

# TreeDiff: AST-Guided Code Generation with Diffusion LLMs

Yiming Zeng<sup>1</sup>, Jinghan Cao<sup>2</sup>, Zexin Li<sup>3</sup>, Yiming Chen<sup>4</sup>,  
Tao Ren<sup>5</sup>, Dawei Xiang<sup>1</sup>, Xidong Wu<sup>6</sup>, Shangqian Gao<sup>7</sup>, Tingting Yu<sup>1</sup>

<sup>1</sup>Computer Science and Engineering, University of Connecticut

<sup>2</sup>Computer Science, San Francisco State University

<sup>3</sup>Electrical and Computer Engineering, The University of California, Riverside

<sup>4</sup>Electrical and Computer Engineering, National University of Singapore

<sup>5</sup>Information Science, University of Pittsburgh

<sup>6</sup>Electrical and Computer Engineering, University of Pittsburgh

<sup>7</sup>Computer Science, Florida State University

## Abstract

Recent advances in diffusion-based language models have opened new possibilities for controllable and bidirectional sequence generation. These models provide an alternative to traditional autoregressive approaches by framing text generation as an iterative denoising process. However, applying diffusion models to structured domains such as source code remains a significant challenge. Programming languages differ from natural language in that they follow strict syntactic and semantic rules, with hierarchical organization that must be preserved for correctness. Standard token-level corruption techniques used during training often ignore this structure, which may hinder the model’s ability to learn meaningful representations of code. To address this limitation, we propose a syntax-aware diffusion framework that incorporates structural priors from Abstract Syntax Tree (AST) into the denoising process. Instead of masking individual tokens at random, we selectively corrupt syntactically meaningful code spans derived from AST subtrees. This enables the model to reconstruct programs in a way that respects grammatical boundaries and captures long-range dependencies. By aligning the corruption process with the underlying structure of code, our method encourages the model to internalize the compositional nature of programming languages. Experimental results demonstrate that syntax-aware corruption significantly improves syntactic correctness, reconstruction accuracy, and generalization to unseen code patterns. These findings highlight the potential of incorporating structural information into diffusion-based training and suggest that syntax-guided denoising is a promising direction for advancing diffusion-based language models in code generation tasks.

## Introduction

Diffusion-based language models have recently emerged as a promising alternative to autoregressive decoders for natural language generation. By learning to iteratively denoise corrupted sequences, these models enable bidirectional reasoning and flexible conditioning, which have shown strong empirical results across open-domain text generation, dialogue modeling, and document completion (Nie et al. 2025; Austin et al. 2021b; Li et al. 2022b). At the same time, code generation has become an increasingly important benchmark for evaluating language models, with wide-reaching

applications in software development, education, and automation (Chen et al. 2024; Chen, Pusalra, and Ray 2025; Chen et al. 2025; Babe et al. 2024; Jiang et al. 2024; Zhao et al. 2024b). LLMs such as Gemini, DeepSeek, and ChatGPT, have achieved impressive performance on tasks like function synthesis, bug fixing, and code completion (DeepSeek-AI 2024; Team, Anil et al. 2023; OpenAI 2022). These models power real-world developer tools, such as GitHub Copilot and Amazon CodeWhisperer, and have become the dominant approach to program synthesis (Cursor 2023; Amazon Web Services 2023; GitHub 2021).

Despite the rise of diffusion-based language models and the importance of code generation, their intersection remains largely underexplored. Existing attempts to apply diffusion-based methods typically rely on unstructured, token-level corruption strategies that are not designed with the structural characteristics of source code in mind (Li et al. 2022b; Sahoo et al. 2024b; Nie et al. 2025). Existing diffusion-based language models were initially proposed for discrete text generation (Austin et al. 2021a; Li et al. 2022a), where random token-level corruption was sufficient. However, when applied to code, such models often generate syntactically invalid intermediate sequences and fail to capture scope and control flow dependencies (Singh et al. 2023), resulting in weaker accuracy (Sahoo et al. 2024a; Nie et al. 2025). Such limitations hinder the diffusion-based language model’s ability to learn and generalize effectively to complex programming tasks. Consequently, there remains a considerable gap in developing corruption strategies better suited for structured code generation tasks.

To address this gap, we revisit the fundamental principles underlying diffusion models, recognizing that effective denoising is contingent upon employing structured and semantically meaningful noise patterns that closely mirror the intrinsic properties of the target data. Given the inherently hierarchical and compositional nature of source code, we posit that corruption techniques explicitly informed by syntactic structures could significantly enhance diffusion-based code generation. To this end, we introduce a syntax-aware diffusion framework that utilizes Abstract Syntax Trees (ASTs), hierarchical representations of source code capturing its grammatical composition, to guide masking operations dur-

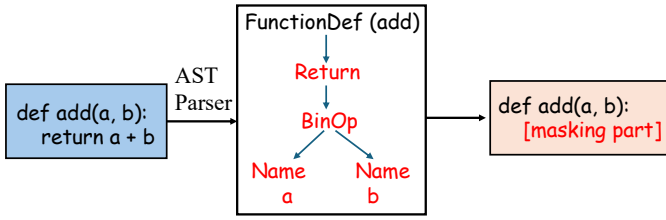


Figure 1: Overview of our AST-aware training setup. The source code is parsed into an AST, from which structured subtrees are selected and masked during training. The masked part is marked by red color.

ing the training process (Neamtiu, Foster, and Hicks 2005). By strategically aligning the corruption and subsequent denoising procedures with coherent code segments identified through AST nodes (illustrated in Figure 1), our approach enables the model to develop robust reconstruction capabilities. This approach effectively preserves local syntactic coherence while capturing critical long-range structural dependencies inherent in the source code.

