Constrained Code Generation with Diffusion Language Models

Lize Shao* **Michael Cardei*** **Zichen Xie** **Ferdinando Fioretto†** **Wenxi Wang†**

University of Virginia

{zgr3et, ntr2rm, graysonxie, fioretto, wenxiw}@virginia.edu

Abstract

Diffusion language models (DLMs) are a powerful, emerging paradigm for code generation. They construct programs through iterative refinement of partially corrupted token sequences and enable parallel token refinement. Importantly, this paradigm exposes a global program state at each denoising step, which provides a natural intervention point for enforcing program-level functionality and security constraints, guiding the generation before the final code is committed. Building on this observation, the paper introduces *Constrained Diffusion for Code* (CDC), an inference-time neurosymbolic framework that integrates constraint satisfaction directly into the reverse denoising process without retraining the base diffusion model. CDC augments the base diffusion sampler with constraint-aware denoising operators that combine mathematical optimization with program analysis to identify constraint-relevant regions of the intermediate program state and locally adjust the denoising trajectory, steering generation toward feasible programs while remaining close to the base model. Across code generation benchmarks, CDC consistently improves constraint satisfaction in functional correctness, security, and even syntax, outperforming diffusion and autoregressive baselines with less corrective computation and more localized edits.

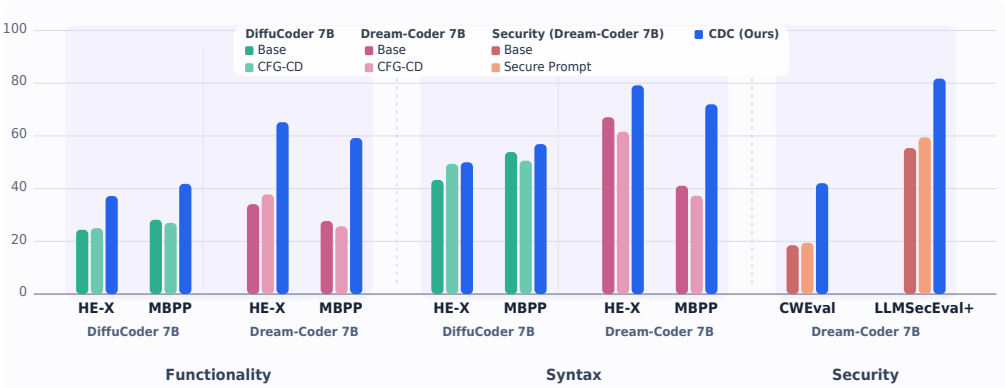

Figure 1: CDC vs. other diffusion code baselines on HumanEval-X (HE-X), MBPP, CWEval and LLMSecEval+.

1 Introduction

Code generation has become an increasingly important capability in modern generative modeling systems, with notable applications in program synthesis, developer assistance, and agentic software workflows [1, 2, 3]. In this setting, diffusion language models (DLMs) are emerging as a highly

*Equal contribution.

†Equal senior-author contribution.

40th Conference on Neural Information Processing Systems (NeurIPS 2026).

promising paradigm, constructing programs through iterative refinement of partially corrupted token sequences [4, 5]. However, the objective used to train generative models does not by itself capture the requirements that make generated code correct or secure. These models are trained to match a target distribution and produce high-fidelity samples, but a program may be highly plausible under the base model while still producing incorrect behavior, or containing an exploitable vulnerability. In practice, generated code must satisfy explicit program-level constraints, including syntactic validity and, more importantly, functional correctness and security properties. Since programs are executable objects, violations of these constraints can lead directly to runtime errors, incorrect outputs, or security failures [6, 7]. Incorporating such constraints is therefore critical for both code quality and safety.

These requirements expose a central challenge for code generation: *Program-level constraints often depend on global structure or semantic properties that are not visible from any single local token decision*. More broadly, common guidance mechanisms such as prompting, reranking, rejection sampling, or post-hoc repair typically intervene only after candidate programs have already been generated [8, 9]. In code generation, this is particularly limiting because the underlying search space is highly combinatorial, while the subset of programs satisfying semantic or security constraints may constitute only a small feasible region under the model distribution. Thus, post-hoc approaches can be both computationally inefficient and unreliable, especially when constraint satisfaction requires steering generation away from the model's dominant unconstrained modes and toward rarer but feasible solutions. In contrast, as this paper shows, diffusion language models offer a different opportunity: their approach to refine the entire sequence through intermediate denoising states exposes a global program representation throughout sampling. This structure creates natural intervention points for integrating constraints directly into the generated code during the denoising process.

Building on this opportunity, this paper introduces *Constrained Diffusion for Code* (CDC), an inference-time neurosymbolic framework that integrates program-level constraints (i.e., functionality and security) directly into the reverse denoising process of diffusion language models, without retraining the base model. CDC treats diffusion sampling as an editable program trajectory: at each denoising step, the model proposes a full clean-state distribution over the program; CDC decodes this proposal, evaluates the resulting candidate against program-level constraints, localizes feedback to relevant regions, and applies a constraint-aware correction before sampling continues. Each correction follows a localized constrained-sampling principle: reduce constraint violation while remaining close to the base denoiser, revising constraint-relevant regions without unnecessarily perturbing the rest of the program. The framework supports both soft and hard feedback. For functional correctness, CDC instantiates this principle as GradGuide, which uses soft surrogate execution signals to localize likely errors and guide targeted denoising updates. For security, CDC instantiates it as MDFI, which uses static program analysis to identify vulnerable regions and correct them through localized remasking, insertion, and feedback injection. In this way, CDC exploits the global, editable states exposed by diffusion sampling to perform focused constraint correction during generation, rather than relying on post-hoc filtering or whole-program regeneration after failure.

This work makes three main contributions:

1. **Constraint-aware localization.** It introduces a general localization mechanism for constrained diffusion code generation, mapping program-level feedback from surrogates or static analyzers to the token regions that should be revised during denoising.
2. **Constraint-aware correction.** It proposes inference-time correction operators that revise these localized regions during reverse diffusion while staying close to the base denoiser. The operators support differentiable surrogate guidance through gradient and KL-anchored updates, and non-differentiable symbolic analyzer feedback through targeted remasking, mask insertion, and feedback injection.
3. **Empirical evaluation.** It provides empirical evidence that denoising-time constraint integration improves constrained code generation across functional-correctness and security-oriented benchmarks. As summarized in Figure 1, CDC improves functionality from 34.1% to 65.2% on HumanEval-X C++ and from 27.7% to 59.2% on MBPP-C++ over Dream-Coder 7B, while MDFI improves CWEval joint functionality-security success from 12.0% to 34.3% over the state-of-the-art diffusion baseline with far few edited tokens. Although CDC targets functionality and security, it also improves syntactic correctness from 67.1% to 79.3% on HumanEval-X C++ and from 41.1% to 72.0% on MBPP-C++.

2 Related Work

Discrete diffusion language models generate sequences through iterative denoising rather than left-to-right decoding, exposing partially generated global states throughout sampling. Recent code-oriented diffusion models including DiffuCoder [4] and Dream-Coder 7B [5] show that this paradigm can achieve competitive code-generation performance while supporting flexible, non-autoregressive refinement. CDC builds on this property: rather than using diffusion only as an alternative decoder, it treats intermediate denoising states as editable program representations where constraints can be evaluated, localized, and incorporated before generation is complete.

Controllable code generation has been studied primarily in autoregressive models through constrained decoding, reranking, rejection sampling, verification, and iterative repair [10, 11, 9, 12, 13]. These methods can improve syntactic validity, type consistency, functional correctness, or security, but they typically operate either at the next-token level or after a complete candidate has been generated. This is limiting for program-level constraints: semantic and security violations often depend on global dataflow or execution behavior, and repairing them post hoc may require regenerating large suffixes or entire programs. CDC instead injects feedback during denoising, when the full program state is visible and localized regions can still be revised.

Recent work has explored controllable generation for discrete diffusion models through classifier guidance, classifier-free guidance, hidden-state optimization, and projection-based constrained sampling [14, 15, 16, 17]. Closest to our setting, Mündler et al. [18] enforce context-free grammar constraints during diffusion-LLM decoding, improving syntactic correctness for code. CDC targets a broader class of program-level constraints, including functional and security requirements that cannot generally be expressed as grammar membership or local token restrictions. Rather than applying global guidance or syntax-only filtering, CDC localizes surrogate or analyzer feedback to constraint-relevant code regions and revises those regions during the reverse diffusion trajectory.

3 Problem Setting

Code generation with target specifications. Let $\mathbf{x}_0 = (x_0^1, \ldots, x_0^L)$ denote a length-L sequence of discrete tokens, where each token takes values in a vocabulary $\mathcal{V}$. In code generation, $\mathcal{V}$ may include programming-language keywords, identifiers, literals, punctuation, operators, and special formatting tokens (e.g., `def`, `if`, `=`, `(`, `)`, `:`, variable names). A code generation model defines a conditional distribution over token sequences given a context c, where the functionality or security requirements may be specified through a problem description, function signature, partial program, unit tests, or other task-specific information. We refer to these requirements as the *target specification*; Figure 2 shows an example. The goal is therefore to generate a sequence $\mathbf{x}_0$ that is both plausible under the learned code distribution and valid with respect to the target specification.

Constrained sampling with hard and soft constraints. Let $p_\theta(\mathbf{x} \mid c)$ denote a base code generation model. Unconstrained decoding samples likely programs from p_θ; constrained decoding instead defines a sampler $p_\theta^{\mathcal{C}}(\mathbf{x} \mid c)$ that remains close to the base model while steering generation toward the feasible set $\mathcal{C}(c)$ specified by the task. We distinguish two kinds of constraints. *Hard constraints* are feasibility conditions that must hold exactly, such as parsing, compiling, passing required tests, or satisfying security checks. *Soft constraints* provide graded feedback before exact feasibility is reached, such as surrogate correctness scores, partial test-pass rates, or analyzer severity scores. CDC uses both: hard constraints define the target feasible set, while soft constraints provide useful denoising-time guidance toward it.

4 Preliminaries: Masked Diffusion Models

Masked diffusion language models generate a sequence by reversing a discrete noising process [19, 20, 21, 4]. Starting from a clean token sequence $\mathbf{x}_0$, the forward process gradually corrupts tokens into an absorbing mask token `[MASK]`; once a token is masked, it remains masked at later timesteps [20, 22]. For each position $i \in [L]$, the forward transition is

$$q(x_t^i \mid x_{t-1}^i) = \mathrm{Cat}\big(x_t^i;\ \alpha_{t|t-1} x_{t-1}^i + \big(1 - \alpha_{t|t-1}\big)\mathtt{M}\big)\,, \tag{1}$$

where $\mathrm{Cat}(\cdot; \pi)$ is a categorical distribution, $\mathtt{M}$ is the one-hot representation of `[MASK]`, and $\alpha_{t|t-1} := \alpha_t / \alpha_{t-1}$ for a decreasing noise schedule with $\alpha_0 = 1$ and $\alpha_T = 0$. Thus, an unmasked token is preserved with probability $\alpha_{t|t-1}$ and masked otherwise.

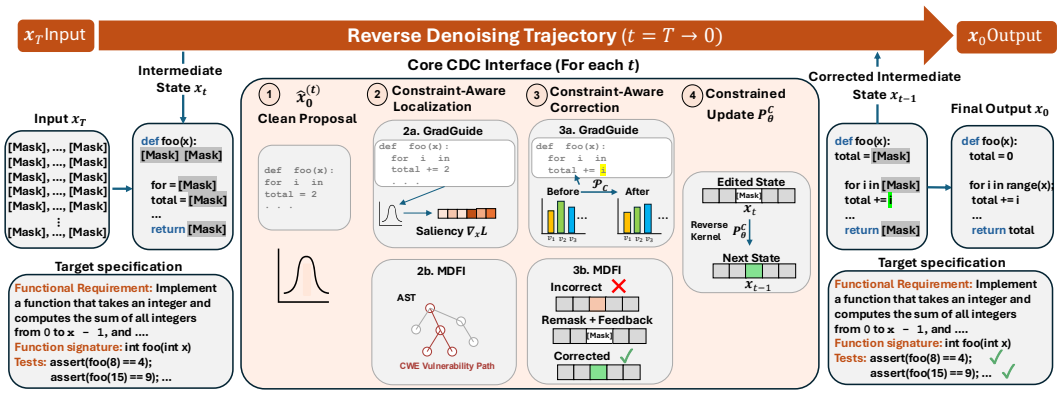

Figure 2: Overview of CDC.

The reverse process uses a neural denoiser to fill in masked positions. Given a partially masked sequence $\mathbf{x}_t$ at timestep t, the denoiser predicts a clean-token distribution $\hat{\mathbf{x}}_0^{(t)} = \mathbf{x}_\theta(\mathbf{x}_t, t)$. The reverse transition is

$$p_\theta(x_{t-1}^i \mid \mathbf{x}_t) = \begin{cases} \mathrm{Cat}(x_{t-1}^i; x_t^i), & \text{if } x_t^i \neq \texttt{[MASK]}, \\ \mathrm{Cat}\big(x_{t-1}^i;\ \gamma_t \mathtt{M} + \eta_t \mathbf{x}_\theta^i(\mathbf{x}_t, t)\big), & \text{if } x_t^i = \texttt{[MASK]}, \end{cases} \tag{2}$$

for each position $i \in [L]$ and timestep $t = T, \ldots, 1$, with $\gamma_t = \frac{1-\alpha_{t-1}}{1-\alpha_t}, \eta_t = \frac{\alpha_{t-1}-\alpha_t}{1-\alpha_t}$ balancing mask retention and sampling from the denoiser's clean-token prediction.

At inference time, sampling starts from the fully masked sequence $\mathbf{x}_T$ and repeatedly applies Eq. 2 until it obtains the generated sequence $\mathbf{x}_0$.

5 Constrained Diffusion for Code Generation

The formulation above motivates *Constrained Diffusion for Code* (CDC): at each timestep, the denoiser proposes a full clean program distribution $\hat{\mathbf{x}}_0^{(t)}$, creating a natural point to evaluate, localize, and incorporate program-level constraints during reverse denoising.

5.1 CDC Framework

CDC preserves the base diffusion model without additional training and formulates reverse diffusion as an editable trajectory that integrates base denoising with constraint-aware interventions. Starting from a fully masked program $\mathbf{x}_T = (\texttt{[MASK]}, \ldots, \texttt{[MASK]})$, sampling proceeds backward through timesteps $t = T, \ldots, 1$ until a clean program $\mathbf{x}_0$ is produced. The overview of CDC is shown in Figure 2. At each timestep, CDC performs four steps:

Step 1: Clean-state proposal. Given the current partially masked sequence $\mathbf{x}_t$, the base denoiser predicts a per-position categorical distribution over clean tokens,

$$\hat{\mathbf{x}}_0^{(t)} = \mathbf{x}_\theta(\mathbf{x}_t, t) \in \Delta^{L \times |\mathcal{V}|}. \tag{3}$$

This distribution represents the model's unconstrained clean-state proposal at timestep t, which can be decoded into an intermediate program candidate for constraint evaluation, for example by taking the per-position maximum,

$$\bar{x}_0^{(t),i} = \arg\max_{v \in \mathcal{V}} \hat{x}_0^{(t),i}(v), \qquad i \in [L]. \tag{4}$$

Step 2: Constraint-aware localization. A constraint evaluator $\mathcal{E}$ then evaluates this candidate $\bar{\mathbf{x}}_0^{(t)}$ under the task context c and constraint specification $\mathcal{C}$. The evaluator $\mathcal{E}$ may be instantiated by any constraint-scoring mechanism, such as a symbolic program analyzer, a verifier, or a learned surrogate model. The evaluator returns violation scores $\boldsymbol{\nu}_t$ and structured feedback r_t (e.g., a vulnerability report), which are mapped by a constraint-localization map $\mathcal{M}$ to editable regions $\mathcal{S}_t$:

$$(\boldsymbol{\nu}_t, r_t) = \mathcal{E}(\bar{\mathbf{x}}_0^{(t)}, c;\ \mathcal{C}), \qquad \mathcal{S}_t = \mathcal{M}(\bar{\mathbf{x}}_0^{(t)}, c, r_t;\ \mathcal{C}) \subseteq [L]. \tag{5}$$

The map $\mathcal{M}$ is induced by the form of feedback returned by $\mathcal{E}$, and translates evaluator feedback into token positions or spans that are responsible for the constraint violation. The set $\mathcal{S}_t$ identifies where the denoising trajectory should be revised, with specific instantiations given in Section 5.2. This separates the question of *where* to edit from the question of *how* to modify the denoising trajectory, which is the focus of the next step.

