TU Delft Delft University of Technology

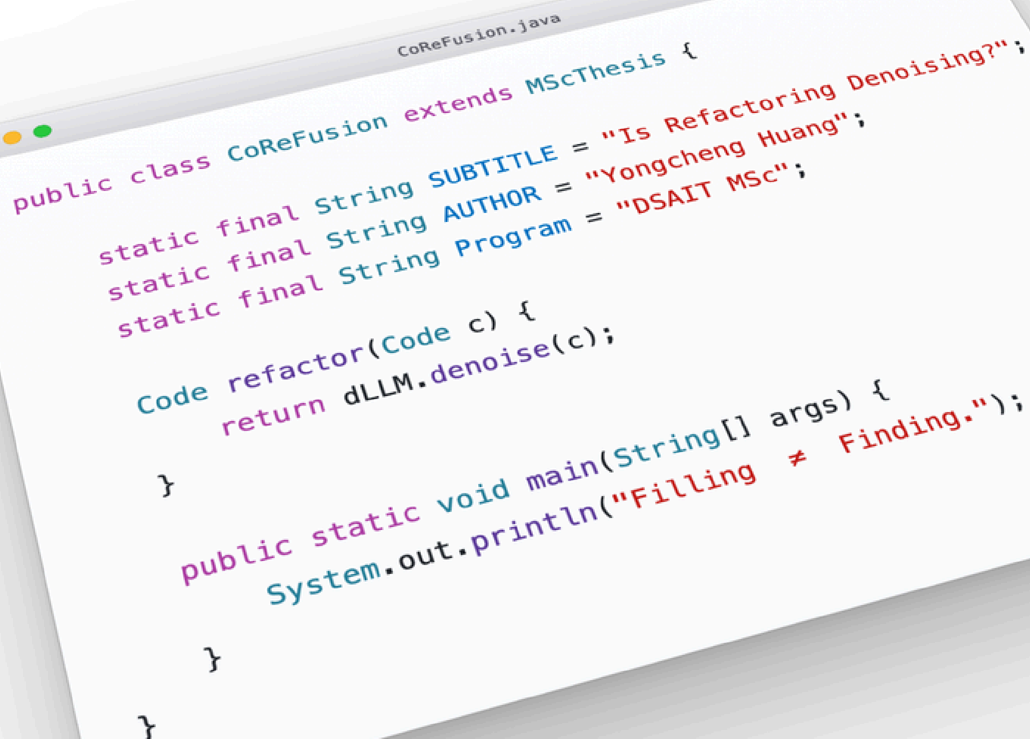
CoReFusion

Refactoring Identifier Names with Diffusion Language Models

Name: Yongcheng Huang

Supervisor: Arie van Deursen, Maliheh Izadi, Jonathan Katzy

Defense Date: June.3rd, 2026



CoReFusion: Refactoring Identifier Names with Diffusion Language Models

Version of May 26, 2026

Yongcheng Huang

CoReFusion: Refactoring Identifier Names with Diffusion Language Models

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

Data Science and Artificial Intelligence Technology

by

Yongcheng Huang
born in Shanghai, China

Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

©2026 Yongcheng Huang. All rights reserved.

CoReFusion: Refactoring Identifier Names with Diffusion Language Models

Author: Yongcheng Huang
Student id: 5560950
Email: Y.Huang-51@student.tudelft.nl

Abstract

Refactoring is a critical part of the software development lifecycle, and identifier renaming accounts for roughly 15% of all agentic refactoring work driven by large language models. Yet the dominant model families fit the task poorly. Autoregressive decoders generate left to right, and even with the fill-in-the-middle extension they resolve masked positions one at a time, so a renaming decision at one site cannot inform a decision at another. Identifier renaming, however, demands consistency across every affected site at once. Diffusion Large Language Models (dLLMs) generate by iteratively denoising a masked sequence under full bidirectional attention, with every prediction conditioned on every other. This matches what renaming needs: if a poorly named identifier is viewed as a small amount of semantic noise overlaid on correct code, then renaming becomes a targeted denoising task that can be solved jointly across all affected sites.

