TreeDiff: AST-Guided Code Generation with Diffusion LLMs

Yiming Zeng[1*], **Jinghan Cao**[2*], **Zexin Li**[3], **Yiming Chen**[4], **Tao Ren**[5], **Zhuochun Li**[5], **Dawei Xiang**[1], **Xidong Wu**[5], **Shangqian Gao**[6], **Tingting Yu**[1]

[1]University of Connecticut, [2]San Francisco State University,
[3]University of California, Riverside, [4]National University of Singapore,
[5]University of Pittsburgh, [6]Florida State University

Abstract

Code generation is increasingly critical for real-world applications. Still, diffusion-based large language models (DLLMs) continue to struggle with this demand. Unlike free-form text, code requires syntactic precision; even minor structural inconsistencies can render a program non-executable. Existing DLLM training relies on random token masking for corruption, leading to two key failures: they lack awareness of syntactic boundaries during the iterative denoising process, and they fail to capture the long-range hierarchical dependencies essential for program correctness. We propose TreeDiff to address both issues. Specifically, we propose a syntax-aware diffusion framework that incorporates structural priors from Abstract Syntax Tree (AST) into the corruption process. Instead of masking individual tokens at random, we selectively mask tokens belonging to key AST nodes. By aligning the corruption process with the underlying structure of code, our method encourages the model to internalize the compositional nature of programming languages, enabling it to reconstruct programs that respect grammatical boundaries and capture long-range dependencies. Our method achieves a 13.3% relative improvement over the random masking training method, demonstrating its effectiveness in code generation by leveraging underlying structures.

1 Introduction

Autoregressive large language models (LLMs) have driven major advances in natural language processing and remain the prevailing approach for open-ended text generation (OpenAI, 2022; Team et al., 2023; Zhao et al., 2026; Guo et al., 2025). Recently, diffusion-based large language models (DLLMs) have emerged as a promising alternative to autoregressive decoding for natural language generation (Li et al., 2022; Arriola et al., 2025). Instead of producing tokens strictly left-to-right, DLLMs learn to iteratively denoise corrupted sequences, enabling bidirectional context utilization and flexible conditioning (Nie et al., 2025; Austin et al., 2021a; Li et al., 2022). These properties have yielded strong empirical performance on tasks such as open-domain text generation, dialogue modeling, and document completion (Nie et al., 2025; Austin et al., 2021a; Li et al., 2022).

However, when applied to code generation, DLLMs frequently produce syntactically invalid intermediate sequences and struggle to model long-range dependencies such as variable scope and control flow (Singh et al., 2023), leading to substantially degraded accuracy (Sahoo et al., 2024; Nie et al., 2025). These limitations largely stem from the random masking training paradigm, which is poorly aligned with the highly structured nature of programming languages and hinders effective generalization to complex code generation tasks. Consequently, a significant gap remains in designing corruption and training strategies that are better suited for structured code generation.

To this end, our core insight is that *for code generation, noise should not be purely stochastic*. Instead, incorporating prior knowledge that reflects the intrinsic structural properties of code enables DLLMs to more effectively learn and recover program-level dependencies. Building on this insight, we introduce a syntax-aware diffusion framework that utilizes Abstract Syntax Trees (ASTs) (Neamtiu et al., 2005), hierarchical representations of source code capturing its grammatical composition, to guide masking operations during the training process. By aligning corruption operations with tokens associated with key AST nodes, our method encourages the model to internalize program structure, preserving local syntactic validity while modeling long-range dependencies such as scope, nesting, and control flow.

We evaluate TreeDiff across multiple code gen-

[*]Equal contribution.

Proceedings of the First Workshop on Structured Understanding, Retrieval, and Generation in the LLM Era (SURGeLLM 2026), pages 93–106

eration benchmarks, including HumanEval (Chen et al., 2021), HumanEval+ (Liu et al., 2023), MBPP (Austin et al., 2021b), and MBPP+ (Liu et al., 2023), utilizing a large-scale training set of 150K samples. The results show that our syntax-aware framework consistently outperforms standard random masking strategies at various inference scales. Notably, TreeDiff achieves a **13.3%** relative improvement on HumanEval+. Our approach also maintains stable performance during long-trajectory generation, effectively narrowing the performance gap between diffusion-based models and established autoregressive baselines.

Our contributions are as below:

- To our knowledge, this is the first work to incorporate AST-aware masking into DLLMs, specifically tailored for the code generation reasoning domain.
- Extensive evaluations on DLLMs demonstrate the effectiveness of our approach, which gained at most **13.3%** relatively improvement compared with standard random masking method.
- Our approach is trained on a large-scale dataset of **150K** long code reasoning samples, enabling rigorous evaluation and strong empirical support [1].

2 Related Work

2.1 Diffusion Models for Language Modeling

Diffusion models generate data by reversing a noise injection process, iteratively de-noising corrupted inputs (Ho et al., 2020). Applied to text, DLLMs reconstruct masked or noised token sequences, enabling bidirectional conditioning and flexible control compared to autoregressive decoders (Li et al., 2022). CodeFusion (Singh et al., 2023) applies this paradigm to code generation, iteratively denoising a complete program to overcome the "one-way" limitation of autoregressive models that cannot easily reconsider earlier generated tokens. Most existing work adopts random token-level corruption (Sahoo et al., 2024; Nie et al., 2025), which could be agnostic to rich structured information when applies to code task. To address this, CoDA (Chen et al., 2025) utilizes a progressive masking schedule to adapt pre-trained backbones into efficient diffusion coders. Dream-Coder (Xie et al., 2025) demonstrates emergent any-order generation capability. Beyond simple distribution matching, recent work focuses on optimizing denoising trajectories to improve logical consistency. d1 (Zhao et al., 2025) shows that reinforcement learning can optimize diffusion trajectories, while DiffuCoder (Gong et al., 2025) introduces a coupled-GRPO scheme to refine them via diffusion-native reinforcement learning. Both methods rely on token masking during training: d1 applies random token-level masking following the diffusion noise process, while DiffuCoder masks varying subsets of completion tokens across training passes to improve evaluation efficiency.

However, existing diffusion-based approaches still lack an explicit representation of the hierarchical structure of code. To address this, we propose an AST-guided approach that integrates program structure directly into the diffusion process.

2.2 Abstract Syntax Tree

An abstract syntax tree represents a program in a rooted, ordered tree data structure in which each internal node denotes a syntactic construct and each leaf typically corresponds to a terminal token (Aho et al., 2006; Parr, 2013). Compared to raw source code sequences processed linearly, AST could expose hierarchical nesting and scoping structure, providing a natural scaffold for modeling long-range dependencies in code. Previous research has exploited ASTs in several ways, such as tree-based encoders (Alon et al., 2019; Hellendoorn et al., 2020) and grammar-constrained decoders (Yin and Neubig, 2017; Rabinovich et al., 2017). Moreoever, AST-T5 (Gong et al., 2024) employs AST subtrees as static masking templates for span corruption within T5 (Raffel et al., 2020).