Step 3: Constraint-aware correction. Given the localized edit region $\mathcal{S}_t$ and feedback r_t, CDC applies a constraint-aware denoising operator $\mathcal{P}_{\mathcal{C}}$ to produce a corrected clean-state proposal:

$$\boldsymbol{y}_t = \mathcal{P}_{\mathcal{C}}(\hat{\mathbf{x}}_0^{(t)}, \mathcal{S}_t, r_t, c). \tag{6}$$

The operator is designed to reduce constraint violation while keeping $\boldsymbol{y}_t$ close to the base denoiser output $\hat{\mathbf{x}}_0^{(t)}$. CDC represents constraint feedback through violation penalties that may come from differentiable surrogate scores, verifier outputs, or security diagnostics. For constraint j at timestep t, let $\nu_{t,j}(\boldsymbol{y}; r_t, c, \mathcal{S}_t)$ denote the feedback-induced penalty under candidate proposal $\boldsymbol{y}$, and let $\lambda_j \geq 0$ denote its penalty coefficient. The resulting localized trust-region update is:

$$\boldsymbol{y}_t = \arg\min_{\boldsymbol{y} \in \Delta^{L \times |\mathcal{V}|}} \Big\{ D_{\mathrm{KL}}\Big(\boldsymbol{y} \,\|\, \hat{\mathbf{x}}_0^{(t)}\Big) + \underbrace{\sum_{j=1}^{m} \lambda_j \, \nu_{t,j}(\boldsymbol{y}; r_t, c, \mathcal{S}_t)}_{V_t(\boldsymbol{y}; r_t, c, \mathcal{S}_t)} \Big\}. \tag{7}$$

Here, D_{KL} denotes Kullback–Leibler divergence, which anchors the correction to the base proposal, and V_t is the aggregate violation penalty. The localization set $\mathcal{S}_t$ scopes the correction: some operators enforce locality by anchoring positions outside $\mathcal{S}_t$, while others use $\mathcal{S}_t$ as a remasking target or feedback anchor. Section 5.2 instantiates this interface for differentiable surrogate feedback for program functionality, and symbolic program-analysis feedback for program security.

Step 4: Constrained reverse update. Finally, CDC advances the reverse chain using the corrected proposal $\boldsymbol{y}_t$ through a constraint-aware reverse kernel,

$$\mathbf{x}_{t-1} \sim p_\theta^{\mathcal{C}}(\cdot \mid \mathbf{x}_t, \boldsymbol{y}_t). \tag{8}$$

A representative transition replaces the base clean-token distribution prediction with the corrected proposal:

$$p_\theta^{\mathcal{C}}(x_{t-1}^i \mid \mathbf{x}_t, \boldsymbol{y}_t) = \begin{cases} \mathrm{Cat}(x_{t-1}^i; x_t^i), & \text{if } x_t^i \neq \texttt{[MASK]}, \\ \mathrm{Cat}\big(x_{t-1}^i;\ \gamma_t \mathtt{M} + \eta_t \boldsymbol{y}_t^i\big), & \text{if } x_t^i = \texttt{[MASK]}. \end{cases} \tag{9}$$

Here, $\boldsymbol{y}_t^i$ is the corrected clean-token distribution at position i. Depending on the operator, positions outside $\mathcal{S}_t$ may either remain anchored to the base proposal or receive weaker global adjustments, while positions inside $\mathcal{S}_t$ receive the strongest constraint-aware corrections through modified token distributions, targeted remasking, or feedback-conditioned revision, as described in Section 5.2. The resulting reverse trajectory is illustrated below.

$$\mathbf{x}_t \xrightarrow{\mathbf{x}_\theta(\cdot,t)} \hat{\mathbf{x}}_0^{(t)} \xrightarrow{\text{Decode}} \bar{\mathbf{x}}_0^{(t)} \xrightarrow{\mathcal{E},\mathcal{M}} (\boldsymbol{\nu}_t, r_t, \mathcal{S}_t) \xrightarrow{\mathcal{P}_{\mathcal{C}}} \boldsymbol{y}_t \xrightarrow{p_\theta^{\mathcal{C}}} \mathbf{x}_{t-1}.$$

5.2 Operator Instantiations

We now show how CDC becomes concrete by instantiating the three components of the framework: evaluator $\mathcal{E}$, localization map $\mathcal{M}$, and correction operator $\mathcal{P}_{\mathcal{C}}$. Following the problem setting (Section 3), hard constraints define feasibility, while soft constraints provide graded signals that can guide the sampler before feasibility is reached. Our first instantiation, GradGuide, uses this soft-feedback route for program functionality: a learned surrogate relaxes hard execution outcomes into soft scores, localizes likely functional errors with gradients, and corrects the localized region through KL-anchored proposal updates and remasking. Our second instantiation, Mid-Diffusion Feedback Injection (MDFI), uses the hard-feedback route for program security: a static analyzer reports concrete vulnerability witnesses on partial programs, localizes them on a program graph, and corrects the corresponding regions through targeted remasking and feedback injection.

Surrogate-gradient operator: GradGuide. Functional correctness is ultimately a hard constraint: the final program must pass the required tests. During denoising, however, this hard constraint is not

differentiable, so GradGuide instantiates the evaluator $\mathcal{E}^{\mathrm{GG}}$ with an auxiliary surrogate g_ϕ trained ahead of time with execution-driven labels (e.g., test-pass outcomes), while keeping the diffusion model parameters θ fixed. Given the clean-state proposal $\hat{\mathbf{x}}_0^{(t)}$, GradGuide maps each predicted token distribution to an expected embedding,

$$\mathrm{Emb}(\hat{\mathbf{x}}_0^{(t)})^i = \sum_{v\in\mathcal{V}} \hat{x}_0^{(t),i}(v)\mathbf{e}_v = \hat{x}_0^{(t),i}\mathbf{E}_{\mathrm{tok}}, \qquad i \in [L], \tag{10}$$

where $\mathbf{E}_{\mathrm{tok}}$ is the token-embedding matrix, and $\mathbf{e}_v$ denotes the embedding of token v.

The surrogate predicts a satisfaction score $\nu_{\phi,j}^{\mathrm{GG}}(\mathrm{Emb}(\hat{\mathbf{x}}_0^{(t)}), c)$ for each constraint j, which is converted into the relaxed violation

$$\Delta\nu_j^{\mathrm{GG}}(\hat{\mathbf{x}}_0^{(t)}, c) = \max\Big(0,\, \tau_j - \nu_{\phi,j}^{\mathrm{GG}}(\mathrm{Emb}(\hat{\mathbf{x}}_0^{(t)}), c)\Big), \qquad V_t^{\mathrm{GG}} = \sum_{j=1}^{m} \Delta\nu_j^{\mathrm{GG}}. \tag{11}$$

GradGuide localizes edits with one backward pass through g_ϕ. The saliency of position i is

$$a_i = \Big\|\nabla_{\mathrm{Emb}(\hat{\mathbf{x}}_0^{(t)})^i} V_t^{\mathrm{GG}}(\hat{\mathbf{x}}_0^{(t)}, c)\Big\|_2, \qquad \mathcal{S}_t^{\mathrm{GG}} = \mathrm{Expand}\Big(\text{Top-}k(a_1,\ldots,a_L)\Big), \tag{12}$$

where $\mathrm{Expand}(\cdot)$ maps selected token positions to syntactically coherent edit spans by enlarging each position to a local syntactic window. Thus, $\mathcal{S}_t^{\mathrm{GG}}$ contains the syntactically expanded token spans that most influence the surrogate-predicted violation.

Given $\mathcal{S}_t^{\mathrm{GG}}$, GradGuide computes a corrected proposal by approximately minimizing Eq. 7 with V_t replaced by V_t^{GG}.

The resulting $\boldsymbol{y}_t$ is used in the constrained reverse kernel. When the intermediate program is sufficiently decoded but still violates the hard functional constraint, GradGuide also reopens the localized region by setting $x_t^i = \texttt{[MASK]}$ for $i \in \mathcal{S}_t^{\mathrm{GG}}$, allowing the remaining denoising steps to regenerate those positions under the corrected proposal. Additional details, including surrogate training and other implementation details, are provided in Appendix C.

Program-analysis-guided operator: MDFI. Security requirements are hard constraints: a program is feasible only if the relevant vulnerability checks pass. For these constraints, we use a static program analyzer to instantiate the evaluator $\mathcal{E}$, which is applied to the intermediate proposal $\bar{\mathbf{x}}_0^{(t)}$ at selected denoising checkpoints. The analyzer constructs a partial program graph $\mathcal{G}_t$ from the partial program structure and returns vulnerability witnesses

$$w_k = (n_k, \tau_k, h_k), \tag{13}$$

where n_k is the offending graph node, $\tau_k \in \{\mathrm{sub}, \mathrm{ins}\}$ indicates whether the repair is substitution- or insertion-like, and h_k is a remediation hint. The witness set $\mathcal{W}_t = \{w_k\}_{k=1}^{K_t}$ defines the structured feedback r_t^{MDFI} and the corresponding hard violation vector $\boldsymbol{\nu}_t^{\mathrm{MDFI}}$.

MDFI localizes each witness by taking its analyzer-supported abstract syntax tree and dataflow neighborhood and projecting it back to token positions:

$$\mathcal{S}_t^{\mathrm{MDFI}} = \mathrm{TopBudget}_B\left(\bigcup_{k=1}^{K_t} \mathrm{Tok}\big(\mathcal{N}(n_k; \mathcal{G}_t)\big)\right) \subseteq [L]. \tag{14}$$

Here, $\mathrm{TopBudget}_B(\cdot)$ selects top token spans under a token budget B, prioritizing higher-confidence analyzer witnesses, while $\mathcal{N}(n_k; \mathcal{G}_t)$ denotes the relevant syntactic or dataflow neighborhood around the witness, and $\mathrm{Tok}(\cdot)$ maps graph nodes back to token spans. Thus, $\mathcal{S}_t^{\mathrm{MDFI}}$ contains diagnostic-supported regions.

Given $\mathcal{S}_t^{\mathrm{MDFI}}$ and r_t^{MDFI}, MDFI applies three discrete interventions. Let $\mathcal{S}_t^{\mathrm{sub}}$ and $\mathcal{S}_t^{\mathrm{ins}}$ denote the portions of $\mathcal{S}_t^{\mathrm{MDFI}}$ supported by substitution-type and insertion-type witnesses, respectively:

$$\mathcal{S}_t^{a} = \mathcal{S}_t^{\mathrm{MDFI}} \cap \bigcup_{k:\tau_k=a} \mathrm{Tok}(\mathcal{N}(n_k; \mathcal{G}_t)), \qquad a \in \{\mathrm{sub}, \mathrm{ins}\}. \tag{15}$$

For substitution-type witnesses, MDFI remasks the offending committed region:

$$x_t^{\star,i} = \begin{cases} \texttt{[MASK]}, & i \in \mathcal{S}_t^{\mathrm{sub}}, \\ x_t^i, & i \notin \mathcal{S}_t^{\mathrm{sub}}. \end{cases} \tag{16}$$

Table 1: Functional correctness (%) on HumanEval-X C++ and MBPP-C++. Parenthesized values report the absolute change from the Vanilla model: green indicates an improvement of $\geq$10%, yellow an improvement of $<$10%, and red a regression. Within each model, the best performance is **bolded**.

		HumanEval-X C++			MBPP-C++		
Model	**Method**	Syn. (compile)	Fun. (p@1)	Fun. (p@10)	Syn. (compile)	Fun. (p@1)	Fun. (p@10)
Dream 7B	Vanilla	40.2	10.4	51.8	60.7	25.4	59.7
	CFG-CD	**70.7** (+30.5)	11.0 (+0.6)	60.4 (+8.6)	59.4 (-1.3)	24.4 (-1.0)	55.9 (-3.8)
	CDC (Ours)	44.5 (+4.3)	**20.7** (+10.3)	**64.6** (+12.8)	**61.2** (+0.5)	**33.8** (+8.4)	**63.7** (+4.0)
DiffuCoder 7B	Vanilla	43.3	24.4	56.1	53.9	28.2	53.1
	CFG-CD	49.4 (+6.1)	25.0 (+0.6)	50.0 (-6.1)	50.6 (-3.3)	27.0 (-1.2)	51.1 (-2.0)
	CDC (Ours)	**50.0** (+6.7)	**37.2** (+12.8)	**68.3** (+12.2)	**56.9** (+3.0)	**41.8** (+13.6)	**68.0** (+14.9)
Dream-Coder 7B	Vanilla	67.1	34.1	55.5	41.1	27.7	54.2
	CFG-CD	61.6 (-5.5)	37.8 (+3.7)	53.7 (-1.8)	37.3 (-3.8)	25.7 (-2.0)	49.1 (-5.1)
	CDC (Ours)	**79.3** (+12.2)	**65.2** (+31.1)	**83.5** (+28.0)	**72.0** (+30.9)	**59.2** (+31.5)	**74.8** (+20.6)
DeepSeek-Coder 6.7B	Vanilla	**94.5**	59.8	80.5	88.2	59.7	73.0
	Reprompt	90.9 (-3.6)	**64.0** (+4.2)	**82.9** (+2.4)	**89.2** (+1.0)	**60.7** (+1.0)	**74.1** (+1.1)
CodeLlama 7B	Vanilla	**90.2**	30.5	54.9	**83.9**	46.1	61.7
	Reprompt	81.7 (-8.5)	**31.1** (+0.6)	**55.5** (+0.6)	83.6 (-0.3)	**47.4** (+1.3)	**64.7** (+3.0)

For insertion-type witnesses, MDFI inserts K fresh `[MASK]` tokens near a structural anchor selected from $\mathcal{S}_t^{\text{ins}}$:

$$\mathbf{x}_t^\star \leftarrow \text{Insert}_K\big(\mathbf{x}_t^\star, \text{anchor}(w_k; \mathcal{S}_t^{\text{ins}})\big), \qquad k : \tau_k = \text{ins}, \tag{17}$$

where $\text{Insert}_K(x, a)$ inserts K `[MASK]` tokens into sequence x at anchor position a. The anchor function selects a syntactically valid insertion point near the insertion-type witness.

The MDFI update is therefore

$$\big(\mathbf{x}_t^\star,\, c^\star,\, \mathcal{S}_t^{\text{MDFI}}\big) = \mathcal{P}_{\mathcal{C}}^{\text{MDFI}}\big(\hat{\mathbf{x}}_0^{(t)},\, \mathcal{S}_t^{\text{MDFI}},\, r_t^{\text{MDFI}},\, c\big). \tag{18}$$

Here, $\mathcal{P}_{\mathcal{C}}^{\text{MDFI}}$ is the MDFI instantiation of the constraint-aware projection operator defined in Eq. 6 and operationalized by the objective in Eq. 7. The chain continues with $\mathbf{x}_t$ replaced by $\mathbf{x}_t^\star$ and c replaced by $c^\star$. MDFI does not update θ or run an inner re-denoising loop; it edits the current state/context in place and lets the remaining reverse steps regenerate the affected regions. If no checkpoint fires or no witness is detected, MDFI reduces to the identity operator. Additional details are in Appendix D.