Our contributions are as follows:

- To the best of our knowledge, this is the first work to incorporate AST-aware masking into large language diffusion models, specifically tailored for the code generation reasoning domain.
- Different from existing methods that rely primarily on unstructured random masking, we propose an AST-guided masking strategy that encourages the model to recover structurally meaningful spans and capture long-range syntactic dependencies more effectively.
- Extensive evaluations on diffusion-based language models and widely-used code generation benchmarks demonstrate the effectiveness and generality of our approach. Our approach is trained on a large-scale dataset of 150K code reasoning samples, enabling rigorous evaluation and strong empirical support.

## Background

### Diffusion Models for Language Modeling

Diffusion models generate data by reversing a noise injection process, iteratively de-noising corrupted inputs (Ho, Jain, and Abbeel 2020; Song et al. 2021). When applied to text, diffusion-based language models reconstruct masked or noised token sequences, enabling bidirectional conditioning and flexible control compared to autoregressive decoders (Austin et al. 2021b; Li et al. 2022b; Nie et al. 2025). However, most existing work adopts random token-level corruption (Sahoo et al. 2024b), which ignores code structure and often produces invalid or hard-to-recover programs, motivating syntax-aware alternatives like the AST-guided approach explored in this work.

### Abstract Syntax Tree

An abstract syntax tree (AST) represents a program as a rooted, ordered tree in which each internal node denotes a

syntactic construct (e.g., a function definition, loop, or conditional) and each leaf typically corresponds to a terminal token or literal (Aho et al. 2006; Parr 2011). Compared to a flat token sequence, an AST exposes (i) hierarchical nesting, (ii) scoping and binding structure, and (iii) typed relations between nodes. These properties make ASTs a natural scaffold for modeling long-range dependencies in code.

Previous approaches exploit ASTs in several ways in enhancing deep neural networks: tree-based encoders (Alon et al. 2019; Hellendoorn et al. 2020), grammar-constrained decoders (Yin and Neubig 2017; Rabinovich, Stern, and Klein 2017), and pointer networks over symbol tables (Li et al. 2022c). However, these researches (a) focus on autoregressive predictions or (b) apply random corruption independent of syntax. In contrast, we use the AST both to decide where and how to inject noise and to condition the denoising process, resulting in a diffusion model that respects the structure of the program throughout the generation. Concretely, we design corruption operators that (i) drop or shuffle whole subtrees, (ii) replace node labels while preserving arity, or (iii) inject type-consistent placeholders, ensuring that intermediate noisy states remain syntactically plausible and thus recoverable.

## Long Chain-of-Thought Training

Recent studies have shown that long CoT supervision encourages generation of multi-step reasoning, typically ranging from 10 to 20 steps per example depending on task complexity (Zhu et al. 2025a; Lin et al. 2025; Zhu et al. 2025b). Unlike natural language, code exhibits deep nesting, hierarchical scope, and precise dependency structures, making long-form reasoning especially vital. Works like DeepSeek-Coder (Guo et al. 2024), CodeGemma (Zhao et al. 2024a), and CodeLLaMA (Rozière et al. 2023) highlight the importance of training in multi-hop programming trajectories and long-range edit sequences. In particular, DeepSeek-R1 (DeepSeek-AI 2024) introduces an explicit CoT-style reasoning corpus containing step-by-step programming plans and rationales, enabling models to generate code with greater structural alignment and task decomposition capability. Our method is based on diffusion-based language modeling, offering a fundamentally different approach to learning long-range structured reasoning for code.

## Methodology

### Diffusion LLM

**Task definition.** Given a natural language specification  $p$  (e.g., problem description), an intermediate reasoning trace  $r$ , and the corresponding target program  $c$  (tokenized code), the training objective is to reconstruct the full sequence  $x_0 = [p \parallel r \parallel c]$  from a corrupted version  $x_t$  using a denoising objective. The complete sequence  $x_0 \in \mathcal{V}^L$  is sampled from a dataset  $\mathcal{D}$  and consists of three semantic regions: prompt, reasoning, and code. Reasoning and code region is corrupted differently during training to reflect its structure and semantics.

