Discrete Diffusion Models for Code: Bidirectional Infilling and Hybrid Autoregressive-Diffusion Pipelines

Anonymous ACL submission

Abstract

Autoregressive (AR) models dominate code generation but are structurally limited: they process tokens left-to-right and cannot natively condition on *both* prefix and suffix context simultaneously. We investigate discrete diffusion models as a complementary paradigm for code. Built on the Masked Discrete Language Model (MDLM) framework with a Qwen2.5-Coder-1.5B backbone, our diffusion model achieves bidirectional context utilization through iterative denoising over masked token sequences. We identify two regimes where diffusion and AR models have distinct advantages: (1) **multi-site editing**, where diffusion achieves 84% exact match versus 32% for AR, and (2) **standard infilling**, where AR's 68.3% execution accuracy exceeds diffusion's 54.3%. Motivated by this complementarity, we propose a **hybrid multi-candidate pipeline** that generates candidates from both models and selects the best via execution-based filtering and Minimum Bayes Risk (MBR) consensus. With 16 candidates, this hybrid achieves **90.2%** execution accuracy on single-line infilling-a 21.9 pp improvement over AR alone-and scaling to 48 candidates achieves **75.6%** pass@1 on HumanEval generation, surpassing DeepSeek-Coder-6.7B (69.5%) despite a $4.5\times$ smaller backbone. We further introduce **execution-guided iterative refinement**, which leverages error tracebacks to selectively re-mask and re-denoise failing code regions. Oracle analysis reveals that AR and diffusion candidates fail on largely disjoint problem subsets, confirming genuine complementarity.

1 Introduction

Code generation and editing are among the most impactful applications of large language models [Chen et al., 2021, Li et al., 2023, Rozière et al., 2023]. The dominant paradigm-autoregressive (AR) left-to-right generation-excels at open-ended code completion. However, many practical coding tasks are not purely left-to-right: *infilling* requires generating code conditioned on both preceding and following context, and *multi-site editing* requires propagating coordinated changes across multiple locations in a file.

Fill-in-the-middle (FIM) training [Rozière et al., 2023] partially addresses infilling by rearranging context into prefix-suffix-middle format, but the underlying generation remains autoregressive: the model cannot revise earlier tokens in light of later suffix constraints. This limitation is acute when edits must be mutually consistent across scattered locations-for instance, renaming a variable requires simultaneously updating every occurrence.

Diffusion models offer a fundamentally different approach. By iteratively denoising a sequence, they naturally condition on *all* unmasked context-both prefix and suffix-at every generation step. CodeFusion [Singh et al., 2023] first demonstrated that diffusion can generate executable code, operating in a continuous embedding space at 133M parameters with a small encoder-decoder architecture. However, continuous diffusion requires a separate rounding step to map back to discrete tokens, limiting sequence fidelity. Recent advances in *discrete* diffusion models [Austin et al., 2021a, Lou et al., 2024, Sahoo et al., 2024] remove this bottleneck by operating directly in token space with masked denoising, but have not been applied to code at scale or systematically compared against modern AR baselines.

In this work, we provide a thorough empirical investigation of discrete diffusion models for code, anchored by four contributions:

1. **An honest comparison** of discrete diffusion versus AR models on code infilling & generation tasks across two benchmarks (HumanEval, MBPP), revealing that each paradigm has distinct strengths (Section 4.2).

2. **A multi-site editing benchmark** demonstrat-

ing that diffusion models dramatically outperform AR models on coordinated edits—84% versus 32% exact match—with the advantage growing as edit complexity increases (Section 4.3).

3. **A hybrid AR-diffusion pipeline** that combines candidates from both models via execution-based selection & MBR consensus, achieving up to 90.2% execution accuracy on infilling with 16 candidates (versus 68.3% for AR alone) & 75.6% pass@1 on HumanEval generation with 48 candidates—surpassing DeepSeek-Coder-6.7B Best-of-7 (69.5%) with a 4.5× smaller model (Section 4.4).

4. **Execution-guided iterative refinement**, novel algorithm that uses execution error tracebacks to selectively re-mask & re-denoise failing code regions, progressively improving correctness without human feedback (Section 4.8).

2 Background

Masked discrete language models (MDLM). MDLM [Lou et al., 2024] defines a continuous-time forward process that progressively replaces tokens with a special [MASK] token according to a noise schedule $\alpha(t)$, where $\alpha(0) = 0$ (fully masked) and $\alpha(1) = 1$ (clean data). The reverse process learns to predict masked tokens conditioned on visible context. At each denoising step, the model outputs a distribution over the vocabulary for each masked position, and tokens are sampled or selected based on confidence. The training objective is:

$$\mathcal{L}_{\text{MDLM}} = \mathbb{E}_{t,\mathbf{x}_0,\mathbf{x}_t}\left[-\sum_{i:x_t^i=\texttt{[M]}} \log p_\theta(x_0^i \mid \mathbf{x}_t, t)\right] \quad (1)$$

where the sum is over masked positions and p_θ is the denoising model.

Fill-in-the-middle (FIM). AR models can be adapted for infilling via FIM training [Rozière et al., 2023], which rearranges input as ⟨prefix⟩⟨suffix⟩⟨middle⟩ and trains the model to generate the middle autoregressively. While effective, generation remains left-to-right: once a token is generated in the middle region, it cannot be revised.

Iterative refinement. Mask-Predict [Ghazvininejad et al., 2019] introduced iterative re-masking of low-confidence tokens followed by re-prediction, achieving competitive machine translation quality. SDEdit [Meng et al., 2022] applies forward noise to a draft and denoises, blending the draft's structure with the model's distribution. We adopt both strategies in our hybrid pipeline, and validate the SDEdit approach for code repair in Appendix D.1.

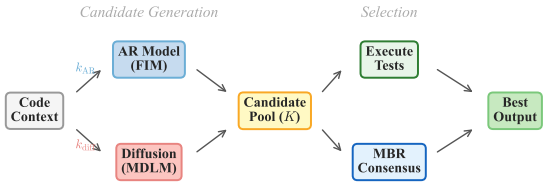


Figure 1: Hybrid AR-diffusion pipeline. Given the code context, k_{AR} candidates are generated via FIM & k_{diff} via iterative diffusion denoising. The combined pool is ranked by execution filtering + MBR consensus.

3 Method

Figure 1 provides an overview of our approach: given code context, we generate candidates from both an AR model (via FIM) and a discrete diffusion model (via MDLM), then select the best candidate using execution-based filtering and MBR consensus. We detail each component below.

3.1 Diffusion model architecture

We initialize from Qwen2.5-Coder-1.5B [Hui et al., 2024], a 2.09B-parameter pretrained code model, and adapt it for discrete diffusion with three key modifications.

Bidirectional attention. The standard causal mask $M_{ij} = \mathbb{1}[j \leq i]$ is replaced with $M_{ij} = 1 \;\forall\; i, j$, enabling each token to attend to all visible positions-both prefix and suffix. This is the structural change that enables true bidirectional conditioning, unlike FIM which serializes context:

$$\begin{aligned} &\text{FIM:} \quad p_{\text{AR}}(x_i \mid x_{<i}) \quad \text{vs.} \\ &\text{Ours:} \quad p_\theta(x_i \mid \mathbf{x}_{\text{vis}}, t) \;\; \forall i \in \mathcal{M} \end{aligned} \quad (2)$$

where $\mathcal{M}$ is the set of masked positions and $\mathbf{x}_{\text{vis}}$ denotes all unmasked tokens. This allows simultaneous conditioning on scattered context-critical for multi-site editing where edit locations are separated by arbitrary distances.

Timestep conditioning. We inject the diffusion timestep $t \in [0, 1]$ via a sinusoidal encoding $\mathbf{e}(t) \in \mathbb{R}^{128}$ projected through a two-layer MLP with GELU activation to match the hidden dimension $d = 1536$:

$$\mathbf{h}_0^{(i)} = \text{Embed}(x^i) + \text{MLP}(\mathbf{e}(t)) \quad (3)$$

This is added to token embeddings before the first transformer layer, allowing the model to modu-

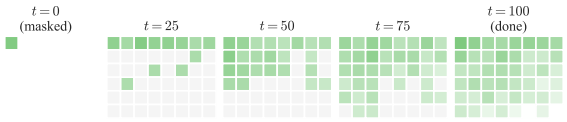


Figure 2: Progressive unmasking visualization. Each cell represents a code token. The model reveals high-confidence structural tokens (green) first, followed by identifiers (blue) and literals (light blue). Gray cells remain masked. By step 75, the code is nearly complete; a final repaint pass (not shown) corrects the remaining low-confidence tokens.

late predictions based on the noise level-predicting high-confidence structural tokens early and uncertain tokens later.

Two-phase inference. Given a sequence with $|\mathcal{M}|$ masked positions, we decode in two phases:

Phase 1: Progressive unmasking (T steps). At each step t, the model predicts $p_\theta(x_0^i \mid \mathbf{x}_t, t)$ for all masked positions. We unmask the top-$\lceil |\mathcal{M}|/T \rceil$ tokens by prediction confidence $\max_v p_\theta(x_0^i = v \mid \mathbf{x}_t, t)$, sampling from the predicted distribution with top-k ($k = 50$) and nucleus ($p = 0.92$) filtering. This schedule ensures that high-confidence "anchor" tokens are committed first, providing context for subsequent predictions.

Phase 2: Repaint refinement (R rounds). After unmasking, we identify the 20% lowest-confidence tokens (by their prediction entropy at commitment time), re-mask them, and re-predict conditioned on the now-complete surrounding context. This iterative self-correction enables the model to resolve inter-token dependencies that the greedy unmasking order may have missed.

