-Sampler (Ben-Hamu et al., 2025) uses an entropy-bounded rule to dynamically determine how many tokens can be unmasked in a single model evaluation while controlling an approximate error tolerance. Confidence-based decoding has also received a formal efficiency analysis: Cai & Li (2026) analyze an entropy-sum unmasking strategy and provide iteration-complexity guarantees in terms of the entropy of the target distribution.

These methods are directly relevant to our use of entropy. Their primary role for entropy is to determine *which tokens are sufficiently certain to commit* or *how many tokens can safely be committed*. Our work extends this perspective to a second decision: whether the current state is ready for commitment or whether another recursive computation should first be performed.

Planning and alternative token orders. Several works frame decoding as an explicit planning problem. Path Planning (Peng et al., 2025) introduces a planner for choosing which tokens to unmask and shows that learned or model-based planning can improve over standard unmasking rules. Another approach separates planning from denoising and allows the planner to identify positions requiring further refinement (Liu et al., 2025). These approaches emphasize that the order of commitments can have a substantial effect on the final sample, complementary to our focus on whether additional computation should occur before a commitment.

Position and interaction-aware decoding. Recent work also points out limitations of purely token-wise confidence scores. UNCODE (Huang et al., 2026b) studies position-dependent biases in uncertainty-based decoding and reports a tendency for trivial positions to be selected early. ADAS (Sahin et al., 2026) explicitly accounts for interactions among candidate tokens by discounting candidates that attend strongly to uncertain positions already selected for decoding. This is closely related to our interest in inter-token dependencies, but ADAS addresses the problem through attention-aware reranking of the commit set, whereas we study whether recursive computation can resolve those dependencies before commitment.

Inference Schedules and Information-Theoretic Decoding The number and placement of denoising steps constitute another major axis of MDM inference. Li & Cai (2025) provide information-theoretic convergence guarantees in which sampling error depends on token mutual information. Chen et al. (2026) give an exact characterization of sampling error for arbitrary unmasking schedules and derive schedules based on total correlation and dual total correlation. These results provide a formal perspective on the amount of parallelism that can be safely exploited when tokens are dependent.

Other work optimizes the discretization schedule directly by optimizing discrete diffusion timesteps using a bound on compounding decoding error (Park et al., 2025). Zhang & Syed (2025) provide an information-geometric analysis showing that the cosine schedule is Fisher–Rao optimal for masked discrete diffusion. These approaches primarily optimize the *outer diffusion schedule*; our setting introduces an additional inner computation axis and asks how compute should be allocated between another recursive pass and an additional denoising transition.

Remasking, Predictor–Corrector Methods, and Refinement A complementary family of approaches addresses the irreversibility of standard MDM decoding by allowing previously generated tokens to be revised. ReMDM (Wang et al., 2025) introduces inference-time remasking so that tokens can be unmasked again when later context indicates that an earlier commitment was unreliable. WINO (Hong et al., 2026) similarly enables revokable decoding through parallel drafting, verification, and re-masking of suspicious tokens. These approaches provide a form of *post-commitment correction*.

Our recursive mechanism is conceptually different: recursion operates *before* the current set of tokens is committed, repeatedly refining the same partially masked state. Thus, remasking asks whether an existing commitment should be revoked, whereas our decoding policy asks whether commitment should be postponed in favor of additional computation. Predictor–corrector and planned-denoising approaches provide related perspectives on separating prediction, planning, and refinement (Zheng et al., 2024; Liu et al., 2025; Peng et al., 2025).

In other lines of work, Chao et al. (2025) explore partial masking strategies that interpolate between absorbing and uniform noise, while Fox et al. (2026) learn generation orders via variational inference. Finally, Horvitz et al. (2026) also study inference-time compute allocation in MDMs and investigate entropy-based decoding choices alongside reasoning-oriented infilling. It is particularly relevant as an empirical study of the interaction between decoding strategy and reasoning performance, including structured reasoning tasks. Our focus is narrower: we study entropy and hidden-state change as signals for allocating compute between recursive refinement and denoising.

Efficient and Adaptive MDM Samplers A growing set of training-free samplers targets the speed–quality trade-off created by parallel MDM decoding. Fast-dLLM (Wu et al., 2026) combines KV-cache reuse with parallel decoding and identifies disrupted token dependencies as a key source of quality degradation. TABES (Saini et al., 2026) uses a backward entropy signal to estimate how revealing a candidate token will affect the future entropy of the remaining masked positions, thereby introducing a form of lookahead into token selection. These methods are closely related to our use of uncertainty and future effects, but operate primarily at the token-selection level. Fast solvers for discrete diffusion models (Ren et al., 2025) instead reduce the number of numerical evaluations through higher-order integration schemes. These methods are complementary to our work: they accelerate the underlying diffusion trajectory or reduce the cost of individual evaluations, whereas our objective is to understand how to spend a fixed evaluation budget across recursive and denoising computation.

B.3 Recursive and Fixed-Point Computation

Weight sharing across depth has a long history in Transformer architectures, from Universal Transformers and ALBERT (Dehghani et al., 2019; Lan et al., 2020) to recent looped Transformers that trade parameter count for repeated computation (Giannou et al., 2023; Yang et al., 2024; Saunshi et al., 2025; Geiping et al., 2025; Yu et al., 2026; Xu & Sato, 2026). Complementary analyses study what such recurrence actually buys: LT2 interprets looped attention as iterative memory refinement and receptive-field expansion (Deng et al., 2026), DeepLoop examines the optimization and stability consequences of repeatedly revisiting shared parameters (Li et al., 2026), and Zhang et al. (2026b) ask whether weight-tied recurrence implements an iterative algorithm with identifiable convergence and propagation behavior. Closely related fixed-point reasoners adapt computation by monitoring hidden-state convergence on structured tasks (Movahed et al., 2026), while self-conditioned flow maps (Yoo et al., 2026) and Recursive Flow Matching (Huang et al., 2026a) cast self-consistency as fixed-point iteration in continuous generative settings. Collectively, these works support treating recursion as a computational dimension distinct from parameter scaling. In AR models, however, that extra computation is typically spent before emitting the next token; masked diffusion raises a qualitatively different allocation problem, because each unit of compute can either refine the current partially masked state or advance the outer denoising trajectory by committing tokens.

Within diffusion and masked generation, two concurrent works are closest to ours. LoopMDM selectively loops Transformer layers inside an MDM and shows that additional loops improve performance, including by allocating loops adaptively across denoising timesteps (Lee et al., 2026b); its analysis also reports increased interactions among masked positions. Fixed-Point Masked Generative Modeling instead embeds a fixed-point solver into masked generation and uses adaptive depth together with cross-step consistency (Miele et al., 2026). Both establish that repeated computation is useful for masked generation. Our contribution differs in what is treated as the primary object of study. Rather than asking only whether more loops help, we ask how recursive refinement interacts with token commitment: loops update model marginals while the masked state is held fixed, whereas denoising steps change that state by revealing tokens. This distinction motivates the dependence–fidelity decomposition in Section 4 and the two-axis decoder in Section 5, which jointly choose *how many* recursive evaluations to run and *which* tokens to commit. Architecturally, we also make different design choices: we loop the entire shared transformer block rather than selected layers, which enables deep supervision across loops, and we inject an explicit recursive step embedding so the shared block is conditioned on loop progress. In short, concurrent work shows that recursion is a viable compute axis inside MDMs; we characterize when that axis is interchangeable with denoising, when it is not, and how to allocate both jointly at inference time.

C Dataset and Metrics

We evaluate our models on a diverse set of datasets spanning structured combinatorial reasoning (Sudoku), symbolic arithmetic planning (Countdown), biological sequence modeling (RNA), and natural-language arithmetic reasoning (TinyGSM). This selection allows us to test different aspects of sequence modeling: constraint satisfaction, multi-step symbolic reasoning with intermediate state tracking, biological sequence generation, and programmatic mathematical reasoning. For each dataset, we construct train and validation splits and report both training objectives and task-specific evaluation metrics. Further information on split sizes, vocabulary sizes, and sequence lengths are detailed in Table 1.

For all datasets during training, we monitor the average masked cross-entropy loss over mini-batches, computed for both the train and validation splits by averaging over `eval_iters` randomly sampled batches at the end of each evaluation interval. For a batch $\{(x_t^{(i)}, x_0^{(i)}, m^{(i)})\}_{i=1}^{B}$ of size B, the estimated validation loss is

$$\hat{\mathcal{L}}_{\text{val}} = \frac{1}{B} \sum_{i=1}^{B} \mathcal{L}(y^{(i)};\, x_0^{(i)},\, m^{(i)}), \tag{18}$$

where $\mathcal{L}$ is whichever training objective is active (depending on the choice of supervision, see Appendix D.4).

Table 1: Summary statistics for all datasets used in our experiments. Sequence length refers to the fixed context length used during training. Vocabulary size includes special tokens such as masks where applicable.

Dataset	Train Size	Val Size	Vocab Size	Seq Length
Sudoku 9×9	1.8M	100k	10	81
Sudoku 25×25	100k	20k	26	625
Countdown ($k = 3$)	100k	20k	18	48
Countdown ($k = 4, 5$)	100k	20k	18	64
RNA	38.8M	125k	25	256
TinyGSM	11.7M	118k	49,154	512

C.1 Sudoku

Sudoku 9×9 For the standard Sudoku, we adapt the dataset from Shah et al. (2024), which consists of pre-existing puzzle files containing solved 9×9 puzzles encoded as move sequences. Each sample in the raw data is a sequence of 81 moves, where each move records a quadruple (r, c, v, s) that indicates the row index, column index, digit value, and a strategy tag respectively. Since we are interested in training on the raw Sudokus, we preprocess the data and store the boards row-wise as flattened sequences of length 81 with digits in $\{1, \ldots, 9\}$. The vocabulary consists of digits $\{0, 1, \ldots, 9\}$, where 0 serves as the mask token.

Extended sudoku For experiments at larger scales, we construct a synthetic dataset of $n \times n$ Sudoku boards, where n is chosen such that a valid rectangular block decomposition exists. Specifically, we require $n = b_r \times b_c$ for integers $b_r, b_c \geq 2$ chosen as the factor pair closest to $\sqrt{n}$. Each board is a valid Sudoku with digits in $\{1, \ldots, n\}$, stored as a flattened sequence of length n^2.

Boards are generated by first constructing a single deterministic base solution using the closed-form pattern

$$G_{r,c} = (b_c \cdot (r \bmod b_r) + \lfloor r/b_r \rfloor + c) \bmod n + 1,$$

which yields a valid completed $n \times n$ grid for any compatible block shape. Each subsequent sample is derived from this base by applying a random symmetry-preserving permutation: row-bands and rows within each band are independently shuffled, column-stacks and columns within each stack are independently shuffled, and digit labels are remapped via a random permutation of $\{1, \ldots, n\}$. All three operations preserve Sudoku validity. The vocabulary consists of digits $\{0, \ldots, n\}$, with 0 again serving as the mask token, giving a vocabulary size of $n + 1$.

We generate 100,000 boards for training and 20,000 for validation with $n = 25$. A fixed set of binary blank masks is pre-generated for the validation split, where each mask is drawn i.i.d. with a per-cell blank probability (0.7, 0.8 or 0.9), subject to the constraint that at least one cell is blank and at least one is given.

Evaluation protocol Sudoku evaluation is conditional infilling on held-out boards: the given cells (clues) remain visible, while blank cells are replaced by the mask token and filled by the decoder. For 9×9, blanks follow the dataset's strategy annotations (non-clue cells); for 25×25, validation blanks are sampled independently.

Metrics and evaluation Here n denotes the grid side length and the board has n^2 cells; b_r and b_c denote the row and column dimensions of each subgrid block, with $n = b_r b_c$.

- **Valid Puzzle Rate (VPR)**: A generated $n \times n$ board $y \in \{1, \ldots, n\}^{n^2}$ is valid if and only if every row, every column, and every $b_r \times b_c$ block contains each digit in $\{1, \ldots, n\}$ exactly once. Any out-of-range digit, repeated digit in a unit, or board of incorrect length counts as invalid.
- **Soft Constraint Loss (SCL)**: This metric is adapted from He et al. (2026) and allows a more fine-grained evaluation of the quality of the generated sudokus. For a board $y \in \mathbb{Z}^{n^2}$ with vocabulary $\mathcal{V} = \{1, \ldots, n\}$, it measures the fractional coverage of the required digit set for each constraint unit:

$$\ell(u;\, y) = 1 - \frac{|\,\{y_j : j \in u\} \cap \mathcal{V}\,|}{n}, \tag{19}$$

  where u ranges over all $3n$ units (rows, columns, blocks). A perfectly valid board has $\ell(u;\, y) = 0$ for all u; a board where every unit contains only one distinct valid digit has $\ell(u;\, y) = (n-1)/n$ for all u. The maximum value $3(n-1)$ corresponds to a board where every unit contains a single repeated in-vocabulary digit. The board-level metric is $\mathrm{SCL}(y) = \sum_u \ell(u; y)$ over the $3n$ rows, columns, and blocks. Mask and out-of-vocabulary tokens are absent from the covered set, and therefore $0 \le \mathrm{SCL}(y) \le 3(n-1)$.

C.2 Countdown

Dataset The Countdown dataset (Yao et al., 2023; Gandhi et al., 2024) is a synthetic arithmetic reasoning task designed to evaluate a model's ability to perform multi-step planning, maintain a dynamic state (the "pool"), and execute precise arithmetic operations. The task requires generating a sequence of operations that transform a set of initial numbers into a specific target value.

Each sample is generated procedurally to ensure the existence of at least one valid solution. The generation follows the logic:

1. **Initialization:** A set of k initial operands $\mathcal{N} = \{n_1, n_2, \ldots, n_k\}$ is sampled uniformly from the range $[1, 100]$. In our experiments, $k = 3, 4, 5$.
2. **Chain Construction:** The algorithm iteratively reduces the pool of numbers until a single value remains. In each step, two numbers a and b are sampled from the current pool and combined using an operator $\odot \in \{+, -, *, /\}$.
3. **Validation:** Operations must result in integers. For division, it is required that $b \neq 0$ and $a \equiv 0 \pmod{b}$.
4. **Target Assignment:** The final remaining number in the pool after $k-1$ steps is designated as the target τ.

Each example is represented as a fixed-length character sequence of length $N_{seq} = 48$ for $k = 3$; and $N_{seq} = 64$ for $k = 4, 5$. The dataset uses a restricted vocabulary of 18 tokens: digits 0–9, operators $+$, $-$, $*$, $/$, and delimiters $=$, ,, and $\backslash n$. The mask token is _.

Evaluation protocol Countdown is prefix-conditioned infilling: the operand list and target stay visible, and only the arithmetic completion region is masked.

Metrics and evaluation

- **Reaches Target Rate (RTR):** We consider a solution *valid* if and only if: (i) each step is arithmetically correct ($a \odot b = c$ evaluates correctly); (ii) at each step both operands a and b are available in the current pool, which is initialized to $\mathcal{N}$ and updated by removing a and b and inserting c after each step; and (iii) after all steps the pool contains exactly one element equal to τ. The Reaches Target Rate is then:

$$\mathrm{RTR} = \frac{1}{N}\sum_{i=1}^{N} \mathbf{1}\Big[y^{(i)} \text{ is a valid Countdown solution}\Big]. \tag{20}$$

- **Local Arithmetic Fraction (LAF):** LAF measures the fraction of generated steps whose arithmetic is locally correct, regardless of whether the operands were available in the pool at that point in the solution:

$$\mathrm{LAF}(y) = \frac{1}{|S|}\sum_{s\in S} \mathbf{1}[\text{parse}(s) \neq \varnothing \text{ and } a \odot b = c \text{ is numerically correct}], \tag{21}$$

  where S is the set of step strings in y that match the expected format $a \odot b = c$. LAF distinguishes between arithmetic errors (low LAF, low RTR) and correct arithmetic with pool-management errors (high LAF, low RTR).

- **Pool Prefix Fraction (PPF):** PPF measures how far through a *pool-consistent* simulation the model gets before its first failure, normalized by the expected number of steps $k-1$ where $k = |\mathcal{N}|$:

$$\mathrm{PPF}(y) = \frac{\max\{j : \text{steps } s_1, \ldots, s_j \text{ are all valid and pool-consistent}\}}{k-1} \in [0,1]. \tag{22}$$

  A model that always fails on the first step has PPF $= 0$; RTR $= 1$ implies PPF $= 1$. PPF tracks progress toward the solution and is particularly informative at intermediate loop counts.

- **Target Residual Norm (TRN):** After following the longest valid pool-consistent prefix of the generated solution, TRN measures how close the remaining pool values are to the target τ:

$$\mathrm{TRN}(y) = \frac{\min_{x\in\mathcal{P}^*} |x-\tau|}{\max(1,\,|\tau|)} \in [0,\,+\infty), \tag{23}$$

  where $\mathcal{P}^*$ is the pool state after the longest pool-consistent prefix. TRN $= 0$ means the correct answer appears somewhere in the pool (a necessary condition for RTR $= 1$); TRN > 0 quantifies how far the best available value is from the target, scaled by the magnitude of the target to be comparable across instances. An unparseable or entirely invalid chain gives TRN $= 1.0$ by convention.

C.3 RNA

Dataset The dataset corpus is built from RNAcentral (The RNAcentral Consortium, 2026) (39,048,508 sequences). Removing exact duplicates leaves 39,035,267 sequences, each carrying the RNA type assigned by RNAcentral. We retain the nineteen most frequent types as families and collapse the remainder into a single *other* family, giving 20 families in total. Held-out sets are drawn per family with a fixed quota rather than proportionally: 10,000 sequences for each of rRNA, lncRNA, tRNA, misc_RNA and *other*, and 5,000 for the remaining fifteen. This yields 125,000 sequences per held-out split and leaves 38,785,267 for training. The quota makes the held-out sets quasi-uniform over families, so validation loss is not dominated by ribosomal RNA, which accounts for 72.5% of the training pool.