**Discrete diffusion formulation.** Instead of next-token prediction, a diffusion language model learns to *denoise* progressively corrupted versions of  $x_0$ . Let  $t \in \{1, \dots, T\}$  index the diffusion timestep. The forward (noising) process samples a corrupted sequence  $x_t$  from  $x_0$  via a corruption kernel  $q(x_t | x_0, t)$ ; the reverse (denoising) model  $p_\theta$  learns to predict a cleaner sequence  $x_{t-1}$  from  $x_t$ :

$$\begin{aligned} q(x_t | x_0, t) &= \text{Corrupt}(x_0; \varepsilon_t), \\ p_\theta(x_{t-1} | x_t, t) &= \text{LM}_\theta(x_t, t). \end{aligned} \quad (1)$$

Here  $\varepsilon_t \in (0, 1)$  controls the corruption strength (e.g., masking rate) at step  $t$ . In the simplest token-masking instantiation, we sample a binary mask vector  $m_t \sim \text{Bernoulli}(\varepsilon_t)^L$  and replace masked positions with a special token  $\langle \text{mask} \rangle$ :

$$x_t[i] = \begin{cases} \langle \text{mask} \rangle, & \text{if } m_t[i] = 1, \\ x_0[i], & \text{otherwise.} \end{cases} \quad (2)$$

**Optimization objective.** Following prior work on discrete diffusion LLMs (Nie et al. 2025), we optimize a denoising loss that encourages  $p_\theta$  to reconstruct the less corrupted sequence  $x_{t-1}$  (or directly  $x_0$ ) from  $x_t$ . We uniformly sample  $t$  and minimize the token-level cross-entropy over the unmasked positions of  $x_{t-1}$ :

$$\mathcal{L}_{\text{diff}}(\theta) = \mathbb{E}_{x_0 \sim \mathcal{D}} \mathbb{E}_{t \sim \mathcal{U}(1, T)} \mathbb{E}_{x_t \sim q(\cdot | x_0, t)} [\text{CE}(p_\theta(x_{t-1} | x_t, t), x_{t-1})] \quad (3)$$

In practice, we implement  $p_\theta$  as a Transformer decoder conditioned on the timestep  $t$  via learned embeddings. The corruption schedule  $\{\varepsilon_t\}_{t=1}^T$  controls the masking difficulty over training steps. In the next section, we describe how to replace uniform token-level corruption with a structure-aware alternative based on code syntax.

## 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  $x_0 = [p \parallel r \parallel c]$ . The reasoning chain  $r$  is a sequence of natural language tokens, enclosed in special tags (e.g.,  $\langle \text{think} \rangle \dots \langle / \text{think} \rangle$ ), that articulates the high-level plan or logic for solving the problem described in  $p$ . This inclusion is inspired by chain-of-thought methodologies 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  $\varepsilon_t$ . 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 rigidity of code syntax.

## Abstract Syntax Trees as Structural Priors

Programming languages, unlike natural language, are governed by strict syntactic rules that define their hierarchical structure. Abstract Syntax Trees (ASTs) capture this structure by representing the grammatical composition of source code as a tree, 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 (e.g., IFSTMT, FORSTMT, IDENTIFIER). Leaves optionally store lexical tokens. Two auxiliary structures are often useful: (1) a linearization  $\pi : V \rightarrow \{1, \dots, |V|\}$  that orders nodes for sequence-based models, and (2) cross-tree links  $R \subseteq V \times V$  encoding non-tree dependencies (e.g., data flow or symbol resolution). While standard compilers rely on  $G$  to perform semantic checks, learning-based models can leverage  $G$  to define structure-aware objectives, mask/noise schedules, and decoding constraints.

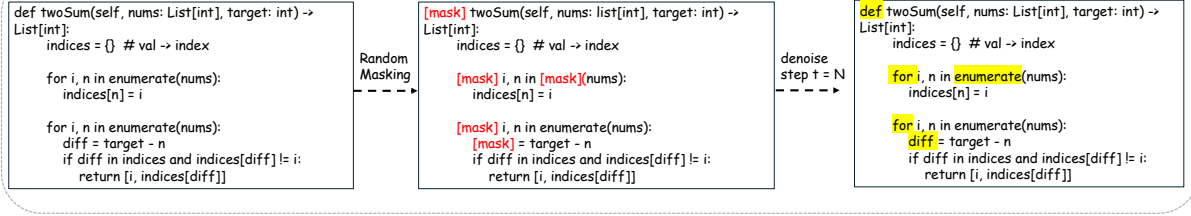
For a concrete example, the Python statement `x = 1` is parsed into an ASSIGN node with two children: a NAME node containing the token "x" and a CONSTANT node containing the token "1". In this case, ASSIGN, NAME, and CONSTANT are syntactic labels, while "x" and "1" are lexical tokens. Another example 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 token "0".

We propose leveraging ASTs as a form of domain-specific prior to inform the corruption process in diffusion-based language models. Given a program snippet, we parse its AST using language-specific parsers (e.g., `ast` module for Python), and map each AST node to its corresponding span in the original source code. To integrate this information into token-level modeling, we convert character-level AST spans into tokenizer-indexed ranges. As Algorithm 1 shows, let  $x_0$  denote the input sequence of tokens; for each AST node, we extract a span  $(s_i, e_i)$  indicating the start and end positions (inclusive) of the corresponding code fragment in  $x_0$ . This corresponds to Lines 9–12 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.

## AST-Guided Masking with Span-Level Control

The standard forward corruption in diffusion LMs applies independent Bernoulli masking at the token level. While effective for natural language, this unstructured corruption may break syntactic units and hinder learning for code generation, where structural coherence is crucial. We therefore propose a span-level corruption mechanism that operates on AST-derived spans and preserves a global corruption budget.