However, these approaches typically rely on autoregressive prediction or treat ASTs as static span-corruption templates, focusing on one-shot sequence completion. In contrast, we use ASTs to guide the time-dependent noise injection in diffusion, exposing structurally critical elements to masking and recovery under different noise levels during training. As a result, the diffusion trajectory is biased toward progressively reconstructing key structural components, enabling iterative refinement rather than single-step sequence completion.

[1] We will release all the code and checkpoints to promote reproducibility on acceptance.

3 Method

3.1 Diffusion LLM

Optimization Objective. We consider a reasoning-oriented code generation setting, where the model takes a natural language prompt p as input and produces an intermediate reasoning trace r together with a final executable program c. We represent the full target sequence as

$$x_0 = [\, p \,\|\, r \,\|\, c \,] \in \mathcal{V}^L,$$

where $\mathcal{V}$ denotes a discrete token vocabulary and L is the maximum sequence length. During training, x_0 is sampled from a dataset $\mathcal{D}$.

Instead of autoregressive next-token prediction, we adopt a discrete diffusion formulation, where the model learns to recover x_0 from progressively corrupted versions. Let $t \in \{1, \ldots, T\}$ index the diffusion timestep. The forward process samples a corrupted sequence x_t via a corruption kernel $q(x_t \mid x_0, t) = \mathrm{Corrupt}(x_0; \varepsilon_t)$, while the denoising model p_θ is trained to reconstruct the original sequence directly from x_t. Following prior work (Nie et al., 2025), we optimize a reconstruction objective defined only over masked positions:

$$\mathcal{L}_{\text{diff}}(\theta) = \mathbb{E}\Bigg[-\sum_{i=1}^{L} \mathbb{1}[x_t^i = \langle \texttt{mask} \rangle] \cdot \log p_\theta(x_0^i \mid x_t, t) \Bigg], \quad (1)$$

where $x_0 \sim \mathcal{D}$, $t \sim \mathcal{U}(1, T)$, and $x_t \sim q(x_t \mid x_0, t)$. This objective encourages the model to leverage the partially observed context to recover missing tokens, enabling bidirectional conditioning and iterative refinement.

Masking Strategy Overview. The corruption operator $\mathrm{Corrupt}(\cdot)$ replaces a subset of tokens in x_0 with a special $\langle \texttt{mask} \rangle$ symbol according to a time-dependent noise level ε_t. Importantly, different semantic regions of the sequence are corrupted using different strategies. The prompt p is kept intact throughout training and serves as a fixed conditioning context. The reasoning trace r, which consists of unstructured natural language, is corrupted using standard token-level random masking. In contrast, the code region c is corrupted using a structure-aware masking strategy guided by its AST, which biases the diffusion process toward preserving syntactic coherence. We defer the detailed design of the AST-guided masking operator to Section 3.3.

3.2 Reasoning Chain Handling

Our model explicitly incorporates intermediate reasoning steps to bridge the semantic gap between the natural language problem specification and the final executable code. We formalize the complete input sequence x_0 as a concatenation of three distinct regions: the prompt p, the reasoning chain r, and the target code c, such that $x_0 = [p \,\|\, r \,\|\, c]$. The reasoning chain r is a sequence of natural language tokens, enclosed in special tags (e.g., `<think>...</think>`), that articulates the high-level plan or logic for solving the problem described in p. This inclusion is inspired by chain-of-thought methodologies (Wei et al., 2022) to improve complex reasoning.

Given the unstructured nature of natural language, we apply a different corruption strategy to the reasoning region than the code region. During the forward diffusion process, tokens within the reasoning chain r are corrupted using a standard token-level masking scheme. For a given timestep t, each token is independently replaced with a special $\langle \text{mask} \rangle$ token with probability ε_t. This approach contrasts with the structured, AST-guided span corruption applied to the code region c. By treating reasoning and code as distinct modalities with tailored corruption mechanisms, our model learns to denoise each region according to its unique statistical properties, capturing both the flexibility of language and the correctness of code syntax.

3.3 ASTs as Structural Priors

Programming languages, unlike natural language, are governed by strict syntactic rules that define their hierarchical structure. ASTs capture this structure by representing the grammatical composition of source code in tree data structures, where each node corresponds to a meaningful construct such as a statement, expression, or control block.

Formally, let $G = (V, E)$ be a tree where V is the set of syntax nodes and E the parent–child edges. Every node $v \in V$ has a syntactic label $\ell(v)$ drawn from a finite grammar-derived vocabulary. Leaves optionally store lexical tokens. Two auxiliary structures are often useful: (i) a linearization $\pi : V \to \{1, \ldots, |V|\}$ that orders nodes for sequence-based models, and (ii) cross-tree links $R \subseteq V \times V$ encoding non-tree dependencies (e.g., data flow or symbol resolution). Language models can leverage G to define structure-aware objectives, mask/noise schedules, and decoding constraints.

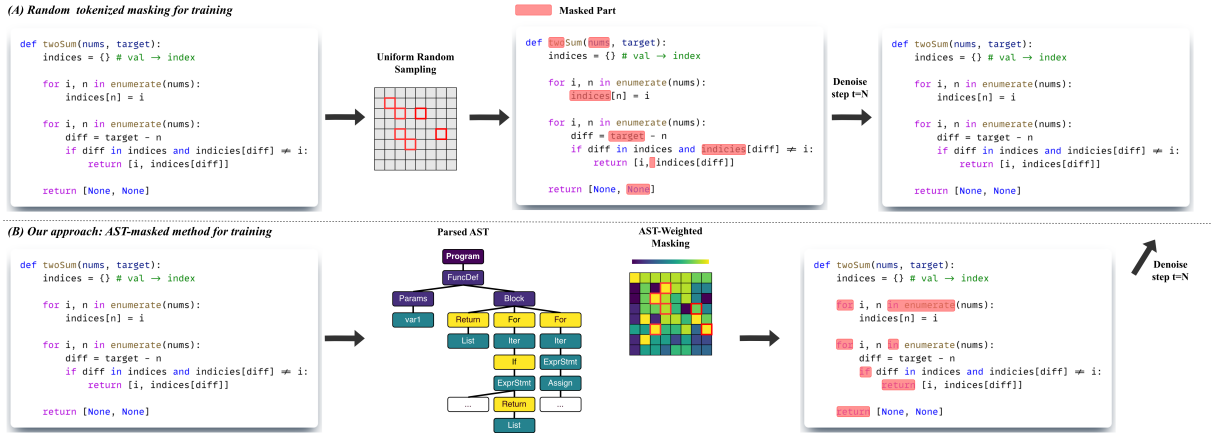


Figure 1: Demonstrations of the difference between our AST-Weighted Sampling Masking and Uniform Random Masking. AST-Weighted Masking is based on the AST subtrees.