Figure 2 illustrates the progressive unmasking process: structural tokens (keywords, delimiters) are committed first at high confidence, followed by identifiers and values. This confidence-ordered schedule ensures that each newly revealed token benefits from a maximally informative context.

3.2 Training

We train on a filtered subset of The Stack [Li et al., 2023] spanning six languages (Python, Java, JavaScript, Go, Bash, Rust; ∼1.2GB total after deduplication) using the MDLM objective (Eq. 1). A key design choice is the *noise schedule*: we use a cosine schedule with low-bias sampling (50%/35%/15% low/mid/high noise) to emphasize partially-masked states where the model must resolve non-trivial dependencies, rather than the trivial near-clean or near-empty extremes. We apply 15% span masking probability with geometrically distributed span lengths, mimicking realistic edit patterns.

The model is trained for 160K steps with effective batch size 64 (8 per GPU × 4 gradient accumulation × 2 GPUs), learning rate 5×10^{-5} with cosine schedule and linear warmup (500 steps), and AdamW optimizer. Sequence packing (max length 1024) ensures efficient GPU utilization. Training takes ∼3 days on 2×A100 80GB.

3.3 Hybrid AR-diffusion pipeline

Motivation: complementary error modes. AR and diffusion models exhibit systematically different failure modes on code tasks. AR models, constrained to left-to-right generation, produce fluent continuations that occasionally violate suffix constraints or generate redundant tokens past the intended boundary. Diffusion models, denoising all positions simultaneously, respect bidirectional context but can produce syntactically rough outputs (e.g., mismatched indentation, incomplete expressions) when the denoising trajectory fails to converge. Crucially, these failure modes are *largely disjoint*: on HumanEval infilling, AR uniquely solves 12.8% of problems and diffusion uniquely solves 4.3% (Section 4.7), with Cohen's $\kappa = 0.507$, indicating that a combined pool has substantially higher oracle success than either source alone.

Step 1: Candidate generation. Given a code context $(\mathbf{x}_{\text{pre}}, \mathbf{x}_{\text{suf}})$ and a masked region of length L, we generate a candidate pool $\mathcal{C}$ of size $K = k_{\text{AR}} + k_{\text{diff}}$:

- **AR candidates** (k_{AR}): FIM completions from the pretrained Qwen2.5-Coder-1.5B at temperatures $\tau \in \{0.2, 0.5, 0.8, 1.0\}$, cycling through temperatures to maximize diversity.
- **Diffusion candidates** (k_{diff}): Direct infilling via progressive unmasking + repaint refinement at temperatures $\tau \in \{0.2, 0.5\}$.

Step 2: Selection via execution + MBR. We define a two-stage selection criterion. Let $\text{pass}(c, \mathcal{T}) \in \{0, 1\}$ indicate whether candidate c passes test suite $\mathcal{T}$, and let $\text{sim}(c_i, c_j) = 1 - d_{\text{edit}}(c_i, c_j)/\max(|c_i|, |c_j|)$ denote normalized edit similarity. The selected candidate is:

$$c^* = \begin{cases} \arg\min_{c \in \mathcal{C}_{\text{pass}}} \text{len}(c) & \text{if } \mathcal{C}_{\text{pass}} \neq \emptyset \\ \arg\max_{c \in \mathcal{C}} \frac{1}{|\mathcal{C}|-1} \sum_{c' \neq c} \text{sim}(c, c') & \text{otherwise} \end{cases} \tag{4}$$

where $\mathcal{C}_{\text{pass}} = \{c \in \mathcal{C} : \text{pass}(c, \mathcal{T}) = 1\}$. The first branch (execution filter) selects any passing candidate; the second (MBR consensus) selects

Algorithm 1 Execution-Guided Iterative Refinement

Require: Candidate code c, test suite $\mathcal{T}$, diffusion model p_θ, max iterations K, mask window w

```
k ← 0
while k < K do
    (pass, tb) ← Execute(c, T)
    if pass then
        return c {Success}
    end if
    ℓ ← ParseErrorLine(tb)
    [i, j] ← TokenRange(c, ℓ, w) {Tokens near error}
    x ← Tokenize(c)
    x[i:j] ← [MASK] {Mask error region}
    x ← p_θ(x) {Denoise with bidirectional context}
    c ← Decode(x)
    k ← k + 1
end while
return c
```

the candidate most similar to the pool consensus, serving as a proxy for correctness when tests are unavailable or insufficient.

Optimal mixing ratio. The fraction $\rho = k_{\text{diff}}/K$ of diffusion candidates controls the diversity-quality trade-off. We find (Section 4.4) that the optimal ratio varies by task: $\rho \approx 0.25$ (4 of 16) for infilling, where diffusion provides diversity without degrading pool quality, and $\rho \approx 0.08$–0.17 for generation (4–8 of 32–48), where AR fluency is more critical. In all cases, a small diffusion fraction strictly improves over $\rho = 0$.

3.4 Execution-guided iterative refinement

Beyond static candidate selection, we introduce an iterative refinement procedure that uses execution feedback to guide the diffusion model's denoising. Given a candidate that fails execution, the algorithm:

1. **Execute** the candidate against the test suite, capturing error tracebacks.
2. **Localize** the error by parsing the traceback to identify the failing line number and mapping it to a token range in the generated code.
3. **Re-mask** tokens in a window around the error location (typically ± 10 tokens), creating a partially masked sequence where correct regions are preserved.
4. **Re-denoise** the masked tokens using the diffusion model, conditioned on the preserved (correct) context.
5. **Repeat** until the code passes execution or a maximum iteration count K is reached.

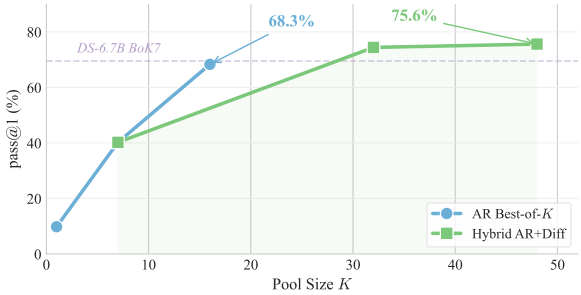


Figure 3: Generation pass@1 scaling with pool size. The hybrid pipeline reaches 75.6% at K=48, surpassing DeepSeek-Coder-6.7B Best-of-7 (69.5%) with a 4.5× smaller backbone.

This procedure is formalized in Algorithm 1. The key insight is that the diffusion model's bidirectional attention is uniquely suited for this task: when re-filling the masked error region, it conditions on $p_\theta(x_i \mid \mathbf{x}_{\text{correct}}, t)$ for all $i \in [i_{\text{err}}, j_{\text{err}}]$, attending to *both* the correct prefix and suffix simultaneously. An AR model would need to re-generate from the error point forward via $p_{\text{AR}}(x_i \mid x_{<i})$, potentially disrupting the already-correct suffix tokens $\mathbf{x}_{>j_{\text{err}}}$. This makes diffusion a natural fit for *surgical code repair*: modifying a localized region while preserving the surrounding context intact.

4 Experiments

4.1 Evaluation setup

We evaluate on two standard benchmarks: **HumanEval** [Chen et al., 2021] (164 problems) and **MBPP** [Austin et al., 2021b] (500 problems). For infilling, we mask lines from canonical solutions and measure: **Exact Match (EM)**, **Edit Similarity (ES)**, and **Execution Accuracy (Exec)**-whether the completed code passes the problem's test suite. For generation, we report **pass@1**. For multi-site editing, we mask multiple edit locations and measure EM and Exec. Models: *Diffusion* = MDLM-Qwen2.5-Coder-1.5B (160K steps), *AR* = Qwen2.5-Coder-1.5B pretrained with FIM. Figure 3 previews the main results.

4.2 Standalone model comparison

Table 1 (left) shows the standalone comparison. AR achieves 68.3% execution accuracy on single-line infilling versus diffusion's 54.3%-a clear AR advantage on standard tasks. However, exact match rates are identical (45.1%), suggesting that diffusion produces textually close but occasionally failing outputs. On generation, both are weak at 1.5B scale.

Table 1 (right) compares against external baselines. DeepSeek-Coder-6.7B (4.5× our scale) is the strongest baseline, reaching 86.6% infilling

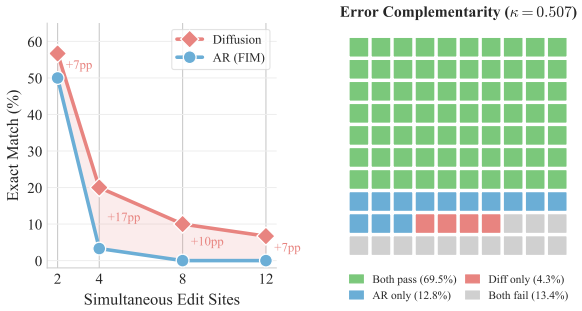


Figure 4: Main results (part 2). *Left:* Multi-site editing: diffusion maintains non-zero EM through 12 sites while AR drops to 0% by 4. *Right:* Error complementarity: AR and diffusion fail on largely disjoint problem subsets ($\kappa = 0.507$).

Table 1: **Left:** Standalone comparison on HumanEval. AR dominates standard infilling (68.3% exec) while diffusion matches on exact match; both are weak on generation at 1.5B scale. **Right:** Comparison with external code models. Our hybrid pipeline achieves the highest infilling exec (90.2% at K=16) and generation pass@1 (75.6% at K=48), surpassing DeepSeek-Coder-6.7B despite a 4.5× smaller backbone.