The RNA dataset therefore consists of non-coding RNA sequences stored as per-family HuggingFace Arrow datasets, with separate train and validation directories. The training split contains 38,785,267 sequences and the validation split contains 125,000 sequences, distributed across 20 RNA families. During training, family streams are interleaved with temperature-scaled sampling probabilities $p_f \propto n_f^{\tau}$, where n_f is the number of sequences in family f and we use $\tau = 0.7$. This

damped sampling rule interpolates between uniform family sampling ($\tau = 0$) and size-proportional sampling ($\tau = 1$).

Sequences are uppercased and tokenized with a RiNALMo-style (Penić et al., 2025) RNA vocabulary of size 25: the special tokens <pad>, <unk>, <mask>, <cls>, <eos>, and the 20 sequence symbols A C G U N R Y S W K M B D H V . X * - I. Unless otherwise stated, each sequence is wrapped with <cls> and <eos>, truncated if necessary, and padded to a fixed length of 256 tokens. Masking is applied only to non-special tokens. For each sequence, the masking probability is sampled uniformly from $[0, 1]$, selected positions are replaced by the <mask> token.

Metrics and evaluation All metrics compare an infilled sequence s^{inf} to its held-out native counterpart s^{nat}. Both sequences have the same length and agree on every unmasked position. We write $\mathcal{M}$ for the set of masked positions.

- **Structural Z-score:** Secondary-structure stability is evaluated with ViennaRNA folding. Let $\text{MFE}(s)$ be the predicted minimum free energy of s in kcal/mol. To control for nucleotide composition, we compare it to 20 mononucleotide-shuffled copies of s:

$$Z(s) = \frac{\text{MFE}(s) - \mu_{\text{shuffle}}(s)}{\sigma_{\text{shuffle}}(s)}. \tag{24}$$

  More negative values mean the sequence folds to a lower energy than its composition alone would predict.

- **Paired Structural Gap (ΔZ):** The primary RNA infilling metric:

$$\Delta Z(s) \;=\; Z(s^{\text{inf}}) \;-\; Z(s^{\text{nat}}), \tag{25}$$

  averaged over the evaluation set. Z is already standardized against length- and composition-matched shuffles, and the difference is taken against each sequence's *own* native fold. As a result, ΔZ is comparable across lengths and can be averaged over length strata. Lower is better. $\Delta Z = 0$ means the completion is as structured as the native molecule, and $\Delta Z > 0$ means it folds less tightly than the original.

- **Masked Recovery:** The fraction of masked positions whose nucleotide matches the native sequence,

$$\text{Rec}(s) = \frac{1}{|\mathcal{M}|} \sum_{t \in \mathcal{M}} \mathbf{1}\left[s_t^{\text{inf}} = s_t^{\text{nat}}\right]. \tag{26}$$

  Uniform guessing over $\{\text{A}, \text{C}, \text{G}, \text{U}\}$ gives 25%. We report this metric as a diagnostic, not as a target: values well above chance show that the completion is consistent with its context, while values close to 100% would mean the model copies the native molecule instead of designing a new one.

- **Exact Recovery:** The fraction of evaluated sequences in which *every* masked position is restored to the native nucleotide, i.e. $\mathbf{1}[\text{Rec}(s) = 1]$ averaged over the evaluation set. Lower is better: a low exact-recovery rate shows that good ΔZ does not come from reproducing the validation set.

Infilling evaluation protocol The RNA results of Section 5.2 are conditional infilling on the validation split, not unconditional design. For each evaluated sequence of length n, exactly $\text{round}(\rho n)$ nucleotide positions are masked uniformly at random and decoded. The remaining positions, together with the special and padding tokens, stay fixed, so the model restores the blanks in the context of the native molecule. The main text uses a mask fraction of $\rho = 40\%$, and Appendix G.7 also reports $\rho \in \{50\%, 60\%\}$. Sampling is restricted to the main nucleotides $\{\text{A}, \text{C}, \text{G}, \text{U}\}$, under the decoders and hyperparameters of Appendix F. We stratify by length and draw 500 held-out sequences at each of $n \in \{76, 110, 254\}$ nucleotides. These are decoded at $T = 6$, 9 and 20 respectively, so that at $\rho = 40\%$ every stratum commits roughly five positions per denoising step. The sequences and masked positions are fixed by a seed and shared across all decoders and models. We fold both the native and the infilled sequence with ViennaRNA to obtain ΔZ. Because ΔZ is length-normalized by construction, the three strata are averaged into the single RNA panel of Figure 8. Per-decoder values for all mask fractions are in Table 14.

C.4 TINYGSM

Dataset TinyGSM (Liu et al., 2023) is a large synthetic grade-school mathematics dataset consisting of natural-language questions paired with executable Python solutions. We use the HuggingFace `TinyGSM/TinyGSM` training split. After filtering examples that exceed the model context length, the resulting split contains 11,703,733 training examples and 118,220 validation examples.

Each example is tokenized with the `HuggingFaceTB/SmolLM-135M` tokenizer and represented as

`BOS + question + separator + Python solution + EOS`,

then padded to a fixed sequence length of 512. Since the tokenizer does not provide a dedicated mask token, we append an explicit mask token, yielding a vocabulary size of 49,154. Training follows an answer-only masked infilling setup: prompt tokens corresponding to the question are kept fixed, while the solution region is eligible for masking and supervision.

Metrics and evaluation

- **GSM8K Accuracy:** We evaluate TinyGSM models on the `openai/gsm8k` test split. For each problem, the tokenized question and separator are kept fixed and all post-prefix positions are masked, up to the 512-token canvas. The model generates a Python completion, from which we extract and execute a function named `simple_math_problem` in a restricted sandbox. A sample is correct if the returned value, or a numeric answer extracted from its output, matches the GSM8K gold answer:

$$\text{GSM8KAcc} = \frac{1}{N}\sum_{i=1}^{N} \mathbf{1}\left[\text{exec}(y^{(i)}) = a^{(i)}\right]. \tag{27}$$

  Execution failures, missing functions, timeouts, unparseable outputs, or numerically incorrect answers are counted as incorrect.

D TRAINING ALGORITHM AND DESIGN CHOICES

D.1 TRAINING PROCEDURE

Training uses a masked denoising objective (Sahoo et al., 2024). We randomly mask positions within the completion region, and the model is trained to minimize the cross-entropy loss at those positions. Further details are described in Algorithm 1.

Algorithm 1 Training Iteration with Recursive Refinement

Input: Dataset $\mathcal{D}$, Model f_θ, Optimizer $\mathcal{O}$, Accumulation Steps A
for iteration $it = 1$ **to** I_{max} **do**
  $\mathcal{O}.zero_grad()$
  $\mathcal{L}_{accum} = 0$
  for micro-step $a = 1$ **to** A **do**
    Sample batch $(\mathbf{x}, \mathbf{y}, \mathbf{mask})$ from $\mathcal{D}$
    Resolve training loop count L_t (Fixed, Sample, or Schedule)
    logits, $\mathcal{L}, \{\mathcal{L}_k\} \leftarrow f_\theta(\mathbf{x}, \mathbf{y}, \mathbf{mask}, L_t, \text{step_idx} = it)$
    Calculate $\mathcal{L}_{scaled} = \mathcal{L}/A$
    Compute gradients: $\mathcal{L}_{scaled}.backward()$
    $\mathcal{L}_{accum} \leftarrow \mathcal{L}_{accum} + \mathcal{L}.item()$
  end for
  $\mathcal{O}.step()$
  Update Learning Rate Scheduler
end for

Models for Sudoku and Countdown were trained on a single NVIDIA A100-SXM4-80GB GPU. The model for RNA was trained on a single NVIDIA H200-SXM5-141GB GPU, and the model for TinyGSM on 8 NVIDIA H200-SXM5-141GB GPUs.

Across all runs, the base optimizer settings are shared: AdamW with learning rate $\mathrm{LR} = 3 \times 10^{-4}$ and a cosine scheduler with warmup (`warmup_steps`=2000, `min_lr_ratio`=0.1). Further details on the models used are detailed in Table 2.

Table 2: Architecture and optimization settings, sequence length, masking, batch size, and effective training epochs for each dataset. Parameter counts include the step-embedding MLP ($\approx d^2$, i.e., 0.15M for d=384) that is not a part of the non-recursive baselines.

Dataset	n_{layer}	n_{head}	n_{embd}	Params	Seq. len	Masking	Batch size	Epochs
Sudoku 9×9	6	6	384	10.8M	81	Uniform	64	10
Sudoku 25×25	6	6	384	10.8M	625	Uniform	32	75
Countdown	3	12	384	5.5M	48/64	Uniform	512	300
RNA	12	20	480	33.7M	256	Uniform MLM	128	1
TinyGSM	12	12	768	161M	512	Log-linear MDLM	64	1.4

Positional encodings The model uses two positional-encoding regimes. For non-Sudoku datasets, we employ standard 1D rotary positional embeddings (RoPE, (Su et al., 2023)). For Sudoku datasets, we explore a 2D variant: each flattened token index is mapped back to grid coordinates (r, c), rotary frequencies are computed separately for row and column components, and then concatenated so the attention mechanism is aware of both horizontal and vertical structure. In addition, we add a learned *block embedding* identifying each subgrid (e.g., 3×3 blocks in 9×9 Sudoku), which explicitly encodes Sudoku block constraints beyond pure sequence order.

Masking strategies All datasets use a masked denoising objective (Sahoo et al., 2024): a subset of positions is replaced by a mask token and the model is trained with cross-entropy on those positions. The datasets differ in *which* positions are eligible and in *how* the per-sequence mask rate is sampled. For Sudoku, every grid cell is eligible; each sequence draws a mask probability $p \sim \mathrm{Unif}[0, 1)$ and masks cells independently with probability p. Countdown uses the same uniform rate, but only within the completion region after the fixed prompt prefix (padding newlines are never masked). TinyGSM likewise freezes the question/prompt tokens and applies masking only to the answer span (including padding under our default settings), but samples the mask rate from the MDLM *log-linear* noise schedule rather than a uniform draw: $t \sim \mathrm{Unif}(\varepsilon_{\text{samp}}, 1)$ with $\varepsilon_{\text{samp}} = 10^{-3}$, total noise $\sigma(t) = -\log\big(1-(1-\varepsilon)t\big)$ with $\varepsilon = 10^{-3}$, and move chance $1 - e^{-\sigma(t)}$. RNA is closest to classical MLM: special tokens (`<pad>`, `<unk>`, `<mask>`, and optionally `<cls>`/`<eos>`) are excluded, and remaining nucleotide positions are masked with a uniform per-sequence rate $p \sim \mathrm{Unif}[0, 1)$.

Further RNA training details RNA uses the RiNALMo-micro encoder architecture (Penić et al., 2025) rather than the GPT-style backbone used on the other datasets. We adopt only the architecture: the model is trained from scratch. Sequences are fixed to length 256 (including optional `<cls>`/`<eos>`), with truncation/padding as needed. The recursive RNA model was trained with Schedule $5 \to 1$ loop count.

Further TinyGSM training details TinyGSM uses a larger GPT-style transformer over length-512 sequences tokenized with the SmolLM tokenizer. Examples are question–code pairs from TinyGSM; sequences longer than 512 tokens are filtered out. Training freezes the prompt and applies log-linear MDLM masking on the answer region. TinyGSM additionally uses bf16 precision, global gradient clipping at norm 1.0, and EMA of the weights with decay 0.9999. The recursive TinyGSM model model was trained with Schedule $5 \to 1$ loop count.

D.2 Choosing the training loop count L_t

We study several strategies for selecting L_t, the number of recursive loops run per forward pass during training. This is distinct from the loop budget L_s used at sampling time and from the denoising budget T. These strategies control the amount of iterative refinement applied by the model and can be broadly categorized into *fixed*, *stochastic*, and *scheduled* schemes.

Fixed-depth recursion In the fixed-depth setting, the number of recursive iterations is held constant, $L_t = L$, where L is a predefined hyperparameter. This approach provides stable optimization

and predictable computational cost, and is commonly used in looped or recurrent transformer architectures. Notably, Yang et al. (2024) demonstrate that Transformers trained with a fixed number of recurrent loops can improve algorithmic generalization when unrolled for multiple iterations. Similarly, RecursiveVLM (Xu et al., 2026) trains with a fixed recursion depth and evaluates the model at different sampling recursive steps.

Random-depth sampling In the stochastic depth setting, the recursion depth is sampled per example or batch, $L_t \sim \Lambda$, where Λ denotes a predefined probability distribution over positive integers. This formulation exposes the model to a distribution over computational budgets during training, improving robustness to variable inference-time compute. This follows the line of work from Geiping et al. (2025), which samples recurrent iteration counts using log-normal or Poisson-like distributions combined with truncated backpropagation through time. This enables the model to allocate computation adaptively while maintaining efficiency on average. In our implementation, this category includes:

- Uniform distribution: $L_t \sim \mathcal{U}(L_{\min}, L_{\max})$
- Poisson distribution: $L_t \sim \text{Poisson}(\lambda) + 1$, where λ is drawn from a log-normal prior with parameters chosen to match a desired mean depth

Curriculum-based scheduling In curriculum-based approaches, the recursion depth is deterministically varied over the course of training, $L_t = f(i)$ where i denotes the training iteration and $f(\cdot)$ is a monotonic schedule mapping training progress to computation depth. This idea has been widely used in recurrent and looped architectures (Dehghani et al., 2019; Fan et al., 2025). Typically, L_t is gradually increased or decreased from an initial value L_{start} to a final value L_{end}, enabling a curriculum over looping depth:

$$L_t = \text{round}\left(L_{\text{start}} + \alpha(i)(L_{\text{end}} - L_{\text{start}})\right),$$

with $\alpha(i) \in [0, 1]$.

In our implementation, we evaluate both **increasing** and **decreasing linear schedules**, corresponding respectively to $L_{\text{start}} < L_{\text{end}}$ and $L_{\text{start}} > L_{\text{end}}$. This allows us to study whether progressively allocating more or less recursive steps affects downstream performance.

Figure 9 evaluates the impact of different training schemes for the number of recursive steps on the final performance across evaluation depths $L_s \in \{1, 3, 5, 7, 10, 15, 20\}$ and decoding budgets $T \in \{1, 5, 20\}$. We compare a constant baseline (*Fixed* L=5), stochastic sampling methods (*Uniform* and *Poisson*), a standard linear curriculum that increases recursive steps during training (*Schedule* $1 \to 10$), and a reverse linear curriculum (*Schedule* $10 \to 1$). The reverse curriculum (*Schedule* $10 \to 1$) emerges as the superior training strategy by a substantial margin, particularly at small-to-moderate evaluation depths and under tight decoding budgets.

For instance, at T=1 and L_s=5, *Schedule* $10 \to 1$ achieves a 63.6% valid puzzle rate, outperforming the constant baseline *Fixed* L=5 (21.2%), stochastic *Poisson* (21.0%), and the forward curriculum *Schedule* $1 \to 10$ (8.0%). This performance gap is even more pronounced at L_s=7 under T=1, where the reverse curriculum reaches 89.8% compared to just 61.6% for *Fixed* L=5 and 33.2% for *Schedule* $1 \to 10$. Even when more decoding steps are allowed, the reverse curriculum retains its advantage: at T=5 and L_s=3, it reaches 95.6% validity, whereas *Fixed* L=5 and *Schedule* $1 \to 10$ lag behind at 79.4% and 73.8%, respectively. These results suggest that introducing deep recursion early in training forces the model to learn high-capacity, multi-step coordination, while progressively reducing the depth down to 1 fine-tunes its ability to execute highly efficient, single-step updates. Conversely, starting with shallow steps (*Schedule* $1 \to 10$) or sampling them randomly (*Uniform*, *Poisson*) fails to establish the same level of systematic multi-step alignment, leading to sub-optimal performance across all evaluation depths.

D.3 Step embedding ablation

We investigate the role of the recursive step embedding introduced in Section 3. In its basic form, Equation 1 applies f_θ repeatedly with shared parameters, where each loop operates on the evolving hidden state but otherwise receives the same conditioning inputs. The model has no explicit signal indicating which loop is currently being executed; it must infer this from the evolving statistics of

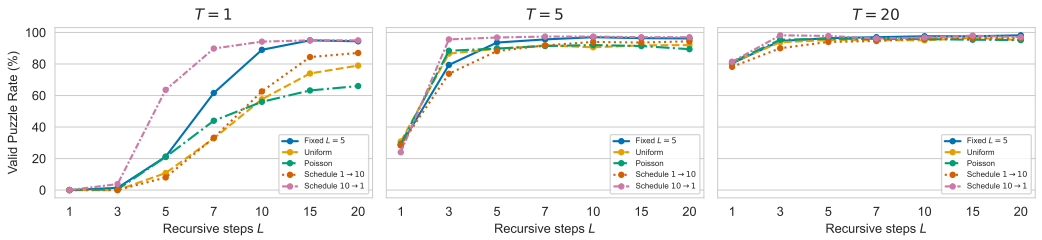


Figure 9: Performance comparison between models trained with different recursive steps choices on 9×9 Sudoku puzzles. We report the Valid Puzzle Rate across varying loop counts at sampling L_s and denoising steps T.

$\mathbf{h}^{(\ell)}$ alone. To give the model direct access to its position in the recursive computation, we introduce a *step embedding*.

Let $s_\ell = (\ell - 1)/(L - 1) \in [0, 1]$ be the normalized loop progress (with $s_1 = 0$ for $L = 1$). At inference, L is the sampling loop budget L_s, which under adaptive stopping is the upper bound on the number of loops. We map this scalar to a d-dimensional vector via a two-layer MLP:

$$\mathbf{v}^{(\ell)} = W_2\, \sigma(W_1\, s_\ell + b_1)\,, \qquad W_1 \in \mathbb{R}^{d\times 1},\ W_2 \in \mathbb{R}^{d\times d}, \tag{28}$$

where σ denotes the SiLU activation. The step vector $\mathbf{v}^{(\ell)}$ is broadcast over the sequence and added to the hidden state before each loop application. The step embedding parameters (W_1, b_1, W_2) are *shared* across loops (they depend on ℓ only through s_ℓ) and add $\mathcal{O}(d^2)$ parameters to the model, which is a negligible overhead relative to the transformer block.