(A) Random tokenized masking for training



(B) Our approach: AST-masking method for training

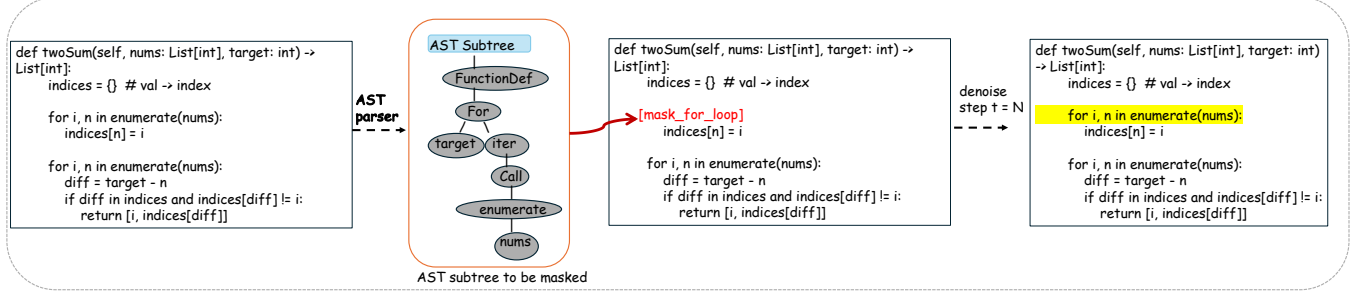


Figure 2: Our AST-guided masking strategy. We extract syntactic subtrees from the abstract syntax tree (AST) and mask entire spans (e.g., for-loops) during training. The model is then trained to reconstruct the missing structures. This approach leverages code structure to guide denoising and improves robustness over random masking.

**Expectation-preserving span sampling.** Let  $x_0 \in \mathcal{V}^L$  be the token sequence of length  $L$ , and let  $\mathcal{S}_x = \{(s_i, e_i)\}_{i=1}^n$  be the set of AST spans (half-open intervals) with lengths  $\ell_i = e_i - s_i$ . At timestep  $t$ , the target corruption rate is  $\varepsilon_t \in [0, 1]$  (equivalently a budget of  $N = \lfloor \varepsilon_t L \rfloor$  tokens). We independently sample a binary variable  $z_i \sim \text{Bernoulli}(p_i)$  for each span, with

$$p_i = 1 - (1 - \varepsilon_t)^{\ell_i}. \quad (4)$$

If  $z_i = 1$ , we mask *all* tokens in  $[s_i, e_i)$ . This choice preserves the global corruption budget in expectation:

$$\mathbb{E} \left[ \sum_{i=1}^n z_i \ell_i \right] = \sum_{i=1}^n p_i \ell_i = \varepsilon_t L. \quad (5)$$

Intuitively, Eq. (4) is the probability that at least one of the  $\ell_i$  tokens in the span would be masked under a token-wise Bernoulli( $\varepsilon_t$ ) process, thus matching the token-level expectation while operating at span granularity.

**Instantiation.** Let  $\mathcal{T}$  be the AST of the code region. For every node  $v \in \mathcal{T}$  we extract the half-open token interval  $(s_v, e_v)$  that covers the subtree rooted at  $v$  (length  $\ell_v = e_v - s_v$ ). We form the candidate set  $\mathcal{S}_x = \{(s_v, e_v)\}$  after two simple filters: (i) remove spans shorter than two tokens to avoid masking trivial leaves; and (ii) collapse duplicate intervals that map to the same token range. We then sample each remaining span independently with  $z_i \sim \text{Bernoulli}(p_i)$ , where  $p_i$  is calculated by Eq. 4. If  $z_i = 1$ , all tokens in  $[s_i, e_i)$  are replaced by the mask token. We do not enforce an exact global budget; overlaps are allowed, and later selections simply overwrite already-masked tokens. When a sequence contains no AST spans, e.g., pure natural language,

we fall back to standard token-wise Bernoulli( $\varepsilon_t$ ) masking. This simple scheme approximates the target corruption rate in expectation while staying faithful to syntactic units when they are available.

**Algorithm Description.** Algorithm 1 summarizes our AST-guided masking procedure. Initially, we compute the target number of masked tokens based on the corruption rate and initialize the mask vector (Lines 1-2). We randomly shuffle AST spans and iterate through them, computing the masking probability for each span (Lines 3-8). Selected spans are entirely masked, and the masked token count is updated until reaching the masking budget (Lines 9-13). If additional masking is needed, random unmasked tokens are selected to fulfill the budget (Lines 14-17). Finally, we replace the masked tokens with the special [MASK] token to produce the corrupted sequence (Lines 18-20).

**Complexity Analysis.** Let  $n$  be the number of AST spans and  $L$  the sequence length. The main loop in Phase 1 iterates over up to  $n$  spans and performs constant-time checks and updates per span. The fallback mask selection in Phase 2 operates over  $O(L)$  tokens. Therefore, the total time complexity of the algorithm is  $O(n + L)$ . In practice,  $n$  is sparse compared to  $L$ , making the method highly efficient for long-sequence training.