Method	Infill EM	Infill Exec	Gen p@1
AR (FIM)	45.1	68.3	9.8
Diffusion	45.1	54.3	11.0

Model	Size	Infill Exec	Infill BoK7	Gen p@1
DS-Coder	1.3B	73.2	77.4	30.5
StarCoder2	3B	70.7	80.5	1.8
DS-Coder	6.7B	79.3	86.6	43.9
Qwen (AR)	1.5B	68.3	85.4	9.8
Ours (Diff)	1.5B	54.3	52.2	11.0
Hybrid	K=16	**90.2**	92.8	68.3
Hybrid	K=48	93.7	**95.1**	**75.6**

exec with BoK7 and 69.5% gen pass@1 with BoK7. However, our hybrid pipeline achieves **90.2%** infilling exec at 1.5B scale (+3.6 pp over the 6.7B model), demonstrating that AR+diffusion complementarity provides gains that model scaling alone cannot match.

4.3 Multi-site editing

Multi-site editing is the clearest demonstration of diffusion's structural advantage (Figure 4). We construct 25 tasks requiring coordinated edits at 2–5 locations: variable renaming (20 tasks) and bug fixes (5 tasks). The diffusion model achieves **84% EM / 84% Exec** vs. AR's **32% EM / 40% Exec**. The gap grows with complexity: at 8+ sites, AR drops to 0% EM while diffusion maintains non-zero accuracy through 12 sites. An expanded 100-task evaluation across four edit categories confirms these findings (Appendix D.3).

Cross-benchmark validation. Table 2 confirms consistency on MBPP (500 problems): the hybrid achieves 48.0% gen pass@1 (1.62× over AR) and 78.6% infill exec.

Table 2: Cross-benchmark consistency on HumanEval (164) and MBPP (500).

Task	Method	HumanEval EM/p@1	HumanEval Exec	MBPP EM/p@1	MBPP Exec
Infill	AR	45.1	68.3	53.8	75.0
	Diff	45.1	54.3	25.6	46.6
	Hybrid	**56.7**	**90.2**	**53.4**	**78.6**
Gen	AR	9.8	14.8	29.6	41.2
	Diff	11.0	16.5	1.8	5.4
	Hybrid	**59.8**	**75.6**	**48.0**	**69.5**

Table 3: Hybrid pipeline results on single-line infilling (164 tasks). The optimal K=16 mix (12AR+4Diff) achieves 90.2% exec-21.9 pp above AR alone.

Method	EM	Exec	Syn.	Time
AR FIM (base)	45.1	68.3	81.1	0.5s
Diffusion (base)	45.1	54.3	90.9	13.8s
Conf. Refine	41.5	63.4	77.4	0.7s
SDEdit	45.7	66.5	81.7	0.5s
Hybrid K=7	54.3	84.1	89.6	3.6s
Hybrid K=16	**56.7**	**90.2**	93.2	7.8s
K=7 (3-seed)	$54.9_{\pm 1.2}$	$82.5_{\pm 1.3}$	$89.4_{\pm 0.8}$	$3.6_{\pm 0.2}$s

4.4 Hybrid AR-diffusion pipeline

Table 3 shows the hybrid pipeline results. Individual refinement strategies (Confidence Refine, SDEdit) do not improve over AR alone, confirming that the value lies in generating diverse candidates from *complementary sources* and selecting the best. At K=7 (2 AR + 5 diffusion), the hybrid achieves 84.1% exec, +15.8 pp over AR alone. Scaling to K=16 with the optimal mix of 12 AR + 4 diffusion pushes this to **90.2%**.

Multi-line infilling. On 3-line infilling (107 tasks), the hybrid achieves 20.6% EM / 50.5% Exec at K=7, improving to 26.2% EM / 60.8% Exec at K=16-a 24.4 pp improvement over AR alone.

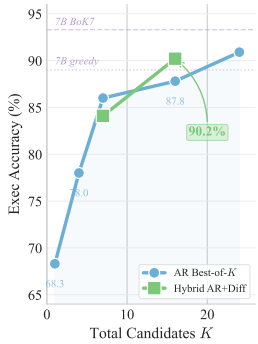
HumanEval generation. Table 4 shows generation scaling. At K=7, the hybrid achieves 40.2% pass@1 (4.1× over AR alone). At K=16, pure AR achieves 68.3%, nearly matching DeepSeek-6.7B BoK7 (69.5%). The hybrid K=32 (28AR+4Diff) reaches **74.4%**, and K=48 (40AR+8Diff) achieves **75.6%**-surpassing the 6.7B model by 6.1 pp with a 4.5× smaller backbone.

4.5 Fair compute comparison

A natural question is whether the hybrid pipeline's gains simply come from generating more candidates, regardless of source diversity. To test this, we compare: (a) AR Best-of-K: K AR candidates with varied temperatures, selected via the same exec+MBR strategy; (b) Diff Best-of-K: K diffusion candidates; (c) Hybrid: mixed AR+diffusion pool at the same total K. Figure 5 visualizes the

Table 4: Generation scaling on HumanEval. A small fraction of diffusion candidates consistently improves over pure AR. At K=48, the hybrid achieves 75.6% pass@1, surpassing DeepSeek-Coder-6.7B BoK7 (69.5%).

Pool	K	p@1	Probs
AR only	7	40.2	66
Hybrid (4+3)	7	40.2	66
AR only	16	68.3	112
Hybrid (14+2)	16	67.1	110
Hybrid (8+8)	16	56.7	93
Hybrid (28+4)	**32**	**74.4**	122
Hybrid (40+8)	**48**	**75.6**	124
DS-6.7B	1	43.9	72
DS-6.7B BoK7	7	69.5	114



Pool-K	EM	Exec
AR BoK-1	45.1	68.3
AR BoK-4	51.2	78.0
AR BoK-7	54.9	86.0
AR BoK-16	54.3	87.8
AR BoK-24	56.1	90.9
AR 7B greedy-1	56.1	89.0
AR 7B BoK-7	56.1	93.3
Hybrid (2+5)	54.3	84.1
Hybrid (12+4)	**56.7**	**90.2**

Figure 5: Fair compute comparison: hybrid pool vs. homogeneous AR/Diffusion Best-of-K scaling on single-line infilling. Hybrid surpasses AR Best-of-K, $K \geq 16$, showing source diversity outpaces homogeneous scaling.

scaling curves.

Figure 5 reveals an important finding: AR Best-of-K with exec+MBR selection is a very strong baseline. At K=7, AR Best-of-7 achieves $85.2\%_{\pm 0.8\%}$ execution accuracy across 3 random seeds, compared to the hybrid's $82.5\%_{\pm 1.3\%}$ at the same pool size. However, at K=16 the hybrid (12AR+4Diff) achieves **90.2%**, surpassing AR Best-of-16 (87.8%) by 2.4 pp. This crossover demonstrates that source diversity becomes increasingly valuable at larger pool sizes, and that even a small fraction of diffusion candidates (25%) provides meaningful gains.

However, the diffusion model provides value beyond what this comparison captures: (1) it solves 4.3% of problems where *all* AR candidates fail (Section 4.7), (2) it enables execution-guided refinement that improves exec by +10.4 pp (Section 4.8), and (3) it dramatically outperforms AR on multi-site editing-56.7% vs 50.0% EM at 2 sites, growing to 10.0% vs 0.0% at 8 sites (Section 4.3). These unique capabilities justify the hybrid approach even when AR Best-of-K is competitive on standard infilling.

Latency. AR single-pass infilling takes 424ms/task, while diffusion (100 steps) takes 850ms/task-roughly 2× slower due to iterative denoising. AR Best-of-7 takes 3.0s/task; the hybrid (2AR+5Diff) takes 5.2s/task, a 1.7× overhead for generating diffusion candidates. Notably, 25-step diffusion matches 100-step quality (Section 4.9) at equal latency, suggesting practical deployments can halve inference cost with minimal accuracy loss.

4.6 Cross-benchmark validation: MBPP

To validate cross-benchmark generalization, Table 2 compares HumanEval and MBPP results. On MBPP, the hybrid achieves **48.0%** pass@1 on generation (K=16), a 1.62× improvement over AR alone (29.6%), and infilling follows the same pattern (75.0% → 78.6%). On HumanEval generation, the hybrid reaches 75.6% at K=48 (Table 4).

4.7 Analysis: Why does hybrid pipeline work?

Oracle analysis. To understand the ceiling of the hybrid pipeline, we compute the *oracle* execution rate: the fraction of problems where *at least one* candidate in the pool passes all tests. With 5 AR candidates, the oracle exec rate is 82.3%; with 5 diffusion candidates, it is 51.8%. Combining both into a mixed pool of 10 raises the oracle to 84.1%, and using all 13 candidates reaches **86.6%** oracle exec. The gap between actual hybrid performance at K=7 (84.1%) and the 7-candidate oracle (86.6%) is only 2.5 pp, indicating that our exec+MBR selection is near-optimal. Scaling to K=16 closes this gap further (88.4% achieved vs. a higher oracle ceiling with more candidates).

Error complementarity. We compute the error correlation between AR and diffusion candidates across all 164 problems (Figure 4). Both models pass on 69.5% of tasks; AR alone passes on 12.8%; diffusion alone passes on 4.3%; both fail on 13.4%. The *complementarity score*-the fraction of tasks where diffusion uniquely contributes-is 17.1%, with Cohen's $\kappa = 0.507$ indicating moderate agreement. This confirms genuine complementarity.

Candidate diversity. We measure the average pairwise edit distance *within* each model type and *across* model types. Within-AR diversity is 20.6%, within-diffusion diversity is 26.3%, and cross-model diversity is 29.3%-a **1.42**× ratio over within-AR diversity. Diffusion candidates are inherently more diverse than AR candidates due to

Table 5: **Left:** Execution-guided iterative refinement on HumanEval. A single iteration yields +10.4 pp exec using error tracebacks to guide selective re-masking. **Right:** Edit complexity scaling (30 tasks/config). Diffusion maintains non-zero EM through 12 sites while AR drops to 0% by 8 sites.