The step embedding allows the model to specialize computation by loop index while using normalized progress s_ℓ to remain well-conditioned across different loop counts. To evaluate its impact, we compare three variants: (i) a model with a *learned* step embedding, where the normalized recursion progress is projected into the hidden space through a small trainable network; (ii) a model with a *fixed* step embedding, constructed directly from the normalized recursion progress (ranging from 0 to 1) using a deterministic encoding; and (iii) a model with *no step embedding*, in which all recursive iterations share exactly the same computation and receive no explicit indication of the current recursion depth.

Figure 10 shows a relatively close comparison between learned embeddings and the no-embedding baseline, with both outperforming fixed embeddings in the most important low-depth regimes. Learned embeddings perform strongly across the sweep and give the model an explicit, trainable signal for the current recursion step, which makes them a natural general-purpose choice. However, the no-embedding baseline is also very competitive. The advantage of learned embeddings is therefore not absolute; rather, they offer a balanced tradeoff: more flexible and robust than fixed embeddings, while remaining close to the best observed performance across the full grid.

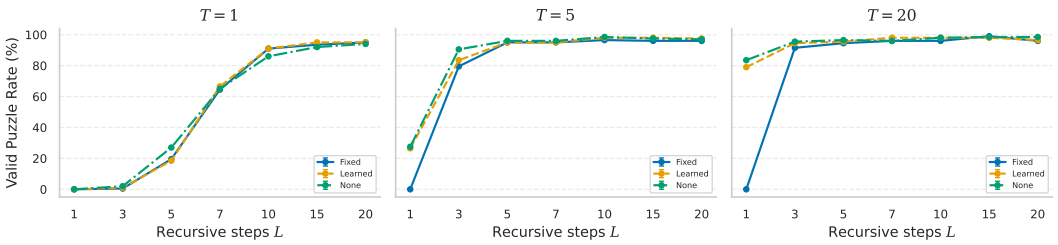


Figure 10: Performance comparison between models trained with learned, fixed and without step-embedding on 9×9 Sudoku puzzles. We report the Valid Puzzle Rate across varying sampling loop counts L_s and denoising steps T.

D.4 Training objectives

Here we extend the discussion on the different supervision methods, and report the results from the ablation study.

Final-step loss (FINAL). Only the logits produced at the last loop, $\hat{\mathbf{y}}^{(L)}$, are supervised:

$$\mathcal{L}_{\text{final}}(\theta) = \mathbb{E}_{t,\, x_0,\, x_t}\left[\frac{1}{|\mathcal{M}|}\sum_{i\in\mathcal{M}} \mathrm{CE}\Big(\hat{\mathbf{y}}_i^{(L)},\ x_0^{(i)}\Big)\right], \tag{29}$$

where $\mathcal{M} = \{i : x_t^{(i)} = \texttt{[MASK]}\}$ is the set of masked positions, $|\mathcal{M}|$ is its cardinality, and CE denotes the cross-entropy loss. This is the natural extension of the standard single-pass MDM objective to the looped setting: the intermediate loop outputs are used only as hidden-state refinements, not directly supervised. The gradient signal reaches early loops only through backpropagation through the full chain of shared blocks.

All-steps loss (ALL). All L sets of logits are supervised, with the loss averaged across loops:

$$\mathcal{L}_{\text{all}}(\theta) = \mathbb{E}_{t,\, x_0,\, x_t}\left[\frac{1}{L}\sum_{\ell=1}^{L}\frac{1}{|\mathcal{M}|}\sum_{i\in\mathcal{M}} \mathrm{CE}\Big(\hat{\mathbf{y}}_i^{(\ell)},\ x_0^{(i)}\Big)\right]. \tag{30}$$

This provides a denser gradient signal: every loop receives direct supervision rather than relying entirely on backpropagation through subsequent loops. This idea is closely related to deeply supervised learning, where intermediate representations are explicitly trained to be predictive (Lee et al., 2015). It also regularizes the model to produce semantically meaningful intermediate predictions, not only a high-quality final output, the MDM analogue of the per-iteration supervision in RELAY for AR looped models (Yu et al., 2026). The all-steps loss slightly increases training memory due to storing intermediate activations for backpropagation through all loops, but it does not increase forward-pass compute.

Figure 11 reports Sudoku 9×9 Valid Puzzle Rate performance for models trained with the different losses and 5 loops at training. We show the results sampling recursive steps $L_s \in \{1, 3, 5, 7, 10, 15, 20\}$ and denoising steps $T \in \{1, 5, 20\}$.

Weighted loss (WEIGHTED). Rather than supervising all loops equally, the weighted objective assigns a non-uniform weight w_ℓ to the loss at each loop:

$$\mathcal{L}_{\text{weighted}}(\theta) = \mathbb{E}_{t,\, x_0,\, x_t}\left[\sum_{\ell=1}^{L} w_\ell \frac{1}{|\mathcal{M}|}\sum_{i\in\mathcal{M}} \mathrm{CE}\Big(\hat{\mathbf{y}}_i^{(\ell)},\ x_0^{(i)}\Big)\right], \tag{31}$$

where $w_\ell \geq 0$ and $\sum_{\ell=1}^{L} w_\ell = 1$. In our implementation, we consider two weighting schemes. For *linear* weighting,

$$w_\ell = \frac{\ell^\alpha}{\sum_{j=1}^{L} j^\alpha}, \tag{32}$$

while for *exponential* weighting,

$$w_\ell = \frac{\exp\big(\alpha(\ell - L)\big)}{\sum_{j=1}^{L}\exp\big(\alpha(j - L)\big)}, \tag{33}$$

where $\alpha > 0$ controls how strongly supervision is concentrated toward later loops. Larger values of α place increasing emphasis on the final recursive iterations while still providing direct supervision to earlier loops. This objective is related to weighted auxiliary-loss training in anytime and deeply supervised neural networks, where intermediate predictions are optimized with unequal importance to balance early predictive capability against final performance (Hu et al., 2019).

Truncated loss (TRUNCATED). The truncated objective supervises only the final k recursive loops. Let $k_{\text{eff}} = \min(k, L)$ and let $\mathcal{S}_k = \{L - k_{\text{eff}} + 1, \ldots, L\}$ denote the set of the last k_{eff} loops. The loss is then

$$\mathcal{L}_{\text{truncated}}(\theta) = \mathbb{E}_{t,\, x_0,\, x_t}\left[\frac{1}{k_{\text{eff}}}\sum_{\ell\in\mathcal{S}_k}\frac{1}{|\mathcal{M}|}\sum_{i\in\mathcal{M}} \mathrm{CE}\Big(\hat{\mathbf{y}}_i^{(\ell)},\ x_0^{(i)}\Big)\right], \tag{34}$$

This objective provides direct supervision only to the later stages of the recursive computation, which are expected to be closer to the final prediction. FINAL corresponds to truncated with $k = 1$.

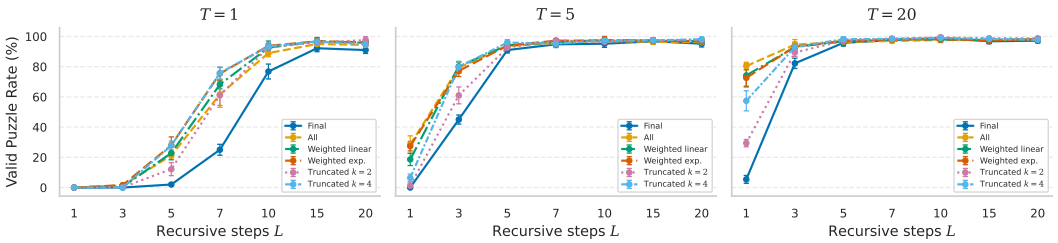


Figure 11: Performance comparison between supervision losses on 9×9 Sudoku puzzles for the model trained with $L_t = 5$ recursive steps. We report the Valid Puzzle Rate across varying loop counts at sampling L_s and denoising steps T.

In the more general setting, multiple late-loop predictions receive explicit supervision, improving gradient flow, while compared to ALL, early loops are not constrained to produce accurate predictions and can instead focus on building useful intermediate representations. This strategy is similar in spirit to the truncated step-range averaging used in looped transformer architectures, where training emphasizes a subset of later recurrent iterations rather than supervising the entire trajectory equally (Yang et al., 2024).

We observe that across panels, ALL yields the strongest and most robust performance, particularly in the regime with less recursion (L_s small to moderate) and where inference is budget-constrained (T small). At L_s=1, the gap between ALL and FINAL is notable. This supports the view that dense, per-step supervision is necessary to align every recursive layer with the denoising objective, rather than relying on error signals that propagate only through the terminal state. As L_s increases, all methods improve and the curves compress, yet ALL remains at or near the top through moderate depth. We therefore adopt ALL as the default supervision strategy in subsequent experiments.

E EXACT DEPENDENCE–FIDELITY MEASUREMENT

This appendix details the experiment of Section 4.2. On the held-out 9×9 boards of Section 3.2, every puzzle has a unique solution so given the clues, $\pi_{\text{data}}(x_M \mid x_O)$ is therefore a point mass. Every marginal $p_i^\star(\cdot \mid s)$ along the oracle process is a point mass as well, so $D_{\text{dep}}(U \mid s) = 0$ for every commit set, including one-shot decoding (Corollary A.7), and $D_{\text{KL}}(P\|Q)$ reduces to the fidelity term $\sum_t \sum_{i \in U_t} -\log q_i(x_i^\star)$. Any conflict among cells committed in the same step is then a fidelity error, which is why the within-step conflict rate is only a proxy for D_{dep} (Remark A.9). Separating the terms requires states at which several completions remain valid.

Board construction. We start from solved held-out grids, blank a fixed number of cells uniformly at random, and enumerate every completion with an exact solver. Most random masks do not have a unique solution (Table 3): the fraction of unique boards drops from 43% at 40 blanks to 6% at 48, and beyond 52 blanks most boards have hundreds of completions. We fix the number of blanks at 48, where all groups occur, so that the commit size $k = 48/T$ is identical across groups and the solution count is not confounded with the number of masked cells. Boards are drawn by rejection sampling until each group $S \in \{1,\ 2\text{–}5,\ 6\text{–}50,\ 51\text{–}500\}$ holds 50 boards; boards with more than 500 completions are discarded, so every board has an exactly enumerated solution set. The unique group is the control, in which $D_{\text{dep}} \equiv 0$.

Target distribution. We take $\pi_{\text{data}}(\cdot \mid x_O)$ uniform over the valid completions of a board. Strictly, π_{data} is the empirical distribution of the 1.8M training grids, but with about 6.7×10^{21} valid 9×9 grids a board's completions are almost never in the training set, and uniformity over valid completions is the natural reading.

Computing both terms exactly. Let $\mathcal{S}$ be the set of completions of a board and, at a state $s = (x_O, M)$, let $\mathcal{C}(s) = \{\sigma \in \mathcal{S} : \sigma_O = x_O\}$ be the completions that agree with every committed cell.

Table 3: Share of random masks by number of solutions S, as a function of the number of blanked cells (300 masks on held-out grids per row). We use 48 blanks.

Blanks	S=1	2–5	6–50	51–500	> 500
40	43%	44%	13%	0%	0%
44	25%	49%	26%	0%	0%
48	**6%**	**28%**	**56%**	**11%**	**0%**
52	0%	4%	38%	46%	12%
56	0%	0%	2%	19%	79%

Under the uniform target,

$$p^\star(x_U \mid s) \;=\; \frac{\big|\{\sigma \in \mathcal{C}(s) : \sigma_U = x_U\}\big|}{|\mathcal{C}(s)|}, \qquad p_i^\star(v \mid s) \;=\; \frac{\big|\{\sigma \in \mathcal{C}(s) : \sigma_i = v\}\big|}{|\mathcal{C}(s)|}. \tag{35}$$

The joint entropy $H(p^\star(x_U \mid s))$ is obtained by grouping $\mathcal{C}(s)$ by the projection σ_U, and $D_{\mathrm{dep}}(U \mid s)$ and $D_{\mathrm{fid}}(U \mid s, q)$ then follow from Lemma 4.1, with q_i the untruncated softmax. Along the oracle process of Theorem A.5, the trajectory costs $\mathcal{D}_{\mathrm{dep}}(\mathcal{U})$ and $\sum_t \mathbb{E}_P[D_{\mathrm{fid}}]$ are averages of these per-step quantities.

The same enumeration gives the exact probability that a step fails. For the model's product of marginals,

$$P_q(\text{some value infeasible alone}) = 1 - \prod_{i \in U} q_i\big(\operatorname{supp} p_i^\star\big), \tag{36}$$

$$P_q(\text{jointly infeasible only}) = \prod_{i \in U} q_i\big(\operatorname{supp} p_i^\star\big) \;-\; \sum_{y \in \operatorname{supp} p_U^\star} \prod_{i \in U} q_i(y_i). \tag{37}$$

Replacing q_i by $p_i^\star$ gives the failure probability of a model with *perfect marginals*, which predicts every cell exactly but still samples the cells of a step independently. Its failures are pure dependence errors, and it is the reference shown in red in Figure 5. Its final survival is the product over steps of its per-step success probability along the oracle process, averaged over boards. This is a close approximation rather than an identity, since a perfect-marginal sampler that survives a step does not visit states in exactly the proportions of P.

Model and sampler. We use the recursive Sudoku checkpoint of Section 4.2 (trained with a fixed L_t=5), with $L \in \{1, 3, 5, 10\}$ and $T \in \{1, 2, 4, 8, 16, 48\}$. Each step commits the next k_t cells of a random permutation of the blanks drawn per board and sample before decoding. The permutation is shared across L, T and trajectory modes, so the commit sets never depend on q and $\mathcal{D}_{\mathrm{dep}}$ is identical for every L by construction.

In the *model* mode, committed tokens are sampled from q and the run stops at its first failure. In the *oracle* mode, a completion is drawn uniformly from $\mathcal{S}$ and revealed along the same commit order; this is the process P of Theorem A.5, along which D_{dep}, D_{fid} and the failure probabilities above are evaluated (Figure 4).

Failure labels. Figure 5 uses the rule-based labels of Section 4.2: a run fails at its first rule violation, either a *parallel* clash between two cells committed in the same step or an *old-cell* clash between a new cell and one already on the board (no invalid tokens occurred). Because every completion is known, we can also label each run by the step at which the board *became uncompletable*. That step is a *marginal* error if some committed value appears in no completion consistent with the board, so that it would fail even if committed alone, and a *joint* error if every committed value appears in some consistent completion but no single completion contains all of them. The two labellings answer different questions:

- A board can become uncompletable without any clash (a *silent dead end*), for example when A=3 forces B=7 through other cells and B=3 is committed alongside A=3 although they share no unit. The clash then surfaces at a later step.
- Old-cell clashes are almost never the original mistake: in 97–100% of them, across all groups, L and T, the board had already become uncompletable at an earlier step.

- Parallel clashes are mostly the original mistake. On multi-solution boards, 83–88% of them coincide with a joint error at L=5 (83–92% across $L \geq 5$), against 0% on unique boards, where joint errors are impossible.

Final completion rates are the same under both labelings: an uncompletable board always ends in a rule violation, and a run without one ends on a valid board, which is one of the enumerated completions. Only the timing and label of failures differ.

Completion rates. Table 4 reports completion rates for all groups. On unique boards, all step budgets converge to about 98% as L grows: at L=10, one-shot decoding completes 97.9% of boards against 0.4% at L=1. On multi-solution boards, completion plateaus from L=5 on, 2–6 points below the perfect-marginal ceiling set by T. Summed along the oracle process, the model's probability of a joint failure reaches 93–98% of that of the perfect-marginal model at L=5 ($T \in \{2, 4, 8, 16\}$) and does not move at L=10.

Table 4: Completion rate (%) by solution-count group, denoising steps T and recursive depth L (48 blanks, 50 boards × 64 samples per cell). Last column: perfect-marginal ceiling.

Group	T	L=1	L=3	L=5	L=10	Perfect marginals
S=1	1	0.4	50.3	89.9	97.9	100
	4	9.7	73.0	94.1	98.5	100
	16	27.5	85.9	97.8	99.0	100
	48	33.7	90.8	98.7	99.1	100
S=2–5	1	0.0	1.7	2.9	2.8	3.0
	4	7.6	33.4	42.4	43.4	47.0
	16	31.8	78.1	84.5	84.2	87.2
	48	43.4	90.7	97.3	98.1	100
S=6–50	1	0.0	0.0	0.0	0.0	0.0
	4	4.4	12.8	14.3	16.3	19.4
	16	31.6	65.9	70.6	71.7	76.1
	48	46.6	89.7	95.8	95.6	100
S=51–500	1	0.0	0.0	0.0	0.0	0.0
	4	1.7	3.6	3.5	4.4	5.9
	16	27.0	54.4	59.6	61.1	64.7
	48	48.3	89.3	95.7	95.9	100

F THE DECODING ZOO

Every decoding method used in this paper is a pair (loop-stopping rule, commit rule). We name a method `loop_commit`. This appendix fixes the semantics of each component, gives the full cross product of the two axes that we actually implement and sweep, and documents two relevant ablations that some of these methods additionally expose.

F.1 ZOO OVERVIEW

Not all combinations of the loop-stopping and commit rules below are meaningful or implemented: `sched` is only ever paired with `uniform` (its schedule *is* the loop budget, so it has nothing to add to a threshold commit rule), and the two adaptive loop rules (`entropy`, `hidden`) are paired with all three of `uniform`, `confidence`, `entropy`. This gives the ten methods of Table 5.