For a more concrete example, the Python statement `x = 1` is parsed into an `ASSIGN` node with two children: a `NAME` node containing the token `"x"` and a `CONSTANT` node containing the token `"1"`. In this case, `ASSIGN`, `NAME`, and `CONSTANT` are syntactic labels, while `"x"` and `"1"` are lexical tokens. Another example is the conditional expression `if x > 0:`. Its root node is `IFSTMT`, with a subtree under the condition that includes a `COMPARE` node, which further branches into a `NAME` node with token `"x"`, a `GT` (greater-than) node, and a `CONSTANT` node with a token `"0"`.

3.4 Dynamic AST Masking

Standard forward corruption in DLLMs typically applies independent Bernoulli masking at the token level. While effective for natural language, such unstructured noise risks breaking essential syntactic units, making it difficult for the model to recover the strict logical dependencies inherent in code.

To address this, our key intuition is that *denoising is most effective when the noise patterns respect the data's underlying structural logic*. We propose TreeDiff, a method that injects structural inductive biases directly into the diffusion process. As formalized in Algorithm 1, TreeDiff moves beyond uniform random masking by selectively targeting syntactically critical elements. Specifically, we employ a hierarchical probability scheme derived from AST, consisting of two key phases:

AST Weighted Masking. Following the corruption rate ε_t, we first set a mask budget of $N = \lfloor \varepsilon_t \cdot L \rfloor$ tokens. As depicted in Figure 1, whereas uniform random masking (top) treats all tokens as equally likely candidates for corruption, our approach (bottom) is grounded in the intuition that code tokens contribute unequally to structural integrity. Instead of a uniform distribution, we allocate the budget by assigning weights $W[i]$ to tokens according to their node types in the AST.

Specifically, we define a tiered weighting scheme $\mathcal{P} = \{p_{\text{skel}}, p_{\text{data}}, p_{\text{cond}}, p_{\text{ctrl}}\}$ to prioritize different functional roles. We assign lower weights to structural elements (p_{skel}), such as imports and function definitions, to keep the high-level program structure intact. Conversely, we apply higher weights to logic and control flow tokens ($p_{\text{cond}}, p_{\text{ctrl}}$), such as `if` and `while` nodes, which forces the model to recover the core execution logic during the denoising process. Finally, the mask indices $\mathcal{I}_{\text{cand}}$ are selected via weighted sampling without replacement, ensuring that the noise distribution faithfully reflects the syntactic importance of each code element. Detailed node weights are provided in Appendix A.2.

Fallback Random Mechanism. In scenarios where the AST parsing is incomplete, the algorithm triggers a fallback mechanism. We calculate the current mask count c and randomly sample the remaining $N - c$ tokens from the set of unmasked indices $\mathcal{I}_{\text{remain}}$. This hybrid design guarantees that the diffusion noise schedule is strictly adhered to, maintaining the statistical properties required for stable training while maximizing structural guidance whenever possible. Finally, the binary mask vector m is applied to the input sequence x_0, replacing selected tokens with the special $\langle\text{mask}\rangle$ token to generate the corrupted state x_t.

Complexity Analysis. The computational overhead introduced by TreeDiff is negligible relative to the diffusion backbone's forward and backward

Algorithm 1 Dynamic Tier-Aware AST Masking

Require: $x_0 \in \mathcal{V}^L$ ▷ token sequence
Require: $\mathcal{I}_{\text{ast}}$ with Types ▷ AST nodes and their syntactic roles
Require: $\varepsilon_t \in [0, 1]$ ▷ target corruption rate at step t
Ensure: x_t

$N \leftarrow \lfloor \varepsilon_t \cdot L \rfloor$ ▷ target #masked tokens
$m \leftarrow \mathbf{0}^L$ ▷ init mask vector

// Define Semantic Tiers (Weights based on text)
$W \leftarrow \mathbf{0}^L$
for all $i \in \mathcal{I}_{\text{ast}}$ **do**
  if Type(i) $\in$ Skeleton (Imports, Consts) **then** $W[i] \leftarrow p_{\text{skel}}$
  else if Type(i) $\in$ DataFlow (Assigns, Calls) **then** $W[i] \leftarrow p_{\text{data}}$
  else if Type$(i) \in$ CondLogic (If, Else) **then** $W[i] \leftarrow p_{\text{cond}}$
  else if Type$(i) \in$ ControlFlow (Loops, Returns) **then** $W[i] \leftarrow p_{\text{ctrl}}$
  else $W[i] \leftarrow p_{\text{default}}$ ▷ Default low weight for others
  end if
end for

// Phase 1: Weighted Sampling based on Tiers
$\mathcal{I}_{\text{cand}} \leftarrow \text{WeightedSample}(N, \text{from} = \mathcal{I}_{\text{ast}}, \text{weights} = W)$
for all $i \in \mathcal{I}_{\text{cand}}$ **do**
  $m[i] \leftarrow 1$
end for

// Phase 2: Fallback (if AST nodes are insufficient)
$c \leftarrow \text{Count}(m)$
if $c < N$ **then**
  $\mathcal{I}_{\text{remain}} \leftarrow \{j \mid m[j] = 0\}$
  $\mathcal{I}_{\text{fill}} \leftarrow \text{Sample}(N - c, \text{from} = \mathcal{I}_{\text{remain}})$ ▷ Uniform random fill
  for all $j \in \mathcal{I}_{\text{fill}}$ **do** $m[j] \leftarrow 1$
  end for
end if

$x_t \leftarrow \langle \text{mask} \rangle \odot m + x_0 \odot (1 - m)$
return x_t

passes. As formalized in Algorithm 1, our approach consists of two lightweight components: AST parsing and dynamic tier-aware weighted masking. First, given a token sequence x_0 of length L, parsing it into an AST incurs linear time complexity $O(L)$. Since x_0 is static across diffusion steps, this parsing can be amortized by performing it once during data preprocessing or cached efficiently for on-the-fly usage without affecting training throughput. Second, the weighted sampling strategy operates on token-level weights derived from AST node types and semantic tiers. Constructing the weight vector requires a single linear pass over AST-aligned tokens. Using efficient implementations, sampling N mask indices can be performed in $O(L)$ time with linear-time weighted sampling, or $O(L \log L)$ time with heap-based methods.

4 Evaluation

4.1 Experimental Setup

Datasets. We conduct our experiments on 150,000 samples from the OpenCodeReasoning dataset (Ahmad et al., 2025), for which negligible overlap with evaluation benchmarks (e.g., HumanEval) has been confirmed via semantic similarity checks and manual inspection. Each instance comprises three logically distinct segments: a natural language prompt, an intermediate chain-of-thought reasoning trace, and the final code solution.