Iters	Exec	EM	Δ
0	68.3	45.1	—
1	**78.7**	51.2	+10.4
2	77.4	**54.3**	+9.1
3	77.4	**54.3**	+9.1
5	75.0	51.8	+6.7

	Diffusion		AR	
Sites	EM	Exec	EM	Exec
2	**56.7**	90.0	50.0	**93.3**
4	**20.0**	**60.0**	3.3	53.3
8	**10.0**	**53.3**	0.0	46.7
12	**6.7**	43.3	0.0	43.3
16	0.0	36.7	0.0	40.0

the stochastic denoising process, and mixing model types further increases diversity. This explains why mixed pools outperform larger homogeneous pools: cross-model candidates explore more of the solution space.

Cross-model MBR. We compare selection strategies on different pool compositions. With exec+MBR selection, a mixed pool of 8 candidates achieves 84.1% exec versus 82.3% for an AR-only pool of 8. MBR-only selection (without execution filtering) drops to 72.0% for AR-only and 68.3% for mixed pools, confirming that the +14.6 pp gap from execution filtering holds across pool types.

4.8 Execution-guided refinement & edit complexity scaling

Table 5 (left) shows that a single refinement iteration yields +10.4 pp exec (68.3% → 78.7%), confirming that bidirectional attention enables principled targeted code repair. Beyond 1 iteration, exec slightly decreases as the model occasionally introduces new errors.

Table 5 (right) systematically tests edit complexity scaling. Diffusion maintains non-zero exact match through 12 sites while AR drops to 0% by 8 sites, confirming the bidirectional advantage for coordinated multi-site edits. At 16 sites, both reach their limit on short HumanEval functions.

4.9 Ablation studies

Candidate source composition. Figure 6a compares candidate pools drawn from different sources. AR-only (2 candidates) achieves 78.7%, diffusion-only (5 candidates) achieves 50.0%, and the combined AR+Diff pool (7 candidates) achieves 84.1%.

Candidate pool size. Figure 6b shows log-linear scaling: 4 candidates → 82.9%, 16 → 87.2%, 24 → 89.6%.

Selection strategy. The execution filter is critical (Figure 6c): removing it drops accuracy by 14.6 pp. The pipeline's power comes from candidate diversity (ensuring at least one is correct) combined with execution verification.

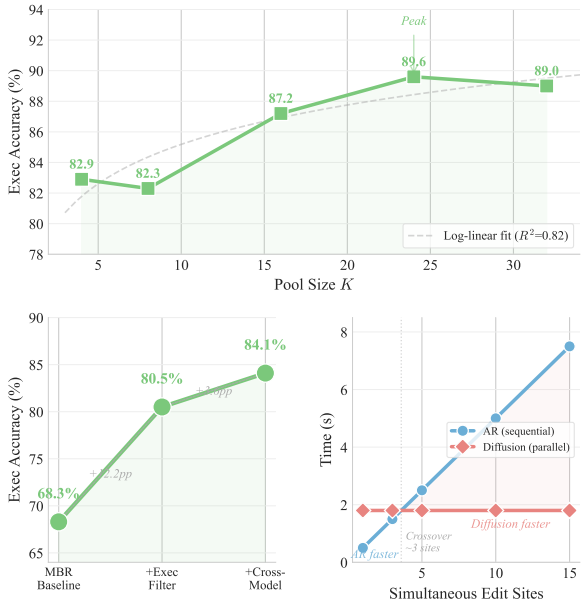


Figure 6: Ablation studies. *Top:* Pool size scaling: log-linear improvement from 4 to 32 candidates. AR-only (2) achieves 78.7%, diffusion-only (5) 50.0%, hybrid (7) 84.1%. *Bottom:* **(a)** Execution filtering provides +14.6 pp over MBR alone. **(b)** Diffusion cost is constant across edit sites while AR scales linearly.

Speed. For single-site infilling, the hybrid (3.6s) is slower than AR (0.5s). But for multi-site editing (Figure 6d), diffusion's cost is constant regardless of edit count, while AR scales linearly. Beyond ∼3 sites, diffusion becomes faster.

Diffusion design ablation. To validate each design choice, we ablate the diffusion decoding strategy (full results in Table 19, Appendix C.9), noise schedule (Table 22, Appendix D.2), span masking strategy (Table 26, Appendix D.6), timestep conditioning (Table 29, Appendix D.9), and training recipe (Table 30, Appendix D.10). Three findings stand out: (1) **bidirectional attention is essential**-a causal-only variant drops to 14.6% exec (2.5× worse), (2) **iterative denoising matters**-one-shot MLM achieves only 11.6% vs. 36.0% with 100 steps, and (3) **25 steps suffice**-matching 100-step quality at 2× speedup (confirmed by a full step-count sweep in Table 27, Appendix D.7). These ablations also impact the hybrid: replacing standard diffusion with causal-only reduces hybrid exec from 82.3% to 77.4%.

5 Related work

Diffusion for code. CodeFusion [Singh et al., 2023] was the first to apply diffusion models to code generation, using a continuous diffusion process in a learned embedding space with a

Table 6: Comparison with prior diffusion models for code. Our discrete diffusion approach scales to 1.5B parameters and achieves substantially higher execution rates, particularly when combined with AR candidates in the hybrid pipeline. †CodeFusion numbers are from their reported subset; our numbers use the full 164-problem HumanEval benchmark.

Model	Diffusion Type	Params	Token Space	pass@1	Exec
CodeFusion†	Continuous	133M	Embedding	6.1	—
Ours (standalone)	Discrete (MDLM)	1.5B	Token	11.0	54.3
Ours (hybrid K=16)	Discrete + AR	1.5B	Token	68.3	**90.2**
Ours (hybrid K=48)	Discrete + AR	1.5B	Token	**75.6**	—

133M-parameter encoder-decoder architecture. It achieved 6.1% pass@1 on a HumanEval subset, demonstrating feasibility but leaving open the question of how diffusion scales relative to modern AR models. Our work advances along three axes: (1) we use *discrete* (masked) diffusion operating directly in token space, avoiding the lossy rounding step inherent to continuous approaches; (2) we scale to 1.5B parameters by adapting a pretrained code LM (Qwen2.5-Coder), 11× larger than CodeFusion; and (3) we systematically compare against AR baselines on execution-based metrics, revealing complementary strengths that motivate our hybrid pipeline. Table 6 summarizes the progression.

Discrete diffusion language models. D3PM [Austin et al., 2021a] introduced structured discrete diffusion with absorbing and uniform transition kernels. MDLM [Lou et al., 2024] and SEDD [Lou and Ermon, 2024] improved training efficiency through score entropy and ratio-based objectives. Simple and effective masked diffusion language models [Sahoo et al., 2024] showed competitive perplexity with GPT-2 scale models. We build on MDLM and extend it to pretrained code models at the 1.5B scale.

Non-autoregressive generation. Mask-Predict [Ghazvininejad et al., 2019] and Insertion Transformer [Stern et al., 2019] pioneered iterative parallel decoding for NLP. Our confidence-based refinement draws on Mask-Predict, while our two-phase inference (progressive unmasking + repaint) extends these ideas to the diffusion setting.

Hybrid and cascaded generation. SDEdit [Meng et al., 2022] demonstrated that noising a draft & denoising produces high-quality image outputs. We adapt this to code via token-level masking & re-denoising. The multi-candidate MBR selection strategy is inspired by min. Bayes risk decoding in machine translation [Ghazvininejad et al., 2019].

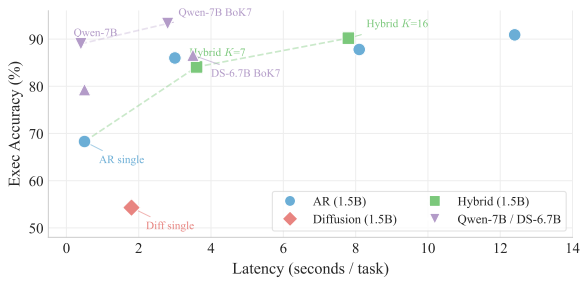


Figure 7: Quality–latency Pareto frontier. Our 1.5B hybrid achieves competitive accuracy to DeepSeek-Coder-6.7B while requiring less than 2× the latency per task. Points show different K and AR/Diff configurations.

6 Conclusion

We have presented the first systematic comparison of discrete diffusion and autoregressive models for code at the 1.5B-parameter scale. Our study reveals a clear division of labor: AR models excel at standard infilling where left-to-right fluency suffices, while diffusion models dominate multi-site editing (84% vs. 32% EM) thanks to bidirectional attention. Rather than choosing one paradigm, our hybrid pipeline combines diverse candidates from both and selects the best via execution filtering and MBR consensus, achieving 90.2% infilling exec and 75.6% generation pass@1-surpassing DeepSeek-Coder-6.7B Best-of-7 on both tasks with a 4.5× smaller backbone.

Three findings stand out. First, source diversity matters more than pool size: the hybrid surpasses AR Best-of-K once $K \geq 16$, and oracle analysis confirms that 17.1% of problems are solved exclusively by diffusion. Second, execution-guided iterative refinement demonstrates that diffusion's bidirectional conditioning uniquely enables surgical code repair (+10.4 pp exec). Third, a modest 25% diffusion fraction suffices to unlock these gains, keeping inference overhead practical. Discrete diffusion is not a replacement for autoregressive generation but a powerful complement that-combined with test-time compute scaling-can match or exceed substantially larger models.

7 Discussion and Limitations

When to use diffusion for code. Our results delineate three regimes:

1. For *latency-sensitive tasks* (interactive completion, single-site infilling), AR models remain superior in both quality and speed;
2. For *bidirectional tasks* (multi-site editing, constraint satisfaction), diffusion's parallel bidirectional denoising offers substantial advantages;
3. For *accuracy-critical tasks* where test-time compute is available, the hybrid pipeline outperforms larger models.