The nominal step count T enters in only two places: the `uniform` commit rule, which reveals $|M_0|/T$ positions per step, and the `sched` loop schedule, which anneals over T steps. Every method pairing a `fixed` or adaptive loop rule with a *threshold* commit rule therefore never reads T: its outer loop runs until no position is masked, and both its quality and its NFE are constant along the T axis, and it is why those methods are reported once per L rather than once per (L, T) pair.

Compute is billed as NFE $= \sum_t \ell_t$. For the two `uniform`-commit, T-scheduled methods with a non-adaptive loop rule, the number of outer steps is $\min(T, |M_0|)$, since `uniform` commits

Table 5: The decoding zoo. $\ell_t \leq L$ is the number of loops run at outer step t; U_t is the commit set; n_{out} is the (data-dependent) number of outer steps a T-invariant method takes. Threshold parameters are defined in Section F.2–Section F.3.

Method	Loop rule	Commit rule	Uses T?	Threshold(s)
`sched_uniform`	`sched`: $L \to 1$	`uniform`	Yes	—
`fixed_uniform`	`fixed`: $\ell_t = L$	`uniform`	Yes	—
`fixed_confidence`	`fixed`: $\ell_t = L$	`confidence`	No	c
`fixed_entropy`	`fixed`: $\ell_t = L$	`entropy`	No	δ
`entropy_uniform`	`entropy`-gain, $\leq L$	`uniform`	Yes	λ
`entropy_entropy`	`entropy`-gain, $\leq L$	`entropy`	No	λ, δ
`entropy_confidence`	`entropy`-gain, $\leq L$	`confidence`	No	λ, c
`hidden_uniform`	`hidden`-drift, $\leq L$	`uniform`	Yes	ε
`hidden_entropy`	`hidden`-drift, $\leq L$	`entropy`	No	ε, δ
`hidden_confidence`	`hidden`-drift, $\leq L$	`confidence`	No	ε, c

at least one position per step; `fixed_uniform` therefore bills NFE $= L\min(T, |M_0|)$, which equals TL whenever $T \leq |M_0|$ (`sched_uniform` instead averages to NFE $\approx T(L{+}1)/2$). For every other method NFE $= \sum_t \ell_t \leq n_{\text{out}} L$, with equality only for the two `fixed`-loop, threshold-commit methods (`fixed_confidence`, `fixed_entropy`); the four `entropy`- and `hidden`-loop methods with a threshold commit rule additionally have $\ell_t \leq L$ per step, so their realized NFE is data-dependent on *both* axes at once.

Unless noted otherwise, decoding uses the `fixed_uniform` policy: over T outer denoising steps the remaining masked positions are scored by top-k confidence, and at each step a nearly equal share of them is committed (highest confidence first). Token values are drawn from temperature-scaled top-k logits, while already-revealed (and prompt) tokens stay fixed. Per-dataset evaluation protocols are given in Appendix C, and the temperature and top-k values in Table 6.

F.2 Loop-stopping rules (how many inner loops at a state)

At outer step t the policy executes $\ell_t \leq L$ loops at the frozen state s_t, where L is the loop budget and ℓ_t is chosen by one of:

- `fixed` — always $\ell_t = L$. Non-adaptive; bills NFE $= L \times T$ exactly.
- `sched` — a decreasing schedule that anneals the per-step loop count linearly from L at $t = 0$ down to 1 at $t = T - 1$, on the premise that the early, mostly-masked states need more recursion to resolve than the late, mostly-committed ones. This gives a mean depth of roughly $(L{+}1)/2$.
- `entropy` — *entropy-gain* stopping: run the first loop unconditionally, then after each loop $\ell \geq 2$ compute the mean Shannon entropy over still-masked positions $\bar{H}(q^{(\ell)})$, and continue while the gain

$$\bar{H}(q^{(\ell-1)}) - \bar{H}(q^{(\ell)}) \;\geq\; \lambda.$$

- `hidden` — *hidden-drift* stopping: continue while the relative hidden-state drift

$$\frac{\|\mathbf{h}^{(\ell)} - \mathbf{h}^{(\ell-1)}\|}{\|\mathbf{h}^{(\ell)}\|} \;\geq\; \varepsilon.$$

Adaptive rules (`entropy`, `hidden`) never exceed the budget L, so $\ell_t \leq L$ always. Paired with `uniform` commits the outer loop runs exactly T steps, giving NFE $\leq LT$ with the gap being the compute the rule saves; paired with a threshold commit rule the number of outer steps n_{out} is data-dependent, so the bound is NFE $\leq n_{\text{out}} L$ instead. When an `entropy`-loop rule stops, the marginals it deploys may be taken from the last loop, the previous loop, or the best-scoring loop seen (the *commit-loop policy*, see ablation in Appendix F.4). Note that a small entropy gain does not imply that the two loops produce similar distributions: entropy is a scalar summary, and two distributions with equal entropy can be far apart in total variation. The choice is therefore settled empirically, not by the analysis.

F.3 COMMIT RULES (WHICH MASKED POSITIONS TO REVEAL)

Given the deployed marginals q_i at the masked positions, the commit set U_t is chosen by one of:

- `uniform` — reveal a fixed number $k = |M_0|/T$ of positions per step, taken in order of decreasing confidence, so that all n positions are committed in T steps. This method carries no dependence certificate: it fixes $|U_t|$ regardless of realized dependence.
- `confidence` — reveal every position whose top-1 probability exceeds a threshold c. This is the standard MaskGIT-style schedule of Chang et al. (2022).
- `entropy` — reveal the largest low-entropy set satisfying the budget $\sum_{i\in U} H_i \le \delta$, built greedily by revealing the lowest-entropy masked positions first.

Both threshold rules (`confidence`, `entropy`) force at least one commit (either the most confident position, or the single lowest-entropy one) when no candidate qualifies, so decoding terminates in at most n steps.

Algorithm 2 Entropy-guided recursive decoding (`entropy_entropy`)

Input: R-MDM (f_θ, W_θ), state $s_0 = (x_O, M_0)$, loop budget L, thresholds λ, δ
while $M_t \neq \emptyset$ **do**
  for $\ell = 1, \dots, L$ **do**
    compute $\mathbf{h}^{(\ell)}$ and marginals $q_i^{(\ell)}(\cdot \mid s_t)$ for $i \in M_t$ (Equation 1)
    if $\ell \ge 2$ **and** $\bar{H}(q^{(\ell-1)}) - \bar{H}(q^{(\ell)}) < \lambda$ **then break** ▷ loop stopping
  $U_t \leftarrow$ lowest-entropy positions of M_t with $\sum_{i\in U_t} H(q_i^{(\ell)}) \le \delta$, at least one ▷ commit
  sample $x_i \sim q_i^{(\ell)}$ independently for $i \in U_t$; $M_{t+1} \leftarrow M_t \setminus U_t$
return the completed sequence

F.4 ABLATION ON THE DECODING MECHANISMS

We perform two focused ablations to isolate design choices in the adaptive recursive decoder. Both experiments use the Sudoku generation setting and evaluate 50 generated boards per configuration. We report Valid Puzzle Rate (VPR) as the task-quality metric and the mean number of recursive forward passes as the compute metric. The latter is important because adaptive decoding policies do not necessarily spend the full nominal recursion budget: they stop once the mean entropy gain of the inner refinement loop falls below λ. Thus, the relevant comparison is not only which method reaches high validity, but which method reaches it at the lowest effective number of function evaluations (NFE).

Commit-Loop Policy For entropy-controlled recursive decoders, the inner loop runs until the average entropy over still-masked positions stops decreasing by more than a threshold λ. If this stopping condition triggers after loop ℓ, there are two nearby predictive distributions available: the previous distribution $q^{\ell-1}$ and the current distribution q^{ℓ}. We ablate three policies for choosing which distribution is used to sample and commit tokens:

$$\texttt{prev}: q^{\ell-1}, \qquad \texttt{curr}: q^{\ell}, \qquad \texttt{best}: \begin{cases} q^{\ell}, & \text{if entropy improved at the confirming step,} \\ q^{\ell-1}, & \text{otherwise.} \end{cases}$$

The `prev` policy is conservative: it avoids committing from the step that first indicates saturation. The `curr` policy is more aggressive: it uses the latest refined logits, even though the marginal entropy improvement has become small. The `best` policy is an intermediate rule that attempts to keep the better of the two local iterates.

The ablation (see Figure 12) supports `curr` as the default policy: it achieves the strongest performance in the adaptive entropy-budget setting, while remaining competitive under fixed-T decoding. This suggests that using the current loop's logits for commitment is preferable to reverting to stale logits, particularly when commit size is adaptively determined from model uncertainty.

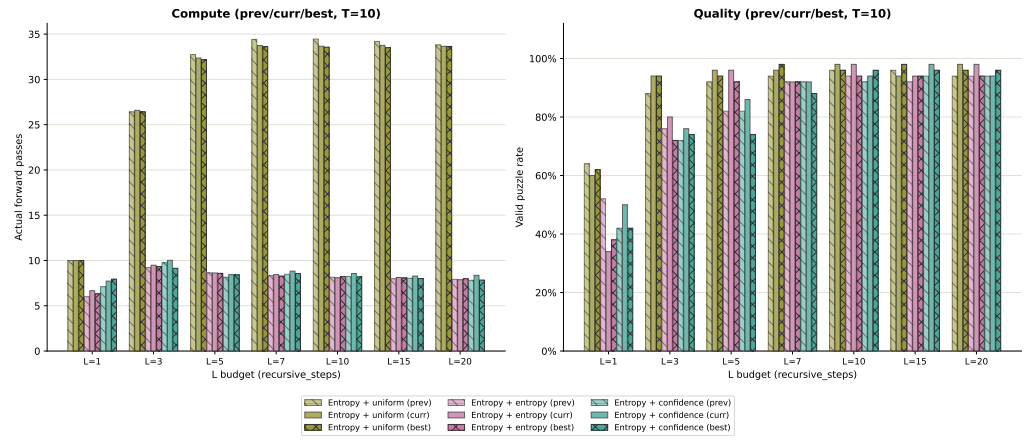


Figure 12: Commit-loop policy ablation for entropy-controlled recursive decoding.

Entropy versus EB Commit Rule We next compare the standard entropy-budget commit rule against an EB-style alternative (Ben-Hamu et al., 2025). The standard entropy rule sorts still-masked positions by predictive entropy and commits the largest low-entropy set S satisfying

$$\sum_{i \in S} H(q_i) \leq \delta,$$

with at least one token committed at every outer step. The EB variant replaces this with the looser bound

$$\sum_{i \in S} H(q_i) - \max_{i \in S} H(q_i) \leq \gamma,$$

using the same numerical budget value as the standard entropy rule. Intuitively, the EB rule permits larger simultaneous commits by subtracting the largest individual uncertainty term. This can reduce compute, but it also risks committing mutually incompatible tokens earlier.

The results (see Figure 13) do not show a consistent benefit from the EB rule. While `entropy` and `entropy_eb` perform similarly, the standard entropy budget is more reliable for adaptive entropy and hidden-entropy decoding: `adaptive_entropy` reaches 0.90 and 0.94 VPR at $L = 7$ and $L = 10$, compared with 0.76 and 0.92 for its EB variant, with similar trends for hidden entropy. This suggests that the stricter sum-entropy criterion better preserves validity under Sudoku's global constraints, where parallel commitment of uncertain cells can create irrecoverable conflicts. We therefore use the standard entropy rule.

F.5 Hyperparameter tuning

We tune four decoding hyperparameters per dataset:

- λ on `entropy_`$\star$, $\lambda \in \{10^{-3}, 5\times10^{-3}, 10^{-2}, 5\times10^{-2}, 10^{-1}, 5\times10^{-1}\}$
- δ on $\star$`_entropy`, $\delta \in \{0.25, 0.5, 1, 2, 4, 8\}$,
- ε on `hidden_`$\star$, $\varepsilon \in \{0.01, 0.05, 0.1, 0.2, 0.5, 1.0\}$,
- c on $\star$`_confidence`, $c \in \{0.5, 0.7, 0.8, 0.9, 0.95, 0.99\}$.

We pick one value per dataset from quality–NFE plots on the validation split, favoring near-frontier settings at moderate L and lower NFE.

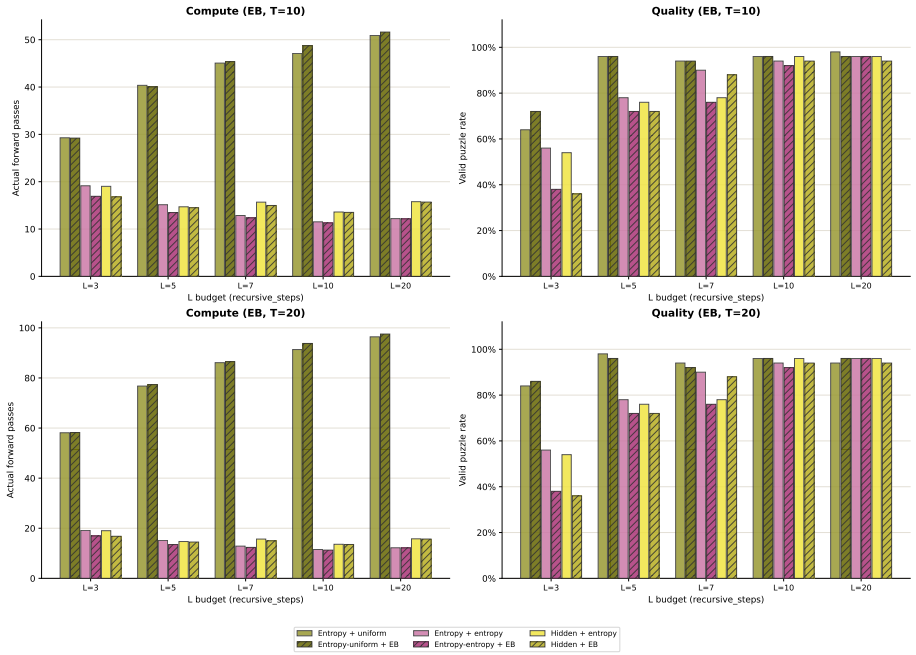


Figure 13: Comparison between standard entropy-budget decoding and the EB commit rule.

Table 6: Decoding settings per dataset. The thresholds λ, δ, ε and c are selected as described above; the sampling temperature and top-k are fixed per dataset and not tuned.

Dataset	λ	δ	ε	c	Temp.	top-k
Sudoku 9×9	0.01	0.25	0.20	0.95	0.8	2
Countdown-3	0.1	0.25	0.50	0.90	0.5	1
Countdown-4	0.01	0.25	0.10	0.95	0.5	1
Countdown-5	0.05	0.50	0.10	0.95	0.5	1
RNA	0.05	4.00	0.05	0.95	0.8	2
TinyGSM	0.01	2.00	0.05	0.95	0.8	1

G Extended Results

G.1 Sudoku 9×9: Full Tables

Table 7 presents the comprehensive Sudoku 9×9 results across recursion depth and denoising step budget, for the all-loops objective $\mathcal{L}_{\text{all}}$ used throughout the paper. Latency and throughput figures are measured at batch size 1 on a single A100. Rows labelled *Baseline* correspond to $(6 \otimes 1)$ without any recursion, and rows labeled k *rec steps* to a $(6 \otimes k)$ model.

Table 7: Sudoku 9×9: valid puzzle rate, soft constraint loss, latency and throughput across recursion depths and denoising budgets. All models are trained with the all-loops objective $\mathcal{L}_{\text{all}}$; the comparison against $\mathcal{L}_{\text{final}}$ is in Appendix D.4. We report mean and standard deviation across 5 sampling runs of 100 samples each.

Model	Steps	VPR %	SCL	Latency (s)	Throughput
Baseline	1	0.0% ± 0.0	3.077 ± 0.039	0.007	149.6
	5	65.2% ± 6.9	0.226 ± 0.052	0.022	44.9
	10	85.0% ± 2.7	0.088 ± 0.024	0.043	23.3
	20	92.2% ± 2.7	0.031 ± 0.017	0.084	11.9
	40	95.8% ± 1.9	0.011 ± 0.006	0.169	5.9
2 rec steps	1	0.0% ± 0.0	2.443 ± 0.080	0.010	102.1
	5	83.2% ± 3.6	0.120 ± 0.026	0.039	25.8
	10	91.2% ± 1.6	0.055 ± 0.017	0.079	12.7
	20	95.4% ± 2.8	0.024 ± 0.015	0.153	6.6
	40	96.4% ± 1.8	0.016 ± 0.009	0.298	3.4
3 rec steps	1	8.6% ± 3.0	1.645 ± 0.051	0.020	50.5
	5	92.4% ± 2.5	0.056 ± 0.018	0.072	13.9
	10	95.6% ± 2.1	0.023 ± 0.015	0.131	7.7
	20	96.0% ± 2.2	0.024 ± 0.016	0.234	4.3
	40	98.0% ± 0.7	0.008 ± 0.005	0.456	2.2
5 rec steps	1	21.2% ± 2.4	1.093 ± 0.113	0.084	11.9
	5	93.6% ± 2.9	0.050 ± 0.026	0.210	4.8
	10	97.4% ± 1.3	0.016 ± 0.014	0.362	2.8
	20	96.4% ± 1.5	0.016 ± 0.007	0.696	1.4
	40	98.0% ± 1.6	0.008 ± 0.006	1.119	0.9
10 rec steps	1	46.4% ± 8.0	0.670 ± 0.156	0.036	27.8
	5	95.2% ± 2.4	0.034 ± 0.016	0.171	5.8
	10	96.0% ± 1.6	0.026 ± 0.016	0.504	2.0
	20	97.2% ± 1.3	0.017 ± 0.012	1.232	0.8
	40	97.0% ± 1.9	0.014 ± 0.010	3.457	0.3

Table 8 presents the Sudoku 9×9 results for the (*Schedule*$10 \rightarrow 1$) training recursion depth and for different evaluated sampling recursion depths (L_s).