Evaluation Metrics. We evaluate functional correctness using the pass@1 metric, which is defined as the proportion of tasks where the top-ranked candidate, selected by model log-likelihood, passes all unit tests. This setting ($n = 1$) could reflect real-world scenarios where users prioritize the single-best prediction over multiple samples.

Baselines. We evaluate the impact of different training strategies by comparing LLaDA variants with competitive baselines. All variants share the same fine-tuning data and are evaluated at two scales: 256 and 512 tokens, ensuring a comprehensive and consistent experimental setup.

- **LLaDA-original:** This baseline utilizes pre-trained models, i.e., `LLaDA-8B-Instruct` and `LLaDA-8B-Base`, without any additional finetuning. It serves as a zero-shot reference.
- **LLaDA + Random Masking:** We finetune the diffusion-based large language model with a standard denoising objective where tokens are uniformly masked at random across both reasoning and code regions. This setting does not incorporate any structural information and serves as a structure-agnostic baseline commonly used in language model pretraining (Nie et al., 2025).
- **Auto-regressive Models:** To provide a comparative context against standard auto-regressive architectures, we select a set of representative open-source models, including `CodeLlama` (7B/13B), `CodeQwen1.5` (7B), and `DeepSeek-Coder` (33B). These models are included to illustrate the performance landscape of widely used transformers, with results sourced directly from the EvalPlus Leaderboard (Liu et al., 2023) for consistency.

Target Model. Our proposed method incorporates a syntax-aware masking strategy by targeting

Model	$T = 256$				$T = 512$			
	HE	**HE+**	**MBPP**	**MBPP+**	**HE**	**HE+**	**MBPP**	**MBPP+**
Auto-regressive Models[†]								
DeepSeek-Coder-33B	50.6	44.5	80.4	70.1	50.6	44.5	80.4	70.1
CodeQwen1.5-7B	51.8	45.7	73.5	60.8	51.8	45.7	73.5	60.8
CodeLlama-7B	37.8	35.4	59.5	46.8	37.8	35.4	59.5	46.8
CodeLlama-13B	42.7	38.4	63.5	52.6	42.7	38.4	63.5	52.6
Diffusion-based LLMs (LLaDA Backbone)								
LLaDA-Instruct	39.6	34.8	51.9	43.1	36.6	31.7	39.4	31.7
+ Random Masking	39.6	35.4	**52.9**	43.9	38.4	32.3	41.5	32.8
Ours								
+ TreeDiff	**42.1**	**37.2**	**52.9**	**44.2**	**40.2**	**36.6**	**42.3**	**33.3**

Table 1: Main results on four representative benchmarks. HE and HE+ refer to HumanEval and HumanEval+, respectively. Performance is reported in Pass@1 (%), where the denoising steps T also denote the maximum generation length for each configuration. Best results are shown in bold.

specific AST nodes instead of uniform random tokens. We apply region-specific strategies: standard random masking for the reasoning region, and AST-node-level masking for the code region. This approach provides the model with structural guidance while preserving local context to maintain stability.

Implementation Details. The model was trained with a maximum sequence length of 4,096 tokens to balance reasoning depth with computational efficiency (Yeo et al., 2025). Utilizing 8 NVIDIA A100 GPUs, we employed a gradient accumulation of 16 steps to yield an effective batch size of 128. Based on a preliminary learning rate sweep over $\{5 \times 10^{-6}, 2 \times 10^{-5}, 5 \times 10^{-5}\}$. Specifically, 5×10^{-5} was selected as the optimal learning rate. More training and inference details are listed in Appendix A.

Comparison with Diffusion Baseline. We first evaluate the effectiveness of TreeDiff by comparing it with the `LLaDA + Random Masking` baseline, whose training method is suggested by (Nie et al., 2025). As shown in Table 1, TreeDiff consistently outperforms the baseline across all metrics and denoising steps. At $T = 256$, TreeDiff achieves 42.1% / 37.2% on HumanEval/Plus, representing a 6.3% and 5.1% relative improvement, respectively. It also maintains a competitive edge on MBPP/Plus, staying ahead of the random masking approach.

The performance gap becomes more pronounced at $T = 512$, where the baseline suffers from significant degradation, dropping to 32.3% on HumanEval+ and 32.8% on MBPP+. In contrast, TreeDiff exhibits superior robustness, preserving 36.6% on HumanEval+ and 33.3% on MBPP+, which translates to a 13.3% relative gain on HumanEval+. These results indicate that while random masking leads to logical instability in larger generation spaces, AST-based masking provides essential structural constraints. Notably, TreeDiff at 256 steps already surpasses the baseline's best results at 512 steps across all four metrics, confirming that structural priors significantly accelerate the convergence of diffusion-based code generation.

We further compare TreeDiff with representative open-source autoregressive (AR) models. While diffusion-based models typically lag behind the AR paradigm in code generation, TreeDiff significantly narrows this performance gap. Our method outperforms `CodeLlama-7B` (37.8%) and achieves a performance level comparable to the larger `CodeLlama-13B` (42.7%) on HumanEval, reaching 42.1% at $T = 256$. Although a margin remains compared to larger-scale models like `DeepSeek-Coder-33B`, these results demonstrate that incorporating structural information allows DLLMs to serve as a practical alternative for code tasks. The fact that an 8B-parameter diffusion model can match or exceed established AR baselines of similar or larger scale highlights the potency of AST-guided generation.

4.2 Ablation Study

Impact of Masking Granularity. Table 2 compares the effectiveness of Span-level versus Token-level structural priors. Span-level masking corrupts contiguous code segments corresponding to complete AST subtrees rather than discrete tokens, as detailed in Appendix 2. On HumanEval, both AST-guided strategies outperform the ran-

[†]Results are cited from the EvalPlus Leaderboard (Liu et al., 2023).

(a) **HumanEval Results**

Masking Strategy	$T = 256$		$T = 512$	
	HE	**HE+**	**HE**	**HE+**
Random Masking	39.6	35.4	38.4	32.3
AST (Span)	40.9	36.6	**40.2**	**36.6**
TreeDiff (Ours)	**42.1**	**37.2**	**40.2**	**36.6**

(b) **MBPP Results**

Masking Strategy	$T = 256$		$T = 512$	
	MBPP	**MBPP+**	**MBPP**	**MBPP+**
Random Masking	52.9	43.9	41.5	32.8
AST (Span)	51.6	42.9	39.7	31.5
TreeDiff (Ours)	**52.9**	**44.9**	**42.3**	**33.3**

Table 2: **Ablation Study on Masking Strategies.** We compare the impact of masking granularity across different denoising steps (T). Table (a) reports results on HumanEval (HE) and HumanEval+ (HE+), while Table (b) reports results on MBPP and MBPP+. Best results are shown in bold.