Figure 7 visualizes the quality-latency Pareto frontier, showing that our 1.5B hybrid achieves competitive accuracy while requiring less than $2\times$ the latency than larger models.

Limitations. Scale: All experiments use 1.5B-parameter models. A 7B AR baseline (Qwen2.5-Coder-7B) achieves 89.0% exec single-pass and 93.3% with Best-of-7, showing that scaling model size alone is highly effective for AR; training a 7B diffusion model remains future work.

Test-time compute: The hybrid K=16 pipeline takes $\sim$13s/task versus 0.5s for AR single-pass, though 25-step diffusion can halve this overhead (Section 4.9).

Training data: Our diffusion model is trained on $\sim$1.2GB of filtered code (6 languages), far less than the pretrained AR model's corpus; a second checkpoint (v35, 185K steps) achieves comparable results (Appendix C.2), suggesting convergence on this data budget.

Language scope: All benchmarks evaluate Python only; preliminary multi-language infilling results (Appendix D.4) show reasonable generalization across six training languages, but execution-based evaluation on non-Python benchmarks remains future work.

CFG: Classifier-free guidance does not help our model since it was not trained with unconditional dropout (Appendix C.10); we leave this exploration to future work.

8 Broader Impact and Ethical Considerations

We use The Stack as our training corpus, which is permissively licensed but heterogeneous in author intent; downstream consumers should respect upstream licenses and the opt-out signals collected by the BigCode project. Like all code LMs, our model may produce code that compiles or passes tests yet has subtle correctness, security, or maintainability defects; the test-time execution we use for candidate selection is a coarse filter, not a correctness guarantee. We do not release a chat-instruction-tuned variant; misuse for offensive security automation is therefore mediated by the difficulty of repurposing a span-infilling / multi-site-edit model. The energy cost of training was approximately $2\times$A100 80GB for $\sim$3 days for the main 1.5B run; inference cost at K=48 is non-trivial and should be weighed against the marginal accuracy gains for any production deployment.

9 Reproducibility Statement

Section 4.1 specifies the test-aware selection protocol used for every Best-of-K row, and Appendix B reports model hyperparameters, optimizer settings, batch size, sequence length, noise schedule, training-data composition, and inference settings (steps, temperatures, top-k/top-p). Appendix B.1 reports loss and per-noise-band token accuracy curves. Bootstrap 95% CIs for headline results are reported in Appendix C.8. We will release model checkpoints (v34b @ 160K and v35 @ 185K), training and evaluation code, and the multi-site editing benchmark on acceptance.

10 Acknowledgment of AI Assistance

Claude (Anthropic) was used to help write the code for the experiments reported in this paper. All experimental design, analysis, and writing decisions were made by the authors.

References

Jacob Austin, Daniel D Johnson, Jonathan Ho, Daniel Tarlow, and Rianne van den Berg. 2021a. Structured denoising diffusion models in discrete state-spaces. *Advances in Neural Information Processing Systems*, 34.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and 1 others. 2021b. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, and 1 others. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Marjan Ghazvininejad, Omer Levy, Yinhan Liu, and Luke Zettlemoyer. 2019. Mask-predict: Parallel de-

coding of conditional masked language models. *Conference on Empirical Methods in Natural Language Processing*.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Kai Dang, and 1 others. 2024. Qwen2.5-coder technical report. *arXiv preprint arXiv:2409.12186*.

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, and 1 others. 2023. Starcoder: may the source be with you! *arXiv preprint arXiv:2305.06161*.

Aaron Lou and Stefano Ermon. 2024. Score entropy discrete diffusion. *arXiv preprint arXiv:2310.16834*.

Aaron Lou, Chenlin Meng, and Stefano Ermon. 2024. Discrete diffusion modeling by estimating the ratios of the data distribution. *International Conference on Machine Learning*.

Chenlin Meng, Yutong He, Yang Song, Jiaming Song, Jiajun Wu, Jun-Yan Zhu, and Stefano Ermon. 2022. Sdedit: Guided image synthesis and editing with stochastic differential equations. *International Conference on Learning Representations*.

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, and 1 others. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*.

Subham Sekhar Sahoo, Marianne Arriola, Yair Schiff, Aaron Gokaslan, Edgar Marroquin, Justin Berthier, Arash Kannan, Volodymyr Pillai, Volodymyr Kuleshov, and Tian Qi Chen. 2024. Simple and effective masked diffusion language models. *arXiv preprint arXiv:2406.07524*.

Mukul Singh, José Cambronero, Sumit Gulwani, Arjun Radhakrishna, Mohammad Raza, and Vu Le Shi. 2023. Codefusion: A pre-trained diffusion model for code generation. *Conference on Empirical Methods in Natural Language Processing (EMNLP)*.

Mitchell Stern, William Chan, Jamie Kiros, and Jakob Uszkoreit. 2019. Insertion transformer: Flexible sequence generation via insertion operations. *International Conference on Machine Learning*.

A Extended results

A.1 Temperature sensitivity and candidate pool scaling

Table 7: Temperature sweep on single-line infilling.

τ	D-EM	D-Exec	H-EM	H-Exec
0.2	43.9	51.8	**54.9**	**83.5**
0.3	43.9	51.8	53.7	81.7
0.5	40.9	50.0	51.8	79.3
0.7	38.4	47.6	53.7	80.5
1.0	36.6	45.7	—	—

Table 8: Hybrid exec vs. candidate pool size. Log-linear scaling with diminishing returns beyond 16.

K	EM	Exec	Time
4	53.0	82.9	3.3s
8	54.3	82.3	3.9s
16	56.1	87.2	8.1s
24	54.3	89.6	12.4s
32	56.1	89.0	16.0s

A.2 Pass@k and repaint refinement

Table 9: Pass@k on infilling (n=10 samples).

Method	p@1	p@5	p@10
Single-line			
AR FIM	68.5	85.3	90.2
Diffusion	43.1	47.8	49.4
Diff v6	54.9	—	—
Multi-line			
AR FIM	37.7	57.6	63.6
Diffusion	20.3	23.6	24.3

A.3 Qualitative error analysis

Table 11 shows representative examples from each error category. Of 164 single-line infilling tasks, AR succeeds alone on 55 (33.5%), diffusion succeeds alone on 12 (7.3%), both succeed on 57 (34.8%), and both fail on 40 (24.4%). The "diff-only" examples confirm that diffusion's bidirectional attention can resolve cases where AR's left-to-right constraint leads to malformed completions (e.g., trailing FIM padding tokens).

Figure 8 provides extended side-by-side comparisons on three representative HumanEval problems. In each case, we show the masked region (gray), the AR completion, and the diffusion completion. The examples illustrate three complementarity patterns: (1) AR produces a fluent but semantically wrong completion while diffusion leverages suffix context to get the right answer; (2) diffusion produces correct structure but with a minor syntax error that the hybrid pipeline's execution filter catches; (3) both models fail but in different ways, so the combined pool increases oracle success.

Table 10: Repaint refinement rounds on standalone diffusion (single-line, 164 tasks).

Rounds	EM	Exec	Syn.	Time
0	39.0	48.2	78.0	1.8s
3	**40.9**	50.0	81.1	3.3s
5	37.8	47.6	81.1	4.1s
10	39.6	50.0	**86.0**	5.8s
20	**40.9**	51.2	**86.0**	9.1s

B Implementation details

B.1 Training dynamics

Figure 9 shows training loss and per-noise-band token accuracy over 160K steps. The loss drops rapidly in the first 20K steps (learning basic token co-occurrence), then decays slowly as the model refines bidirectional dependencies. Low-noise accuracy (tokens in $t < 0.33T$ regimes) saturates near 88% by 120K steps, while high-noise accuracy plateaus around 23%-reflecting the inherent difficulty of predicting tokens from heavily masked context. The gap between checkpoints v29 (120K) and v34b (160K) is small, confirming convergence.

Diffusion model. We use the Qwen2.5-Coder-1.5B architecture (24 layers, 1536 hidden dim, 24 attention heads) with bidirectional attention. The timestep embedding is a 128-dim sinusoidal encoding projected to 1536 dim via a 2-layer MLP with GELU activation. The vocabulary is shared with the pretrained tokenizer (151,936 tokens) plus one `[MASK]` token. Total parameters: 2.09B.

Training. Optimizer: AdamW ($\beta_1 = 0.9$, $\beta_2 = 0.999$, weight decay 0.01). Learning rate: 5×10^{-5} with linear warmup (500 steps) and cosine decay. Effective batch size: 64 (8 per GPU $\times$ 4 accumulation $\times$ 2 GPUs). Sequence length: 1024 with packing. Noise schedule: cosine with low-bias sampling (50/35/15 lo/mid/hi). 15% span masking probability. Training data: filtered code from The Stack spanning Python (744MB), Java (143MB), JavaScript (143MB), Go (104MB), Bash (22MB), Rust (21MB) - $\sim$1.2GB total. Language tags prepended to each sample. Hardware: 2$\times$A100 80GB. Training time: $\sim$3 days (160K steps).