Table 8: Sudoku Performance: comprehensive comparison across sampling recursion depths (L_s) and decoding steps (T). We report mean and standard deviation across 5 sampling runs of 100 samples each.

Recursion L_s	Steps	VPR %	SCL	Latency (s)	Throughput
1 rec step	1	0.0% ± 0.0	3.729 ± 0.034	0.018	55.0
	5	24.0% ± 2.5	0.587 ± 0.033	0.045	22.3
	10	63.6% ± 2.5	0.210 ± 0.034	0.087	11.5
	20	81.4% ± 2.5	0.084 ± 0.008	0.143	7.0
	40	85.2% ± 0.8	0.050 ± 0.003	0.286	3.5
3 rec steps	1	3.8% ± 1.3	1.791 ± 0.046	0.024	41.8
	5	95.6% ± 0.9	0.025 ± 0.009	0.081	12.3
	10	95.8% ± 1.8	0.021 ± 0.007	0.161	6.2
	20	98.2% ± 0.8	0.012 ± 0.011	0.276	3.6
	40	99.0% ± 1.0	0.004 ± 0.004	0.608	1.6
5 rec steps	1	63.6% ± 1.5	0.426 ± 0.019	0.032	30.9
	5	96.8% ± 0.8	0.022 ± 0.007	0.120	8.3
	10	97.0% ± 1.6	0.020 ± 0.007	0.235	4.2
	20	97.8% ± 0.8	0.012 ± 0.006	0.446	2.2
	40	99.8% ± 0.4	0.001 ± 0.002	0.940	1.1
7 rec steps	1	89.8% ± 0.8	0.113 ± 0.007	0.040	25.1
	5	97.4% ± 0.9	0.021 ± 0.004	0.159	6.3
	10	96.4% ± 0.9	0.015 ± 0.004	0.339	3.0
	20	96.2% ± 0.8	0.018 ± 0.004	0.573	1.7
	40	97.6% ± 0.5	0.011 ± 0.002	1.279	0.8
10 rec steps	1	94.2% ± 0.4	0.082 ± 0.013	0.049	20.4
	5	97.4% ± 0.5	0.021 ± 0.002	0.212	4.7
	10	98.4% ± 0.9	0.005 ± 0.003	0.435	2.3
	20	96.8% ± 0.4	0.014 ± 0.004	0.806	1.2
	40	98.0% ± 0.0	0.007 ± 0.000	1.565	0.6
15 rec steps	1	95.0% ± 0.0	0.062 ± 0.010	0.053	18.9
	5	97.0% ± 0.0	0.016 ± 0.001	0.326	3.1
	10	97.2% ± 0.4	0.014 ± 0.002	0.613	1.6
	20	98.0% ± 0.0	0.010 ± 0.002	1.183	0.8
	40	98.0% ± 0.0	0.007 ± 0.000	1.822	0.5
20 rec steps	1	95.0% ± 0.0	0.060 ± 0.006	0.071	14.0
	5	97.0% ± 0.0	0.018 ± 0.003	0.373	2.7
	10	97.6% ± 0.5	0.007 ± 0.002	0.726	1.4
	20	97.0% ± 0.0	0.018 ± 0.003	1.579	0.6
	40	98.0% ± 0.0	0.007 ± 0.000	2.416	0.4

G.2 SUDOKU 9×9: DEPTH SCALING WITHOUT RECURSION

Table 9 reports Valid Puzzle Rate and Soft Constraint Loss metrics for non-recursive baselines with 6, 8, 10, 12, 18, 24, and 30 transformer layers. Parameter counts range from 10.6M to 53.1M. All models share the same d=384, H=6 architecture and are trained for the same number of gradient steps as the recursive models.

Notably, increasing depth beyond 18 layers produces only marginal improvements on VPR (96.6%→97.2%→97.2% for 18→24→30 layers at T=40), whereas increasing L from 1 to 5 at fixed 6-layer parameter count produces a larger gain (95.8%→98.0%).

Table 9: Sudoku performance for baseline models without recursion across different depths (6–30 layers). We report mean and standard deviation across 5 sampling runs of 100 samples each.

Model	Steps	VPR %	SCL	Latency (s)	Throughput
6 layers (10.6M)	1	0.0% ± 0.0	3.077 ± 0.039	0.007	149.6
	5	65.2% ± 6.9	0.226 ± 0.052	0.022	44.9
	10	85.0% ± 2.7	0.088 ± 0.024	0.043	23.3
	20	92.2% ± 2.7	0.031 ± 0.017	0.084	11.9
	40	95.8% ± 1.9	0.011 ± 0.006	0.169	5.9
8 layers (14.2M)	1	0.0% ± 0.0	3.209 ± 0.053	0.010	104.8
	5	58.6% ± 6.4	0.291 ± 0.038	0.033	30.6
	10	83.0% ± 3.7	0.099 ± 0.018	0.072	13.9
	20	90.2% ± 3.5	0.045 ± 0.016	0.151	6.6
	40	94.4% ± 2.0	0.020 ± 0.007	0.370	2.7
10 layers (17.7M)	1	0.0% ± 0.0	3.100 ± 0.098	0.010	102.4
	5	60.4% ± 5.9	0.302 ± 0.055	0.046	22.0
	10	83.8% ± 3.0	0.098 ± 0.016	0.094	10.6
	20	91.4% ± 4.3	0.047 ± 0.028	0.211	4.7
	40	94.6% ± 1.1	0.024 ± 0.004	0.470	2.1
12 layers (21.2M)	1	0.0% ± 0.0	3.025 ± 0.057	0.010	101.5
	5	65.4% ± 3.4	0.262 ± 0.034	0.040	24.8
	10	87.6% ± 4.8	0.064 ± 0.022	0.082	12.1
	20	93.6% ± 2.9	0.031 ± 0.015	0.276	3.6
	40	93.8% ± 1.9	0.018 ± 0.006	1.005	1.0
18 layers (31.9M)	1	0.0% ± 0.0	2.622 ± 0.083	0.023	43.0
	5	80.6% ± 3.2	0.144 ± 0.021	0.085	11.7
	10	95.2% ± 2.4	0.028 ± 0.017	0.187	5.4
	20	94.8% ± 1.5	0.027 ± 0.008	0.535	1.9
	40	96.6% ± 0.6	0.014 ± 0.005	1.112	0.9
24 layers (42.5M)	1	0.2% ± 0.5	2.710 ± 0.056	0.048	21.1
	5	82.8% ± 3.6	0.127 ± 0.025	0.167	6.0
	10	92.2% ± 2.8	0.042 ± 0.019	0.365	2.7
	20	95.0% ± 2.5	0.023 ± 0.013	0.724	1.4
	40	97.2% ± 0.8	0.008 ± 0.002	1.435	0.7
30 layers (53.1M)	1	0.2% ± 0.5	2.624 ± 0.093	0.050	19.9
	5	77.8% ± 2.2	0.159 ± 0.018	0.160	6.2
	10	91.4% ± 1.3	0.044 ± 0.010	0.284	3.5
	20	95.0% ± 2.0	0.021 ± 0.006	0.547	1.8
	40	97.2% ± 0.8	0.012 ± 0.004	1.059	0.9

G.3 SUDOKU 9×9: CROSS-RECURSION TRADE-OFF AND EFFICIENT COMPUTE

Figure 14 shows the effect of varying the training loop count L_t and sampling loop count L_s on Sudoku 9×9 validity under tight decoding budgets ($T = 1, 5$). Interestingly, for one-shot generation ($T = 1$), increasing L_s above L_t consistently improves performance for all models. For $T = 5$ decoding steps, moderate extrapolation still helps, but performance eventually degrades when $L_s \gg L_t$, suggesting that representations are calibrated to converge in approximately L_t loops, which could also be explained by the step embedding. Critically, performance degrades sharply when $L_s \ll L_t$ (e.g. L_s=1 for an L_t=5 model), confirming that early-loop representations are optimized to be refined by subsequent loops rather than decoded directly. Overall, this flexibility means that recursive loops and denoising steps can be traded off continuously at inference time. In Section D.2, we further investigate alternative schedules for the number of recursive steps using training, showing that additional performance gains can be obtained by varying the number of recursive steps during training.

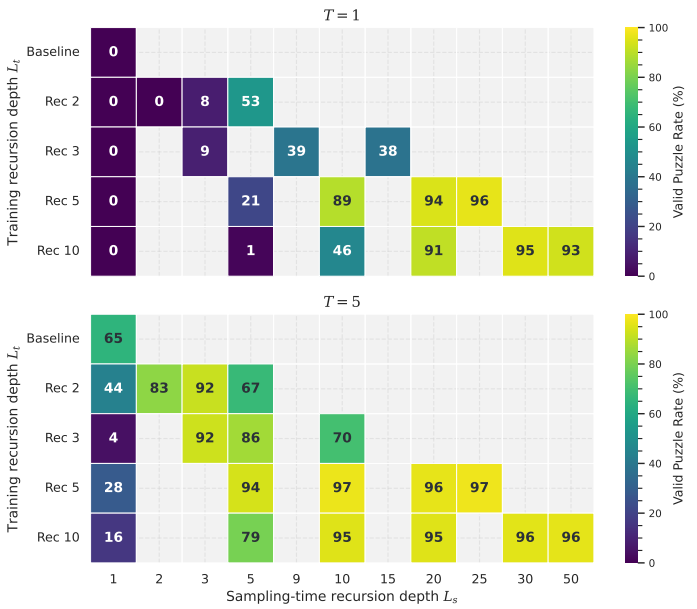


Figure 14: Cross-recursion trade-off between training recursion depth (L_t) and sampling recursion depth (L_s) for 9×9 Sudoku tasks. The top heatmap shows one-step decoding, while the bottom shows 5 decoding steps.

Table 10 extends the cross-recursion analysis by reporting both Valid Puzzle Rate and Soft Constraint Loss across training depths $L_t \in \{1, 2, 3, 5, 10\}$ and a range of sampling depths L_s.

Note that the $(L_t = 2, L_s = 3)$ setting, which uses one more loop at sampling than training, achieves 91.6% at T=5 already surpassing the L_t=2 baseline at T=5 (83.2%) and comparable to the L_t=3 model at T=5 (92.4%). This suggests that moderate extrapolation is possible, likely because the step embedding (Eq. 28) is parameterized by normalized progress $s_\ell \in [0, 1]$ rather than the raw loop index.

Table 10: Sudoku Performance: Cross-Recursion Trade-off. Training Recursion Depth vs. Sampling Recursion Depth across various decoding steps (T). We report mean and standard deviation across 5 sampling runs of 100 samples each.

Tr. Rec	Sa. Rec	$T = 1$		$T = 5$		$T = 10$		$T = 20$		$T = 40$	
Steps	Steps	VPR %	SCL	VPR %	SCL	VPR %	SCL	VPR %	SCL	VPR %	SCL
Baseline	1	0.0 ± 0.0	3.077 ± 0.039	65.2 ± 6.9	0.226 ± 0.052	85.0 ± 2.7	0.088 ± 0.024	92.2 ± 2.7	0.031 ± 0.017	95.8 ± 1.9	0.011 ± 0.006
Rec 2	1	0.0 ± 0.0	3.355 ± 0.069	43.6 ± 3.0	0.415 ± 0.048	77.6 ± 2.5	0.129 ± 0.017	88.2 ± 3.8	0.052 ± 0.024	92.6 ± 3.2	0.023 ± 0.011
	2	0.0 ± 0.0	2.443 ± 0.080	83.2 ± 3.6	0.120 ± 0.026	91.2 ± 1.6	0.055 ± 0.017	95.4 ± 2.8	0.024 ± 0.015	96.4 ± 1.8	0.016 ± 0.009
	3	8.2 ± 3.1	1.625 ± 0.042	91.6 ± 2.7	0.060 ± 0.025	96.2 ± 2.2	0.026 ± 0.015	96.2 ± 1.1	0.021 ± 0.008	98.4 ± 0.9	0.007 ± 0.004
	5	52.8 ± 5.0	0.658 ± 0.110	67.2 ± 5.6	0.153 ± 0.023	64.6 ± 3.2	0.160 ± 0.027	67.6 ± 2.7	0.137 ± 0.019	67.0 ± 6.5	0.130 ± 0.019
Rec 3	1	0.0 ± 0.0	4.485 ± 0.058	4.0 ± 3.9	0.884 ± 0.020	24.2 ± 3.1	0.457 ± 0.029	38.6 ± 7.1	0.277 ± 0.028	50.8 ± 9.3	0.165 ± 0.043
	3	8.6 ± 3.0	1.645 ± 0.051	92.4 ± 2.5	0.056 ± 0.018	95.6 ± 2.1	0.023 ± 0.015	96.0 ± 2.2	0.024 ± 0.016	98.0 ± 0.7	0.008 ± 0.005
	9	38.8 ± 5.5	1.304 ± 0.188	86.2 ± 3.0	0.174 ± 0.040	91.6 ± 4.4	0.073 ± 0.035	93.6 ± 3.4	0.044 ± 0.025	95.2 ± 1.3	0.033 ± 0.021
	15	38.0 ± 3.7	1.796 ± 0.285	70.0 ± 6.6	0.817 ± 0.325	80.0 ± 5.4	0.424 ± 0.168	84.6 ± 1.1	0.260 ± 0.067	82.8 ± 2.7	0.290 ± 0.037
Rec 5	1	0.0 ± 0.0	3.645 ± 0.060	28.4 ± 5.8	0.605 ± 0.054	61.6 ± 4.6	0.219 ± 0.035	80.4 ± 2.5	0.097 ± 0.022	83.4 ± 3.3	0.061 ± 0.014
	5	21.2 ± 2.4	1.093 ± 0.113	93.6 ± 2.9	0.050 ± 0.026	97.4 ± 1.3	0.016 ± 0.014	96.4 ± 1.5	0.016 ± 0.007	98.0 ± 1.6	0.008 ± 0.006
	10	89.0 ± 2.5	0.127 ± 0.033	97.0 ± 1.4	0.022 ± 0.011	95.8 ± 2.5	0.025 ± 0.013	97.6 ± 0.9	0.012 ± 0.004	98.4 ± 0.9	0.008 ± 0.007
	20	94.4 ± 2.9	0.080 ± 0.052	96.2 ± 1.9	0.027 ± 0.018	97.4 ± 1.3	0.018 ± 0.011	98.2 ± 0.8	0.011 ± 0.007	97.4 ± 2.4	0.012 ± 0.012
	25	96.4 ± 2.1	0.050 ± 0.027	97.4 ± 1.1	0.018 ± 0.013	98.4 ± 1.8	0.009 ± 0.012	97.4 ± 1.7	0.014 ± 0.010	99.2 ± 0.8	0.002 ± 0.002
Rec 10	1	0.0 ± 0.0	3.935 ± 0.044	16.0 ± 3.1	0.686 ± 0.035	51.6 ± 5.0	0.259 ± 0.041	70.4 ± 3.8	0.132 ± 0.012	73.8 ± 4.6	0.081 ± 0.015
	5	1.2 ± 0.8	2.186 ± 0.097	79.4 ± 0.9	0.163 ± 0.025	88.4 ± 2.7	0.069 ± 0.027	92.4 ± 1.5	0.048 ± 0.010	94.8 ± 1.9	0.026 ± 0.005
	10	46.4 ± 8.0	0.670 ± 0.156	95.2 ± 2.4	0.034 ± 0.016	96.0 ± 1.6	0.026 ± 0.016	97.2 ± 1.3	0.017 ± 0.012	97.0 ± 1.9	0.014 ± 0.010
	20	90.6 ± 2.1	0.120 ± 0.050	95.0 ± 2.1	0.041 ± 0.019	96.4 ± 3.1	0.026 ± 0.018	98.4 ± 0.9	0.006 ± 0.003	97.6 ± 1.5	0.013 ± 0.009
	30	95.2 ± 2.5	0.098 ± 0.053	95.6 ± 1.5	0.026 ± 0.009	95.6 ± 2.0	0.021 ± 0.009	97.0 ± 1.0	0.018 ± 0.007	98.4 ± 1.1	0.007 ± 0.006
	50	92.8 ± 2.5	0.122 ± 0.042	96.0 ± 1.0	0.024 ± 0.004	97.2 ± 1.9	0.018 ± 0.012	97.8 ± 2.3	0.012 ± 0.013	98.8 ± 0.8	0.005 ± 0.003

Having established that in our recursive paradigm the number of recursive steps can be flexibly changed at inference time, we now proceed to investigate to what extent using recursive steps (rather than denoising steps) can lead to more efficient inference. We quantify "efficiency" using the total number of forward passes through the model, ($x = T \times L$). Specifically, we compare a non-recursive baseline ($L_t = 1$) against a looped model trained with a reverse curriculum (*Schedule* $10 \to 1$), evaluated at multiple inference depths $L_s \in \{1, 3, 5, 7, 10\}$.

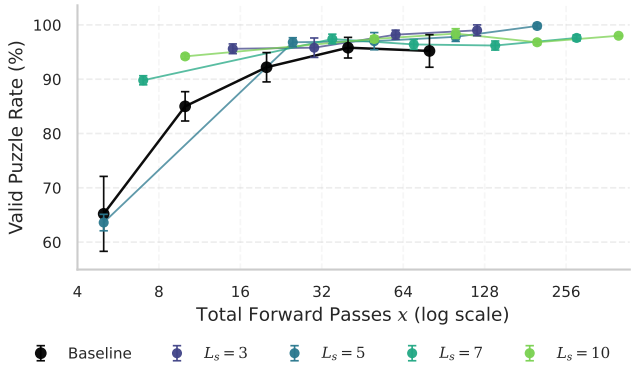


Figure 15: Valid Puzzle Rate versus total computational budget (forward passes $x = T \times L$) on Sudoku 9×9.