## Evaluation

### Experiment Setup

**Datasets.** We conduct our experiments on 150,000 samples from the OpenCodeReasoning dataset (Ahmad et al.

---

**Algorithm 1: AST-Guided Masking with Expected Span Corruption**


---

**Require:**  $x_0 \in \mathcal{V}^L$  ▷ token sequence  
**Require:**  $\mathcal{S}_x = \{(s_i, e_i)\}_{i=1}^n$  ▷ AST spans (half-open intervals  $[s_i, e_i)$ )  
**Require:**  $\varepsilon_t \in [0, 1]$  ▷ target corruption rate  
**Ensure:**  $x_t$

```

1:  $N \leftarrow \lfloor \varepsilon_t \cdot L \rfloor$  ▷ target #masked tokens
2:  $m \leftarrow \mathbf{0}^L$  ▷ mask vector
3:  $\mathcal{S}_{\text{cand}} \leftarrow \text{Shuffle}(\mathcal{S}_x)$ 
4:  $c \leftarrow 0$  ▷ current masked-token count

5: // Phase 1: span-level masking on code regions
6: for all  $(s, e) \in \mathcal{S}_{\text{cand}}$  do
7:    $\ell \leftarrow e - s$ 
8:    $p \leftarrow 1 - (1 - \varepsilon_t)^\ell$  ▷ Eq. (4)
9:   if  $\text{Bernoulli}(p) = 1$  and  $m[s : e)$  has no 1 then
10:     $m[s : e) \leftarrow 1$ ;  $c \leftarrow c + \ell$ 
11:    if  $c \geq N$  then
12:      break
13:    end if
14:  end if
15: end for

16: // Phase 2: fallback token masking
17: if  $c < N$  then
18:    $\mathcal{I}_{\text{unmasked}} \leftarrow \{i \mid m[i] = 0\}$ 
19:   Randomly pick  $N - c$  indices from  $\mathcal{I}_{\text{unmasked}}$  and set them
   to 1 in  $m$ 
20: end if

21: // Phase 3: materialize
22:  $x_t \leftarrow [\text{MASK}] \odot m + x_0 \odot (1 - m)$ 
23: return  $x_t$ 

```

---

2025), holding out an additional 1,000 examples for validation. Each instance comprises three logically distinct segments: a natural language prompt, an intermediate chain-of-thought reasoning trace, and the final code solution. To facilitate region-aware learning, we explicitly tokenize and label these three segments into prompt, reasoning, and solution spans. This segmentation enables us to apply semantically aligned masking strategies: we use dynamic token-level corruption on reasoning traces to encourage robustness in logical generation, and AST-guided span-level masking on the code solution to promote structural understanding of programs. When AST spans are unavailable, we fall back to token-level masking in the code region. This design allows the model to learn with fine-grained inductive bias across natural language, reasoning, and code generation stages.

**Evaluation Metrics.** We measure generation quality using the *pass@1* metric, which computes the probability that the single top-scoring sample produced by the model exactly matches the reference solution. Concretely, for each test instance we draw  $n$  candidate outputs (here  $n = 1$ ), score them according to model log-likelihood, and regard the instance as solved if the highest-scoring candidate compiles

and passes all unit tests. Let  $N$  be the total number of test instances and  $c$  the number of instances solved in this way, *pass@1* can be calculated by:

$$\text{pass@1} = \frac{c}{N}.$$

We choose *pass@1* to reflect the practical requirement that users inspect only the single best prediction, and to avoid variability introduced by sampling multiple outputs. This metric aligns well with real-world code-generation scenarios where a single correct suggestion is preferable.

**Baselines** To evaluate the impact of different training strategies, we compare four variants of LLaDA under controlled conditions. All models are fine-tuned on the same dataset and evaluated under two generation lengths: 512 and 1024 tokens.

- **(1) LLaDA-original:** This baseline uses the pretrained LLaDA-8B-Instruct model without any additional fine-tuning. It serves as a zero-shot reference and reflects the model’s out-of-the-box reasoning and generation capability.
- **(2) LLaDA + Random Masking:** We fine-tune the 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.
- **(3) LLaDA + AST-guided Token Masking:** This variant introduces fine-grained structure-aware denoising by leveraging the abstract syntax tree (AST) of the code. Instead of applying uniform random masking, we assign masking probabilities based on the semantic roles of AST nodes. Specifically, we use the following probability table:

Node Type	Masking Prob.
Others (punctuation, constants)	0.15
Assignments / Function calls	0.42
Conditionals (if, elif)	0.58
Control flow (for, while, return)	0.60

Each code token is assigned a probability according to its AST node type. Masking is applied at the token level. This allows the model to learn structure-sensitive representations while avoiding excessive corruption of syntactically critical elements.

**Target Model.** Our proposed method further extends the AST-based strategy by masking entire AST spans instead of individual tokens. This respects syntactic boundaries and avoids fragmenting code constructs. We apply region-specific masking strategies: random token-level masking for the reasoning region, and span-level masking for the code region. To further improve training stability, we design an adaptive masking schedule where the corruption rate  $\varepsilon$  decays over time using a cosine schedule. This curriculum-based strategy exposes the model to low-noise inputs in the

Table 1: HumanEval pass@1 with 512/1024-token prompts.