6 Experiments

This section evaluates CDC for constrained code generation under four objectives: functional correctness, syntactic validity, security, and edit localization. For functional correctness and syntactic validity, we instantiate CDC with GradGuide. For security, we instantiate CDC with MDFI. We evaluate edit localization for both instantiations to measure whether CDC concentrates corrections on constraint-relevant regions. Additional experimental details and results are provided in Appendix B.

6.1 Code Functionality and Syntax

We first evaluate CDC on the core requirements of code generation: producing programs that compile (syntactic validity) and implement the intended behavior (functional correctness).

Baselines. We evaluate CDC on three diffusion language models: Dream-Coder-7B [5], DiffuCoder-7B [4], and Dream-7B [19]. We compare against two diffusion baselines: Vanilla, which uses the base denoising sampler without constraint intervention, and CFG-CD [18], which uses grammar-constrained diffusion decoding for syntax-level constraints. We also compare against two autoregressive code models, DeepSeek-Coder-Instruct-6.7B [23] and CodeLlama-7B [24], under vanilla and reprompt configurations. Reprompt first performs vanilla generation; if the output fails the constraint, test pass/fail feedback is used to launch one additional generation round. **Benchmarks.** We evaluate C++ program synthesis on HumanEval-X C++ (164 tasks) and MBPP-C++ (397 tasks) [25, 26, 27]. **Evaluation Metrics.** Functionality is measured by pass@1 and pass@10: a task is solved if at least one of 1 or 10 generated samples compiles and passes all tests. Syntax is the compile-success rate.

Table 1 reports syntax and functional correctness on HumanEval-X C++ and MBPP-C++. Across diffusion backbones, CDC improves both pass@1 and pass@10 over vanilla diffusion, with the

Table 2: Security-aware code generation on CWEval and LLMSecEval+. Parenthesized values give the absolute change relative to the vanilla model: green marks gains of $\geq$10%, yellow gains of $<$10%, and red regressions. Within each model, the best performance is **bolded**.

		CWEval					LLMSecEval+				
Model	**Method**	func@1	sec@1	fs@1	fs@5	fs@10	func@1	sec@1	fs@1	fs@5	fs@10
Diffusion	Vanilla	26.9	18.5	12.0	25.9	26.9	32.0	54.7	14.7	22.0	24.0
	Sec prompt	26.9 (+0)	19.4 (+1)	13.9 (+2)	23.2 (-3)	27.8 (+1)	**34.7** (+3)	58.7 (+4)	16.0 (+1)	22.7 (+1)	26.7 (+3)
	CDC	**39.8** (+13)	**41.7** (+23)	**34.3** (+22)	**49.1** (+23)	**51.9** (+25)	30.7 (-1)	**80.7** (+26)	**24.7** (+10)	**40.7** (+19)	**45.3** (+21)
AR	Vanilla	**48.2**	24.1	21.3	36.1	40.7	37.3	55.3	21.3	30.7	38.0
	Reprompt	44.4 (-4)	**47.2** (+23)	**39.8** (+19)	**63.0** (+27)	**68.5** (+28)	**40.0** (+3)	**64.7** (+9)	**26.0** (+5)	**40.0** (+9)	**48.7** (+11)

largest gains on stronger code-diffusion models. On HumanEval-X C++, CDC improves Dream-Coder 7B from 34.1% to 65.2% pass@1, while also increasing compile success from 67.1% to 79.3%. On MBPP-C++, the same model improves from 27.7% to 59.2% pass@1, with compile success increasing from 41.1% to 72.0%. CDC also improves weaker diffusion backbones, increasing Dream 7B from 10.4% to 20.7% on HumanEval-X and DiffuCoder 7B from 24.4% to 37.2%. The improvements are consistent at pass@10.

While CFG-CD improves grammar or compile validity in several cases, it does not consistently improve functional correctness. For example, on MBPP-C++, CFG-CD decreases Dream-Coder 7B pass@1 from 27.7% to 25.7%, whereas CDC increases it to 59.2%. This difference suggests that syntax constraints alone are insufficient for program synthesis: CDC improves both validity and test-pass rates because its surrogate signal is trained from execution outcomes rather than grammar membership alone.

Compared with autoregressive baselines, CDC substantially narrows the functionality gap while retaining the advantages of diffusion-style correction. On HumanEval-X C++, CDC with Dream-Coder 7B achieves 65.2% pass@1 and 83.5% pass@10, outperforming all 7B autoregressive baselines, including DeepSeek-Coder with re-prompting at 64.0% pass@1 and 82.9% pass@10. On MBPP-C++, CDC reaches 59.2% pass@1 and 74.8% pass@10, approaching DeepSeek-Coder and substantially outperforming CodeLlama. These results suggest that denoising-time constraint integration can make diffusion code models competitive with strong autoregressive code generators, without relying on full-program re-generation after feedback. Additional regression analysis and ablations are provided in Appendix B.3.

6.2 Code Generation Security

We next evaluate whether CDC can improve security-oriented code generation, where outputs are evaluated to satisfy both security and functionality constraints. This setting is challenging as security fixes often require localized semantic changes while preserving the surrounding program behavior.

Baselines. We use Dream-Coder-7B as the diffusion backbone for CDC and DeepSeek-Coder-Instruct-6.7B as the autoregressive baseline. For autoregressive models, we compare vanilla generation with AR re-prompting [9], where an insecure or failing solution is returned to the model with feedback from a static program analyzer and the model regenerates a complete program. For diffusion models, we compare against vanilla diffusion and security prompting. **Benchmarks.** We evaluate CDC on CWEval and LLMSecEval+ [28, 29] (see Appendix B.1). **Metrics.** We report func@1 and sec@1, the fractions of single samples that pass the functional and security oracles, respectively, and joint func-sec@k (fs@k) for $k \in \{1, 5, 10\}$, where a task is solved if any of k sampled outputs is both functional and secure.

Table 2 shows that CDC substantially improves security-aware generation over vanilla diffusion and secure prompting. On CWEval, CDC increases fs@1 from 12.0% to 34.3%, nearly tripling the joint functionality-security success rate, and lifts both component metrics in tandem (func@1 26.9%$\rightarrow$39.8%, sec@1 18.5%$\rightarrow$41.7%). On LLMSecEval+, CDC improves security most strongly, increasing sec@1 from 54.7% for vanilla diffusion and 58.7% with secure prompting to 80.7%—higher than every other method in the table, including AR re-prompting (64.7%)—while also improving fs@1 from 14.7% to 24.7%. The improvement extends to multi-sample regimes: CDC raises CWEval fs@5/fs@10 from 25.9%/26.9% to 49.1%/51.9%, and matches AR re-prompting on LLMSecEval+ fs@5 (40.7% vs. 40.0%).

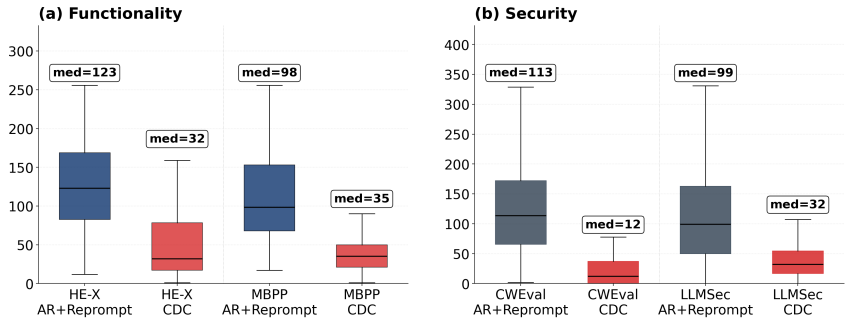

Figure 3: Edited tokens per correction attempt (fewer means more localized edits): (a) functionality corrections on HumanEval-X and MBPP, and (b) security corrections on CWEval and LLMSecEval+.

Compared with AR re-prompting, CDC is competitive despite using a weaker diffusion backbone for functionality. AR re-prompting obtains the highest overall fs@1 on both benchmarks, but CDC closes much of the gap while performing interventions during generation rather than regenerating complete programs after failure. The per-language CWEval results (Table 5, Appendix B.3) show that CDC is especially strong on Python, where it reaches 72.0% fs@1, and remains competitive on JavaScript, C++, and Go, suggesting that localized mid-diffusion repair can improve security without relying only on post-hoc complete-program correction. Additional efficiency analysis, regression analysis and ablations are provided in Appendix B.3.

6.3 Edit Locality

In addition to task success, we evaluate whether CDC performs targeted constraint correction rather than broad regeneration. We measure edit locality across the same functional-correctness and security benchmarks described above. For each intervention, we compare the constrained output to the corresponding unconstrained/base output and report edit locality statistics, including the fraction of changed tokens, number of changed spans or edit clusters, and tokens inserted or remasked. We also measure corrective cost, including the number of additional denoising interventions, remasked tokens, and generated tokens. These metrics are used to assess whether CDC improves constraint satisfaction while concentrating changes on constraint-relevant regions and avoiding repeated full-program regeneration.

Figure 3 reports the number of edited tokens for security and functionality corrections, where fewer edited tokens indicate a more targeted intervention. For security, CDC is substantially more local than AR re-prompting on both benchmarks: AR re-prompting rewrites a median of 113 tokens on CWEval and 99 tokens on LLMSecEval+, while CDC edits only 12 and 32 tokens, respectively. This supports the main motivation of CDC: because diffusion states remain editable during generation, analyzer feedback can be applied directly to the vulnerable region without regenerating the full program.

For functionality, CDC also produces more localized corrections than AR re-prompting. On HumanEval-X C++, CDC edits a median of 32 tokens compared with 123 tokens for AR re-prompting. On MBPP-C++, CDC edits a median of 35 tokens compared with 98 tokens for AR re-prompting. Thus, across both security and functionality settings, CDC reduces the amount of code rewritten during correction, suggesting that its gains come from focused denoising-time interventions rather than full-program repair. Additional locality analysis and ablations are included in Appendix B.3.

6.4 Inference Cost and Surrogate Reliability

We also evaluate the end-to-end inference cost of CDC. Table 3 reports per-sample latency and peak GPU memory, with each method using the same backbone, decoding configuration, hardware, and 256-step schedule as its vanilla baseline (Appendix B.3.2). GradGuide intervenes on 66–84% of samples at 48–53 ms per surrogate pass, raising mean latency to 1.72–2.01$\times$ that of vanilla decoding and peak memory by 53–61%; the median latency rises only to 1.41–1.47$\times$, as the cost concentrates in trajectories that require repeated correction. In return, GradGuide improves pass@1 by 31.1 and 31.5 points on HumanEval-X C++ and MBPP-C++, compared with 3.1 and 3.5 points for Best-of-N sampling ($N = 2$) and 4.2 and 1.0 points for AR re-prompting. MDFI adds only 2.3–6.1% latency from seven CPU-based analyzer calls (46 ms in total) and achieves 2.79$\times$ and 1.58$\times$ the successful-output throughput of vanilla decoding on CWEval and LLMSecEval+.

Table 3: Inference cost of CDC versus vanilla decoding (Van.) under identical settings. Gray values give the ratio to vanilla; memory is the increase in peak GPU memory.

	Mean latency (s)		Median latency (s)		Peak mem.	Success (%)	
Setting	Van.	CDC	Van.	CDC	Δ	Van.	CDC
GradGuide on HumanEval-X C++ (pass@1)							
Dream-Coder 7B	4.23	7.99 (1.89×)	4.26	6.10 (1.43×)	+53.0%	34.1	**65.2** (+31.1)
Dream 7B	4.18	7.20 (1.72×)	4.21	5.93 (1.41×)	+57.8%	10.4	**20.7** (+10.3)
DiffuCoder 7B	4.16	8.37 (2.01×)	4.19	6.16 (1.47×)	+60.5%	24.4	**37.2** (+12.8)
MDFI with Dream-Coder 7B (fs@1)							
CWEval	12.98	13.28 (1.02×)	12.85	12.97 (1.01×)	+4.2%	12.0	**34.3** (+22.3)
LLMSecEval+	10.12	10.73 (1.06×)	9.62	11.00 (1.14×)	+2.6%	14.7	**24.7** (+10.0)

Given the additional inference cost of GradGuide, we next examine the reliability of the surrogate that guides its corrections. The surrogate is trained once on CodeContests problems disjoint from both benchmarks, with a reported training cost of 1.8 GPU-hours. It accurately predicts execution outcomes on HumanEval-X C++ and MBPP-C++ (Spearman ρ of 0.848 and 0.837, ECE of 0.024 and 0.030), whereas substituting the non-fine-tuned backbone (ρ of 0.056 and 0.117) lowers pass@1 by 8.5 and 5.3 points relative to vanilla decoding. These results highlight the importance of surrogate quality in achieving effective guidance (Appendix C.2).

7 Limitations

CDC depends on the quality and coverage of the feedback mechanisms used during denoising. Specifically, GradGuide relies on a learned surrogate, which may provide noisy or misleading estimates of functional correctness, while MDFI relies on static analysis, which may miss vulnerabilities or produce false positives. The empirical evaluation is also limited to a fixed set of benchmarks, languages, and diffusion backbones, so performance may differ for larger codebases, longer-context tasks, or more complex software-engineering workflows. Finally, although CDC performs more localized edits than post-hoc regeneration, it still adds inference-time overhead from surrogate evaluation, static analysis, and constraint-aware correction.

8 Conclusion

This paper introduced CDC, an inference-time neurosymbolic framework that integrates program-level constraints into the reverse denoising process of diffusion language models without retraining the base model. CDC uses the global, editable states exposed during sampling to evaluate intermediate programs, localize constraint feedback, and apply targeted corrections that reduce violation while staying close to the base denoiser. The framework supports both soft and hard feedback: GradGuide uses soft surrogate signals for functional correctness, while MDFI uses hard symbolic program-analysis signals for security repair through remasking, insertion, and feedback injection. Across functional and security-oriented benchmarks, CDC improves test pass rates, compile success, and joint functionality-security success over standard diffusion decoding and syntax-only constrained diffusion. These results show that discrete diffusion language models are a natural substrate for constrained code generation, enabling focused correction during generation instead of post-hoc filtering or whole-program regeneration.

Broader Impact

This work aims to improve the reliability and security of code generation by steering diffusion models toward programs that better satisfy functional and security constraints. Potential benefits include fewer syntax errors, higher task correctness, and reduced vulnerabilities in generated code. However, stronger code generation is dual-use and could be misused if constraint mechanisms are adapted toward harmful objectives.

Acknowledgments and Disclosure of Funding

This research is partially supported by NSF awards 2533631, 2401285, 2334936, and 2226816, and by DARPA under Contract No. #HR0011252E005. The authors acknowledge the Research Computing at the University of Virginia. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF or DARPA. This material is based upon work supported by the National Science Foundation Graduate Research Fellowship Program under Grant No. 2234693. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

Competing Interests. The authors declare no competing financial interests relevant to this work. All research funding and computational support are disclosed above.

References

[1] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. Competition-level code generation with AlphaCode. *Science*, 378(6624):1092–1097, 2022. doi: 10.1126/science.abq1158. URL https://doi.org/10.1126/science.abq1158.

[2] Fengji Zhang, Bei Chen, Yue Zhang, Jacky Keung, Jin Liu, Daoguang Zan, Yi Mao, Jian-Guang Lou, and Weizhu Chen. Repocoder: Repository-level code completion through iterative retrieval and generation, 2023. URL https://arxiv.org/abs/2303.12570.

[3] John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. Swe-agent: Agent-computer interfaces enable automated software engineering, 2024. URL https://arxiv.org/abs/2405.15793.

[4] Shansan Gong, Ruixiang Zhang, Huangjie Zheng, Jiatao Gu, Navdeep Jaitly, Lingpeng Kong, and Yizhe Zhang. DiffuCoder: Understanding and improving masked diffusion models for code generation, 2025. URL https://arxiv.org/abs/2506.20639.