Figure 15 shows that for a fixed compute budget x, allocating computation to recursive inference depth L_s can be more efficient than increasing decoding steps T in a non-recursive model. At $x = 10$ forward passes, the baseline (T=10) achieves 85.0% validity, whereas the looped model with L_s=10, T=1 reaches 94.2%, a 9.2% absolute improvement under identical compute. At $x = 15$, the configuration L_s=3, T=5 achieves 95.6%, matching the baseline performance at $x = 40$ (95.8%), corresponding to a 2.7× reduction in compute. Beyond efficiency, recursion also raises the performance ceiling: while the baseline reaches 95.8% validity at $x = 40$, recursive models exceed this level already at T=5 (96.8% for L_s=5 and 97.4% for L_s=7), and reach up to 99.8% (L_s=5, T=40).

G.4 Sudoku 25×25: Effect of Recursion and Masking

Table 11 and Figure 16 show Valid Puzzle Rate and Soft Constraint Loss for the baseline and recursive models on the 25×25 Sudoku task under three masking regimes: 70%, 80%, and 90% of cells masked at inference time.

At 70% masking, the task is tractable for all models at high T: the baseline reaches 95.2% VPR at T=100. However, both L=3 and L=5 achieve 100% at T=1, a single denoising step (i.e., L passes of the 6-layer block), whereas the baseline does not reach 100% at any tested budget. At 80% masking, the baseline degrades severely (69.6% VPR at T=100), while L=3 achieves 100% VPR at T=100 and 99.6% at T=10. At 90% masking, all models struggle, but the gap between baseline (2.0% at T=100) and L=3 (52.2%) reveals the qualitative benefit of recursive loops for highly under-determined constraint satisfaction.

These results suggest that the benefits of recursive depth are not simply additive with problem size but compound with masking difficulty: as fewer clues are provided, the model's need to propagate global constraints increases, and recursive loops (each a full pass through the 6-layer bidirectional block) provide increasingly valuable additional computation.

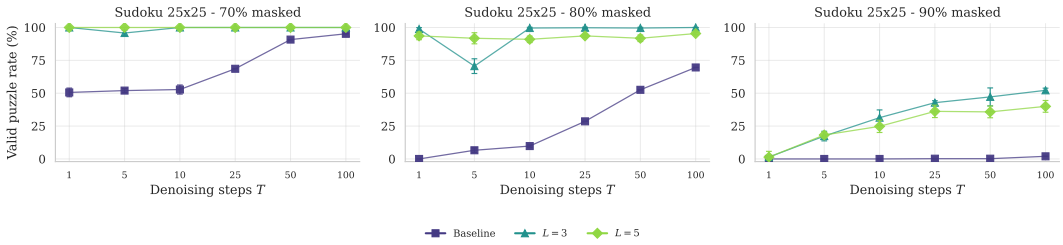


Figure 16: Effect of recursive refinement depth (fixed L at training and sampling) on 25×25 Sudoku under different masking conditions. Across all masking regimes, recursion improves VPR.

Table 11: Sudoku 25 $\times$ 25 generative performance: Comparison between different percentages of masks (M). We report mean and standard deviation across 5 sampling runs of 100 samples each.

		M = 70%		M = 80%		M = 90%	
Model	**Steps**	**VPR %**	**SCL**	**VPR %**	**SCL**	**VPR %**	**SCL**
Baseline	1	50.6% $\pm$ 3.3	0.189 $\pm$ 0.021	0.0% $\pm$ 0.0	4.587 $\pm$ 0.127	0.0% $\pm$ 0.0	24.978 $\pm$ 0.226
	5	52.0% $\pm$ 1.6	0.225 $\pm$ 0.008	6.6% $\pm$ 1.8	1.031 $\pm$ 0.080	0.0% $\pm$ 0.0	12.286 $\pm$ 0.453
	10	52.8% $\pm$ 3.4	0.221 $\pm$ 0.023	9.8% $\pm$ 0.8	0.719 $\pm$ 0.034	0.0% $\pm$ 0.0	6.363 $\pm$ 0.240
	25	68.6% $\pm$ 3.9	0.116 $\pm$ 0.026	28.6% $\pm$ 5.2	0.335 $\pm$ 0.032	0.2% $\pm$ 0.5	3.272 $\pm$ 0.188
	50	90.8% $\pm$ 2.5	0.026 $\pm$ 0.006	52.6% $\pm$ 4.2	0.142 $\pm$ 0.007	0.2% $\pm$ 0.5	2.060 $\pm$ 0.058
	100	95.2% $\pm$ 2.4	0.008 $\pm$ 0.004	69.6% $\pm$ 5.6	0.432 $\pm$ 0.040	2.0% $\pm$ 1.6	1.428 $\pm$ 0.120
3 rec. steps	1	100.0% $\pm$ 0.0	0.000 $\pm$ 0.000	98.8% $\pm$ 1.1	0.018 $\pm$ 0.019	1.4% $\pm$ 1.3	10.426 $\pm$ 0.323
	5	95.8% $\pm$ 2.2	0.007 $\pm$ 0.004	70.6% $\pm$ 5.6	0.082 $\pm$ 0.029	17.4% $\pm$ 3.7	2.803 $\pm$ 0.243
	10	100.0% $\pm$ 0.0	0.000 $\pm$ 0.000	99.6% $\pm$ 0.6	0.001 $\pm$ 0.001	31.4% $\pm$ 5.9	0.256 $\pm$ 0.050
	25	100.0% $\pm$ 0.0	0.000 $\pm$ 0.000	99.8% $\pm$ 0.5	0.001 $\pm$ 0.001	42.8% $\pm$ 1.5	0.680 $\pm$ 0.056
	50	100.0% $\pm$ 0.0	0.000 $\pm$ 0.000	99.6% $\pm$ 0.6	0.002 $\pm$ 0.003	47.2% $\pm$ 6.8	0.480 $\pm$ 0.058
	100	100.0% $\pm$ 0.0	0.000 $\pm$ 0.000	100.0% $\pm$ 0.0	0.000 $\pm$ 0.000	52.2% $\pm$ 1.6	0.430 $\pm$ 0.051
5 rec. steps	1	100.0% $\pm$ 0.0	0.000 $\pm$ 0.000	93.6% $\pm$ 2.8	0.083 $\pm$ 0.035	1.4% $\pm$ 1.7	9.298 $\pm$ 0.754
	5	100.0% $\pm$ 0.0	0.000 $\pm$ 0.000	91.8% $\pm$ 2.3	0.118 $\pm$ 0.030	18.2% $\pm$ 4.4	2.891 $\pm$ 0.187
	10	100.0% $\pm$ 0.0	0.000 $\pm$ 0.000	91.0% $\pm$ 2.5	0.101 $\pm$ 0.027	24.8% $\pm$ 3.5	1.691 $\pm$ 0.158
	25	100.0% $\pm$ 0.0	0.000 $\pm$ 0.000	93.6% $\pm$ 1.3	0.039 $\pm$ 0.012	36.2% $\pm$ 1.1	0.936 $\pm$ 0.093
	50	100.0% $\pm$ 0.0	0.000 $\pm$ 0.000	91.8% $\pm$ 2.6	0.030 $\pm$ 0.013	35.8% $\pm$ 4.8	0.835 $\pm$ 0.168
	100	100.0% $\pm$ 0.0	0.000 $\pm$ 0.000	95.4% $\pm$ 5.5	0.017 $\pm$ 0.001	40.0% $\pm$ 3.7	0.600 $\pm$ 0.088

G.5 Countdown: Extended Results

Table 12 reports Countdown performance for recursive models and the baseline across $k \in \{3, 4, 5\}$ operands, varying both recursion depth and denoising step budget. Table 13 reports the equivalent depth-scaling analysis for non-recursive models with 3, 6, 9, 15, and 30 layers.

For Countdown-3, the task is tractable even for the baseline at high T (96.0% RTR at T=30), but recursion substantially accelerates convergence: L=3 reaches 92.4% at T=10, a step budget at which the baseline achieves only 59.4%. For Countdown-4 and -5, the baseline saturates at substantially lower accuracy than the recursive models, suggesting that the difficulty of multi-step arithmetic planning grows faster than what additional denoising steps alone can resolve.

For the depth-scaling analysis (Table 13), results show that a 15-layer non-recursive model (26.7M) on Countdown-4 reaches 80.0% RTR at T=30, while the $(3 \otimes 5)$ model (5.5M) achieves 81.8%, comparable performance at about one-fifth of the parameter count.

Table 12: Countdown performance with different recursive steps. We report mean and standard deviation across 5 sampling runs of 100 samples each as well as the total sampling time.

Model	Dec. steps	RTR %	PPF %	LAF %	TRN	Sampling time (s)
Countdown 3						
Baseline	1	7.6% ± 1.1	19.0% ± 1.2	26.2% ± 2.7	1.046 ± 0.300	2.79
	5	44.8% ± 3.0	51.0% ± 3.4	64.5% ± 4.1	0.520 ± 0.027	8.27
	10	59.4% ± 5.0	63.5% ± 4.2	75.8% ± 3.9	0.346 ± 0.061	7.32
	20	93.2% ± 1.6	94.0% ± 1.7	94.3% ± 1.4	0.068 ± 0.033	41.51
	30	96.0% ± 0.7	96.3% ± 0.5	96.7% ± 0.6	0.037 ± 0.011	56.41
3 recursive steps	1	67.0% ± 4.6	70.1% ± 5.6	76.9% ± 4.6	0.307 ± 0.053	3.52
	5	87.6% ± 1.8	89.2% ± 2.0	92.9% ± 1.9	0.145 ± 0.086	14.43
	10	92.4% ± 2.5	93.2% ± 2.7	95.5% ± 2.0	0.061 ± 0.017	28.24
	20	97.4% ± 1.5	97.6% ± 1.6	98.0% ± 1.2	0.021 ± 0.012	52.85
	30	97.8% ± 1.6	98.2% ± 1.5	98.3% ± 1.3	0.030 ± 0.024	55.96
5 recursive steps	1	67.4% ± 2.1	70.3% ± 2.0	79.5% ± 1.5	0.317 ± 0.088	5.57
	5	85.8% ± 6.8	86.6% ± 6.0	93.0% ± 3.7	0.122 ± 0.059	23.32
	10	91.0% ± 1.6	91.5% ± 1.5	95.4% ± 0.9	0.071 ± 0.015	46.97
	20	98.4% ± 0.9	98.7% ± 0.8	98.7% ± 0.8	0.012 ± 0.011	86.52
	30	98.2% ± 1.9	98.5% ± 1.5	98.8% ± 0.9	0.016 ± 0.016	91.50
Countdown 4						
Baseline	1	0.0% ± 0.0	1.9% ± 0.6	4.1% ± 1.1	1.054 ± 0.169	2.55
	5	6.6% ± 3.1	16.7% ± 3.2	36.2% ± 1.2	1.001 ± 0.231	5.67
	10	18.0% ± 2.5	31.6% ± 3.5	49.3% ± 3.6	0.817 ± 0.140	10.82
	20	38.2% ± 6.2	52.3% ± 4.4	64.5% ± 3.8	0.648 ± 0.206	21.36
	30	44.8% ± 6.0	59.5% ± 5.2	71.5% ± 3.3	0.626 ± 0.046	31.88
3 recursive steps	1	0.8% ± 0.8	5.5% ± 1.2	15.0% ± 1.7	1.243 ± 0.401	4.51
	5	11.6% ± 2.4	28.4% ± 2.3	53.2% ± 2.2	0.908 ± 0.180	16.52
	10	39.6% ± 1.8	56.9% ± 1.7	72.6% ± 1.0	0.593 ± 0.143	32.83
	20	68.6% ± 6.2	77.6% ± 4.0	85.1% ± 3.0	0.364 ± 0.150	63.77
	30	76.8% ± 2.6	84.4% ± 1.6	89.6% ± 1.0	0.256 ± 0.131	83.95
5 recursive steps	1	0.6% ± 0.6	6.7% ± 2.6	17.2% ± 1.9	1.021 ± 0.145	7.46
	5	15.4% ± 3.0	30.9% ± 2.0	54.2% ± 1.6	0.802 ± 0.075	28.37
	10	37.0% ± 2.9	53.7% ± 3.1	69.9% ± 2.0	0.580 ± 0.088	50.52
	20	70.2% ± 7.0	78.3% ± 5.2	85.6% ± 3.3	0.269 ± 0.055	93.76
	30	81.8% ± 2.5	86.9% ± 1.0	90.4% ± 1.1	0.212 ± 0.116	137.65
10 recursive steps	1	0.8% ± 0.8	5.6% ± 2.4	14.8% ± 3.2	0.876 ± 0.043	10.50
	5	12.8% ± 2.2	27.6% ± 1.2	48.1% ± 1.9	0.848 ± 0.118	46.15
	10	43.6% ± 4.2	56.7% ± 4.1	70.2% ± 3.5	0.676 ± 0.317	91.65
	20	74.4% ± 5.3	81.1% ± 4.6	86.1% ± 2.8	0.261 ± 0.102	179.72
	30	86.2% ± 4.1	89.7% ± 3.0	91.7% ± 2.5	0.211 ± 0.184	264.44
Countdown 5						
Baseline	1	0.0% ± 0.0	0.3% ± 0.3	1.3% ± 0.6	0.925 ± 0.112	2.75
	5	0.2% ± 0.5	11.1% ± 1.3	33.7% ± 0.9	1.023 ± 0.192	5.68
	10	3.8% ± 2.4	20.9% ± 2.8	47.0% ± 2.9	0.909 ± 0.069	11.23
	20	9.6% ± 1.8	28.3% ± 2.3	57.4% ± 1.7	0.886 ± 0.161	21.74
	30	15.6% ± 5.1	37.6% ± 4.1	63.3% ± 3.0	0.865 ± 0.229	32.22
	40	17.0% ± 2.6	40.5% ± 4.4	65.1% ± 3.4	0.889 ± 0.168	166.18
3 recursive steps	1	0.0% ± 0.0	0.5% ± 0.4	1.6% ± 0.6	1.036 ± 0.229	3.50
	5	4.8% ± 1.8	20.9% ± 1.2	47.6% ± 1.7	0.971 ± 0.169	14.00
	10	10.8% ± 2.2	31.0% ± 1.6	60.9% ± 1.9	0.758 ± 0.068	28.00
	20	21.6% ± 4.2	42.6% ± 5.5	70.9% ± 2.5	0.913 ± 0.153	55.50
	30	36.6% ± 4.9	57.5% ± 2.2	79.6% ± 1.0	0.746 ± 0.160	83.00
	40	39.6% ± 7.0	60.3% ± 6.0	80.8% ± 3.7	0.782 ± 0.125	107.00
5 recursive steps	1	0.0% ± 0.0	0.7% ± 0.4	2.6% ± 0.6	1.136 ± 0.123	6.42
	5	4.8% ± 1.9	23.6% ± 1.9	47.0% ± 4.5	1.051 ± 0.275	24.82
	10	14.2% ± 3.4	34.0% ± 2.1	59.6% ± 2.1	0.864 ± 0.177	48.76
	20	24.6% ± 7.2	44.6% ± 6.4	69.7% ± 3.8	0.822 ± 0.303	97.16
	30	44.4% ± 4.7	62.1% ± 4.8	80.4% ± 2.1	0.585 ± 0.122	141.88
	40	46.8% ± 6.5	64.3% ± 4.3	81.3% ± 1.3	0.586 ± 0.270	174.04
10 recursive steps	1	0.2% ± 0.5	1.2% ± 1.0	3.5% ± 0.9	1.107 ± 0.330	10.50
	5	5.8% ± 1.3	22.4% ± 1.9	44.2% ± 2.8	0.856 ± 0.204	43.51
	10	14.4% ± 4.3	35.1% ± 4.1	58.2% ± 4.7	0.880 ± 0.166	86.57
	20	28.4% ± 4.4	49.3% ± 4.2	69.4% ± 2.5	0.654 ± 0.033	173.24
	30	56.4% ± 5.0	71.4% ± 4.2	82.0% ± 2.0	0.671 ± 0.355	260.25
	40	53.8% ± 4.3	69.6% ± 3.4	80.4% ± 2.0	0.538 ± 0.208	330.43

Table 13: Countdown performance with models of different depth. We report mean and standard deviation across 5 sampling runs of 100 samples each as well as the total sampling time.