Model	@512	@1024
<i>w/o length constraint</i>		
LLaMA2-7B		16.50
<i>Length-controlled</i>		
LLaDA-Instruct	28.66	32.32
+ Random Masking	31.71	33.54
+ AST Token Masking	31.71	28.66
<b>+ AST Span + <math>\varepsilon</math> (Ours)</b>	<b>32.93</b>	<b>36.59</b>

early stages of training, allowing it to learn core patterns first, and gradually increases difficulty to enhance generalization to noisier contexts.

**Implementation Details.** We train the model with a maximum input length of 4 096 tokens, getting a balance between expressive reasoning and computational efficiency (Yeo et al. 2025). Each GPU processes a single example, and the gradients are accumulated in 16 steps, yielding an effective batch size of 128. Optimization is performed with AdamW and a cosine-decay learning rate schedule that includes a 10 % warm-up phase. In a preliminary learning-rate sweep of  $5 \times 10^{-6}$ ,  $2 \times 10^{-5}$ , and  $5 \times 10^{-5}$ , the last value achieved the best validation performance, and we therefore adopt  $5 \times 10^{-5}$  for all subsequent experiments. Training resumes from a checkpoint corresponding to 150 000 processed examples to accelerate convergence. We evaluate on the validation set every 50 steps and apply early stopping once the validation loss ceases to improve.

### Overall Effectiveness

We evaluate the effectiveness of our proposed syntax-aware diffusion model on the HumanEval and MBPP benchmarks. Results are summarized in Tables 1 and 2.

On HumanEval, our method achieves a pass@1 of 32.93% with 512-token prompts and 36.59% with 1024-token prompts, outperforming the instruction-tuned LLaDA baseline by 4.27% at both lengths. Compared to the random masking variant, our model achieves gains of 1.22% and 3.05%, respectively. While the AST token masking baseline performs comparably at 512 tokens, its performance drops at 1024 tokens, suggesting that token-level structural modeling is less effective as input length increases. In contrast, our span-level masking maintains performance by preserving syntactic integrity during the denoising process. This improvement can be attributed to the model’s ability to mask and reconstruct complete syntactic units such as function calls, conditional blocks, or loop bodies, which helps it better capture the hierarchical structure of code. Such modeling is especially advantageous for longer prompts, where preserving global coherence and nesting structure becomes increasingly important.

On MBPP, our model reaches 33.07% pass@1 at 512 tokens, outperforming the LLaDA baseline at 25.89% and the random masking variant at 31.13%. The AST token masking baseline again underperforms at 24.51%, likely due to

Table 2: MBPP pass@1 with 512-token prompts.

Model	@512
<i>w/o length constraint</i>	
LLaMA2-7B	20.60
<i>Length-controlled</i>	
LLaDA-Instruct	25.89
+ Random Masking	31.13
+ AST Token Masking	24.51
<b>+ AST Span + <math>\varepsilon</math> (Ours)</b>	<b>33.07</b>

the fragmented nature of token-level corruption, which can disrupt meaningful substructures and reduce the model’s ability to learn robust syntactic patterns. In contrast, our AST-guided span masking mechanism introduces a strong structural inductive bias, encouraging the model to focus on higher-level program semantics rather than surface-level token sequences. Additionally, the use of curriculum noise scheduling during training enables the model to progressively adapt to harder reconstruction tasks, which further improves robustness and generalization in the diffusion generation process.

These results support the conclusion that combining syntax-aware span masking with carefully designed noise scheduling leads to consistent improvements in code generation quality. The proposed method demonstrates particular strength in handling longer and more complex inputs, where structural awareness and long-range reasoning are critical. Overall, our approach effectively leverages program syntax as a guide for denoising, resulting in more accurate and semantically faithful code completions across diverse benchmarks.

### Qualitative Study

We selected several representative examples from the outputs of the random masking and AST-aware diffusion models on the HumanEval benchmark to analyze their behavioral differences in code reasoning.

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

```

1 # AST
2 set0 = set(s0)
3 set1 = set(s1)
4 return set0 == set1

1 # random masking
2 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`.

```

1 # AST
2 total_chars1 = sum(len(s) for s in lst1)
3 total_chars2 = sum(len(s) for s in lst2)
4 if total_chars1 < total_chars2:
5     return lst1
6 elif total_chars1 > total_chars2:
7     return lst2
8 else:
9     return lst1

1 # random masking
2 if len(lst1) < len(lst2):
3     return lst1
4 elif len(lst1) > len(lst2):
5     return lst2
6 else:
7     total1 = sum(len(s) for s in lst1)
8     total2 = sum(len(s) for s in lst2)
9     if total1 < total2:
10        return lst1
11    elif total1 > total2:
12        return lst2
13    else:
14        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.

```

1 # AST
2 import math
3 rounded_lst = [math.ceil(x) for x in lst
4 ]
5 squared_lst = [x**2 for x in rounded_lst
6 ]
7 return sum(squared_lst)

1 # random masking