[5] Zhihui Xie, Jiacheng Ye, Lin Zheng, Jiahui Gao, Jingwei Dong, Zirui Wu, Xueliang Zhao, Shansan Gong, Xin Jiang, Zhenguo Li, and Lingpeng Kong. Dream-Coder 7b: An open diffusion language model for code, 2025. URL https://arxiv.org/abs/2509.01142.

[6] Hammond Pearce, Baleegh Ahmad, Benjamin Tan, Brendan Dolan-Gavitt, and Ramesh Karri. Asleep at the keyboard? assessing the security of github copilot's code contributions, 2021. URL https://arxiv.org/abs/2108.09293.

[7] Enna Basic and Alberto Giaretta. From vulnerabilities to remediation: A systematic literature review of llms in code security, 2024. URL https://arxiv.org/abs/2412.15004.

[8] Catherine Tony, Nicolás E. Díaz Ferreyra, Markus Mutas, Salem Dhiff, and Riccardo Scandariato. Prompting techniques for secure code generation: A systematic investigation, 2025. URL https://arxiv.org/abs/2407.07064.

[9] Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. Teaching large language models to self-debug, 2023. URL https://arxiv.org/abs/2304.05128.

[10] J. Edward Hu, Huda Khayrallah, Ryan Culkin, Patrick Xia, Tongfei Chen, Matt Post, and Benjamin Van Durme. Improved lexically constrained decoding for translation and monolingual rewriting. In Jill Burstein, Christy Doran, and Thamar Solorio, editors, *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 839–850, Minneapolis, Minnesota, June 2019. Association for Computational Linguistics. doi: 10.18653/v1/N19-1090. URL https://aclanthology.org/N19-1090/.

[11] Ansong Ni, Srini Iyer, Dragomir Radev, Ves Stoyanov, Wen tau Yih, Sida I. Wang, and Xi Victoria Lin. Lever: Learning to verify language-to-code generation with execution, 2023. URL https://arxiv.org/abs/2302.08468.

[12] Niels Mündler, Jingxuan He, Hao Wang, Koushik Sen, Dawn Song, and Martin Vechev. Type-constrained code generation with language models. *Proceedings of the ACM on Programming Languages*, 9(PLDI):601–626, June 2025. ISSN 2475-1421. doi: 10.1145/3729274. URL http://dx.doi.org/10.1145/3729274.

[13] Manish Bhatt, Sahana Chennabasappa, Yue Li, Cyrus Nikolaidis, Daniel Song, Shengye Wan, Faizan Ahmad, Cornelius Aschermann, Yaohui Chen, Dhaval Kapil, David Molnar, Spencer Whitman, and Joshua Saxe. Cyberseceval 2: A wide-ranging cybersecurity evaluation suite for large language models, 2024. URL https://arxiv.org/abs/2404.13161.

[14] Yair Schiff, Subham Sekhar Sahoo, Hao Phung, Guanghan Wang, Sam Boshar, Hugo Dallatorre, Bernardo P. de Almeida, Alexander Rush, Thomas Pierrot, and Volodymyr Kuleshov. Simple guidance mechanisms for discrete diffusion models, 2025. URL https://arxiv.org/abs/2412.10193.

[15] Nate Gruver, Samuel Stanton, Nathan C. Frey, Tim G. J. Rudner, Isidro Hotzel, Julien Lafrance-Vanasse, Arvind Rajpal, Kyunghyun Cho, and Andrew Gordon Wilson. Protein design with guided discrete diffusion, 2023. URL https://arxiv.org/abs/2305.20009.

[16] Michael Cardei, Jacob K Christopher, Thomas Hartvigsen, Bhavya Kailkhura, and Ferdinando Fioretto. Constrained discrete diffusion, 2025. URL https://arxiv.org/abs/2503.09790.

[17] Jacob K. Christopher, Michael Cardei, Jinhao Liang, and Ferdinando Fioretto. Neuro-symbolic generative diffusion models for physically grounded, robust, and safe generation, 2025. URL https://arxiv.org/abs/2506.01121.

[18] Niels Mündler, Jasper Dekoninck, and Martin Vechev. Constrained decoding of diffusion LLMs with context-free grammars, 2025. URL https://arxiv.org/abs/2508.10111.

[19] Jiacheng Ye, Zhihui Xie, Lin Zheng, Jiahui Gao, Zirui Wu, Xin Jiang, Zhenguo Li, and Lingpeng Kong. Dream 7b: Diffusion large language models, 2025. URL https://arxiv.org/abs/2508.15487.

[20] Subham Sekhar Sahoo, Marianne Arriola, Yair Schiff, Aaron Gokaslan, Edgar Marroquin, Justin T Chiu, Alexander Rush, and Volodymyr Kuleshov. Simple and effective masked diffusion language models, 2024. URL https://arxiv.org/abs/2406.07524.

[21] Jiaxin Shi, Kehang Han, Zhe Wang, Arnaud Doucet, and Michalis Titsias. Simplified and generalized masked diffusion for discrete data. In *Advances in Neural Information Processing Systems*, volume 37, pages 103131–103167. Curran Associates, Inc., 2024. doi: 10.52202/079017-3277. URL https://proceedings.neurips.cc/paper_files/paper/2024/file/bad233b9849f019aead5e5cc60cef70f-Paper-Conference.pdf.

[22] Shen Nie, Fengqi Zhu, Zebin You, Xiaolu Zhang, Jingyang Ou, Jun Hu, Jun Zhou, Yankai Lin, Ji-Rong Wen, and Chongxuan Li. Large language diffusion models, 2025. URL https://arxiv.org/abs/2502.09992.

[23] Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Yu Wu, Y. K. Li, Fuli Luo, Yingfei Xiong, and Wenfeng Liang. DeepSeek-Coder: When the large language model meets programming – the rise of code intelligence, 2024. URL https://arxiv.org/abs/2401.14196.

[24] Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. Code llama: Open foundation models for code, 2024. URL https://arxiv.org/abs/2308.12950.

[25] 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. Evaluating large language models trained on code, 2021. URL https://arxiv.org/abs/2107.03374.

[26] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. Program synthesis with large language models, 2021. URL https://arxiv.org/abs/2108.07732.

[27] Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Zihan Wang, Lei Shen, Andi Wang, Yang Li, Teng Su, Zhilin Yang, and Jie Tang. CodeGeeX: A pre-trained model for code generation with multilingual benchmarking on HumanEval-X. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, KDD '23, pages 5673–5684, New York, NY, USA, 2023. Association for Computing Machinery. doi: 10.1145/3580305.3599790. URL https://doi.org/10.1145/3580305.3599790.

[28] Jinjun Peng, Leyi Cui, Kele Huang, Junfeng Yang, and Baishakhi Ray. CWEval: Outcome-driven evaluation on functionality and security of LLM code generation, 2025. URL https://arxiv.org/abs/2501.08200.

[29] Catherine Tony, Markus Mutas, Nicolás E. Díaz Ferreyra, and Riccardo Scandariato. LLMSecEval: A dataset of natural language prompts for security evaluations. In *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*, pages 588–592. IEEE, 2023. doi: 10.1109/MSR59073.2023.00084. URL https://doi.org/10.1109/MSR59073.2023.00084.

[30] Jonathan Ho, Ajay Jain, and Pieter Abbeel. Denoising diffusion probabilistic models, 2020. URL https://arxiv.org/abs/2006.11239.

[31] Robin Rombach, Andreas Blattmann, Dominik Lorenz, Patrick Esser, and Björn Ommer. High-resolution image synthesis with latent diffusion models, 2022. URL https://arxiv.org/abs/2112.10752.

[32] Jacob Austin, Daniel D. Johnson, Jonathan Ho, Daniel Tarlow, and Rianne van den Berg. Structured denoising diffusion models in discrete state-spaces, 2021. URL https://arxiv.org/abs/2107.03006.

[33] Yiming Zeng, Jinghan Cao, Zexin Li, Yiming Chen, Tao Ren, Zhuochun Li, Dawei Xiang, Xidong Wu, Shangqian Gao, and Tingting Yu. TreeDiff: AST-guided code generation with diffusion LLMs. In *Proceedings of the First Workshop on Structured Understanding, Retrieval, and Generation in the LLM Era (SURGeLLM 2026)*, pages 93–106, San Diego, California, United States, July 2026. Association for Computational Linguistics. doi: 10.18653/v1/2026.surgellm-1.5. URL https://aclanthology.org/2026.surgellm-1.5/.

[34] Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, Kai Dang, Yang Fan, Yichang Zhang, An Yang, Rui Men, Fei Huang, Bo Zheng, Yibo Miao, Shanghaoran Quan, Yunlong Feng, Xingzhang Ren, Xuancheng Ren, Jingren Zhou, and Junyang Lin. Qwen2.5-Coder technical report, 2024. URL https://arxiv.org/abs/2409.12186.

[35] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. Modeling and discovering vulnerabilities with code property graphs. In *2014 IEEE Symposium on Security and Privacy*, pages 590–604. IEEE, 2014. doi: 10.1109/SP.2014.44. URL https://doi.org/10.1109/SP.2014.44.

[36] Max Brunsfeld. Tree-sitter: An incremental parsing system. https://tree-sitter.github.io/tree-sitter/, 2018. URL https://tree-sitter.github.io/tree-sitter/.

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

[38] A. Singh, A. Fry, A. Perelman, et al. OpenAI GPT-5 system card, 2026. URL https://arxiv.org/abs/2601.03267.

[39] Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. In *International Conference on Learning Representations*, 2019.

A Related work (Extended)

With the success of diffusion models in continuous domains, especially in image and video generation [30, 31], recent work has extended diffusion principles to discrete data. Early work by Austin et al. [32] introduced discrete denoising diffusion models, which generalize multinomial diffusion with structured transition operators for discrete state spaces. Sahoo et al. [20] subsequently showed that masked discrete diffusion with an absorbing-state corruption process can be effective for language modeling, deriving a simplified Rao-Blackwellized training objective and establishing a strong masked diffusion baseline that approaches autoregressive perplexity. Concurrently, Shi et al. [21] introduced MD4, which adopts the same masked diffusion formulation and objective and achieves comparable state-of-the-art language-modeling results. More recently, these models have been scaled to large language model regimes. Ye et al. [19] introduced Dream 7B, a scalable diffusion large language model that performs competitively with similarly sized autoregressive models while emphasizing parallel refinement, planning, and flexible inference. In the coding domain, Gong et al. [4] developed DiffuCoder, a 7B diffusion language model specialized for code generation, and showed that diffusion-specific decoding analysis and reinforcement-learning-based post-training can further improve coding performance. More recently, Xie et al. [5] introduced Dream-Coder 7B, an open diffusion language model for code generation with emergent any-order generation capabilities, demonstrating that diffusion language models can adapt their decoding behavior across coding tasks while achieving competitive performance on standard code benchmarks.

Controllable generation is a central challenge in generative AI, particularly for code generation, where outputs are expected not only to be plausible, but also to satisfy validity, correctness, safety, and other program-level properties. In autoregressive language models, this has been studied through constrained decoding, reranking, rejection sampling, and iterative self-correction [10, 11]. These methods typically enforce constraints by restricting next-token decisions, selecting among multiple completed candidates, or repairing an initial output using verifier or execution feedback. Specific to code generation, this line includes self-debugging and test-driven refinement [9], type-constrained decoding [12], and security-aware code evaluation and repair [13]. More broadly, these approaches inherit the structure of autoregressive decoding: early decisions propagate forward, revising a constraint-relevant region often requires regenerating substantial suffixes, and program-level constraints are therefore typically handled through token-local controls or post-hoc repair.

In contrast, constraint integration is particularly suitable for discrete diffusion models, as they refine the entire sequence over multiple denoising steps, creating natural opportunities for sequence-level constraint enforcement during sampling. This has led to several works on controllable generation. Schiff et al. [14] derive classifier-based and classifier-free guidance for discrete diffusion models, adapting standard diffusion guidance mechanisms to discrete state spaces. For protein design, Gruver et al. [15] introduce diffusioN Optimized Sampling (NOS), which performs guidance through gradients in the denoiser hidden states and supports constrained sequence optimization. Other techniques [16, 17] formulate diffusion sampling as constrained optimization and apply projection-based updates during reverse denoising.

Specific to code generation, Mündler et al. [18] present a constrained decoding method for diffusion LLMs under context-free grammars, showing that syntax-level constraints for programming languages can be enforced during out-of-order denoising with improved syntactic correctness. Complementary to such decoding-time constraints, TreeDiff [33] injects structure at training time, masking the tokens

Table 4: Comparison of CDC with related approaches.

Approach	Base	When	Constraint Scope	Semantic / Verifier
AR constrained decoding, reranking, self-correction [10, 11, 9, 13]	AR	token / post	tests, execution, repair feedback	post-hoc only
Type-constrained decoding [12]	AR	token	syntax, types	No
Discrete diffusion guidance [14, 15]	Diff.	denoise	soft or gradient-based objectives	Limited
Optimization-based constrained diffusion [16, 17]	Diff.	denoise	projection / optimization	Limited
CFG-constrained diffusion for code [18]	Diff.	denoise	syntax (CFG)	No
CDC (ours)	Diff.	denoise	syntax, semantics, verifier feedback	**Yes**

of key abstract-syntax-tree nodes rather than random tokens so that the model learns to respect syntactic boundaries. The work presented in this paper is most closely related to this line of research, but targets a harder code-generation setting in which constraints are semantic and security-related. In this setting, prior guidance-based and syntax-level constrained decoding methods are not sufficient, as program semantic correctness and security generally cannot be captured by local token preferences or grammar constraints alone.

Table 4 summarizes the closest lines of work and the dimensions that most directly separate CDC from prior approaches.

B Extended Experimental Details and Results

B.1 Benchmarks

Syntax and functionality. C++ program synthesis is evaluated with HumanEval-X C++ and MBPP-C++ [25, 26, 27]. HumanEval-X extends HumanEval to multiple programming languages with hand-written reference solutions and execution tests; we use its 164 C++ tasks. MBPP-C++ contains 397 C++ translations of MBPP-style programming problems. For each problem, the model receives the natural-language description and function signature, and must generate a complete solution.

We report compile success (Syntax), functional pass@1, and pass@10. A sample is functionally correct if it compiles and passes all unit tests. For pass@1, every model is run deterministically at temperature 0 with seed 0, 256 denoising / decoding steps, and a maximum of 512 generated tokens. For pass@10 we draw 10 independent samples (seeds 0–9, otherwise the same step and length budget) and report the unbiased estimator $\mathrm{pass@}k = \mathbb{E}\big[1 - \binom{n-c}{k} / \binom{n}{k}\big]$, with $n = 10$ and c the number of correct samples per task.

Security. Security-oriented code generation is evaluated on CWEval and LLMSecEval [28, 29]. CWEval additionally evaluates functionality and security on the same programming tasks, making it suitable for testing a method's impact on correctness under security objectives. This dataset contains 108 prompts spanning C, C++, Go, JavaScript, and Python. LLMSecEval contains natural-language

Table 5: Per-language **fs@1** on CWEval (108 prompts total). Parenthesized values give the absolute change relative to the corresponding model's Vanilla row: yellow marks an improvement of <10 points, green marks an improvement of ≥ 10 points, and red marks a regression. Within each model, the strongest value per language is **bolded**.

Model	Method	py (25)	js (23)	c (20)	cpp (21)	go (19)	ALL (108)
Diffusion	Vanilla	24.0	13.0	0.0	14.3	5.3	12.0
	+ sec prompt	28.0 (+4)	17.4 (+4)	0.0 (+0)	14.3 (+0)	5.3 (+0)	13.9 (+2)
	MDFI (Ours)	**72.0** (+48)	**34.8** (+22)	**5.0** (+5)	**28.6** (+14)	**21.1** (+16)	**34.3** (+22)
AR	Vanilla	44.0	21.7	10.0	14.3	10.5	21.3
	Reprompt	**64.0** (+20)	**43.5** (+22)	**25.0** (+15)	**33.3** (+19)	**26.3** (+16)	**39.8** (+19)

prompts associated with common CWE patterns; LLMSecEval+ is constructed by adding executable functional and security oracles to 150 LLMSecEval prompts.