Model (params)	Dec. steps	RTR %	PPF %	LAF %	TRN	Sampling time (s)
Countdown 3						
3 layers (5.5M)	1	7.6% ± 1.1	19.0% ± 1.2	26.2% ± 2.7	1.046 ± 0.300	2.79
	5	44.8% ± 3.0	51.0% ± 3.4	64.5% ± 4.1	0.520 ± 0.027	8.27
	10	59.4% ± 5.0	63.5% ± 4.2	75.8% ± 3.9	0.346 ± 0.061	7.32
	20	93.2% ± 1.6	94.0% ± 1.7	94.3% ± 1.4	0.068 ± 0.033	41.51
	30	96.0% ± 0.7	96.3% ± 0.5	96.7% ± 0.6	0.037 ± 0.011	56.41
6 layers (10.8M)	1	20.0% ± 4.1	28.5% ± 3.5	39.9% ± 4.8	0.865 ± 0.201	2.52
	5	56.8% ± 6.5	59.9% ± 6.4	73.2% ± 4.9	0.413 ± 0.093	9.36
	10	74.4% ± 5.8	76.4% ± 5.9	85.4% ± 4.0	0.341 ± 0.186	18.48
	20	97.2% ± 1.8	97.4% ± 1.5	97.6% ± 1.4	0.025 ± 0.014	34.41
	30	97.0% ± 1.2	97.3% ± 1.2	97.8% ± 1.2	0.028 ± 0.013	35.78
9 layers (16.1M)	1	36.0% ± 4.6	43.1% ± 3.4	54.8% ± 2.6	0.558 ± 0.043	4.89
	5	72.4% ± 3.6	75.0% ± 3.5	83.2% ± 2.6	0.389 ± 0.261	13.17
	10	86.6% ± 2.7	87.6% ± 2.1	92.1% ± 2.0	0.121 ± 0.013	26.12
	20	96.0% ± 1.9	96.5% ± 1.7	97.2% ± 1.8	0.036 ± 0.015	48.75
	30	95.0% ± 1.9	95.7% ± 1.4	96.2% ± 1.4	0.131 ± 0.208	51.51
15 layers (26.7M)	1	59.0% ± 3.9	64.3% ± 4.0	73.4% ± 3.0	0.405 ± 0.090	5.67
	5	88.0% ± 1.4	90.0% ± 0.7	91.6% ± 0.8	0.119 ± 0.049	20.54
	10	94.4% ± 4.2	95.1% ± 3.6	96.1% ± 2.8	0.052 ± 0.037	40.93
	20	95.6% ± 2.2	96.4% ± 2.0	96.8% ± 1.6	0.036 ± 0.025	76.25
	30	95.2% ± 1.3	96.0% ± 1.2	96.6% ± 1.1	0.038 ± 0.013	83.87
Countdown 4						
3 layers (5.5M)	1	0.0% ± 0.0	1.9% ± 0.6	4.1% ± 1.1	1.054 ± 0.169	2.55
	5	6.6% ± 3.1	16.7% ± 3.2	36.2% ± 1.2	1.001 ± 0.231	5.67
	10	18.0% ± 2.5	31.6% ± 3.5	49.3% ± 3.6	0.817 ± 0.140	10.82
	20	38.2% ± 6.2	52.3% ± 4.4	64.5% ± 3.8	0.648 ± 0.206	21.36
	30	44.8% ± 6.0	59.5% ± 5.2	71.5% ± 3.3	0.626 ± 0.046	31.88
6 layers (10.8M)	1	0.4% ± 0.6	3.8% ± 1.0	10.7% ± 1.2	1.003 ± 0.180	3.33
	5	10.2% ± 3.6	24.0% ± 4.8	47.9% ± 1.2	0.915 ± 0.129	13.44
	10	34.8% ± 4.3	48.9% ± 5.0	65.1% ± 3.2	0.755 ± 0.269	26.62
	20	59.8% ± 3.1	71.7% ± 3.7	79.2% ± 3.3	0.385 ± 0.058	52.90
	30	68.4% ± 4.0	77.6% ± 2.3	83.1% ± 1.3	0.445 ± 0.255	76.06
9 layers (16.1M)	1	0.2% ± 0.5	5.7% ± 1.7	14.1% ± 2.1	1.023 ± 0.108	5.19
	5	15.0% ± 3.5	28.0% ± 2.8	51.3% ± 2.8	0.837 ± 0.140	20.07
	10	38.8% ± 2.7	51.9% ± 1.5	68.2% ± 1.9	0.591 ± 0.071	42.87
	20	64.2% ± 4.4	73.0% ± 3.5	80.0% ± 2.1	0.359 ± 0.091	84.29
	30	74.2% ± 8.0	81.3% ± 5.7	86.1% ± 4.3	0.227 ± 0.086	106.25
15 layers (26.7M)	1	1.4% ± 1.1	9.1% ± 2.1	24.2% ± 2.2	0.999 ± 0.303	5.65
	5	21.4% ± 4.9	33.9% ± 3.1	60.1% ± 3.6	0.781 ± 0.124	22.96
	10	48.2% ± 2.6	59.6% ± 2.2	75.5% ± 1.8	0.453 ± 0.046	41.23
	20	69.4% ± 4.3	78.3% ± 2.6	84.7% ± 1.8	0.269 ± 0.066	81.46
	30	80.0% ± 2.1	85.7% ± 1.9	89.5% ± 2.3	0.183 ± 0.042	121.79
30 layers (53.3M)	1	5.2% ± 2.3	15.2% ± 2.1	32.4% ± 3.5	0.886 ± 0.075	8.69
	5	32.4% ± 4.9	42.6% ± 3.7	67.2% ± 3.7	0.616 ± 0.047	40.17
	10	58.0% ± 2.7	68.7% ± 2.8	80.4% ± 1.3	0.356 ± 0.051	79.95
	20	69.6% ± 2.6	78.2% ± 1.8	85.4% ± 2.0	0.362 ± 0.152	159.88
	30	72.6% ± 2.4	81.3% ± 2.0	86.3% ± 1.6	0.300 ± 0.144	231.88
Countdown 5						
3 layers (5.5M)	1	0.0% ± 0.0	0.3% ± 0.3	1.3% ± 0.6	0.925 ± 0.112	2.75
	5	0.2% ± 0.5	11.1% ± 1.3	33.7% ± 0.9	1.023 ± 0.192	5.68
	10	3.8% ± 2.4	20.9% ± 2.8	47.0% ± 2.9	0.909 ± 0.069	11.23
	20	9.6% ± 1.8	28.3% ± 2.3	57.4% ± 1.7	0.886 ± 0.161	21.74
	30	15.6% ± 5.1	37.6% ± 4.1	63.3% ± 3.0	0.865 ± 0.229	32.22
	40	17.0% ± 2.6	40.5% ± 4.4	65.1% ± 3.4	0.889 ± 0.168	166.18
6 layers (10.8M)	1	0.0% ± 0.0	0.9% ± 0.4	2.1% ± 1.1	0.976 ± 0.162	2.51
	5	1.8% ± 0.8	17.5% ± 2.3	45.7% ± 3.3	0.923 ± 0.150	9.51
	10	8.6% ± 0.6	27.9% ± 1.3	54.4% ± 1.2	1.047 ± 0.397	18.40
	20	15.4% ± 2.9	36.9% ± 5.4	64.3% ± 2.6	1.017 ± 0.176	36.99
	30	28.4% ± 3.8	51.2% ± 4.0	71.9% ± 2.8	0.708 ± 0.121	54.75
	40	29.4% ± 3.8	53.6% ± 3.1	73.3% ± 1.6	0.845 ± 0.279	72.69
9 layers (16.1M)	1	0.0% ± 0.0	1.3% ± 0.4	2.7% ± 0.9	1.128 ± 0.232	4.01
	5	2.8% ± 1.3	18.5% ± 2.2	43.4% ± 1.3	0.953 ± 0.201	13.15
	10	9.4% ± 3.2	28.6% ± 3.4	56.0% ± 3.3	0.992 ± 0.322	25.85
	20	22.0% ± 5.3	40.6% ± 4.3	65.0% ± 3.5	0.718 ± 0.060	56.03
	30	37.8% ± 3.3	57.4% ± 2.6	75.0% ± 2.3	0.714 ± 0.226	87.30
	40	39.6% ± 2.3	60.7% ± 1.6	77.4% ± 1.5	0.604 ± 0.090	99.92
15 layers (26.7M)	1	0.0% ± 0.0	1.2% ± 0.6	3.0% ± 0.7	0.977 ± 0.170	5.81
	5	4.0% ± 1.7	20.3% ± 3.4	44.4% ± 3.2	0.909 ± 0.076	23.30
	10	11.2% ± 2.4	31.7% ± 2.3	60.8% ± 0.9	1.091 ± 0.381	43.96
	20	21.8% ± 5.0	44.6% ± 3.7	69.7% ± 2.2	0.764 ± 0.109	83.75
	30	42.2% ± 4.6	63.5% ± 2.9	81.2% ± 1.6	0.587 ± 0.079	125.12
	40	45.8% ± 3.3	64.9% ± 2.0	82.2% ± 2.0	0.530 ± 0.073	161.05
30 layers (53.3M)	1	0.0% ± 0.0	1.7% ± 0.3	3.6% ± 0.8	0.912 ± 0.047	8.57
	5	5.0% ± 1.6	21.5% ± 1.7	47.1% ± 3.3	0.859 ± 0.117	39.80
	10	15.0% ± 3.9	34.8% ± 3.3	60.3% ± 2.5	0.927 ± 0.347	79.44
	20	28.2% ± 3.9	47.5% ± 3.1	70.0% ± 1.4	0.698 ± 0.165	158.83
	30	45.4% ± 5.2	65.2% ± 4.2	79.6% ± 2.6	0.479 ± 0.048	237.60
	40	47.8% ± 6.0	66.6% ± 3.4	80.6% ± 3.2	0.494 ± 0.149	304.52

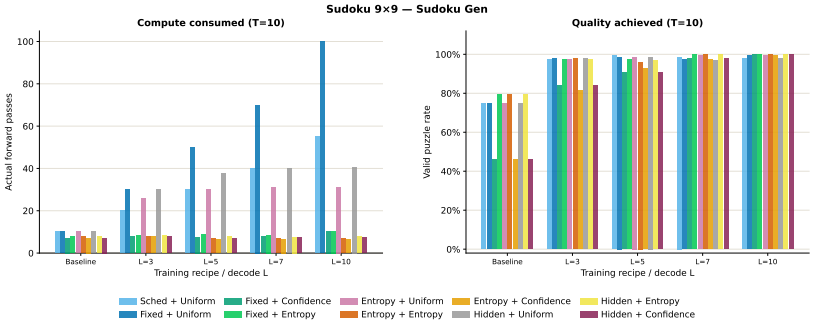
G.6 DECODING ALGORITHMS COMPARISON

Figure 17 compares decoding policies on the four structured tasks (Sudoku 9×9 and Countdown-$\{3, 4, 5\}$), reporting task quality against *realized* forward passes (NFE) at a fixed denoising budget T. Bars contrast the non-recursive Baseline with R-MDM decoded at $L_s \in \{3, 5, 7, 10\}$.

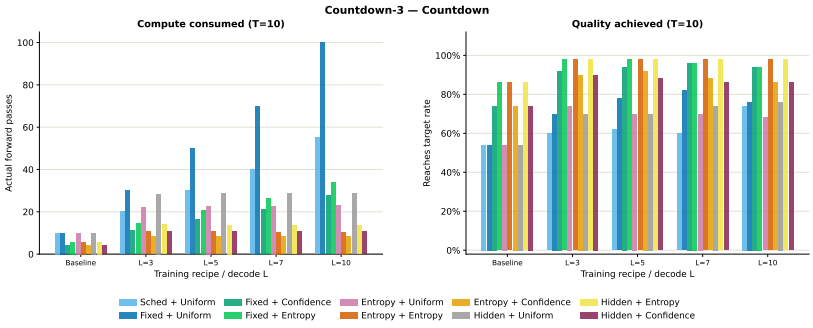
Entropy as a loop-stopping signal. Uniform commits spend a near-constant L loops per denoising step, so NFE scales roughly as $L\times T$. Entropy-based controls break this coupling. **Entropy-budget** commits (`fixed_entropy`, `entropy_entropy`) stop unmasking once a Shannon budget is exhausted, yielding low, nearly T-invariant NFE. **Entropy-loop** early stopping (`entropy_uniform`, `entropy_entropy`) halts recursion when the mean token-entropy gain falls below λ, so extra decode capacity is spent only where the model is still uncertain. Together, these mechanisms convert recursion from a fixed compute tax into an adaptive budget that concentrates loops on hard tokens/steps.

Entropy-gain stopping with entropy commits. Across these four tasks, `entropy_entropy` is consistently near the quality–compute frontier: it matches or exceeds `fixed_uniform` quality while using far fewer forwards. On Sudoku it reaches (near-)perfect VPR at single-digit NFE, whereas `fixed_uniform` needs $\sim L\times T$ forwards for the same saturation. On Countdown-3 it attains the top Reaches-Target rate ($\approx 98\%$) at NFE $\approx$ 10–15, comparable to `fixed_entropy` and well below uniform looping. On the harder Countdown-4/5 instances the gap is sharper: `fixed_uniform` peaks at moderate L and then *degrades* as L grows (quality drops while NFE keeps climbing), whereas `entropy_entropy` stays competitive at high L with roughly stable, low NFE—evidence that entropy-gated looping avoids wasteful over-refinement.

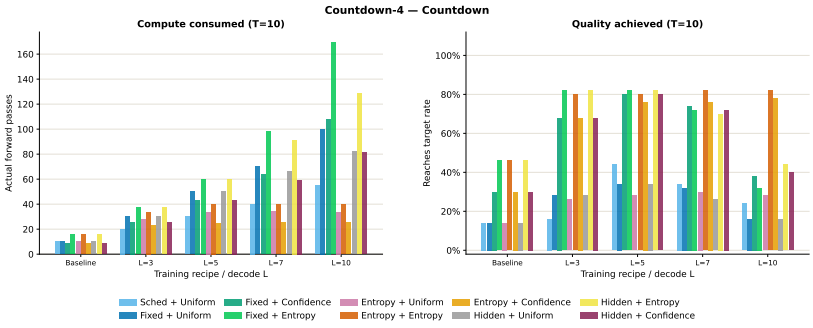
Takeaway. Recursion helps when it is spent selectively. Entropy-based stopping—especially the `entropy_entropy` combination of gain-based early exit and budgeted commits—delivers the best quality per forward pass in these plots, and is the most robust recipe when increasing L on hard Countdown instances. It is not uniformly best in absolute terms: on Countdown-5 the highest reaches-target rate is still obtained by `fixed_uniform`, at several times the NFE (Section 5.2).



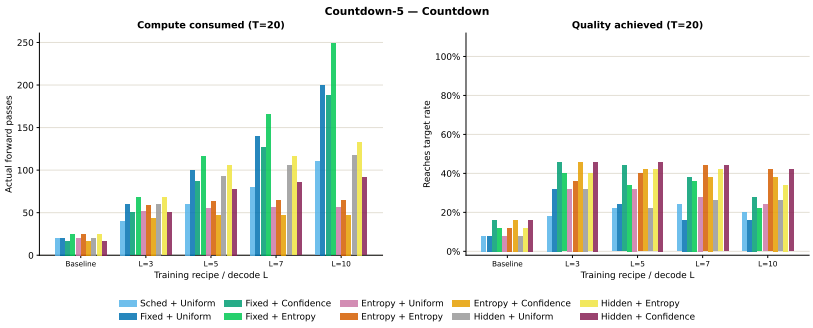
(a) Sudoku 9×9 (VPR), T=10.



(b) Countdown-3 (RTR), T=10.



(c) Countdown-4 (RTR), T=10.



(d) Countdown-5 (RTR), T=20.

Figure 17: Decoding comparison on Baseline vs. R-MDM ($L_s \in \{3, 5, 7, 10\}$). Left: realized NFE (forward passes). Right: task quality at the indicated T. Entropy-gated policies (particularly `entropy_entropy`) match or beat `fixed_uniform` at substantially lower NFE.

G.7 RNA INFILLING: EXTENDED RESULTS

Table 14 reports the paired structural gap ΔZ and the realized NFE for the four decoders of Section 5.1, averaged over the three length strata ($n \in \{76, 110, 254\}$, decoded at $T = 6, 9, 20$) described in Appendix C. Rows with L=1 are the non-recursive baseline, for which the two loop rules coincide; $L \in \{3, 5\}$ use the schedule-trained recursive checkpoint. Lower ΔZ is better. All mean values but one are positive, so infilled sequences are on average somewhat less structured than the native molecules they replace; the exception, `fixed_entropy` at L=5 with 60% masking ($\Delta Z = -0.014 \pm 0.063$), is within one standard error of zero. Since these are averages, individual completions can still fold more tightly than their native counterpart.

Table 14: RNA infilling at 40%, 50% and 60% masked positions. We report the Z-score of the infilled sequence, the paired structural gap ΔZ, masked-position recovery, exact recovery and realized NFE. All values are unweighted means over the three length strata ($n \in \{76, 110, 254\}$, 500 sequences each) at $T = 6$, 9 and 20 for the three strata; uncertainties are standard errors combined across strata.