```

```

2 rounded_lst = [math.ceil(x) for x in lst
3 ]
4 squared_lst = [x**2 for x in rounded_lst
5 ]
6 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.

## Discussion and Limitation

Our syntax-aware diffusion framework highlights the broader potential of incorporating structural priors into generative models, suggesting applications beyond code, such as document generation or dialogue modeling, where domain-specific structure, e.g., discourse or logic trees, may guide denoising more effectively. While our results are promising, there are limitations. We do not evaluate on extremely large-scale models such as GPT-4-class or black-box settings, and our experiments focus on white-box training with full access to ASTs and gradients. Moreover, we limit our evaluation to reconstruction and completion tasks; extending to program synthesis or multi-file reasoning remains future work. Despite these constraints, our findings provide a foundation for integrating structural guidance into diffusion-based LLMs across diverse structured domains.

## Conclusion

We present a syntax-aware diffusion framework for code generation that integrates AST structures into the denoising process of diffusion-based language models. Motivated by the limitations of random token-level corruption in capturing the hierarchical and compositional nature of programming languages, our method selectively masks syntactically coherent code spans derived from AST subtrees. This design enables the model to reconstruct code in a structurally aware manner, preserving grammatical integrity and enhancing its ability to model long-range dependencies. Our experiments show that syntax-guided corruption significantly improves performance across syntactic correctness, reconstruction accuracy, and generalization to unseen code patterns. These results demonstrate the importance of incorporating domain-specific structure into the training dynamics of diffusion models and highlight the potential of structured denoising as a promising direction for advancing controllable and accurate code generation. Future work can further

explore broader structural priors and extend our approach to other structured domains beyond source code.



## References

- Ahmad, W. U.; Narenthiran, S.; Majumdar, S.; Ficek, A.; Jain, S.; Huang, J.; Noroozi, V.; and Ginsburg, B. 2025. OpenCodeReasoning: Advancing Data Distillation for Competitive Coding. *arXiv:2504.01943*.
- Aho, A. V.; Lam, M. S.; Sethi, R.; and Ullman, J. D. 2006. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley.
- Alon, U.; Brody, S.; Levy, O.; and Yahav, E. 2019. code2seq: Generating Sequences from Structured Representations of Code. In *International Conference on Learning Representations*. ArXiv:1808.01400.
- Amazon Web Services. 2023. Announcing Amazon Code-Whisperer. <https://aws.amazon.com/blogs/aws/announcing-amazon-codewhisperer/>. Accessed: 2025-07-24.
- Austin, J.; Johnson, D.; Ho, J.; Tarlow, D.; Shayer, P.; Parmar, N.; Isola, P.; and Thomas, D. J. 2021a. Structured denoising diffusion models in discrete state-spaces. In *Advances in Neural Information Processing Systems*, volume 34, 17984–17997.
- Austin, J.; Johnson, D. D.; Ho, J.; Tarlow, D.; and van den Berg, R. 2021b. Structured Denoising Diffusion Models in Discrete State-Spaces. In *Advances in Neural Information Processing Systems*.
- Babe, H. M.; Nguyen, S.; Zi, Y.; Guha, A.; Feldman, M. Q.; and Anderson, C. J. 2024. StudentEval: A Benchmark of Student-Written Prompts for Large Language Models of Code. In Ku, L.-W.; Martins, A.; and Srikumar, V., eds., *Findings of the Association for Computational Linguistics: ACL 2024*, 8452–8474. Bangkok, Thailand: Association for Computational Linguistics.
- Chen, S.; Chen, Y.; Li, Z.; Jiang, Y.; Wan, Z.; He, Y.; Ran, D.; Gu, T.; Li, H.; Xie, T.; et al. 2025. Recent advances in large language model benchmarks against data contamination: From static to dynamic evaluation. *arXiv preprint arXiv:2502.17521*.
- Chen, S.; Feng, X.; Han, X.; Liu, C.; and Yang, W. 2024. Ppm: Automated generation of diverse programming problems for benchmarking code generation models. *Proceedings of the ACM on Software Engineering*, 1(FSE): 1194–1215.
- Chen, S.; Pusarla, P.; and Ray, B. 2025. Dynamic benchmarking of reasoning capabilities in code large language models under data contamination. *arXiv preprint arXiv:2503.04149*.
- Cursor. 2023. Cursor: The AI Code Editor. <https://www.cursor.so/>. Accessed: 2025-07-24.
- DeepSeek-AI. 2024. DeepSeek LLM Technical Report. *arXiv preprint arXiv:2401.06066*.