For security, we report three pass@1 metrics. Let F_i indicate that sample i passes the functional oracle, and let S_i indicate that it passes the security oracle. Then

$$\text{func@1} = \mathbb{E}[F_i], \qquad \text{sec@1} = \mathbb{E}[S_i], \qquad \text{func-sec@1} = \mathbb{E}[F_i S_i].$$

The joint metric func-sec@1 is our primary security metric, since generated code must be both secure and functionally correct.

B.2 Implementation Details

GradGuide. GradGuide uses a Qwen2.5-Coder-1.5B surrogate [34] trained on CodeContests-derived programs [1] with execution labels (Appendix C.2). The surrogate predicts a continuous correctness score from soft token embeddings and is used for both localization and ALM correction. We apply GradGuide adaptively during reverse denoising, using adaptive editing to localize and refine constraint-relevant regions. Specifically, surrogate-guided proposal corrections are activated when needed, while high-saliency token positions are expanded into local syntactic neighborhoods and remasked when the constraint-violation and edit-budget conditions are satisfied.

MDFI. MDFI invokes the static-analysis evaluator at fixed denoising checkpoints. The analyzer constructs a tolerant partial Code Property Graph [35] using Tree-sitter-style incremental parsing [36], extracts dataflow witnesses with bounded source-to-sink search, obtains structural risk with the product FSM when exact labels are masked, selects semantic neighborhoods from the AST under a token budget, and injects compact remediation feedback into a preallocated prompt buffer.

Efficiency measurements. We measure both outcome quality and corrective cost. For GradGuide, we report total tokens generated by the model, tokens per instance, tokens per passing solution, tokens rewritten per failed instance, and fraction of the program body edited. For MDFI, we additionally measure edit span locality and number of edit clusters. AR re-prompting is counted as regenerating the full program body, while CDC counts only the regions that are remasked or inserted during the denoising trajectory. Efficiency measurements and per-prompt analyses in Appendix B.3 each use one generation per prompt, independently sampled across analyses from the multi-sample runs underlying pass@10 in Table 1 and fs@10 in Table 2 for all methods.

B.3 Additional Experimental Results

In this section we present additional detailed results.

B.3.1 CDC is substantially more local and token-efficient than AR re-prompting

Figure 4 compares GradGuide to AR plus one re-prompt on HumanEval-X C++ and MBPP-C++. On HumanEval-X, AR plus one re-prompt generates 35,640 total model tokens, whereas GradGuide generates 26,723 tokens, a $0.75\times$ cost ratio and a savings of 8,917 tokens. On MBPP-C++, AR plus one re-prompt generates 89,456 tokens, whereas GradGuide generates 38,676 tokens, a $0.43\times$ cost ratio and a savings of 50,780 tokens.

Normalizing by problem count and by successful solution gives the same conclusion. On HumanEval-X, GradGuide uses 163 tokens per instance and 265 tokens per passing solution, compared with

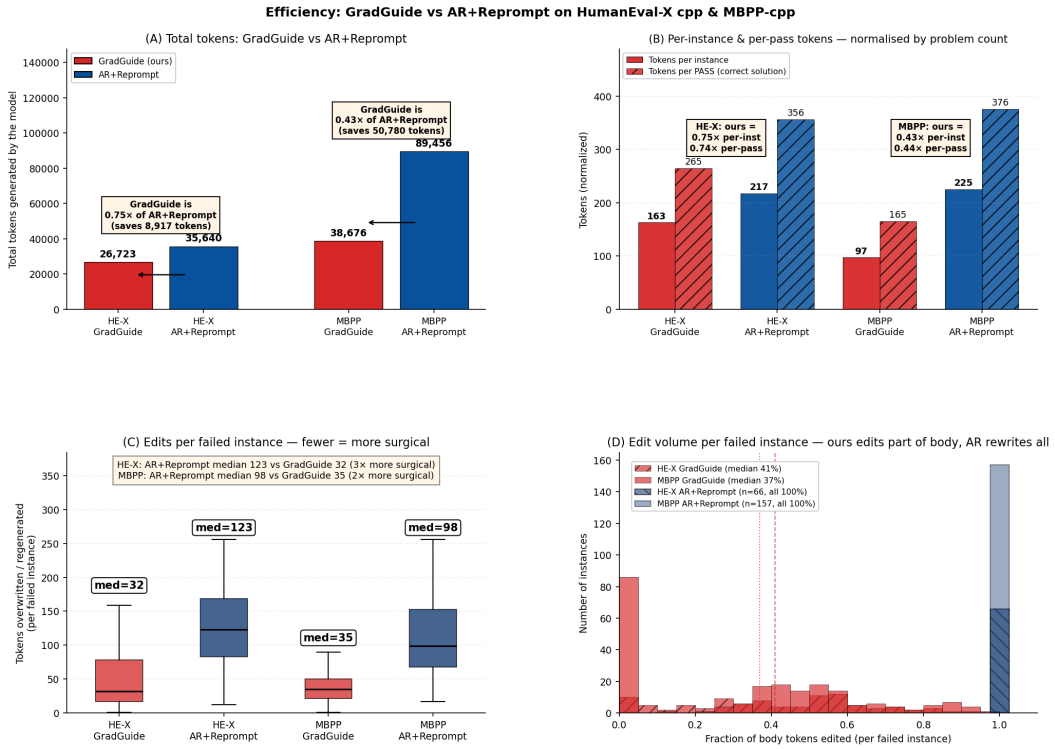

Figure 4: Efficiency and locality of CDC vs. AR+Reprompt

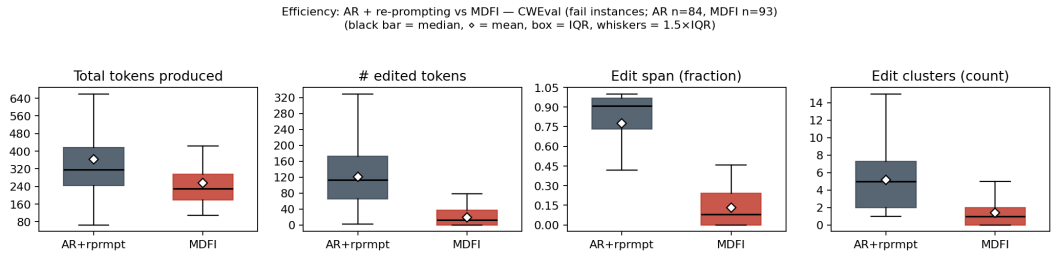

Figure 5: CWEval efficiency comparison between AR re-prompting and MDFI. MDFI lowers pipeline token cost, edited-token count, edit span, and edit clusters by repairing localized vulnerable regions rather than regenerating the entire program.

217 and 356 for AR re-prompting. On MBPP-C++, GradGuide uses 97 tokens per instance and 165 tokens per passing solution, compared with 225 and 376 for AR re-prompting. Thus the efficiency gain is not merely due to differences in benchmark size; it persists when normalized by both instances and successful outputs.

The locality analysis explains the efficiency gain. For failed instances that require correction, AR re-prompting corrects a median of 123 tokens on HumanEval-X and 98 tokens on MBPP-C++, while GradGuide edits medians of 32 and 35 tokens, respectively. In other words, GradGuide is about $3\times$ more surgical on HumanEval-X and $2\times$ more surgical on MBPP-C++. At the body-fraction level, AR re-prompting rewrites essentially the entire program, whereas GradGuide concentrates edits on localized regions identified by surrogate saliency. This supports the focused-editing hypothesis: CDC achieves its gains by revising the parts of the denoising trajectory most relevant to the constraint, not by repeatedly sampling complete programs.

Figure 5 compares MDFI with AR re-prompting on CWEval. AR re-prompting produces roughly a full second program, with a mean pipeline cost around 3.1×10^2 tokens. MDFI reduces the mean pipeline cost to roughly 2.1×10^2 tokens. The difference is larger for edited tokens: AR re-prompting

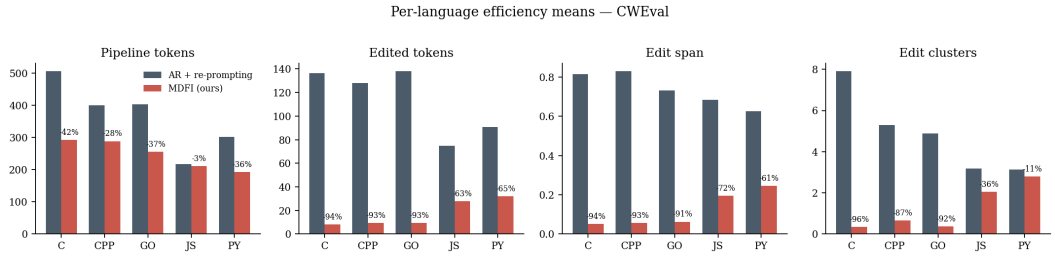

Figure 6: Per-language efficiency means on CWEval. MDFI substantially reduces edited tokens, edit span, and edit clusters across C, C++, Go, JavaScript, and Python.

edits on the order of an entire solution, while MDFI edits a small localized span. Median edit span drops from 0.89 of the program body under AR re-prompting to below 0.1 under MDFI, and median edit clusters drop from 4 to 1.

The per-language analysis in Figure 6 confirms that locality is robust across languages. MDFI reduces edited tokens by 94% for C, 93% for C++, 93% for Go, 63% for JavaScript, and 65% for Python. Edit-span reductions are similarly large: 94% for C, 93% for C++, 91% for Go, 72% for JavaScript, and 61% for Python. Edit-cluster reductions are strongest in C/C++/Go, where vulnerable patterns tend to be localized around library calls or pointer/string operations; they are smaller in Python and JavaScript, where secure fixes often require changing a higher-level API pattern across multiple nearby tokens. Even in those languages, MDFI remains substantially more local than whole-program re-prompting.

B.3.2 Inference cost and throughput

Tables 6–8 give the full measurements summarized in Section 6.4. Vanilla decoding and CDC use the same GPUs, numerical precision, batch size, backbone, decoding configuration, and 256-step schedule, and reported times include all surrogate and analyzer calls. Intervention frequency is the fraction of samples with at least one intervention, and successful-output throughput is the number of functional and secure outputs per unit of wall-clock time relative to vanilla decoding. MDFI reaches 86–95% of the fs@1 of AR re-prompting (34.3 vs. 39.8 on CWEval and 24.7 vs. 26.0 on LLMSecEval+) without generating a second complete program.

Table 6: GradGuide inference cost on HumanEval-X C++ (164 tasks). Latency statistics are per sample; the surrogate column gives the mean (median) cost of one surrogate forward–backward pass.

Backbone	Method	Total time	Mean / p50 / p95 / max (s)	×vanilla	Peak GPU	Surrogate pass	Interv. freq.
Dream-Coder 7B	Vanilla	693 s	4.23 / 4.26 / 4.62 / 5.38	1.00×	15.1 GB	–	–
	GradGuide	1,310 s	7.99 / 6.10 / 17.46 / 20.07	1.89×	23.1 GB	49.5 ms (47.0)	0.66
Dream 7B	Vanilla	685 s	4.18 / 4.21 / 4.57 / 5.34	1.00×	14.7 GB	–	–
	GradGuide	1,181 s	7.20 / 5.93 / 15.55 / 19.41	1.72×	23.2 GB	52.6 ms (48.5)	0.84
DiffuCoder 7B	Vanilla	682 s	4.16 / 4.19 / 4.54 / 5.32	1.00×	14.7 GB	–	–
	GradGuide	1,372 s	8.37 / 6.16 / 16.84 / 21.31	2.01×	23.6 GB	48.3 ms (45.0)	0.66

B.3.3 Ablations

GradGuide components. Figure 7(a) reports per-configuration pass@1 on HumanEval-X C++ (Dream-Coder 7B, greedy): vanilla 34.1, ALM-only 40.2, adaptive editing-only 50.6, full operator 65.2; random editing matches vanilla, so localization must be constraint-steered. Panel (b) decomposes the full-operator gain: ALM-only (+6.1) and adaptive editing-only (+16.5) predict 56.7 under independent action; the full operator reaches 65.2, a **+8.5 pp super-additive synergy** from ALM shaping the local distribution *inside* the region adaptive editing reopens.

Table 7: Pass@1 gain (points) of test-time-compute methods over their own vanilla baselines.

Method	HumanEval-X C++ Δpass@1	MBPP-C++ Δpass@1
Best-of-N ($N=2$)	+3.1	+3.5
AR+Reprompt	+4.2	+1.0
CDC (Ours)	**+31.1**	**+31.5**

Table 8: MDFI inference cost with Dream-Coder 7B (vanilla→MDFI).

Metric	CWEval	LLMSecEval+
Mean latency (s)	12.98→13.28 (+2.3%)	10.12→10.73 (+6.1%)
p50 / p95 latency (s)	12.85→12.97 / 16.73→16.87	9.62→11.00 / 11.82→12.72
Total benchmark time (min)	23.4→23.9	25.3→26.8
Peak GPU memory (GiB)	64.2→66.9 (+4.2%)	30.4→31.2 (+2.6%)
Analyzer calls / CPU time	7 / 46 ms (1 core)	7 / 46 ms (1 core)
fs@1 (%)	12.0→**34.3**	14.7→**24.7**
sec@1 (%)	18.5→**41.7**	54.7→**80.7**
Successful-output throughput (×vanilla)	**2.79**×	**1.58**×

Correction modes. Table 9 isolates GradGuide's two correction modes on both benchmarks, keeping the backbone, surrogate, and localization fixed. Mode A (KL-anchored proposal correction) and Mode B (localized remasking) each improve pass@1, with Mode B contributing more, and their combination exceeds the sum of the individual gains by 8.5 and 3.5 points on HumanEval-X C++ and MBPP-C++.

Table 9: Contribution of GradGuide's two correction modes (Dream-Coder 7B, greedy).

Benchmark	Mode A	Mode B	pass@1 (%)	Δ vs. vanilla (points)
HumanEval-X C++	✗	✗	34.1	–
	✓	✗	40.2	+6.1
	✗	✓	50.6	+16.5
	✓	✓	**65.2**	**+31.1**
MBPP-C++	✗	✗	27.7	–
	✓	✗	35.5	+7.8
	✗	✓	47.9	+20.2
	✓	✓	**59.2**	**+31.5**

MDFI insertion budget and neighborhood scope. Figure 8(a): mask insertion K plateaus at $K \in [8, 12]$ on both benchmarks (CWEval 34.3, LLMSecEval+ 24.7); $K > 16$ regresses functionality. Panel (b): the deployed Parent+Leaf neighborhood (34.3) beats tighter (Token-Window 24.1) and looser (Use–Def Slice 26.9) alternatives.

B.3.4 CDC reduces correction-induced regressions

To complement the headline pass@k rates, we characterize each method's behavior as a *corrector*: how it converts vanilla outputs into corrected outputs at the per-prompt level. Following standard practice for evaluating verifier- or test-feedback-driven correctors [9], we report four complementary

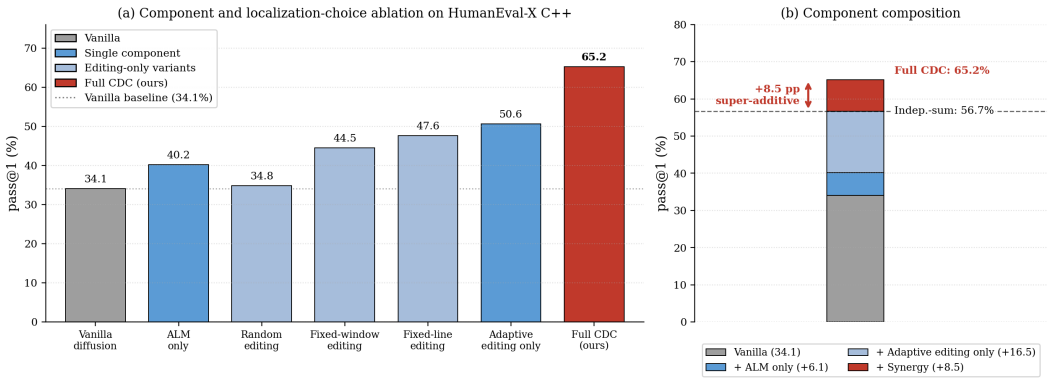

Figure 7: Component and localization-choice ablation of CDC on HumanEval-X C++ (164 tasks, Dream-Coder 7B, greedy). **(a)** Per-configuration pass@1. **(b)** Component composition: ALM-only +6.1 pp and adaptive editing-only +16.5 pp predict 56.7% under independent action; the full operator reaches 65.2%, a +8.5 pp super-additive synergy.