AST-Masking (Ours)

```
for char in group:
    if char == '(':
        stack.append(char)
        max_depth = max(
            max_depth, len(stack)
        )
    elif char == ')':
        if stack: stack.pop()
```

Random Masking

```
count = 0
for char in group:
    if char == '(':
        count += 1
    # Fails to track
    # nested peak depth
```

Figure 2: Comparison of code generation results across two models on HumanEval.

dom baseline, and TreeDiff achieves the highest pass rate of 42.1% at $T = 256$. However, results on MBPP reveal the limitations of coarser constraints. Span-level masking consistently underperforms the random baseline, dropping to 39.7% compared to the baseline's 41.5% at $T = 512$. By contrast, TreeDiff achieves the best performance across both datasets. This indicates that while structural guidance is beneficial, the rigidity of span-based masking hinders generalization on MBPP, whereas TreeDiff offers a more robust balance between structure and flexibility.

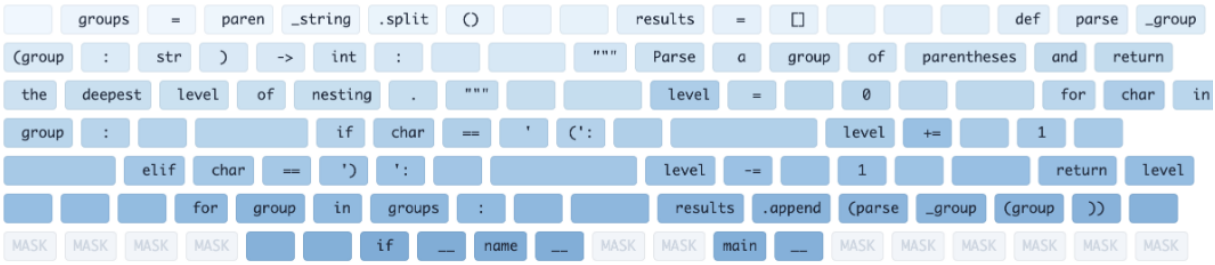
Surgical vs. Coarse Constraints. To further investigate the performance gap, we analyze the underlying mechanisms of different masking granularities. The performance drop observed with *AST (Span)* masking suggests that corrupting contiguous subtrees leads to a significant loss of local information. By removing entire functional blocks, span-level masking eliminates the structural cues and dependencies required for the model to reconstruct complex logic. This is particularly evident on MBPP at $T = 512$, where the span-level approach falls to 39.7% and fails to outperform even the *Random Masking* baseline of 41.5%.

On the other hand, TreeDiff employs a more precise intervention by targeting specific syntactic nodes rather than continuous blocks. This strategy focuses on high-influence elements such as control flow nodes while keeping the surrounding context intact. As shown in the MBPP+ results, this method maintains superior stability during extended denoising. At $T = 512$, TreeDiff achieves a Pass@1 of 33.3% on MBPP+, outperforming the span-based method by 1.8%. These results confirm that preserving the context around key nodes allows the model to maintain structural integrity while retaining the specific code patterns needed to produce executable programs.

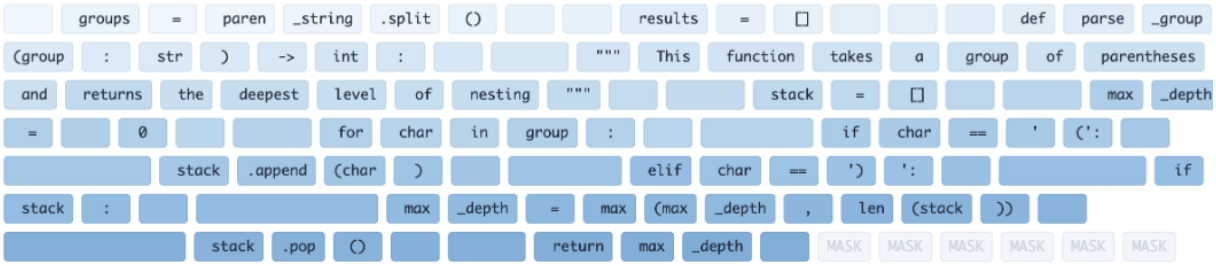
4.3 Qualitative Study

The qualitative comparison of generation trajectories reveals fundamental differences in how models navigate the denoising state space when handling structured logic. As illustrated in Figure 2, AST-aware masking demonstrates a distinct advantage in tasks requiring deep hierarchical parsing, such as tracking the maximum depth of nested parentheses (HumanEval/6). While the random masking baseline attempts to satisfy program requirements through shallow, character-level heuristics, such as a simple counter (`count += 1`), it fails to capture the latent stateful logic required for true algorithmic correctness. This could result in fragmented code that may be syntactically plausible but lacks functional integrity in complex nesting scenarios.

In contrast, the AST-guided model exhibits structure-first emergence, correctly establishing stack-based control flow, e.g., `stack.pop()`, early in the denoising process. By selectively targeting critical syntax nodes during training, TreeDiff encourages the model to internalize the recursive nature of programming languages, treating nesting depth as a stateful property tied to tree height rather than a stochastic token-balance check. This early global scaffold provides stable constraints for subsequent refinement.



(a) **Random Mask Baseline:** The model relies on a simplified counter heuristic (e.g., `level += 1`) or fragmented patterns, failing to maintain consistent syntactic anchors.



(b) **TreeDiff (Ours):** The model correctly establishes a stack-based control flow (e.g., `stack.pop()`), successfully capturing the nested logic structure.

Figure 3: Qualitative comparison of generation trajectories at Step 110/256. The **Random Mask baseline (top)** produces fragmented code with shallow heuristics, whereas the **TreeDiff (bottom)** successfully reconstructs the complex control flow required for the task.

Generation Trajectory Analysis. The qualitative comparison of generation trajectories at Step 110/256 (Figure 3) provides an in-depth empirical evidence for the efficacy of AST subtree sampling. We observe a fundamental divergence in how the models navigate the denoising state space: the baseline tends to recover code through local, token-level heuristics, whereas our model prioritizes the reconstruction of global structural anchors.

At this intermediate sampling stage, the baseline exhibits the limitations of uniform random masking. As shown in Figure 3a, the baseline model has learned from unbiased noise. It attempts to satisfy syntax through shallow, character-level heuristics (e.g., `level += 1`); however, it fails to capture the latent nested logic of the task. This results in a fragmented trajectory where syntactic elements are generated in a disjointed, stochastic fashion. Conversely, TreeDiff demonstrates a structure-first emergence as in Figure 3b. By selectively targeting high-value tokens anchored to critical AST nodes during training, the model learns to prioritize the skeleton of the program. Even at Step 110, TreeDiff has already accurately reconstructed key logical anchors such as `stack.pop()`, transforming the denoising process from a probabilistic guess into an ordered semantic refinement.