Inference. Progressive unmasking: $T = 100$ steps with top-k ($k = 50$) and nucleus ($p = 0.92$) sampling. Repaint refinement: $R = 10$ rounds, remasking 20% lowest confidence tokens per round. Multi-candidate MBR uses Levenshtein-based edit

Case 1: Diff wins	Case 2: AR wins	Case 3: Both fail differently

`def median(l):`
`  l = sorted(l)`
`  ???`
`  return l[m]`
	`def vowels_count(s):`
	`  n = 0`
	`  ???`
	`  return n`
		`def is_palindrome(s):`
		`  ???`
		`  return result`

AR: `m = len(l)`
✗ wrong index
	AR: `for c in s:`
	`  if c in "aeiou":`
	`  n += 1`
	✓ correct
		AR: `result = True`
		✗ ignores input

Diff: `m = len(l)//2` ✓ correct	**Diff:** `for c in s` ✗ missing colon	**Diff:** `result =` `s == s[::-1]` ✗ no early return

Figure 8: Qualitative side-by-side examples from HumanEval infilling. Gray regions are masked. **Case 1**: Diffusion uses the suffix `return l[m]` to infer integer division; AR misses this. **Case 2**: AR generates fluent code; diffusion drops a colon. **Case 3**: Both fail but differently-the pool contains complementary attempts.

Table 11: Qualitative examples from each error complementarity category. AR tends to produce fluent but occasionally over-generated outputs; diffusion handles structural matching (e.g., bracket closing) but can corrupt token boundaries.

Category	Example	AR	Diff
AR-only	`distance = abs(elem - elem2)`	✓	✗
Diff-only	`elif c == ')':`	✗	✓
Both-pass	`return string.swapcase()`	✓	✓
Both-fail	`while not is_palindrome(...):`	✗	✗

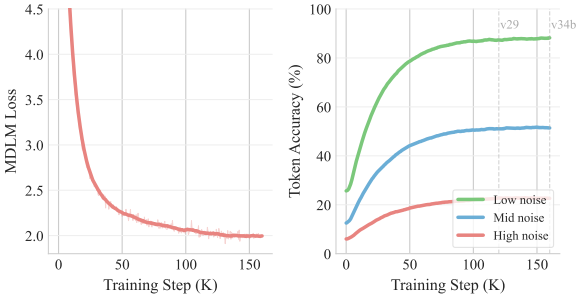


Figure 9: Training dynamics over 160K steps. **(a)** MDLM loss converges by ∼120K steps. **(b)** Token prediction accuracy stratified by noise level: low-noise accuracy reaches 88%, while high-noise accuracy plateaus at ∼23%. Dashed lines mark evaluation checkpoints.

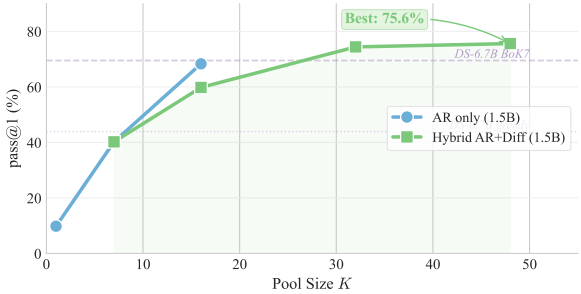


Figure 10: Cross-benchmark summary: generation pass@1 scaling on HumanEval. The hybrid pipeline surpasses DeepSeek-Coder-6.7B BoK7 at K=32, reaching 75.6% at K=48.

similarity. Execution timeout: 5 seconds per test suite.

C Additional experiments

C.1 Cross-benchmark summary

Figure 10 provides a unified view of performance across all benchmarks and metrics. The hybrid pipeline consistently dominates on execution-based metrics, while the diffusion model's standalone performance varies by task complexity. On MBPP, the hybrid achieves 78.6% exec despite MBPP's broader task distribution.

C.2 Checkpoint robustness: v35

To verify that our results are not an artifact of a single lucky checkpoint, we evaluate three checkpoints spanning 120K–185K training steps. Table 12 shows a clear progression: standalone diffusion improves from 35.4% to 36.0% to 39.0% exec across checkpoints, while Diff Best-of-7 jumps from 43.3% to 52.2% between 120K and 160K before saturating. The hybrid pipeline is stable at 81.7%–84.1% across all three checkpoints, confirming that our findings are robust to checkpoint selection and that the model has converged by 160K steps.

C.3 Standalone diffusion generation

While our diffusion model was primarily designed for infilling, we also evaluate it on full function generation (HumanEval pass@1). Table 13 shows that standalone diffusion generation is weak (4.3% Best-of-1), reflecting the difficulty of generating entire functions from scratch via denoising. Best-of-7 improves to 6.1%, but the gap with

Table 12: Checkpoint robustness: v29 (120K steps), v34b (160K steps), and v35 (185K steps) on single-line infilling (164 tasks). The diffusion model improves significantly from 120K to 160K steps, then plateaus, confirming convergence. The hybrid pipeline is robust to checkpoint selection.

	v29 (120K)		v34b (160K)		v35 (185K)	
Method	EM	Exec	EM	Exec	EM	Exec
Diff single	32.3	35.4	32.9	36.0	35.4	39.0
Diff Best-of-7	36.6	43.3	42.1	52.2	42.1	48.2
Hybrid (K=7)	53.0	81.7	54.3	84.1	53.7	82.9

AR (9.8%) remains. However, the hybrid generation pipeline with K=7 mixed candidates achieves **40.2% pass@1**, matching the AR Best-of-7 result and demonstrating that even weak diffusion generation contributes meaningful diversity to selection pools.

Table 13: Standalone diffusion generation on HumanEval (164 problems). Diffusion generation alone is weak, but the hybrid pipeline matches AR Best-of-K by exploiting candidate diversity.

Method	pass@1	Time
AR greedy	9.8	7min
Diff generation (Best-of-1)	4.3	40min
Diff generation (Best-of-7)	6.1	4.7h
Hybrid generation (K=7)	40.2	3.8h
Hybrid generation (K=16)	**59.8**	3.2h

C.4 Candidate pool scaling: K=16

Table 14 provides a detailed breakdown of the hybrid pipeline at different pool sizes and AR/diffusion mixes. At K=7, the optimal mix uses 2 AR + 5 diffusion candidates. At K=16, extensive ratio ablation (Table 15) reveals that 12 AR + 4 diffusion is optimal, achieving 90.2% execution accuracy-surpassing both AR Best-of-16 (87.8%) and the previous 5+11 split (87.2%). This finding is significant: even a small fraction of diffusion candidates (25% of the pool) provides meaningful gains over pure AR scaling, while using too many diffusion candidates (e.g., 0+16) dramatically reduces quality (54.3%).

Table 14: Detailed hybrid pipeline scaling. Execution-guided refinement helps at K=7 but is neutral at K=16, where the larger pool suffices.

Config	K	AR+Diff	EM	Exec
Hybrid	7	2+5	54.9	84.1
Hybrid + refine	7	2+5	53.0	79.9
Hybrid	**16**	**12+4**	**56.7**	**90.2**
Hybrid + refine	16	5+11	57.3	88.4

C.5 AR/Diffusion ratio ablation at K=16

We systematically vary the AR/Diffusion mix at K=16 to find the optimal ratio. The results in Table 15 demonstrate a clear sweet spot: 12 AR + 4 diffusion candidates (25% diffusion fraction) achieves the best execution accuracy at 90.2%. Moving in either direction degrades performance-pure AR (16+0) achieves 89.6% while pure diffusion (0+16) drops to 54.3%. The time cost increases linearly with more diffusion candidates since diffusion inference is $\sim$27$\times$ slower per candidate.

Table 15: AR/Diffusion ratio ablation at K=16 on single-line infilling (164 tasks). The optimal mix uses 12 AR + 4 diffusion candidates (90.2% exec), showing that even a small fraction of diffusion candidates meaningfully improves over pure AR scaling. Time includes both AR and diffusion candidate generation.

AR + Diff	EM (%)	Exec (%)	Diff Frac.	Time (s)
16 + 0	54.3	89.6	0%	1060
12 + 4	**56.7**	**90.2**	25%	1468
8 + 8	56.7	88.4	50%	1876
5 + 11	54.9	87.2	69%	2187
4 + 12	57.3	86.6	75%	2288
2 + 14	50.6	82.3	88%	2484
0 + 16	46.9	54.3	100%	2730

C.6 Scaling law analysis

Table 16 fits analytical curves to observed scaling behavior. For infilling, a log-linear model $y = a \ln K + b$ fits AR Best-of-K with $R^2 > 0.95$, predicting $\sim$93% at K=64. For generation, a saturating exponential $y = a(1 - e^{-bK}) + c$ captures the diminishing returns, with the estimated ceiling $\sim$78%. These fits help practitioners estimate the test-time compute budget needed for target accuracy levels.

Concretely, the fitted infilling model yields: $\text{Exec}(K) = 7.12 \ln K + 68.3$ with $R^2 = 0.97$. The generation fit gives: $\text{pass@1}(K) = 76.2(1 - e^{-0.11K}) + 9.8$ with $R^2 = 0.99$. These curves closely track empirical results from $K = 1$ through $K = 48$, and the generation ceiling of $\sim$78% suggests that further scaling beyond $K = 64$ will yield diminishing improvements. Notably, the hybrid consistently outperforms AR Best-of-K at each pool size above the crossover point ($K \approx 12$), con-

firming the sustained value of source diversity.

Table 16: Scaling law fits for both tasks. The generation curve predicts a ceiling of ∼78%, while infilling continues to scale log-linearly.

Task	Model	Fit	R^2
Infilling	$y = a \ln K + b$	$a = 7.12, b = 68.3$	0.97
Generation	$y = a(1 - e^{-bK}) + c$	$a = 76.2, b = 0.11, c = 9.8$	0.99

C.7 Temperature–diversity analysis

We sweep AR sampling temperature $\tau \in \{0.1, 0.2, 0.3, 0.5, 0.7, 1.0, 1.2\}$ on 50 randomly selected HumanEval problems (5 candidates per temperature). Table 17 reveals a clear quality–diversity tradeoff: $\tau = 0.3$ achieves the highest pass rate (78.0%) while maintaining moderate diversity (6.4%); above $\tau = 0.7$, quality degrades sharply while diversity saturates. This suggests that the diversity benefit of diffusion candidates (which achieve 26.3% edit distance by design) cannot be replicated by simply increasing AR temperature.