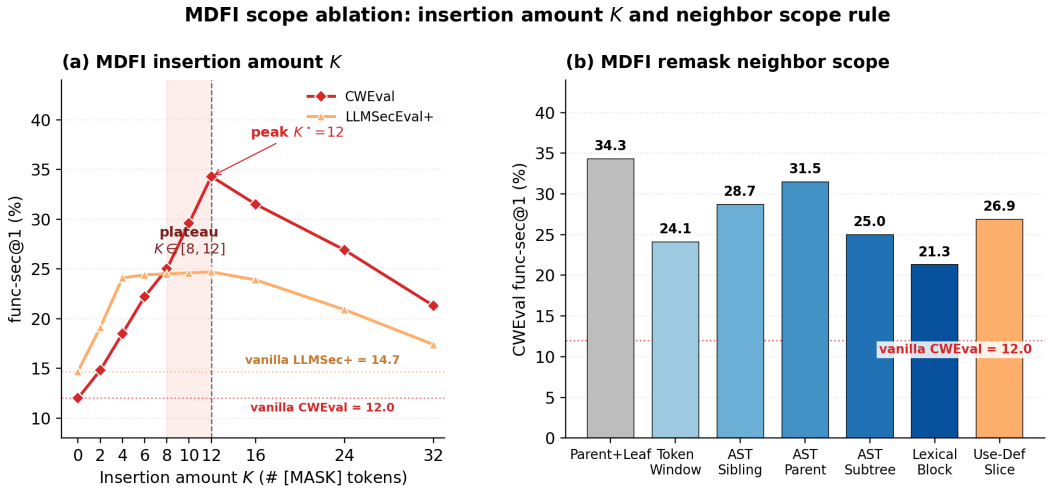

Figure 8: MDFI scope ablation. **(a)** Insertion amount K on CWEval and LLMSecEval+; func-sec@1 plateaus at $K \in [8, 12]$. **(b)** Remasking neighborhood scope on CWEval; the deployed Parent+Leaf rule peaks at 34.3%, beating both Token-Window and broader (Use–Def Slice) alternatives.

measurements that probe different facets of corrective behavior, summarized in Table 10: improvement rate (fraction of base-failing prompts recovered), the raw improvement-to-regression count, net change, and constructive ratio (the share of all verdict changes that are improvements). Together they characterize whether a method recovers failures at scale, whether it does so with high precision, and whether its action is closer to a monotone corrector or to coin-flip churn.

For security, Table 11 compares how MDFI and AR re-prompting change per-prompt verdicts on CWEval. MDFI rarely breaks programs that vanilla decoding already solves (a regression rate of at most 3.1%, versus 21.7–29.4% for AR re-prompting) and achieves a larger net change on every metric; all of its security and joint verdict changes are improvements, whereas AR re-prompting turns 14.7% of its joint verdict changes into regressions.

C Details of GradGuide

This appendix provides the implementation details of the surrogate-gradient operator used in Section 5.2. GradGuide instantiates the constraint-aware operator $\mathcal{P}_C$ for functional correctness and syntactic validity. Its goal is to provide differentiable denoising-time feedback for constraints that

Table 10: Corrective effectiveness on functional benchmarks. For each method we report four complementary per-prompt measurements: **improvement rate** (↑), the fraction of base-failing prompts the method recovers; **Imp. / Reg.** (↑), the raw counts of improvements and regressions induced; **net** change (↑), improvements minus regressions; and **constructive ratio** (↑), the share of all verdict changes that are improvements (100% corresponds to a strictly monotone corrector). For CFG-CD and CDC, the reference vanilla is Dream-Coder 7B; for AR+Reprompt, the reference is DeepSeek-Coder-Instruct-6.7B. Best per row is **bolded**. CDC is the strongest corrector on every measurement on both benchmarks: it recovers 48.0%/43.8% of base-failing programs (vs. AR+Reprompt's 28.8%/4.4%), delivers +46/+120 net improvements (vs. +2/+1), and converts 94–97% of its verdict changes into fixes (vs. AR+Reprompt's near-coin-flip ∼53%). CFG-CD recovers *zero* failing programs on MBPP-C++.

Angle	Benchmark	CFG-CD	AR+Reprompt	CDC (Ours)
Improvement rate ↑	HE-X C++	13.7%	28.8%	**48.0%**
	MBPP-C++	0.0%	4.4%	**43.8%**
Imp. / Reg. (counts)	HE-X C++	14 / 16	19 / 17	**49 / 3**
	MBPP-C++	0 / 8	7 / 6	**124 / 4**
Net ↑	HE-X C++	-2	+2	**+46**
	MBPP-C++	-8	+1	**+120**
Constructive ratio ↑	HE-X C++	46.7%	52.8%	**94.2%**
	MBPP-C++	0.0%	53.8%	**96.9%**

Table 11: Stability and corrective effectiveness on CWEval for AR re-prompting (AR) and CDC with MDFI. Regression rate is the fraction of vanilla-correct prompts that a method breaks; net change is the number of improvements minus regressions on the prompts where CDC's operator triggers; edit precision is the fraction of verdict changes that are improvements.

	Regression rate (%) ↓		Net change ↑		Edit precision (%) ↑	
Metric	AR	CDC	AR	CDC	AR	CDC
func@1	29.4	**3.1**	+1	**+19**	51.6	**95.2**
sec@1	23.1	**0.0**	+29	**+33**	85.4	**100.0**
fs@1	21.7	**0.0**	+24	**+35**	85.3	**100.0**

are ultimately evaluated by non-differentiable program oracles, such as compilation and unit-test execution.

C.1 Surrogate Model

Program-level correctness is non-differentiable: a candidate program either compiles and passes its tests, or it does not. GradGuide therefore trains an auxiliary surrogate g_ϕ ahead of time and uses it only at inference time. The diffusion model parameters θ are never updated.

At reverse step t, the surrogate maps the clean-state proposal $\hat{\mathbf{x}}_0^{(t)} \in \Delta^{L\times|\mathcal{V}|}$ and task context c to one or more continuous correctness scores,

$$g_{\phi,j}(\hat{\mathbf{x}}_0^{(t)}, c) \in [0,1], \qquad j \in \{1,\dots,m\}. \tag{19}$$

To make the surrogate differentiable with respect to token distributions, each soft token distribution is converted into a soft embedding,

$$\mathrm{Emb}(\hat{\mathbf{x}}_0^{(t)})^i = \hat{x}_0^{(t),i}\,\mathbf{E}_{\mathrm{tok}}, \qquad i \in [L], \tag{20}$$

where $\hat{x}_0^{(t),i} \in \Delta^{|\mathcal{V}|}$ is the per-position distribution (the i-th row of $\hat{\mathbf{x}}_0^{(t)}$) and $\mathbf{E}_{\mathrm{tok}} \in \mathbb{R}^{|\mathcal{V}|\times d}$ is the surrogate input embedding matrix. The surrogate is trained using execution-derived labels, including compile success and test-pass outcomes. In our implementation, the surrogate is a LoRA-adapted Qwen2.5-Coder-1.5B backbone [34] with a regression head on the last-token hidden state, trained on CodeContests-derived $(\text{problem}, \text{code})$ pairs labeled by $y = \mathbf{1}[\text{compiles}] \cdot (\text{test-pass ratio}) \in [0,1]$

from real C++ execution; Appendix C.2 gives the architecture, training data, and training protocol. The model is conditioned on a fixed judge prompt prepended to each input – "*You are a strict unit-test judge. Given a programming problem and a solution, assign a functionality correctness score from 0 to 1: 0 = completely wrong or non-functional, 1 = fully correct and passes all intended behavior.\n\nProblem:\n*" followed by the problem text and the candidate code (token IDs at training time, soft embeddings at inference) – so that the same prompt is used for training the regression head, for localization, and for proposal correction.

For each constraint j, we define the surrogate-relaxed violation

$$\Delta g_j(\hat{\mathbf{x}}_0^{(t)}, c) = \max\big(0, \tau_j - g_{\phi,j}(\hat{\mathbf{x}}_0^{(t)}, c)\big), \tag{21}$$

where $\tau_j \in (0, 1]$ is the target satisfaction threshold. The aggregate surrogate violation is

$$\Delta G(\hat{\mathbf{x}}_0^{(t)}, c) = \sum_{j=1}^{m} \Delta g_j(\hat{\mathbf{x}}_0^{(t)}, c). \tag{22}$$

The surrogate functions $g_{\phi,j}$, Emb, Δg_j, and ΔG are well-defined on any soft proposal in $\Delta^{L\times|\mathcal{V}|}$; we have written them at $\hat{\mathbf{x}}_0^{(t)}$ above for clarity, and Mode A below reuses them with its inner-loop iterate in place of $\hat{\mathbf{x}}_0^{(t)}$.

C.2 Surrogate Training and Evaluation

The surrogate fine-tunes Qwen2.5-Coder-1.5B [34] with LoRA adapters [37] on the last 12 of its 28 decoder layers and a four-layer MLP head (dropout 0.1) on the last-token hidden state, keeping the backbone frozen. Its input consists of the judge prompt, the problem statement, and the candidate program, truncated to 2,048 tokens, and it is trained with soft-label binary cross-entropy on $y \in [0, 1]$. We use a causal scorer because it reuses a pretrained code model and its last-token state attends to the entire program.

The 59K training examples are drawn exclusively from CodeContests [1]. Candidates are generated by Dream [19], Dream-Coder [5], and DiffuCoder [4] with $\{32, 64, 128, 256\}$ denoising steps and temperatures $\{0, 0.2, 0.6, 1.0\}$, supplemented by execution-verified GPT-5 [38] solutions for problems on which no diffusion candidate passes, and labeled by execution against the CodeContests tests. No HumanEval-X or MBPP data is used for training or model selection, and duplicate checks confirm that the training problems are disjoint from both benchmarks. We train with AdamW [39] (learning rate 2×10^{-5}, cosine schedule with 10% warmup, weight decay 0.01, effective batch size 16, bf16) for up to five epochs and select the checkpoint with the highest validation Pearson correlation on held-out problems (epoch 3). Training takes about 1.8 GPU-hours on a single NVIDIA RTX PRO 6000 Blackwell GPU.

Table 12 evaluates the surrogate before and after fine-tuning on frozen base-model candidates and on held-out CodeContests problems. MAE and MSE measure the error against y, Spearman ρ measures ranking by test-pass ratio, and AUPRC and ECE measure discrimination and calibration for complete execution success. Guidance Utility is the fraction of same-problem candidate pairs with different outcomes that the surrogate orders correctly, and Net Δpass@1 is the change in GradGuide pass@1 (Dream-Coder 7B) when the scorer is used in the pipeline. Fine-tuning improves every metric on all three datasets and turns the pass@1 change from $-8.5/-5.3$ into +31.1/+31.5 points on HumanEval-X C++/MBPP-C++.

C.3 Gradient-Based Localization

GradGuide computes the gradient of the aggregate surrogate violation with respect to each soft token embedding, evaluated at the clean-state proposal $\hat{\mathbf{x}}_0^{(t)}$:

$$\boldsymbol{g}_i\big(\hat{\mathbf{x}}_0^{(t)}, c\big) = \nabla_{\mathrm{Emb}(\hat{\mathbf{x}}_0^{(t)})^i} \Delta G\big(\hat{\mathbf{x}}_0^{(t)}, c\big), \qquad i \in [L]. \tag{23}$$

The norm $\|\boldsymbol{g}_i(\hat{\mathbf{x}}_0^{(t)}, c)\|_2$ measures how sensitive the predicted violation is to changes at position i. GradGuide combines this violation sensitivity with uncertainty signals from the base denoiser:

$$a_i = \underbrace{\big\|\boldsymbol{g}_i\big(\hat{\mathbf{x}}_0^{(t)}, c\big)\big\|_2}_{\text{violation sensitivity}} + \alpha_H H\big(\hat{x}_0^{(t),i}\big) + \alpha_C \left(1 - \max_{v\in\mathcal{V}} \hat{x}_0^{(t),i}(v)\right), \tag{24}$$

Table 12: Standalone evaluation of the GradGuide surrogate before fine-tuning (the Qwen2.5-Coder-1.5B backbone) and after training. Lower is better for MAE, MSE, and ECE; higher is better otherwise. N/A: not applicable.

Scorer	Dataset	MAE	MSE	Spearman ρ	AUPRC	ECE	Guidance Utility	Net Δpass@1
Backbone (no fine-tuning)	HumanEval-X C++	0.533	0.342	0.056	0.277	0.377	N/A	−8.5
	MBPP-C++	0.472	0.262	0.117	0.533	0.163	N/A	−5.3
	Held-out CodeContests	0.254	0.086	0.581	0.827	0.092	0.723	N/A
Trained surrogate	HumanEval-X C++	**0.074**	**0.022**	**0.848**	**0.946**	**0.024**	N/A	**+31.1**
	MBPP-C++	**0.111**	**0.051**	**0.837**	**0.960**	**0.030**	N/A	**+31.5**
	Held-out CodeContests	**0.042**	**0.010**	**0.921**	**0.995**	**0.030**	**0.933**	N/A

where $H(\hat{x}_0^{(t),i})$ is the entropy of the base proposal at position i, and $\alpha_H, \alpha_C \geq 0$ weight entropy and confidence terms. Importantly, the saliency is computed before any correction has been applied, so all three terms are anchored at $\hat{\mathbf{x}}_0^{(t)}$. GradGuide then selects the top-k saliency positions and expands them to syntactically coherent neighborhoods:

$$\mathcal{S}_t = \mathrm{Expand}(\text{Top-}k(a_1, \ldots, a_L)) \,. \tag{25}$$

The expansion operation maps isolated token positions to coherent edit regions, such as the enclosing line, expression, block, or delimiter-balanced span. This prevents GradGuide from editing individual tokens in ways that break local syntax.

C.4 Mode A: KL-Anchored Augmented-Lagrangian Projection

The first correction mode shifts the clean-state proposal toward satisfying the surrogate constraints while preserving proximity to the base denoiser. Given the localized edit set $\mathcal{S}_t$, GradGuide introduces an auxiliary optimization variable $\boldsymbol{y} \in \Delta^{L\times|\mathcal{V}|}$, initialized at the clean-state proposal $\hat{\mathbf{x}}_0^{(t)}$, and approximately solves

$$\boldsymbol{y}_t \approx \arg\min_{\boldsymbol{y}\in\Delta^{L\times|\mathcal{V}|}} \mathcal{L}_{\mathrm{ALM}}\big(\boldsymbol{y}; \hat{\mathbf{x}}_0^{(t)}, \mathcal{S}_t, c\big), \tag{26}$$

where

$$\mathcal{L}_{\mathrm{ALM}}\big(\boldsymbol{y}; \hat{\mathbf{x}}_0^{(t)}, \mathcal{S}_t, c\big) = D_{\mathrm{KL}}\Big(\boldsymbol{y} \,\|\, \hat{\mathbf{x}}_0^{(t)}\Big) + \sum_{j=1}^{m} \Big[\lambda_j \Delta g_j(\boldsymbol{y}, c) + \frac{\mu_j}{2} \Delta g_j(\boldsymbol{y}, c)^2\Big] + \beta \sum_{i\notin\mathcal{S}_t} D_{\mathrm{KL}}\Big(\boldsymbol{y}^i \,\|\, \hat{x}_0^{(t),i}\Big) \,. \tag{27}$$

The first term is a trust region that anchors the corrected proposal to the base denoiser distribution. The second term is an augmented-Lagrangian penalty that drives surrogate violation toward zero. The final term is a locality anchor that discourages changes outside the localized edit region. In the limit of large β, the update is effectively restricted to $\mathcal{S}_t$.