- GitHub. 2021. GitHub Copilot: Your AI pair programmer. <https://github.blog/2021-06-29-introducing-github-copilot-ai-pair-programmer/>. Accessed: 2025-07-24.
- Guo, D.; Zhu, Q.; Yang, D.; Xie, Z.; Dong, K.; Zhang, W.; Chen, G.; Bi, X.; Wu, Y.; Li, Y.; Luo, F.; Xiong, Y.; and Liang, W. 2024. DeepSeek-Coder: When the Large Language Model Meets Programming – The Rise of Code Intelligence. *arXiv preprint arXiv:2401.14196*.
- Hellendoorn, V. J.; Maniatis, P.; Singh, R.; Sutton, C.; and Bieber, D. 2020. Global Relational Models of Source Code. In *International Conference on Learning Representations*.
- Ho, J.; Jain, A.; and Abbeel, P. 2020. Denoising Diffusion Probabilistic Models. In *Advances in Neural Information Processing Systems*.
- Jiang, J.; Wang, F.; Shen, J.; Kim, S.; and Kim, S. 2024. A survey on large language models for code generation. *arXiv preprint arXiv:2406.00515*.
- Li, X.; Tang, R.; Pan, Y.; Cohn, T.; Goyal, A.; and Cheung, J. C. K. 2022a. Diffusion-LM Improves Controllable Text Generation. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, 7668–7681.
- Li, X. L.; Thickstun, J.; Gulrajani, I.; Liang, P.; and Hashimoto, T. B. 2022b. Diffusion-LM Improves Controllable Text Generation. In *Advances in Neural Information Processing Systems*.
- Li, Y.; Choi, D.; Chung, J.; Kushman, N.; Schrittwieser, J.; Leblond, R.; Eccles, T.; Keeling, J.; Gimeno, F.; Dal Lago, A.; et al. 2022c. Competition-Level Code Generation with AlphaCode. *Science*, 378(6615): 1092–1098.
- Lin, J.; Wong, A.; Xia, T.; He, S.; Wei, H.; Han, M.; and Luo, J. 2025. Facilitating Long Context Understanding via Supervised Chain-of-Thought Reasoning. In *Submitted*.
- Neamtii, I.; Foster, J. S.; and Hicks, M. 2005. Understanding source code evolution using abstract syntax tree matching. In *Proceedings of the 2005 International Workshop on Mining Software Repositories, MSR '05*, 1–5. New York, NY, USA: Association for Computing Machinery. ISBN 1595931236.
- Nie, Y.; Fan, A.; Grave, E.; and Weston, J. 2025. LLaDA: Diffusion Language Models are Large Language Models. *arXiv preprint arXiv:2503.00001*.
- OpenAI. 2022. Introducing ChatGPT. <https://openai.com/blog/chatgpt>. Accessed: 2025-07-24.
- Parr, T. 2011. *The Definitive ANTLR 4 Reference*. Pragmatic Bookshelf.
- Rabinovich, M.; Stern, M.; and Klein, D. 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.
- Rozière, B.; Gehring, J.; Gloeckle, F.; Sootla, S.; Gat, I.; Tan, X. E.; Adi, Y.; Liu, J.; Sauvestre, R.; Remez, T.; et al. 2023. Code Llama: Open Foundation Models for Code. *arXiv preprint arXiv:2308.12950*.
- Sahoo, R.; Hu, H.; Lee, H.; Cho, K.; and Zhang, H. 2024a. Masked Diffusion Language Models are Autoregressive. *arXiv preprint arXiv:2402.12892*.
- Sahoo, S. S.; Arriola, M.; Schiff, Y.; Gokaslan, A.; Marroquin, E.; Chiu, J. T.; Rush, A.; and Kuleshov, V. 2024b. Simple and Effective Masked Diffusion Language Models. *arXiv preprint arXiv:2406.07524*.
- Singh, S.; Chaudhary, V. M.; Liu, J.; Fan, A.; Weston, J.; and Grave, E. 2023. Retrieval-Augmented Diffusion Language Models. In *ICML Workshop on Retrieval-Augmented Generation*.

Song, Y.; Sohl-Dickstein, J.; Kingma, D. P.; Kumar, A.; Ermon, S.; and Poole, B. 2021. Score-Based Generative Modeling through Stochastic Differential Equations. *arXiv preprint arXiv:2011.13456*.

Team, G.; Anil, R.; et al. 2023. Gemini: A Family of Highly Capable Multimodal Models. *arXiv preprint arXiv:2312.11805*.

Yeo, E.; Tong, Y.; Niu, M.; Neubig, G.; and Yue, X. 2025. Demystifying long chain-of-thought reasoning in llms. *arXiv preprint arXiv:2502.03373*.

Yin, P.; and Neubig, G. 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.

Zhao, H.; Hui, J.; Howland, J.; Nguyen, N.; Zuo, S.; Hu, A.; Choquette-Choo, C. A.; Shen, J.; Kelley, J.; Bansal, K.; et al. 2024a. CodeGemma: Open Code Models Based on Gemma. *arXiv preprint arXiv:2406.11409*.

Zhao, Y.; Luo, Z.; Tian, Y.; Lin, H.; Yan, W.; Li, A.; and Ma, J. 2024b. CodeJudge-Eval: Can Large Language Models be Good Judges in Code Understanding? *arXiv preprint arXiv:2408.10718*.

Zhu, D.; Wei, X.; Zhao, G.; Wu, W.; Zou, H.; Ran, J.; Wang, X.; Sun, L.; Zhang, X.; and Li, S. 2025a. Chain-of-Thought Matters: Improving Long-Context Language Models with Reasoning Path Supervision. *arXiv preprint arXiv:2502.20790*.

Zhu, Y.; Li, G.; Jiang, X.; Li, J.; Mei, H.; Jin, Z.; and Dong, Y. 2025b. Uncertainty-Guided Chain-of-Thought for Code Generation with LLMs. *arXiv preprint arXiv:2503.15341*.