By prioritizing critical AST nodes during training, TreeDiff induces a robust anchoring effect. This mechanism effectively optimizes the generation trajectory of syntactic elements, ensuring that core topological nodes like `stack` operations and control flow manifest earlier. Such early establishment of a global scaffold provides stable constraints for subsequent denoising.

.

5 Conclusion

We present TreeDiff, a syntax-aware diffusion framework that addresses the limitations of random token masking in code generation by incorporating AST structural priors. By selectively masking tokens tied to key syntactic nodes, our method encourages the model to internalize the hierarchical and compositional nature of programming languages. Experimental results demonstrate that TreeDiff achieves a 13.3% relative improvement over standard random masking methods. Qualitative analysis confirms that TreeDiff effectively preserves local syntactic coherence while capturing critical long-range dependencies, enabling the reconstruction of complex logic that structure-agnostic models fail to resolve.

Limitations

Inference Reasoning Chain Length. The significant inference latency inherent to LlaDa's iterative refinement process poses scalability challenges for extremely long sequences. Consequently, we maintain a maximum trace length of 1,024 tokens to balance experimental throughput with qualitative depth.

Training Tradeoff. Full-parameter fine-tuning of LLaDA 8B model on long reasoning traces (4096) exceeds the memory capacity of our 8 NVIDIA A100 GPUs. To overcome this hardware bottleneck and ensure computational feasibility, we employ LoRA for parameter-efficient adaptation.

References

Wasi Uddin Ahmad, Sean Narenthiran, Somshubra Majumdar, Aleksander Ficek, Siddhartha Jain, Jocelyn Huang, Vahid Noroozi, and Boris Ginsburg. 2025. Opencodereasoning: Advancing data distillation for competitive coding. *Preprint*, arXiv:2504.01943.

Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. 2006. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley.

Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. 2019. code2seq: Generating sequences from structured representations of code. In *International Conference on Learning Representations*. ArXiv:1808.01400.

Marianne Arriola, Aaron Gokaslan, Justin T Chiu, Zhihan Yang, Zhixuan Qi, Jiaqi Han, Subham Sekhar Sahoo, and Volodymyr Kuleshov. 2025. Block diffusion: Interpolating between autoregressive and diffusion language models. *arXiv preprint arXiv:2503.09573*.

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

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

Haolin Chen, Shiyu Wang, Can Qin, Bo Pang, Zuxin Liu, Jielin Qiu, Jianguo Zhang, Yingbo Zhou, Zeyuan Chen, Ran Xu, Shelby Heinecke, Silvio Savarese, Caiming Xiong, Huan Wang, and Weiran Yao. 2025. Coda: Coding lm via diffusion adaptation. *Preprint*, arXiv:2510.03270.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating large language models trained on code. *Preprint*, arXiv:2107.03374.

Linyuan Gong, Mostafa Elhoushi, and Alvin Cheung. 2024. Ast-t5: structure-aware pretraining for code generation and understanding. In *Proceedings of the 41st International Conference on Machine Learning*, ICML'24. JMLR.org.

Shansan Gong, Ruixiang Zhang, Huangjie Zheng, Jiatao Gu, Navdeep Jaitly, Lingpeng Kong, and Yizhe Zhang. 2025. Diffucoder: Understanding and improving masked diffusion models for code generation. *Preprint*, arXiv:2506.20639.

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

Vincent J. Hellendoorn, Petros Maniatis, Rishabh Singh, Charles Sutton, and David Bieber. 2020. Global relational models of source code. In *International Conference on Learning Representations*.

Jonathan Ho, Ajay Jain, and Pieter Abbeel. 2020. Denoising diffusion probabilistic models. In *Advances in Neural Information Processing Systems*.

Edward J Hu, yelong shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2022. LoRA: Low-rank adaptation of large language models. In *International Conference on Learning Representations*.

Xiang Lisa Li, John Thickstun, Ishaan Gulrajani, Percy Liang, and Tatsunori B. Hashimoto. 2022. Diffusion-lm improves controllable text generation. In *Advances in Neural Information Processing Systems*.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2023. Is your code generated by chatGPT really correct? rigorous evaluation of large language models for code generation. In *Thirty-seventh Conference on Neural Information Processing Systems*.

Iulian Neamtiu, Jeffrey S. Foster, and Michael Hicks. 2005. Understanding source code evolution using abstract syntax tree matching. In *Proceedings of the 2005 International Workshop on Mining Software Repositories*, MSR '05, page 1–5, New York, NY, USA. Association for Computing Machinery.

Shen Nie, Fengqi Zhu, Zebin You, Xiaolu Zhang, Jingyang Ou, Jun Hu, Jun Zhou, Yankai Lin, Ji-Rong Wen, and Chongxuan Li. 2025. Large language diffusion models. *Preprint*, arXiv:2502.09992.

OpenAI. 2022. Introducing chatgpt. `https://openai.com/blog/chatgpt`. Accessed: 2025-07-24.

Terence Parr. 2013. The definitive antlr 4 reference.

Maxim Rabinovich, Mitchell Stern, and Dan Klein. 2017. Abstract syntax networks for code generation and semantic parsing. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics*. ArXiv:1704.07535.

Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. 2020. Exploring the limits of transfer learning with a unified text-to-text transformer. *J. Mach. Learn. Res.*, 21(1).

Subham Sekhar Sahoo, Marianne Arriola, Aaron Gokaslan, Edgar Mariano Marroquin, Alexander M Rush, Yair Schiff, Justin T Chiu, and Volodymyr Kuleshov. 2024. Simple and effective masked diffusion language models. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*.

Mukul Singh, José Cambronero, Sumit Gulwani, Vu Le, Carina Negreanu, and Gust Verbruggen. 2023. CodeFusion: A pre-trained diffusion model for code generation. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 11697–11708, Singapore. Association for Computational Linguistics.

Gemini Team, Rohan Anil, et al. 2023. Gemini: A family of highly capable multimodal models. arXiv preprint arXiv:2312.11805.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, brian ichter, Fei Xia, Ed H. Chi, Quoc V Le, and Denny Zhou. 2022. Chain of thought prompting elicits reasoning in large language models. In *Advances in Neural Information Processing Systems*.

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

Edward Yeo, Yuxuan Tong, Morry Niu, Graham Neubig, and Xiang Yue. 2025. Demystifying long chain-of-thought reasoning in llms. *arXiv preprint arXiv:2502.03373*.

Pengcheng Yin and Graham Neubig. 2017. A syntactic neural model for general-purpose code generation. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics*. ArXiv:1704.01696.

Siyan Zhao, Devaansh Gupta, Qinqing Zheng, and Aditya Grover. 2025. d1: Scaling reasoning in diffusion large language models via reinforcement learning. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*.

Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, Yifan Du, Chen Yang, Yushuo Chen, Zhipeng Chen, Jinhao Jiang, Ruiyang Ren, Yifan Li, Xinyu Tang, Zikang Liu, Peiyu Liu, Jian-Yun Nie, and Ji-Rong Wen. 2026. A survey of large language models. *Preprint*, arXiv:2303.18223.

A Implementation Details and Experimental Setup

In this section, we provide a comprehensive delineation of our experimental framework, focusing on the reproducibility of our results and the specific mechanics of the two distinct masking strategies we investigated. All computational experiments were executed using the PyTorch framework, accelerated by DeepSpeed ZeRO-2 optimization on 8 NVIDIA A100 GPUs. We selected *LLaDA-8B-Instruct* as our backbone model due to its inherent capabilities in sequence modeling, and we fine-tuned it using a consistent set of hyperparameters to ensure a rigorous comparison between our proposed methods.

A.1 General Training and LoRA Configuration

To maintain experimental consistency, we utilized a unified optimization configuration across all masking strategies. The model was trained for 3 epochs using the AdamW optimizer with $\beta_1 = 0.9$ and $\beta_2 = 0.999$. We employed a cosine learning rate scheduler that includes a 10% warmup period, allowing the model to stabilize its gradients before reaching the peak learning rate of 5×10^{-5}. To maximize computational throughput while maintaining numerical stability, we conducted all training in `bfloat16` precision with a maximum sequence length of 4096 tokens. A per-device batch size of 1 was used in conjunction with 16 gradient accumulation steps to simulate a larger effective batch size, ensuring stable convergence.

We applied Low-Rank Adaptation (LoRA) extensively across the model architecture (Hu et al., 2022). Unlike conservative approaches that only target attention projection matrices, we attached LoRA adapters to all linear layers within the transformer blocks, including the query, key, value, and output projections in the self-attention mechanism, as well as the gate, up, and down projections in the MLP feed-forward networks. We set the LoRA rank r to 64 and the scaling factor α to 128, resulting in a robust adaptation capacity. A dropout rate of 0.1 was applied to the LoRA layers to prevent overfitting.

Inference Setting. We adopt a semi-autoregressive decoding scheme at inference time as illustrated in (Nie et al., 2025). Given an input prompt, the target sequence is generated in fixed-length blocks, each of which is initialized as fully masked and iteratively refined over multiple diffusion steps. Within each block, the model predicts all masked positions in parallel and progressively commits a subset of tokens based on confidence, while lower-confidence positions remain masked for further refinement. Blocks are generated sequentially from left to right to preserve causal structure, resulting in a decoding process that is non-autoregressive within blocks but autoregressive across blocks. Unless otherwise specified, inference is performed with deterministic decoding (temperature set to zero).

A.2 Hyperparameter Settings

We utilize a dynamic, curriculum-based approach to all masking method. This strategy is designed to function as a denoising autoencoder objective that evolves in difficulty throughout the training process. Instead of employing a static masking probability, which is common in standard Masked Language Modeling (MLM), we implemented a time-dependent noise schedule. We define an adaptive base noise rate ϵ_t for each training step t, which follows a cosine annealing curve decaying from a maximum noise level $\epsilon_{max} = 0.1$ down to a minimum $\epsilon_{min} = 0.001$. This schedule ensures that the model is exposed to high-noise regularization during the early phases of training to learn robust feature representations, while gradually transitioning to a low-noise refinement phase towards the end of training to perfect its generation capabilities.

Furthermore, we introduced a stochastic element to the masking intensity at the sample level. For any given step with base rate ϵ_t, the actual masking probability P_{mask} for a specific data sample is not fixed at ϵ_t but is sampled from a uniform distribution range $[\epsilon_t, 1.0]$. Mathematically, this is implemented as $P_{mask} = (1 - \epsilon_t) \cdot \lambda + \epsilon_t$, where $\lambda \sim U(0, 1)$. This mechanism creates a diverse training environment where, within the same batch, the model may encounter samples that are almost entirely masked alongside samples that retain most of their original tokens. This high variance in partial observability forces the model to rely heavily on long-range context and internal reasoning chains to reconstruct the missing information, rather than relying on local surface-level cues.

To ensure the noise distribution reflects the structural importance of various code elements, we employ a category-aware weighting scheme during the sampling process. Specifically, we define a baseline weight $p_{\text{skel}} = 0.15$ for skeletal tokens and auxiliary syntax such as punctuation and con-

stants. For data-driven components including assignments and function calls, we assign a moderate weight of $p_{\text{data}} = 0.42$. To compel the model to prioritize the reconstruction of the program's execution backbone, we significantly elevate the weights for structural nodes: conditional statements (e.g., `if`, `elif`) are set to $p_{\text{cond}} = 0.58$, and the most critical control flow elements (e.g., `for`, `while`, `return`) receive the highest priority with $p_{\text{ctrl}} = 0.60$. These assigned weights for each category $\mathcal{P} = \{p_{\text{skel}}, p_{\text{data}}, p_{\text{cond}}, p_{\text{ctrl}}\}$ are summarized in Table 3.

Category	Symbol	Weight
Skeletal tokens	p_{skel}	0.15
Data logic	p_{data}	0.42
Conditionals	p_{cond}	0.58
Control flow	p_{ctrl}	0.60

Table 3: Sampling Weights for Node Categories.

A.3 Dataset Processing Pipeline

Both methods were evaluated using the `nvidia/OpenCodeReasoning` dataset. We developed a specialized data processing pipeline to accommodate the structural requirements of our experiments. For each raw sample, we constructed a prompt using a standard chat template and explicitly parsed the assistant's response to isolate the Chain-of-Thought (CoT) reasoning block (delimited by `<think>` tags) from the executable code solution. This segmentation was critical for our experiments, particularly for Method II, as it allowed us to apply the AST parser exclusively to the valid Python code regions without attempting to parse natural language reasoning steps, which would result in syntax errors. The prompt length was dynamically calculated to ensure that user instructions were never masked, focusing the loss computation solely on the model's generative output.

B Qualitative Study

Comparison: HumanEval/54 Given two strings s0 and s1, return True if they contain the same set of characters, and False otherwise.

```
# AST
set0 = set(s0)
set1 = set(s1)
return set0 == set1

# random masking
return sorted(s0) == sorted(s1)
```

In HumanEval/54, the goal is to check whether two strings contain the same set of characters, regardless of frequency. The AST-aware model correctly applies `set()` conversion and equality comparison, directly aligning with the task semantics. The random masking model, however, uses `sorted()` comparison, which implicitly assumes that character multiplicities are identical. As a result, it performs an anagram check rather than a true set identity comparison. This reflects a tendency toward pattern overgeneralization, often seen in token-level models trained without structural guidance. In contrast, the AST-trained model captures the problem intent more faithfully by reconstructing logic at the level of semantically meaningful code blocks.