Policy	L	Z ↓	ΔZ ↓	Masked rec. %	Exact rec. % ↓	NFE
			40% masked			
`fixed_uniform`	1	-1.381 ± 0.052	0.478 ± 0.056	60.3 ± 0.7	$\mathbf{1.6} \pm 0.3$	11.7
	3	-1.628 ± 0.056	0.229 ± 0.052	62.7 ± 0.7	2.5 ± 0.4	35.0
	5	-1.682 ± 0.054	0.175 ± 0.052	62.4 ± 0.7	2.3 ± 0.4	58.3
`fixed_entropy`	1	-1.408 ± 0.048	0.453 ± 0.055	60.3 ± 0.7	2.0 ± 0.4	13.2
	3	-1.688 ± 0.055	0.169 ± 0.051	62.4 ± 0.7	2.2 ± 0.4	33.5
	5	$\mathbf{-1.716} \pm 0.057$	$\mathbf{0.140} \pm 0.051$	62.4 ± 0.7	1.8 ± 0.3	50.3
`entropy_uniform`	1	-1.381 ± 0.052	0.478 ± 0.056	60.3 ± 0.7	$\mathbf{1.6} \pm 0.3$	11.7
	3	-1.554 ± 0.054	0.304 ± 0.052	62.1 ± 0.7	2.3 ± 0.4	26.7
	5	-1.588 ± 0.053	0.271 ± 0.050	62.3 ± 0.7	2.5 ± 0.4	27.4
`entropy_entropy`	1	-1.408 ± 0.048	0.453 ± 0.055	60.3 ± 0.7	2.0 ± 0.4	13.2
	3	-1.612 ± 0.054	0.247 ± 0.049	62.0 ± 0.7	2.5 ± 0.4	25.9
	5	-1.658 ± 0.056	0.200 ± 0.051	62.3 ± 0.7	2.7 ± 0.4	26.2
			50% masked			
`fixed_uniform`	1	-1.234 ± 0.047	0.625 ± 0.057	58.8 ± 0.7	$\mathbf{1.0} \pm 0.3$	11.7
	3	-1.659 ± 0.057	0.199 ± 0.055	61.1 ± 0.7	1.7 ± 0.3	35.0
	5	-1.767 ± 0.057	0.092 ± 0.057	61.2 ± 0.7	1.6 ± 0.3	58.3
`fixed_entropy`	1	-1.413 ± 0.048	0.447 ± 0.056	58.8 ± 0.7	1.1 ± 0.3	16.5
	3	-1.746 ± 0.054	0.114 ± 0.055	61.0 ± 0.7	1.3 ± 0.3	41.5
	5	$\mathbf{-1.800} \pm 0.058$	$\mathbf{0.056} \pm 0.059$	61.1 ± 0.7	1.5 ± 0.3	61.3
`entropy_uniform`	1	-1.234 ± 0.047	0.625 ± 0.057	58.8 ± 0.7	$\mathbf{1.0} \pm 0.3$	11.7
	3	-1.544 ± 0.053	0.312 ± 0.054	60.9 ± 0.7	1.5 ± 0.3	26.7
	5	-1.512 ± 0.053	0.345 ± 0.055	60.7 ± 0.7	1.7 ± 0.3	27.6
`entropy_entropy`	1	-1.413 ± 0.048	0.447 ± 0.056	58.8 ± 0.7	1.1 ± 0.3	16.5
	3	-1.619 ± 0.054	0.238 ± 0.055	60.9 ± 0.7	1.5 ± 0.3	32.0
	5	-1.584 ± 0.052	0.275 ± 0.054	60.9 ± 0.7	1.4 ± 0.3	32.4
			60% masked			
`fixed_uniform`	1	-1.257 ± 0.049	0.604 ± 0.061	57.4 ± 0.7	$\mathbf{0.7} \pm 0.2$	11.7
	3	-1.588 ± 0.055	0.270 ± 0.060	59.3 ± 0.7	1.3 ± 0.3	35.0
	5	-1.765 ± 0.059	0.094 ± 0.062	59.7 ± 0.7	1.0 ± 0.3	58.3
`fixed_entropy`	1	-1.340 ± 0.047	0.520 ± 0.060	57.3 ± 0.7	$\mathbf{0.7} \pm 0.2$	19.7
	3	-1.769 ± 0.058	0.090 ± 0.060	59.2 ± 0.7	1.0 ± 0.3	48.7
	5	$\mathbf{-1.871} \pm 0.062$	$\mathbf{-0.014} \pm 0.063$	59.5 ± 0.7	0.8 ± 0.2	70.9
`entropy_uniform`	1	-1.257 ± 0.049	0.604 ± 0.061	57.4 ± 0.7	$\mathbf{0.7} \pm 0.2$	11.7
	3	-1.501 ± 0.053	0.358 ± 0.058	58.9 ± 0.7	1.0 ± 0.3	26.7
	5	-1.464 ± 0.054	0.395 ± 0.058	59.1 ± 0.7	0.9 ± 0.2	27.5
`entropy_entropy`	1	-1.340 ± 0.047	0.520 ± 0.060	57.3 ± 0.7	$\mathbf{0.7} \pm 0.2$	19.7
	3	-1.570 ± 0.054	0.289 ± 0.058	59.2 ± 0.7	0.9 ± 0.2	38.3
	5	-1.600 ± 0.055	0.259 ± 0.061	58.9 ± 0.7	0.9 ± 0.2	38.7

We evaluate partial infilling on held-out validation sequences: a fraction of nucleotide positions is blanked uniformly at random and the model fills them with the remaining positions visible.

G.8 TINYGSM: EXTENDED RESULTS

Decoding is infill on a 512-token canvas: the GSM8K question is kept as a prefix and every subsequent position is masked. We evaluate n=512 GSM8K test problems, shared across every cell so paired comparisons use common random numbers. Tokens are decoded greedily (top-k=1, temperature 0.8).

Table 15: TinyGSM GSM8K accuracy, mean NFE and sampling time for the schedule-trained recursive checkpoint ($L \in \{3, 5\}$) and the non-recursive baseline (L=1). Uncertainties are binomial standard errors with $n = 512$ shared test problems. Sampling time is the total wall-clock time to decode all 512 problems.

Policy	L	T	Acc.	NFE	Sampling time (s)
`fixed_uniform`	1	32	6.4% ± 1.1	32	283
	1	128	**15.0%** ± 1.6	128	1118
	1	256	11.5% ± 1.4	256	2237
	1	448	10.5% ± 1.4	440	3792
`entropy_uniform`	1	32	6.4% ± 1.1	32	306
	1	128	15.0% ± 1.6	128	1184
	1	256	11.5% ± 1.4	256	2286
	1	448	10.5% ± 1.4	440	3971
`fixed_entropy`	1	—	13.7% ± 1.5	22	218
`entropy_entropy`	1	—	13.7% ± 1.5	22	213
`fixed_uniform`	3	32	16.0% ± 1.6	96	566
	3	128	21.5% ± 1.8	384	2202
	3	256	**24.2%** ± 1.9	768	4407
	3	448	23.4% ± 1.9	1319	7569
`entropy_uniform`	3	32	13.5% ± 1.5	82	519
	3	128	21.1% ± 1.8	326	2043
	3	256	21.9% ± 1.8	650	4100
	3	448	20.3% ± 1.8	1115	6987
`fixed_entropy`	3	—	24.0% ± 1.9	69	412
`entropy_entropy`	3	—	22.3% ± 1.8	57	374
`fixed_uniform`	5	32	7.8% ± 1.2	160	906
	5	128	16.0% ± 1.6	640	3623
	5	256	15.8% ± 1.6	1280	7230
	5	448	15.8% ± 1.6	2198	12382
`entropy_uniform`	5	32	11.1% ± 1.4	83	525
	5	128	16.8% ± 1.7	329	2066
	5	256	17.4% ± 1.7	653	4103
	5	448	16.4% ± 1.6	1113	6977
`fixed_entropy`	5	—	18.4% ± 1.7	136	790
`entropy_entropy`	5	—	17.2% ± 1.7	63	406

Recursion helps. The schedule-trained model peaks at 24.2% ± 1.9 (`fixed_uniform`, L=3, T=256), against 15.0% ± 1.6 for the non-recursive baseline at its best setting (T=128). Every recursive cell at L=3 with $T \geq 128$, and every entropy-budget cell at L=3, sits above that baseline.

Commit width is the most useful adaptive axis. `fixed_entropy` at L=3 matches the headline accuracy (24.0% ± 1.9 vs 24.2% ± 1.9) at 68.8 NFE instead of 768 (an 11× saving) because δ=2 commits several confident tokens per step rather than spreading them across a long uniform schedule. Pushing uniform T from 256 to 448 does not help (24.2% → 23.4%). Entropy-gain stopping on top of that budget (`entropy_entropy`) is slightly worse (22.3% ± 1.8) at comparable cost (57 NFE): the loop controller trims only about half a loop relative to L=3 and does not improve solves. Entropy-gain stopping with a uniform schedule (`entropy_uniform`) likewise tracks `fixed_uniform` but never overtakes it.

On the baseline, extra denoising after T=128 hurts (15.0% → 11.5% at T=256), and the entropy budget is a cheap alternative (13.7% at 22 NFE).

H SAMPLES

H.1 SUDOKU 9×9

Baseline $L = 1$, $T = 20$



Fixed-L, uniform $L = 10$, $T = 20$



Fixed-L, entropy $L = 10$



Entropy-stop, entropy $L = 10$



Entropy-stop, uniform $L = 10$, $T = 20$



Figure 18: Sudoku trajectories.

H.2 COUNTDOWN-5

Table 16: Countdown-5 samples.

Problem	Model completion	Outcome
Baseline (L=1, T=20)		
79,7,7,72,57=108	72+7=79,79+7=86,79-57=22,22+86=108	valid
41,92,85,18,71=165	85+85=126,92-71=21,18+126=144,144+21=165	arithmetic, number reused
42,23,16,51,53=-980	16+53=69,42-69=-23,-23*53=-929,-929-51=-980	arithmetic, number reused
fixed_uniform (L=5, T=20)		
79,43,65,68,11=244	68+79=147,65+43=108,11-147=-136,108--136=244	valid
79,7,7,72,57=108	72+7=79,7+79=86,79-57=22,86+22=108	valid
41,92,85,18,71=165	18+18=103,103-71=32,92+92=124,124+41=165	arithmetic, number reused
42,23,16,51,53=-980	16+16=96,23-51=-28,-2**43=-984,-984-96=-980	arithmetic, number reused, malformed
fixed_entropy (L=5)		
79,7,7,72,57=108	72+7=79,79-57=22,7+22=29,29+79=108	valid
24,87,74,61,90=214	87-61=26,24+26=50,74+90=164,164+50=214	valid
41,92,85,18,71=165	85+92=177,18+41=59,59-177=-94,71--94=165	arithmetic
42,23,16,51,53=-980	23-53=-30,-30-16=-76,51+-76=-25,42*-25=-980	arithmetic
entropy_entropy (L=5)		
41,92,85,18,71=165	41+85=126,126+18=144,92-71=21,21+144=165	valid
79,43,65,68,11=244	65+79=144,43+144=187,11-187=-176,68--176=244	valid
42,23,16,51,53=-980	53-42=11,16*53=968,11-968=-957,-957-23=-980	arithmetic, number reused
45,66,65,6,34=1485	65-6=59,66-34=32,59-32=27,45*27=1485	arithmetic
entropy_uniform (L=5, T=20)		
79,7,7,72,57=108	7+72=79,7+79=86,79-57=22,86+22=108	valid
4,35,33,62,63=7	33-4=29,62-35=27,63-29=34,34-27=7	valid
41,92,85,18,71=165	18+18=103,103-71=32,32+92=124,124+41=165	arithmetic, number reused
42,23,16,51,53=-980	42-23=19,51*51=946,19-946=-927,-927-53=-980	arithmetic, number reused

H.3 TINYGSM

Problem	Generated program	Outcome
Baseline (L=1, T=128)		
Argo has 200 toys. He gives 40 to Alyssa, 80 to Bonnie, and 30 to Nicky. How many remain?	`toys=200; toys-=40; toys-=80; toys-=30`	valid
Laptop $1000, 20% off. How much to pay?	`result=20`	constant guess
3 rose bushes, 25 roses each, 8 thorns per rose. Total thorns?	`result=18`	constant guess
Ann is 9; her brother is twice her age. Brother's age in 3 years?	`result=...`	truncated
`fixed_uniform` (L=3, T=256)		
Laptop $1000, 20% off. How much to pay?	`price=1000; discount=0.2; discounted_price=price-(price*discount)`	valid
3 rose bushes, 25 roses each, 8 thorns per rose. Total thorns?	`roses=3*25; thorns=roses*8`	valid
22 games, won 8 more than lost. How many wins?	`games_played=22; games_won=games_played+8`	wrong equation
Eraser $2, pencil $3; cost of 6 erasers and 8 pencils?	`eraser=2; pencil=3; total=6*eraser+8*pencil` *(no return)*	no return
`fixed_entropy` (L=3)		
Laptop $1000, 20% off. How much to pay?	`price=1000; discount=0.2; discounted_price=price-(price*discount)`	valid
3 rose bushes, 25 roses each, 8 thorns per rose. Total thorns?	`roses=3*25; thorns=roses*8`	valid
22 games, won 8 more than lost. How many wins?	`total_games=22; games_won=(total_games+8)/2`	valid
Eraser $2, pencil $3; cost of 6 erasers and 8 pencils?	`erasers=6; pencils=8; cost=erasers*2+pencils*3` *(no return)*	no return
`entropy_entropy` (L=3)		
Laptop $1000, 20% off. How much to pay?	`price=1000; discount=0.2; discounted_price=price-(price*discount)`	valid
22 games, won 8 more than lost. How many wins?	`total_games=22; wins=(total_games+8)/2`	valid
Eraser $2, pencil $3; cost of 6 erasers and 8 pencils?	`cost=2*6+3*8`	valid
John collects peaches for 3 hours at 2 per minute. How many peaches?	`hours=3; minutes=2; minutes=hours*60; result=hours*minutes`	wrong equation
`entropy_uniform` (L=3, T=256)		
Laptop $1000, 20% off. How much to pay?	`price=1000; discount=0.2; discounted_price=price-(price*discount)`	valid
3 rose bushes, 25 roses each, 8 thorns per rose. Total thorns?	`roses=3*25; thorns=roses*8`	valid
22 games, won 8 more than lost. How many wins?	`games_played=22; games_won=games_played+8`	wrong equation
Eraser $2, pencil $3; cost of 6 erasers and 8 pencils?	`eraser=2; pencil=3; total=6*eraser+8*pencil` *(no return)*	no return

H.4 RNA

Held-out validation sequence #285 (n=76 **nt**)

40% blanks (30/76)

```
WT AGGGCUAUAGCUCAGUUAGGUAGAGCGCCUCGUUUACACACCGAGAAGGUCUACGGUUCGAAUCCGUAUAGCCCUA
IN AG.GC..UAGCU..G..AG.UAGA...C.UC.UUUACACAC..AGA...U.U.CGG....AA..CGUAU...CCU.
B1 AGGGCUAUAGCUCAGUCAGGUAGAGCACCUCGUUUACACACAGAGAAGGUCUCCGGUUCAAAUCCGUAUAGCCCUA
```
Baseline L=1: recovery 0.83, $Z = -3.58$ (WT $Z = -2.27$)
```
R5 AGGGCUAUAGCUCAGUCAGGUAGAGCACUUCGUUUACACACGAAGAAGGUCUACGGUUCAAAUCCGUAUAGCCCUA
```
Recursive L=5: recovery 0.80, $Z = -3.84$ (WT $Z = -2.27$)

50% blanks (38/76)

```
WT AGGGCUAUAGCUCAGUUAGGUAGAGCGCCUCGUUUACACACCGAGAAGGUCUACGGUUCGAAUCCGUAUAGCCCUA
IN AG.G...UAGCU.....AG.UA.A...C.UC.UU.ACA.AC..AGA...U.U..GG....AA..CGUAU...C...
B1 AGGGCUAUAGCUCAGUCAGUUAGAGCGCCUCGUUCACACACAGAGAAGGUCUACGGUUCAAAUCCGUAUACCCCUA
```
Baseline L=1: recovery 0.84, $Z = -2.42$ (WT $Z = -2.27$)
```
R5 AGGGCUAUAGCUCAGCCAGGUAGAGCACUUCGUUUACACACGAAGAAUGUCUACGGUUCGAAUCCGUAUACCCCUA
```
Recursive L=5: recovery 0.79, $Z = -2.40$ (WT $Z = -2.27$)

60% blanks (46/76)

```
WT AGGGCUAUAGCUCAGUUAGGUAGAGCGCCUCGUUUACACACCGAGAAGGUCUACGGUUCGAAUCCGUAUAGCCCUA
IN AG.G...UA.CU......G..A.A...C.U..U..ACA..C..AGA...U.U..GG....A...CGU.U...C...
B1 AGGGGUAUAGCUCAGUUGGUUAGAGCACCUGUUUCACACUCAGAGAAGGUCUACGGUUCAAUUCCGUAUGUCCCUA
```
Baseline L=1: recovery 0.72, $Z = -0.93$ (WT $Z = -2.27$)
```
R5 AGGGUUAUAGCUCAGUUGGUUAGAGCACUUGUUUUACAUUCUAAGAAAGUCUACGGUUCAAAUCCGUAUGGCCCUA
```
Recursive L=5: recovery 0.70, $Z = -1.47$ (WT $Z = -2.27$)

Held-out validation sequence #463 (n=76 **nt**)

40% blanks (30/76)

```
WT CUCCUGUUAGUUCAAACGGAAGAACAUCGACCUUCUAAGUCGCAGGUUACUGGUUCGAGUCCAGUACGGGAGACAA
IN C.C.UGUUA.U.CAA.C.G.AG.A.AU..ACCUU.U..G.C.C.GGU..C.GG...GA.UC.AGUAC.GGA...A.
B1 CUCCUGUUAGUUCAAUCGGCAGAACAUUGACCUUUUAAGUCACAGGUUACUGGUUCGAGUCCAGUACGGGAGGCAA
```
Baseline L=1: recovery 0.80, $Z = -1.68$ (WT $Z = -2.24$)
```
R5 CUCUUGUUAGUUCAACCGGCAGAACAUUGACCUUUUAAGUCACAGGUUACUGGUUCGAAUCCAGUACAGGAGACAA
```
Recursive L=5: recovery 0.73, $Z = -1.99$ (WT $Z = -2.24$)

50% blanks (38/76)

```
WT CUCCUGUUAGUUCAAACGGAAGAACAUCGACCUUCUAAGUCGCAGGUUACUGGUUCGAGUCCAGUACGGGAGACAA
IN C.C.U.UUA.U.CAA.C...A..A.AU..ACCUU....G.C.C.G.U..C.GG...G..UC...UAC.GGA...A.
B1 CUCCUGUUAGUUCAAUCGGUAGAACAUUGACCUUUAAAGUCGCAGGUUACAGGUUCGAGUCCUGUACGGGAGGCAA
```
Baseline L=1: recovery 0.79, $Z = -1.52$ (WT $Z = -2.24$)
```
R5 CUCCUGUUAGUUCAAUCGGCAGAACAUUGACCUUUUAAGUCGCAGGUUACAGGUUCGAAUCCUGUACGGGAGACAA
```
Recursive L=5: recovery 0.82, $Z = -1.95$ (WT $Z = -2.24$)

60% blanks (46/76)

```
WT CUCCUGUUAGUUCAAACGGAAGAACAUCGACCUUCUAAGUCGCAGGUUACUGGUUCGAGUCCAGUACGGGAGACAA
IN C.....UUA...CAA.C...A..A.AU..AC..U....G.C.C.G.U..C.GG.......C...UAC.GG....A.
B1 CCCCCGUUAGUUCAACCGGCAGAACAUCGACCUUUAAAGUCGCAGGUUACAGGUUCAAGUCCCGUACGGGGGGCAA
```
Baseline L=1: recovery 0.76, $Z = -0.90$ (WT $Z = -2.24$)
```
R5 CCCUCGUUAGUUCAACCGGUAGAACAUCGACCUUUUAAGUCGCAGGUUACAGGUUCAAAUCCUGUACGGGGGACAA
```
Recursive L=5: recovery 0.76, $Z = -1.31$ (WT $Z = -2.24$)