In implementation, we parameterize $\boldsymbol{y} = \mathrm{softmax}(u)$ with unconstrained logits $u \in \mathbb{R}^{L\times|\mathcal{V}|}$, initialize

$$u^{(0)} = \log\Big(\hat{\mathbf{x}}_0^{(t)} + \varepsilon\Big) \,, \tag{28}$$

and take K_{inner} first-order steps on Eq. 27. The corrected proposal after these steps is denoted $\boldsymbol{y}_t$.

The multipliers and penalties are updated using an augmented-Lagrangian rule:

$$\lambda_j \leftarrow \big[\lambda_j + \mu_j \Delta g_j(\bar{\mathbf{x}}_0^{(t)}, c)\big]_+, \tag{29}$$

and

$$\mu_j \leftarrow \begin{cases} \rho\mu_j, & \Delta g_j(\bar{\mathbf{x}}_0^{(t)}, c) \geq \vartheta \Delta g_j^{\mathrm{prev}}, \\ \mu_j, & \text{otherwise}, \end{cases} \tag{30}$$

where $\rho > 1$ is the penalty growth factor and $\vartheta \in (0, 1)$ is a progress tolerance. The update uses the surrogate violation on the argmax-decoded intermediate program $\bar{\mathbf{x}}_0^{(t)}$, rather than only the soft proposal $\boldsymbol{y}$, to reduce drift between soft surrogate satisfaction and decoded-program feasibility.

Mode A is gated by the surrogate score. If the decoded intermediate program already satisfies the surrogate threshold,

$$g_\phi(\bar{\mathbf{x}}_0^{(t)}, c) \geq \tau_{\text{alm}}, \tag{31}$$

then GradGuide skips the inner optimization and returns $\boldsymbol{y}_t = \hat{\mathbf{x}}_0^{(t)}$. Otherwise, the KL-anchored correction is applied.

C.5 Mode B: Constraint-Triggered Local Remasking

The second correction mode reopens already committed tokens for re-denoising. This is necessary because, under the masked diffusion reverse kernel, once $x_t^i \neq \texttt{[MASK]}$, a proposal correction at position i cannot directly rewrite the committed token. Mode B therefore remasks localized positions when the intermediate program is sufficiently decoded and still violates the target constraint.

Let

$$n_{\text{mask}}(\mathbf{x}_t) = \left|\{i \in [L] : x_t^i = \texttt{[MASK]}\}\right| \tag{32}$$

denote the number of masked positions at timestep t, and let b_t denote the number of edits already used along the trajectory. Mode B activates when

$$n_{\text{mask}}(\mathbf{x}_t) \leq m_\star, \qquad b_t < B, \qquad \text{Satisfies}(\bar{\mathbf{x}}_0^{(t)}, c) = \texttt{false}. \tag{33}$$

Here, $m_\star$ is a mask-count threshold ensuring that the partial program is sufficiently decoded for evaluation, and B is the global edit budget. The predicate Satisfies can be an exact oracle when available, such as compilation, unit-test execution, or static analysis. When exact evaluation is not available on the partial program, GradGuide uses the surrogate condition $g_\phi(\bar{\mathbf{x}}_0^{(t)}, c) \geq \tau$.

When Eq. 33 is satisfied, GradGuide constructs a reopened state

$$x_t^{\star,i} = \begin{cases} \texttt{[MASK]}, & i \in \mathcal{S}_t, \\ x_t^i, & i \notin \mathcal{S}_t. \end{cases} \tag{34}$$

The reverse chain then continues from $\mathbf{x}_t^\star$ under the constrained reverse kernel, with Mode A active during the refill steps. The edit counter b_t is incremented after the intervention.

C.6 Composition of the Two Modes

GradGuide uses a single surrogate signal to produce two coupled outputs at each reverse step: a corrected proposal $\boldsymbol{y}_t$ from Mode A and, when triggered, a reopened state $\mathbf{x}_t^\star$ from Mode B. The constrained reverse kernel then advances the chain using the pair $(\mathbf{x}_t^\star, \boldsymbol{y}_t)$. If Mode B does not activate, $\mathbf{x}_t^\star = \mathbf{x}_t$.

The two modes address complementary failure cases. Mode A shifts probability mass toward feasible tokens while the position remains editable through the proposal distribution. Mode B reopens tokens that have already been committed, allowing the denoiser to repair localized regions with the benefit of the surrounding program context. Both modes are scoped by the same localization set $\mathcal{S}_t$ and driven by the same surrogate gradient.

The main configurations are recovered by different choices of the projection threshold and edit budget:

$$\begin{aligned}
&\tau_{\text{alm}} = 0,\ B = 0 && \text{recovers the unconstrained reverse process,} \\
&\tau_{\text{alm}} > 0,\ B = 0 && \text{uses only KL-anchored proposal correction,} \\
&\tau_{\text{alm}} = 0,\ B > 0 && \text{uses only constraint-triggered remasking,} \\
&\tau_{\text{alm}} > 0,\ B > 0 && \text{uses the full GradGuide operator.}
\end{aligned}$$

The deployed configuration uses both modes. Its per-step cost is at most one surrogate forward/backward pass, K_{inner} inner optimization steps when Mode A is active, and K_{edit} additional reverse steps when Mode B activates.

D Details of MDFI

This appendix provides the implementation details of the static-analysis-guided operator used in Section 5.2. MDFI instantiates the constraint-aware operator $\mathcal{P}_{\mathcal{C}}$ for security constraints. Its goal is to provide non-differentiable denoising-time feedback for properties that are characterized by discrete syntactic and dataflow patterns rather than smooth correctness scores. The base diffusion parameters θ are not modified, the analyzer carries no learnable parameters, and the same partial-program analysis pipeline is reused across all benchmarks, languages, and decoding configurations: MDFI is therefore training-free.

D.1 Partial Program Representation

At each fired checkpoint, the analyzer builds a structural representation $\mathcal{G}_t$ from the decoded clean-state proposal $\bar{\mathbf{x}}_0^{(t)}$ defined in Eq. 4. The representation is a partial program graph that combines an abstract syntax tree with a dataflow approximation: $\mathcal{G}_t = (V, E^{\mathrm{AST}}, E^{\mathrm{CFG}}, E^{\mathrm{DFG}}, \ell)$, where V are program nodes (statements, expressions, identifiers), E^{AST} encodes parent–child structure, E^{CFG} encodes control flow between adjacent basic blocks within each function, E^{DFG} encodes intra-procedural dataflow between defining and using occurrences of each program identifier, and ℓ labels each node with its lexical class.

Tokens in $\mathbf{x}_t$ that remain masked at checkpoint time are not carried into the parser literally; instead, each mask is rewritten into a fresh placeholder identifier of the form `__hole_<i>__` before parsing. This rewrite preserves the lexical class of the position—an identifier slot remains an identifier, a literal slot remains a literal—and lets the partial parser produce a well-formed graph even when a non-trivial fraction of the program is still masked. The analyzer is invoked at a sparse schedule of checkpoints (Section D.8), gated by a minimum committed-fraction so that the rewrite is only triggered when enough of $\bar{\mathbf{x}}_0^{(t)}$ has been decoded for the resulting graph to be informative.

D.2 Vulnerability Detection on Partial Programs

The detector inspects $\mathcal{G}_t$ for two complementary symptoms of insecurity that together cover both fully decoded and partially masked vulnerable shapes.

Dataflow witnesses. For each known unsafe sink class, the detector performs a bounded breadth-first search on E^{DFG} from candidate *source* nodes (locations in $\mathcal{G}_t$ that introduce data outside the program's trust boundary, such as user input or environment data) toward candidate *sink* nodes (locations that consume data in a security-sensitive way, such as command execution or unparameterized query construction). The search halts whenever it crosses a sanitizer node and yields a witness whenever it terminates at a sink without ever crossing one. The result is a path of nodes in $\mathcal{G}_t$ that demonstrates the vulnerable flow.

Structural witnesses. For partial programs in which the dangerous identifier is still masked or the dataflow chain has not yet crystallized, the detector also matches local AST shapes that are unsafe regardless of dataflow: a sink callsite whose argument shape bypasses a required guard, an insecure construction template, or a missing structural neighbor that the safe usage requires. These shape matches fire on subgraphs of $\mathcal{G}_t$ rather than on dataflow paths and are essential for catching vulnerabilities at early checkpoints when only the coarse program skeleton is committed.

In both cases, each detection produces a witness $w_k = (n_k, \tau_k, h_k)$ with the offending node $n_k \in V$, the correction type $\tau_k \in \{\mathrm{sub}, \mathrm{ins}\}$, and a structured remediation hint h_k identifying the violated property and the recommended safe pattern. The witnesses across the two detection paths are combined into a single set $\mathcal{W}_t = \{w_k\}_{k=1}^{K_t}$ that populates the violation vector $\boldsymbol{\nu}_t$ and structured feedback r_t in Eq. 5.

D.3 Analyzer Rules and Choice of Static Analysis

The analyzer reuses existing declarative detection rules and intermediate representations from three general-purpose static analyzers, CodeQL[3], Semgrep[4], and Bandit[5], without additional hand-crafted rules; the partial-program layer described above adapts them to masked, incomplete programs and maps each finding to an editable token region. We use static analysis rather than a learned detector because the available labeled data, 258 prompts across CWEval (108) and LLMSecEval+ (150), are too few to train a reliable multilingual detector and would leak evaluation data. A false positive may trigger an unnecessary correction, which mainly adds overhead. As Table 13 shows, substituting the off-the-shelf tools for our analyzer on LLMSecEval+ improves fs@1 by at most 2.6 points at 36–288% additional latency (CodeQL times out), whereas our analyzer improves fs@1 by 10.0 points at 6.1% additional latency.

Table 13: MDFI with different static analyzers on LLMSecEval+ (Dream-Coder 7B).

Analyzer	sec@1 (%)	fs@1 (%)	Δfs@1 vs. vanilla	Latency vs. vanilla
None (vanilla, no MDFI)	54.7	14.7	–	–
Bandit	62.7	17.3	+2.6	+36%
Semgrep	62.7	17.3	+2.6	+288%
CodeQL	54.7	14.7	0.0	timed out
Our partial-program analyzer	**80.7**	**24.7**	**+10.0**	**+6.1%**

D.4 Localization on the Program Graph

For each witness w_k, the localization map $\mathcal{M}^{\mathrm{MDFI}}$ lifts the offending node n_k to a coherent token region by walking $\mathcal{G}_t$ along its surrounding AST and dataflow neighborhood. The neighborhood includes the smallest enclosing statement node, the immediately neighboring AST nodes around n_k, and the dataflow-adjacent nodes that define identifiers used by n_k within a small dataflow radius. The set of structurally adjacent nodes $\mathcal{N}(n_k; \mathcal{G}_t)$ is then projected back to token positions:

$$N(w_k) \;=\; \mathrm{Tok}\big(\mathcal{N}(n_k; \mathcal{G}_t)\big) \;\subseteq\; [L]. \tag{35}$$

The full editable region is the union over witnesses, capped by a token budget B that bounds the total fraction of the program that may be revised at any single checkpoint:

$$\mathcal{S}_t \;=\; \mathrm{TopBudget}_B\Big(\bigcup_{k=1}^{K_t} N(w_k)\Big) \;\subseteq\; [L]. \tag{36}$$

When $\big|\bigcup_k N(w_k)\big| > B$, $\mathrm{TopBudget}_B$ retains the witnesses with the highest analyzer confidence first; ties are broken at the granularity of AST statements so that the resulting localization preserves syntactic coherence.

D.5 Substitute Remasking

For witnesses with $\tau_k = \mathrm{sub}$, the offending construct is already present in the partial program in the wrong form—for instance, an unsafe API call, an unsafe constructor, or a hard-coded credential. The operator opens these positions for re-denoising by setting them back to the mask token. Let

$$\mathcal{S}_t^{\mathrm{sub}} \;=\; \mathcal{S}_t \;\cap \bigcup_{k:\tau_k=\mathrm{sub}} N(w_k). \tag{37}$$

The substitute operation produces

$$x_t^{\star,i} \;=\; \begin{cases} \texttt{[MASK]}, & i \in \mathcal{S}_t^{\mathrm{sub}}, \\ x_t^i, & i \notin \mathcal{S}_t^{\mathrm{sub}}. \end{cases} \tag{38}$$

[3] `https://codeql.github.com`

[4] `https://semgrep.dev`

[5] `https://github.com/PyCQA/bandit`

This is the same form as the standard partial-mask state defined in Section 5.1, with the localization set anchored on a witness node rather than on a heuristic score.

D.6 Mask Insertion

For witnesses with $\tau_k = \mathrm{ins}$, the bug is the *absence* of a needed construct—for example, a missing input-validation guard, a missing length check, or a missing exception handler. Re-denoising existing tokens cannot resolve these failures because the tokens that should be there do not yet exist in $\mathbf{x}_t$. The operator instead allocates K fresh mask positions adjacent to a structurally chosen anchor in $N(w_k)$ and splices them into the partial-mask state:

$$\mathbf{x}_t^\star \;\leftarrow\; \mathrm{Insert}_K\big(\mathbf{x}_t^\star,\; \mathrm{anchor}(w_k)\big), \qquad k : \tau_k = \mathrm{ins}. \tag{39}$$

The anchor is chosen so that the inserted region admits a syntactically coherent infill—typically immediately before the offending sink node, immediately after a relevant assignment, or at the start of the offending block. The insertion extends the sequence length by K and right-shifts the downstream committed tokens to preserve syntactic context. The localization set $\mathcal{S}_t^{\mathrm{sub}}$ used by Eq. 38 is updated to reference the post-insertion sequence whenever both substitute and insert witnesses are present in $\mathcal{W}_t$.

D.7 Pre-Allocated Prompt Buffer

To prevent the next denoiser pass from regenerating the same vulnerable construct under unchanged conditioning, MDFI augments the conditioning context c with the analyzer's remediation hints. At the start of every trajectory we pre-allocate a contiguous mask buffer of fixed length B_p inside the prompt context, immediately following the task description. This buffer is part of the model's input but is flagged as conditioning rather than as part of the output sequence; the constrained reverse kernel never samples into it, and base-model attention over fixed prompt positions is unaffected by it.

At each fired checkpoint, the structured hints $\{h_k\}$ in r_t are tokenized into a natural-language remediation message $\rho_t(r_t)$ that names the violated property and the recommended safe pattern, and $\rho_t(r_t)$ is written into the buffer slots, displacing leading mask tokens. When multiple checkpoints fire along a trajectory, the most recent message overwrites any previously injected message in the same buffer:

$$c^\star \;=\; \mathrm{Write}\big(c,\; \mathrm{Buffer}_{B_p},\; \rho_t(r_t)\big). \tag{40}$$

Because $|c^\star| = |c|$ by construction, no token positions inside the program $\mathbf{x}_t^\star$ are displaced by feedback injection. When the message is shorter than B_p, trailing buffer slots remain mask tokens and the model can still infer "no further hint" from that region.

D.8 Checkpoint Schedule and Composition

The MDFI operator is invoked at a predetermined schedule of denoising checkpoints

$$\mathcal{T}^{\mathrm{ck}} \;\subseteq\; \{1, \ldots, T\}, \tag{41}$$

gated by a minimum committed-fraction $\rho_{\min} \in [0, 1)$:

$$\frac{\big|\{i \in [L] : x_t^i \neq \texttt{[MASK]}\}\big|}{L} \;\geq\; \rho_{\min}. \tag{42}$$

This guarantees that the analyzer is consulted only once enough of $\bar{\mathbf{x}}_0^{(t)}$ has been decoded for the partial program graph $\mathcal{G}_t$ to be informative. A global intervention budget B_{int} caps the total number of fired checkpoints along any single trajectory; once exhausted, the operator collapses to the identity for the rest of the chain.