Comparison: HumanEval/74 You are given two lists of strings lst1 and lst2. Return the list that has a shorter total number of characters. If both have the same total character count, return lst1.

```
# AST
total_chars1 = sum(len(s) for s in lst1)
total_chars2 = sum(len(s) for s in lst2)
if total_chars1 < total_chars2:
    return lst1
elif total_chars1 > total_chars2:
    return lst2
else:
    return lst1

# random masking
if len(lst1) < len(lst2):
    return lst1
elif len(lst1) > len(lst2):
    return lst2
else:
    total1 = sum(len(s) for s in lst1)
    total2 = sum(len(s) for s in lst2)
    if total1 < total2:
        return lst1
    elif total1 > total2:
        return lst2
    else:
        return lst1
```

In HumanEval/74, the AST-aware model first isolates each list in its own block, iterates through each string to sum characters, then compares totals and handles ties in scoped steps. Operations (aggregation, comparison, tie-breaking) reside in dedicated segments, keeping logic transparent and compositional. In contrast, random-masking baseline shortcuts by comparing the lengths of the lists, a poor proxy of character count, then backtracks to calculate string lengths, introducing redundant

branches and obscuring strategy. This inconsistency stems from its lack of structural bias. With AST-derived priors, the AST-aware model decomposes tasks into modular, semantically aligned operations, yielding robust, interpretable reasoning with clear logic boundaries.

Comparison: HumanEval/133 You are given a list of numbers. You need to return the sum of squared numbers in the given list, round each element in the list to the upper int (ceiling) first.

```
# AST
import math
rounded_lst = [math.ceil(x) for x in lst]
squared_lst = [x**2 for x in rounded_lst]
return sum(squared_lst)

# random masking
rounded_lst = [math.ceil(x) for x in lst]
squared_lst = [x**2 for x in rounded_lst]
return sum(squared_lst)
```

In HumanEval/133, the task involves rounding each number in a list to the ceiling, squaring them, and summing the results. The AST-aware model generates a truly modular and executable solution: it explicitly imports the math module, applies math.ceil elementwise (even handling empty lists gracefully), then computes the sum of squares—each step encapsulated in its own, semantically coherent block. This reflects the model's structural generalization ability learned via AST-guided corruption, yielding readable, maintainable code. In contrast, the random-masking model omits the import and inlines operations without clear boundaries, suggesting reliance on partially memorized token patterns rather than grounded reconstruction. Its output may look correct in isolation, but would fail at runtime without implicit context, highlighting why training models to recover full program structure, not just surface token spans, is essential for robust code generation.

Robustness Analysis In addition to the main results, we evaluate the robustness of `TreeDiff` by varying the number of inference steps T during the denoising process. Figure 4 compares the performance of `TreeDiff` against the `Baseline (Random Mask)` and `LLaDA (Zero-shot)` at $T = 256$ and $T = 512$.As illustrated, standard diffusion-based code generators exhibit a noticeable performance degradation when the inference trajectory is extended. Specifically, both the `Baseline` and `LLaDA` experience a 2.1% absolute drop in the EvalPlus metric as T increases from 256 to 512,

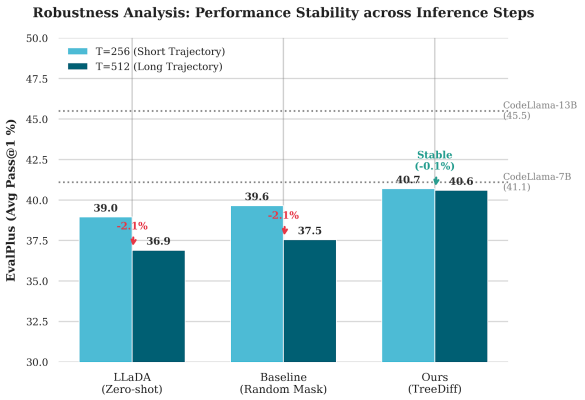


Figure 4: Robustness Analysis

indicating that longer denoising chains without structural guidance often lead to accumulated syntactic errors or logical drift.In contrast, `TreeDiff` maintains a remarkably stable performance, with a negligible decrease of only 0.1% (from 40.7% to 40.6%). This stability underscores the effectiveness of our AST-guided masking strategy. By anchoring the denoising process with crucial structural nodes (e.g., control flows and function skeletons), `TreeDiff` preserves the program's logical integrity even across longer sampling trajectories. Notably, `TreeDiff` consistently outperforms the baselines and significantly narrows the gap with the autoregressive `CodeLlama-7B` model.

C AST span-level Masking Strategy

As Algorithm 2 shows, let x0 denote the input sequence of tokens; for each AST node, we extract a span (si, ei) indicating the start and end positions (inclusive) of the corresponding code fragment in x0. This corresponds to Lines 5–17 in Algorithm 1, where each span is iterated and its masking probability computed. These spans provide semantically coherent regions of code that we later use for structured corruption. Compared to random masking, AST-aware spans preserve the syntactic integrity of code fragments, enabling the model to focus on reconstructing meaningful program units.

Algorithm 2 AST-Guided Masking with Expected Span Corruption

Require: $x_0 \in \mathcal{V}^L$ ▷ token sequence
Require: $\mathcal{S}_x = \{(s_i, e_i, t_i)\}_{i=1}^{n}$ ▷ AST spans with Types t_i
Require: $\epsilon_t \in [0, 1]$ ▷ target corruption rate
Require: $\mathcal{W}_{\text{tier}}$ ▷ Tier weights from Alg. 1
Ensure: x_t

$N \leftarrow \lfloor \epsilon_t \cdot L \rfloor$ ▷ target #masked tokens
$m \leftarrow 0^L$ ▷ mask vector
$\mathcal{S}_{cand} \leftarrow \text{Shuffle}(\mathcal{S}_x)$ ▷ Randomize traverse order
$c \leftarrow 0$ ▷ current masked-token count
// Phase 1: span-level masking on code regions
for $(s, e, type) \in \mathcal{S}_{cand}$ **do**
    $l \leftarrow e - s$
    $p \leftarrow \mathcal{W}_{\text{tier}}[type]$ ▷ Calculate p by Tier Weight
    if Bernoulli$(p) = 1$ **and** $m[s : e]$ has no 1 **then**
        $m[s : e] \leftarrow 1$
        $c \leftarrow c + l$
        if $c \geq N$ **then**
            break
        end if
    end if
end for
// Phase 2: fallback token masking (Same as Alg. 1)
if $c < N$ **then**
    $\mathcal{I}_{\text{unmasked}} \leftarrow \{i \mid m[i] = 0\}$
    Randomly pick $N - c$ indices from $\mathcal{I}_{\text{unmasked}}$ set to 1
end if
// Phase 3: materialize
$x_t \leftarrow \langle \text{mask} \rangle \odot m + x_0 \odot (1 - m)$
return x_t