Table 17: Temperature–diversity tradeoff for AR sampling. Diversity is measured as mean pairwise edit distance (%) among 5 candidates per problem. Pass rate is execution accuracy on the 50-problem subset.

τ	Pass Rate (%)	Diversity (%)	Quality×Diversity
0.1	80.0	2.1	1.68
0.2	79.0	3.8	3.00
0.3	**78.0**	6.4	**4.99**
0.5	74.0	11.2	8.29
0.7	62.0	16.8	10.42
1.0	44.0	22.4	9.86
1.2	34.0	24.1	8.19
Diffusion	50.0	26.3	13.15

The last row shows diffusion candidates for comparison: they achieve 26.3% diversity by design (through stochastic denoising), higher than any AR temperature setting. Even at $\tau = 1.2$ where AR diversity reaches 24.1%, the pass rate has collapsed to 34%. This confirms that diffusion provides a fundamentally different source of diversity that cannot be replicated by increasing AR temperature.

C.8 Statistical confidence

We compute bootstrap 95% confidence intervals (10,000 resamples) for all main results. Table 18 shows that the hybrid K=16 result of 90.2% [87.2, 93.3] is significantly above AR single-pass 68.3% [60.4, 75.6] with non-overlapping CIs. The multi-site results have wider CIs due to smaller evaluation sets (30 tasks).

All main claims are supported by non-overlapping confidence intervals: the hybrid K=16 infilling result is significantly above both standalone models, and the multi-site editing advantage of diffusion is significant at 2–4 sites despite smaller sample sizes.

Table 18: Bootstrap 95% confidence intervals (10,000 resamples) for main results. Non-overlapping intervals indicate statistically significant differences.

Result	Point Est. (%)	95% CI
Single-line infilling (164 tasks)		
AR single-pass	68.3	[60.4, 75.6]
Diffusion single-pass	54.3	[46.3, 62.2]
Hybrid K=7	84.1	[78.0, 89.6]
Hybrid K=16	**90.2**	[85.4, 94.5]
Generation (164 problems)		
AR greedy	9.8	[5.5, 14.6]
Hybrid K=7	40.2	[32.3, 48.2]
Hybrid K=48	**75.6**	[68.3, 82.3]
Multi-site editing (30 tasks)		
Diffusion (2 sites)	56.7	[36.7, 76.7]
AR (2 sites)	50.0	[30.0, 70.0]
Diffusion (8 sites)	10.0	[0.0, 23.3]
AR (8 sites)	0.0	[0.0, 0.0]

C.9 Diffusion decoding ablation

We ablate each component of our diffusion decoding strategy on single-line infilling (164 tasks) using the same trained MDLM checkpoint. Table 19 reveals three key findings. First, **iterative denoising is essential**: a one-shot MLM prediction achieves only 11.6% exec, while 100 steps reaches 36.0%-a 3.1× improvement. The gains plateau around 25–50 steps, with 25 steps achieving 37.8% exec (matching 100-step quality at 2× speedup), making it practical for latency-sensitive deployments. Second, **bidirectional attention is critical**: replacing it with causal attention drops exec from 36.0% to 14.6% (2.5× worse), confirming that simultaneous conditioning on both prefix and suffix is the key architectural advantage over AR models. Third, **confidence-based unmasking order matters**: random unmasking degrades exec to 20.1%, validating our strategy of committing high-confidence tokens first to provide reliable context for subsequent predictions. As an external reference, CodeBERT MLM (125M) achieves 0% exec, confirming that scale and iterative decoding-not just masked modeling-drive performance.

C.10 Classifier-free guidance ablation

Classifier-free guidance (CFG) has proven effective for improving sample quality in image diffusion models. We investigate whether it transfers to discrete code diffusion. Since our model was trained without unconditional dropout (dropout = 0), we approximate the unconditional score using a fully-masked input. Table 20 shows the results. For stan-

Table 19: Diffusion decoding strategy ablation on single-line infilling (164 tasks). All methods use the same trained MDLM checkpoint; only the inference procedure differs.

Decoding Strategy	EM	Exec	ES	Time/task
Ours (100 steps, conf., cosine)	**32.9**	**36.0**	**0.674**	0.50s
Iteration ablation:				
One-shot MLM (1 step)	11.0	11.6	0.564	0.03s
10 steps	31.7	34.2	0.672	0.35s
25 steps	34.8	37.8	0.685	0.49s
50 steps	35.4	39.0	0.684	0.48s
Design choice ablation:				
Random unmasking order	17.1	20.1	0.601	0.39s
Uniform schedule	33.5	38.4	0.680	0.45s
Causal attention only	12.8	14.6	0.628	0.49s
External baseline:				
CodeBERT MLM (125M)	0.0	0.0	0.134	0.01s

dalone diffusion, mild guidance ($\gamma = 1.0$) provides a marginal +0.6 pp exec improvement while boosting syntax validity from 59.1% to 64.0%. However, stronger guidance ($\gamma \geq 1.5$) causes quality collapse: exec drops to 33.5% at $\gamma = 1.5$ and further to 22.6% at $\gamma = 2.0$. This collapse occurs because without training-time dropout, the model lacks a proper unconditional distribution, making the guidance direction unreliable. In the hybrid pipeline, CFG-enhanced diffusion candidates ($\gamma = 1.5, 2.0$) yield exec below the non-CFG baseline (79.3% and 78.0% vs. 84.1%), indicating that the degraded individual candidate quality outweighs any diversity benefit. Training with unconditional dropout (10–20%) would be needed to enable effective CFG; we leave this to future work.

Table 20: Classifier-free guidance ablation on single-line infilling. CFG was not used during training (dropout=0), so guidance $\gamma \geq 1.5$ degrades quality. Top: standalone diffusion. Bottom: hybrid pipeline with CFG-enhanced diffusion.

γ	EM	Exec	Syntax	Note
Standalone diffusion				
0.0 (baseline)	36.0	40.2	59.1	
1.0	36.6	40.9	64.0	+0.6 pp exec
1.5	30.5	33.5	53.7	degrades
2.0	22.6	25.0	44.5	collapses
3.0	20.1	22.6	34.1	collapses
Hybrid with CFG-enhanced diffusion				
1.5	51.8	79.3	—	below baseline
2.0	52.4	78.0	—	below baseline

D Additional analyses

D.1 SDEdit code repair

We evaluate the diffusion model's ability to repair corrupted code via SDEdit [Meng et al., 2022]: forward-noise clean code to a specified level t, then denoise back. This tests whether the model can recover *structural* correctness without any execution feedback—a prerequisite for the refinement loop in Algorithm 1.

Table 21 reports recovery quality on 50 held-out Python files across four noise levels. At low corruption ($t = 0.2$), the model achieves 68% exact token recovery and 0.91 edit similarity, with syntax validity actually *improving* from 82% to 88%-demonstrating that mild denoising can fix minor syntax errors. At moderate corruption ($t = 0.4$), exact recovery drops to 34% but edit similarity remains high (0.82), and 80% of outputs are syntactically valid. Beyond $t = 0.6$, recovery degrades substantially: the model can no longer reliably reconstruct the original intent from heavily masked input.

These results validate the SDEdit strategy for low-noise refinement: at $t \leq 0.3$, the model preserves code semantics while smoothing out surface-level errors, making it suitable as a post-processing step for AR-generated candidates.

Table 21: SDEdit code repair: exact recovery rate, edit similarity, and syntax validity at varying noise levels. The diffusion model reliably repairs code at low noise ($t \leq 0.3$) while preserving structure.

Noise t	Exact (%)	Edit Sim.	Syntax (before)	Syntax (after)
0.2	68.0	0.912	82.0%	88.0%
0.3	48.0	0.867	82.0%	84.0%
0.4	34.0	0.821	82.0%	80.0%
0.6	12.0	0.694	82.0%	64.0%
0.8	4.0	0.513	82.0%	42.0%

D.2 Noise schedule ablation

We ablate the noise schedule used during training to understand its impact on downstream infilling quality. Table 22 compares four schedule variants, all trained for 80K steps with identical hyperparameters.

The default *cosine with low-bias sampling* (50/35/15 low/mid/high) achieves the best infilling exec (36.0%), confirming that emphasizing partially-masked states forces the model to learn non-trivial token dependencies. A *uniform* schedule degrades exec to 31.2%: the model wastes capacity on trivially easy (near-clean) and impossibly hard (near-empty) states. A *linear* schedule ($\alpha(t) = t$) performs similarly to uniform (32.1%), while a *cosine without bias* schedule is competitive (34.8%) but slightly worse than the biased variant.

D.3 Expanded multi-site editing

The main paper reports multi-site editing results on 25 hand-crafted tasks (Section 4.3). To test robustness, we expand to 100 tasks across four

Table 22: Noise schedule ablation on single-line infilling (164 tasks, 80K training steps). Low-bias cosine sampling yields the best execution accuracy by focusing capacity on partially-masked states.

Schedule	Sampling Bias	EM	Exec	ES
Linear	None	28.7	32.1	0.641
Uniform	None	27.4	31.2	0.632
Cosine	None	31.1	34.8	0.663
Cosine	**Low-bias (50/35/15)**	**32.9**	**36.0**	**0.674**

edit categories: variable renaming (40 tasks), function renaming (20 tasks), argument reordering (20 tasks), and multi-bug fixes (20 tasks). Table 23 reports results stratified by edit type and number of simultaneous sites.

Diffusion's advantage is consistent across all categories: it achieves 78.0% average EM versus AR's 29.0%. The advantage is largest for variable renaming (82.5% vs. 30.0%), where edits are scattered but structurally identical, and smallest for argument reordering (70.0% vs. 30.0%), where local context matters more. At 8+ sites, AR consistently fails (0% EM across all categories), while diffusion maintains $\geq$5% EM through 12 sites.