When a checkpoint at step t is active, the operator combines the substitute remask of Eq. 38, the mask insertion of Eq. 39, and the buffer write of Eq. 40 into a single output:

$$\big(\mathbf{x}_t^\star,\, c^\star,\, \mathcal{S}_t\big) \;=\; \mathcal{P}_{\mathcal{C}}^{\mathrm{MDFI}}\big(\hat{\mathbf{x}}_0^{(t)},\, \mathcal{S}_t,\, r_t,\, c\big). \tag{43}$$

The chain then advances from the modified state and conditioning under the constrained reverse kernel of Eq. 8: $\mathbf{x}_{t-1} \sim p_\theta^{\mathcal{C}}(\,\cdot \mid \mathbf{x}_t^\star, \hat{\mathbf{x}}_0^{(t)}, \mathcal{S}_t;\, c^\star)$. Crucially, MDFI does not run an inner re-denoising

loop on the reopened region; it modifies the state and conditioning in place and proceeds directly to the next reverse step. As a result, the freshly masked positions are refilled by the remaining t standard transitions under feedback-aware conditioning, and the total number of reverse transitions along the trajectory is identical to vanilla diffusion.

NeurIPS Paper Checklist

1. **Claims**

   Question: Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope?

   Answer: [Yes]

   Justification: The abstract and introduction (Section 5 and the contributions list in Section 1) state three claims: (i) constraint-aware localization for diffusion code models, (ii) constraint-aware correction operators (GradGuide, MDFI), and (iii) empirical gains on functionality and security benchmarks with more localized edits.

   Guidelines:

   - The answer [N/A] means that the abstract and introduction do not include the claims made in the paper.
   - The abstract and/or introduction should clearly state the claims made, including the contributions made in the paper and important assumptions and limitations. A [No] or [N/A] answer to this question will not be perceived well by the reviewers.
   - The claims made should match theoretical and experimental results, and reflect how much the results can be expected to generalize to other settings.
   - It is fine to include aspirational goals as motivation as long as it is clear that these goals are not attained by the paper.

2. **Limitations**

   Question: Does the paper discuss the limitations of the work performed by the authors?

   Answer: [Yes]

   Justification: Limitations are explicitly discussed in Section 7.

   Guidelines:

   - The answer [N/A] means that the paper has no limitation while the answer [No] means that the paper has limitations, but those are not discussed in the paper.
   - The authors are encouraged to create a separate "Limitations" section in their paper.
   - The paper should point out any strong assumptions and how robust the results are to violations of these assumptions (e.g., independence assumptions, noiseless settings, model well-specification, asymptotic approximations only holding locally). The authors should reflect on how these assumptions might be violated in practice and what the implications would be.
   - The authors should reflect on the scope of the claims made, e.g., if the approach was only tested on a few datasets or with a few runs. In general, empirical results often depend on implicit assumptions, which should be articulated.
   - The authors should reflect on the factors that influence the performance of the approach. For example, a facial recognition algorithm may perform poorly when image resolution is low or images are taken in low lighting. Or a speech-to-text system might not be used reliably to provide closed captions for online lectures because it fails to handle technical jargon.
   - The authors should discuss the computational efficiency of the proposed algorithms and how they scale with dataset size.
   - If applicable, the authors should discuss possible limitations of their approach to address problems of privacy and fairness.
   - While the authors might fear that complete honesty about limitations might be used by reviewers as grounds for rejection, a worse outcome might be that reviewers discover limitations that aren't acknowledged in the paper. The authors should use their best judgment and recognize that individual actions in favor of transparency play an important role in developing norms that preserve the integrity of the community. Reviewers will be specifically instructed to not penalize honesty concerning limitations.

3. **Theory assumptions and proofs**

   Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?

Answer: [N/A]

Justification: The paper does not include theoretical results.

Guidelines:

- The answer [N/A] means that the paper does not include theoretical results.
- All the theorems, formulas, and proofs in the paper should be numbered and cross-referenced.
- All assumptions should be clearly stated or referenced in the statement of any theorems.
- The proofs can either appear in the main paper or the supplemental material, but if they appear in the supplemental material, the authors are encouraged to provide a short proof sketch to provide intuition.
- Inversely, any informal proof provided in the core of the paper should be complemented by formal proofs provided in appendix or supplemental material.
- Theorems and Lemmas that the proof relies upon should be properly referenced.

4. **Experimental result reproducibility**

   Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper (regardless of whether the code and data are provided or not)?

   Answer: [Yes]

   Justification: We provide algorithm, experimental and implementation details at Section 5, Section 6, Appendix B and Appendix C.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - If the paper includes experiments, a [No] answer to this question will not be perceived well by the reviewers: Making the paper reproducible is important, regardless of whether the code and data are provided or not.
   - If the contribution is a dataset and/or model, the authors should describe the steps taken to make their results reproducible or verifiable.
   - Depending on the contribution, reproducibility can be accomplished in various ways. For example, if the contribution is a novel architecture, describing the architecture fully might suffice, or if the contribution is a specific model and empirical evaluation, it may be necessary to either make it possible for others to replicate the model with the same dataset, or provide access to the model. In general. releasing code and data is often one good way to accomplish this, but reproducibility can also be provided via detailed instructions for how to replicate the results, access to a hosted model (e.g., in the case of a large language model), releasing of a model checkpoint, or other means that are appropriate to the research performed.
   - While NeurIPS does not require releasing code, the conference does require all submissions to provide some reasonable avenue for reproducibility, which may depend on the nature of the contribution. For example
     (a) If the contribution is primarily a new algorithm, the paper should make it clear how to reproduce that algorithm.
     (b) If the contribution is primarily a new model architecture, the paper should describe the architecture clearly and fully.
     (c) If the contribution is a new model (e.g., a large language model), then there should either be a way to access this model for reproducing the results or a way to reproduce the model (e.g., with an open-source dataset or instructions for how to construct the dataset).
     (d) We recognize that reproducibility may be tricky in some cases, in which case authors are welcome to describe the particular way they provide for reproducibility. In the case of closed-source models, it may be that access to the model is limited in some way (e.g., to registered users), but it should be possible for other researchers to have some path to reproducing or verifying the results.

5. **Open access to data and code**

Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

Answer: [Yes]

Justification: We use publicly available datasets and models. Information to faithfully reproduce the experimental results is included in Section 6, and Appendix B and Appendix C. Code will be released following the review process.

Guidelines:

- The answer [N/A] means that paper does not include experiments requiring code.
- Please see the NeurIPS code and data submission guidelines (`https://neurips.cc/public/guides/CodeSubmissionPolicy`) for more details.
- While we encourage the release of code and data, we understand that this might not be possible, so [No] is an acceptable answer. Papers cannot be rejected simply for not including code, unless this is central to the contribution (e.g., for a new open-source benchmark).
- The instructions should contain the exact command and environment needed to run to reproduce the results. See the NeurIPS code and data submission guidelines (`https://neurips.cc/public/guides/CodeSubmissionPolicy`) for more details.
- The authors should provide instructions on data access and preparation, including how to access the raw data, preprocessed data, intermediate data, and generated data, etc.
- The authors should provide scripts to reproduce all experimental results for the new proposed method and baselines. If only a subset of experiments are reproducible, they should state which ones are omitted from the script and why.
- At submission time, to preserve anonymity, the authors should release anonymized versions (if applicable).
- Providing as much information as possible in supplemental material (appended to the paper) is recommended, but including URLs to data and code is permitted.

6. **Experimental setting/details**

Question: Does the paper specify all the training and test details (e.g., data splits, hyperparameters, how they were chosen, type of optimizer) necessary to understand the results?

Answer: [Yes]

Justification: Please see Appendix B and C.

Guidelines:

- The answer [N/A] means that the paper does not include experiments.
- The experimental setting should be presented in the core of the paper to a level of detail that is necessary to appreciate the results and make sense of them.
- The full details can be provided either with the code, in appendix, or as supplemental material.

7. **Experiment statistical significance**

Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

Answer: [Yes]

Justification: Please refer to Appendix B and Appendix C.

Guidelines:

- The answer [N/A] means that the paper does not include experiments.
- The authors should answer [Yes] if the results are accompanied by error bars, confidence intervals, or statistical significance tests, at least for the experiments that support the main claims of the paper.
- The factors of variability that the error bars are capturing should be clearly stated (for example, train/test split, initialization, random drawing of some parameter, or overall run with given experimental conditions).

- The method for calculating the error bars should be explained (closed form formula, call to a library function, bootstrap, etc.)
- The assumptions made should be given (e.g., Normally distributed errors).
- It should be clear whether the error bar is the standard deviation or the standard error of the mean.
- It is OK to report 1-sigma error bars, but one should state it. The authors should preferably report a 2-sigma error bar than state that they have a 96% CI, if the hypothesis of Normality of errors is not verified.
- For asymmetric distributions, the authors should be careful not to show in tables or figures symmetric error bars that would yield results that are out of range (e.g., negative error rates).
- If error bars are reported in tables or plots, the authors should explain in the text how they were calculated and reference the corresponding figures or tables in the text.

8. **Experiments compute resources**

   Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

   Answer: [Yes]

   Justification: The one-time training cost of the GradGuide surrogate is reported in Appendix C.2; inference latency, peak GPU memory, and surrogate and analyzer overhead are reported in Section 6.4 and Appendix B.3.2.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The paper should indicate the type of compute workers CPU or GPU, internal cluster, or cloud provider, including relevant memory and storage.
   - The paper should provide the amount of compute required for each of the individual experimental runs as well as estimate the total compute.
   - The paper should disclose whether the full research project required more compute than the experiments reported in the paper (e.g., preliminary or failed experiments that didn't make it into the paper).

9. **Code of ethics**

   Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics `https://neurips.cc/public/EthicsGuidelines`?

   Answer: [Yes]

   Justification: Yes, we conform to the NeurIPS Code of Ethics in every respect.

   Guidelines:

   - The answer [N/A] means that the authors have not reviewed the NeurIPS Code of Ethics.
   - If the authors answer [No], they should explain the special circumstances that require a deviation from the Code of Ethics.
   - The authors should make sure to preserve anonymity (e.g., if there is a special consideration due to laws or regulations in their jurisdiction).

10. **Broader impacts**

    Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

    Answer: [Yes]

    Justification: The Broader Impact section discusses the potential positive impacts on code reliability and security, as well as the potential dual-use risks of stronger constraint-guided code generation.

    Guidelines:

    - The answer [N/A] means that there is no societal impact of the work performed.

- If the authors answer [N/A] or [No], they should explain why their work has no societal impact or why the paper does not address societal impact.
- Examples of negative societal impacts include potential malicious or unintended uses (e.g., disinformation, generating fake profiles, surveillance), fairness considerations (e.g., deployment of technologies that could make decisions that unfairly impact specific groups), privacy considerations, and security considerations.
- The conference expects that many papers will be foundational research and not tied to particular applications, let alone deployments. However, if there is a direct path to any negative applications, the authors should point it out. For example, it is legitimate to point out that an improvement in the quality of generative models could be used to generate Deepfakes for disinformation. On the other hand, it is not needed to point out that a generic algorithm for optimizing neural networks could enable people to train models that generate Deepfakes faster.
- The authors should consider possible harms that could arise when the technology is being used as intended and functioning correctly, harms that could arise when the technology is being used as intended but gives incorrect results, and harms following from (intentional or unintentional) misuse of the technology.
- If there are negative societal impacts, the authors could also discuss possible mitigation strategies (e.g., gated release of models, providing defenses in addition to attacks, mechanisms for monitoring misuse, mechanisms to monitor how a system learns from feedback over time, improving the efficiency and accessibility of ML).

11. **Safeguards**

    Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse (e.g., pre-trained language models, image generators, or scraped datasets)?

    Answer: [N/A]

    Justification: This work does not introduce any datasets or models that pose substantial misuse risks.

    Guidelines:

    - The answer [N/A] means that the paper poses no such risks.
    - Released models that have a high risk for misuse or dual-use should be released with necessary safeguards to allow for controlled use of the model, for example by requiring that users adhere to usage guidelines or restrictions to access the model or implementing safety filters.
    - Datasets that have been scraped from the Internet could pose safety risks. The authors should describe how they avoided releasing unsafe images.
    - We recognize that providing effective safeguards is challenging, and many papers do not require this, but we encourage authors to take this into account and make a best faith effort.

12. **Licenses for existing assets**

    Question: Are the creators or original owners of assets (e.g., code, data, models), used in the paper, properly credited and are the license and terms of use explicitly mentioned and properly respected?

    Answer: [Yes]

    Justification: Appropriate credit has been given to the creators or original owners of all assets.

    Guidelines:

    - The answer [N/A] means that the paper does not use existing assets.
    - The authors should cite the original paper that produced the code package or dataset.
    - The authors should state which version of the asset is used and, if possible, include a URL.
    - The name of the license (e.g., CC-BY 4.0) should be included for each asset.

- For scraped data from a particular source (e.g., website), the copyright and terms of service of that source should be provided.
- If assets are released, the license, copyright information, and terms of use in the package should be provided. For popular datasets, `paperswithcode.com/datasets` has curated licenses for some datasets. Their licensing guide can help determine the license of a dataset.
- For existing datasets that are re-packaged, both the original license and the license of the derived asset (if it has changed) should be provided.
- If this information is not available online, the authors are encouraged to reach out to the asset's creators.

13. **New assets**

    Question: Are new assets introduced in the paper well documented and is the documentation provided alongside the assets?

    Answer: [N/A]

    Justification: No new assets are released in this work.

    Guidelines:

    - The answer [N/A] means that the paper does not release new assets.
    - Researchers should communicate the details of the dataset/code/model as part of their submissions via structured templates. This includes details about training, license, limitations, etc.
    - The paper should discuss whether and how consent was obtained from people whose asset is used.
    - At submission time, remember to anonymize your assets (if applicable). You can either create an anonymized URL or include an anonymized zip file.

14. **Crowdsourcing and research with human subjects**

    Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation (if any)?

    Answer: [N/A]

    Justification: This paper does not involve crowdsourcing, nor does it include research involving human subjects.

    Guidelines:

    - The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
    - Including this information in the supplemental material is fine, but if the main contribution of the paper involves human subjects, then as much detail as possible should be included in the main paper.
    - According to the NeurIPS Code of Ethics, workers involved in data collection, curation, or other labor should be paid at least the minimum wage in the country of the data collector.

15. **Institutional review board (IRB) approvals or equivalent for research with human subjects**

    Question: Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board (IRB) approvals (or an equivalent approval/review based on the requirements of your country or institution) were obtained?

    Answer: [N/A]

    Justification: This paper does not involve crowdsourcing, nor does it include research involving human subjects.

    Guidelines:

    - The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.

- Depending on the country in which research is conducted, IRB approval (or equivalent) may be required for any human subjects research. If you obtained IRB approval, you should clearly state this in the paper.
- We recognize that the procedures for this may vary significantly between institutions and locations, and we expect authors to adhere to the NeurIPS Code of Ethics and the guidelines for their institution.
- For initial submissions, do not include any information that would break anonymity (if applicable), such as the institution conducting the review.

16. **Declaration of LLM usage**

    Question: Does the paper describe the usage of LLMs if it is an important, original, or non-standard component of the core methods in this research? Note that if the LLM is used only for writing, editing, or formatting purposes and does *not* impact the core methodology, scientific rigor, or originality of the research, declaration is not required.

    Answer: [Yes]

    Justification: The core method development in this research does not involve LLMs as any important, original, or non-standard components.

    Guidelines:

    - The answer [N/A] means that the core method development in this research does not involve LLMs as any important, original, or non-standard components.
    - Please refer to our LLM policy in the NeurIPS handbook for what should or should not be described.