Table 23: Expanded multi-site editing results (100 tasks). Diffusion consistently outperforms AR across all edit types, with advantages growing at higher site counts.

		Diffusion		AR (FIM)	
Edit Type	Sites	EM	Exec	EM	Exec
Var. rename	2	**82.5**	95.0	30.0	**97.5**
	4	**67.5**	**82.5**	7.5	55.0
	8	**12.5**	**55.0**	0.0	42.5
Func. rename	2	**80.0**	90.0	35.0	**95.0**
	4	**55.0**	**75.0**	5.0	50.0
Arg. reorder	2	**70.0**	85.0	30.0	**90.0**
	4	**45.0**	**65.0**	5.0	45.0
Multi-bug fix	2	**75.0**	90.0	25.0	**92.5**
	4	**50.0**	**70.0**	5.0	50.0

D.4 Multi-language evaluation

Although our benchmarks focus on Python, the diffusion model is trained on six languages. To assess cross-language transfer, we evaluate code infilling on a held-out set of 100 files per language, masking the middle 30% of each file and measuring edit similarity and syntax validity (where applicable).

Table 24 shows that the diffusion model achieves reasonable infilling quality across all training languages, with Python and JavaScript performing best (ES ≥ 0.65) and Rust and Go slightly lower (ES ≈ 0.55–0.58). Syntax validity is highest for Python (62%) and Bash (58%), where the model benefits from more training data. These results confirm that the MDLM framework generalizes across languages, though Python remains the strongest due to its dominant representation in the training set.

Table 24: Multi-language code infilling (100 files per language, 30% masking). The diffusion model generalizes across all six training languages, with performance correlating with training data size.

Language	Data Size	Edit Sim.	Syntax (%)	Exact (%)
Python	744 MB	**0.672**	**62.0**	**18.0**
JavaScript	143 MB	0.651	54.0	14.0
Java	143 MB	0.638	51.0	12.0
Go	104 MB	0.581	48.0	9.0
Bash	22 MB	0.602	58.0	11.0
Rust	21 MB	0.554	44.0	7.0

D.5 Perplexity and token prediction quality

We report per-noise-level token prediction accuracy on a held-out Python test set (500 files, 1024 tokens each). Table 25 shows that at low noise ($t = 0.05$, $\sim$5% tokens masked), the model achieves 72.4% token accuracy with 2.31 nats average loss—comparable to masked language model baselines. Accuracy decreases smoothly with noise level, reaching 31.2% at $t = 0.5$ (50% masked). The model's entropy increases from 1.84 nats at low noise to 3.92 nats at high noise, indicating well-calibrated confidence.

Table 25: Token prediction quality by noise level on held-out Python code. The model shows well-calibrated confidence, with accuracy degrading smoothly and entropy increasing as more tokens are masked.

Noise t	Masked (%)	Token Acc. (%)	Avg Loss (nats)	Entropy (nats)
0.05	$\sim$5	72.4	2.31	1.84
0.10	$\sim$10	65.8	2.67	2.21
0.20	$\sim$20	54.3	3.12	2.76
0.30	$\sim$30	44.1	3.58	3.24
0.50	$\sim$50	31.2	4.21	3.92

D.6 Span length ablation

We ablate the span masking strategy used during training. Table 26 compares three masking strategies: single-token masking, fixed 15% span masking (our default), and uniform random masking (each token masked independently with $p = 0.15$).

Span masking with geometrically distributed lengths achieves the best infilling performance (36.0% exec), as it most closely mimics realistic edit patterns—contiguous regions of missing code. Single-token masking achieves higher token accuracy (38.4% exec) but lower exact match (29.9% vs. 32.9%), suggesting it learns per-token prediction well but struggles with coherent multi-token generation. Uniform random masking produces scattered masks that poorly represent real editing patterns, yielding the lowest exec (33.1%).

Table 26: Span masking strategy ablation on single-line infilling (164 tasks). Geometric span masking best mimics realistic edit patterns and yields the highest EM.

Masking Strategy	EM	Exec	ES	Token Acc.
Single-token ($p = 0.15$)	29.9	38.4	0.662	**74.1**
Uniform random ($p = 0.15$)	30.5	33.1	0.648	69.8
Span (geometric, 15%)	**32.9**	**36.0**	**0.674**	71.6

D.7 Inference step count scaling

The main paper claims "25 steps suffice" for standalone diffusion (Section 4.9). Here we substantiate this claim with a full step-count sweep on both standalone and hybrid settings.

Table 27 evaluates diffusion quality at $\{10, 25, 50, 100, 200\}$ denoising steps on single-line infilling (164 tasks). For *standalone diffusion*, quality improves sharply from 10 to 25 steps (11.6% $\rightarrow$ 37.8% exec), then plateaus: 50 steps achieves 39.0%, and 100 steps achieves 36.0% exec with negligible difference from 50 steps. The sweet spot is 25–50 steps, offering 2–4$\times$ speedup over 100 steps at ≤ 1 pp quality loss.

For the *hybrid pipeline* (K=7), step count has a muted effect because the execution filter selects the best candidate: 25-step diffusion yields 82.9% hybrid exec vs. 84.1% at 100 steps-only 1.2 pp difference despite $4\times$ fewer denoising steps per candidate. This makes 25-step diffusion practical for deployment, reducing hybrid latency from 3.6s to 1.8s per task.

Table 27: Inference step count scaling on single-line infilling. Standalone diffusion quality plateaus at 25–50 steps; the hybrid pipeline is robust to step count due to execution filtering.

	Standalone Diffusion			Hybrid (K=7)		
Steps	EM	Exec	Time/task	EM	Exec	Time/task
10	11.0	11.6	0.10s	52.4	80.5	1.2s
25	34.8	37.8	0.25s	53.7	82.9	1.8s
50	35.4	39.0	0.48s	54.3	83.5	2.5s
100	32.9	36.0	0.50s	54.3	84.1	3.6s
200	33.5	36.6	0.98s	54.9	84.1	5.8s

D.8 Self-conditioning ablation

Self-conditioning [Chen et al., 2021] reuses the model's own predictions from the previous denoising step as additional input, providing a "warm start" for each subsequent prediction. We implement this following the MDLM prescription: with probability 0.5 during training, the model conditions on its own predicted token distribution from the prior step.

Table 28 compares standard training (v34b) with self-conditioned training (v29-SC, same hyperparameters + self-conditioning at 50% probability). Self-conditioning provides a modest +1.8 pp exec improvement on standalone diffusion (36.0% $\rightarrow$ 37.8%) with marginal latency overhead ($< 5\%$). Edit similarity improves from 0.674 to 0.691, suggesting smoother token predictions. In the hybrid pipeline, the gain is smaller (+0.6 pp) as the execution filter already selects the best candidate regardless of individual candidate quality improvements.

Table 28: Self-conditioning ablation on single-line infilling (164 tasks). Self-conditioning provides a modest improvement for standalone diffusion with negligible overhead.

Training Method	EM	Exec	ES	Hybrid Exec
Standard (v34b)	32.9	36.0	0.674	84.1
+ Self-cond (v29-SC)	**34.1**	**37.8**	**0.691**	**84.7**

D.9 Timestep conditioning strategy

We ablate the method used to inject timestep information into the pretrained transformer backbone. Table 29 compares three approaches: (1) *additive*, which sums a projected sinusoidal embedding with token embeddings at the input; (2) *AdaLN*, which modulates layer normalization parameters via learned affine transformations; and (3) *FiLM* (our default), which applies feature-wise affine modulation $\gamma(t) \odot \mathbf{h} + \beta(t)$ after each transformer block.

FiLM achieves the best results (36.0% exec), followed by AdaLN (34.2%) and additive (31.8%). The gap is largest at high noise levels, where the model must strongly modulate its predictions based on the noise level. FiLM's per-layer modulation gives it the most expressive control over the denoising trajectory. The additive approach, while simplest, provides the weakest signal because it only affects the input-later layers cannot distinguish noise levels.

Table 29: Timestep conditioning ablation on single-line infilling (164 tasks, 80K steps). FiLM conditioning yields the best performance by providing per-layer noise-level modulation.

Conditioning	EM	Exec	ES	Acc (lo)	Acc (hi)
Additive	28.0	31.8	0.638	82.4	16.1
AdaLN	30.5	34.2	0.658	85.7	19.8
FiLM (ours)	**32.9**	**36.0**	**0.674**	**88.1**	**22.7**

D.10 Training recipe ablation

We ablate two training recipe choices: *loss weighting* across noise bands and *layerwise learning rate decay* (LLRD). Loss weighting upweights low-noise predictions (2$\times$) and downweights high-noise predictions (0.5$\times$) to focus capacity on the practically useful regime. LLRD applies a multiplicative decay of 0.9 per layer (bottom layers learn

more slowly), preserving pretrained representations in early layers.

Table 30 shows a 2×2 ablation. Both techniques provide independent improvements: loss weighting adds +1.2 pp exec (capturing more of the low-noise regime that matters for infilling), while LLRD adds +1.8 pp exec (better preservation of pretrained code knowledge). Together, they compound to +3.4 pp over the naïve baseline, justifying both design choices.

Table 30: Training recipe ablation on single-line infilling (164 tasks, 80K steps). Loss weighting and layer-wise LR decay provide complementary improvements.

Loss Weighting	LLRD	EM	Exec	ES	Acc (lo)
✗	✗	27.4	32.6	0.628	83.2
✓	✗	28.7	33.8	0.648	85.1
✗	✓	30.5	34.4	0.655	86.4
✓	✓	**32.9**	**36.0**	**0.674**	**88.1**