We instantiate this view as CoReFusion, the first systematic study of dLLMs on Java identifier renaming, and benchmark them against twelve decoder-only FIM-AR baselines and five encoder-decoder Seq2Seq baselines on the RefineID dataset. DreamCoder-7B and DiffuCoder-7B reach 33.2% and 31.1% Exact Match, beating the best non-dLLM model (CodeT5-large) by more than ten points while being roughly nine times smaller than the largest FIM-AR baseline. The advantage grows with the number of identifiers that must be renamed together: FIM-AR models win the single-site case, but dLLMs pull ahead as soon as the task involves more than one site. When the same dLLMs must instead find the positions on their own, Exact Match drops to about 3%, and most wrong predictions copy the lexical style of the surrounding code rather than improve on it. Probing the internal states of DiffuCoder-7B shows why: the signal that tells a bad name from a good one appears only in the last few layers and the last few denoising steps, after the unmasking schedule has already confirmed most of its predictions. Providing the rename positions as masks bypasses this timing problem, which is why dLLMs work as filling engines but not as standalone refactoring agents.

Thesis Committee:

Chair:	Professor Dr. A. van Deursen, Faculty EEMCS, TU Delft
Daily supervisor:	Professor Dr. M. Izadi, Faculty EEMCS, TU Delft Assistant
Daily co-supervisor:	Ir. J. Katzy, Faculty EEMCS, TUDelft
Committee Member:	Professor Dr. P. Kellnhofer, Faculty EEMCS, TU Delft Assistant

Preface

This thesis was written for the completion of the degree of Master of Science in Data Science and Artificial Intelligence Technology at the Delft University of Technology. I would like to thank the people who have made it possible for this thesis to be completed by giving guidance during the process.

Arie van Deursen, Maliheh Izadi, and Jonathan Katzy, I would like to thank you for the guidance during the process of creating this thesis, the assistance in paper writing, in research skills cultivating, and in the way of doing research.

Yongcheng Huang
Delft, The Netherlands
May 26, 2026

CoReFusion: Refactoring Identifier Names with Diffusion Large Language Models

1st Yongcheng Huang
Faculty Electrical Engineering, Mathematics and Computer Science
Delft University of Technology
Delft, The Netherlands
Y.Huang-51@student.tudelft.nl

***Abstract*—Refactoring is a critical part of the software development lifecycle, and identifier renaming accounts for roughly 15% of all agentic refactoring work driven by large language models. Yet the dominant model families fit the task poorly. Autoregressive decoders generate left to right, and even with the fill-in-the-middle extension they resolve masked positions one at a time, so a renaming decision at one site cannot inform a decision at another. Identifier renaming, however, demands consistency across every affected site at once. Diffusion Large Language Models (dLLMs) generate by iteratively denoising a masked sequence under full bidirectional attention, with every prediction conditioned on every other. This matches what renaming needs: if a poorly named identifier is viewed as a small amount of semantic noise overlaid on correct code, then renaming becomes a targeted denoising task that can be solved jointly across all affected sites.**

We instantiate this view as CoReFusion, the first systematic study of dLLMs on Java identifier renaming, and benchmark them against twelve decoder-only FIM-AR baselines and five encoder-decoder Seq2Seq baselines on the RefineID dataset. DreamCoder-7B and DiffuCoder-7B reach 33.2% and 31.1% Exact Match, beating the best non-dLLM model (CodeT5-large) by more than ten points while being roughly nine times smaller than the largest FIM-AR baseline. The advantage grows with the number of identifiers that must be renamed together: FIM-AR models win the single-site case, but dLLMs pull ahead as soon as the task involves more than one site. When the same dLLMs must instead find the positions on their own, Exact Match drops to about 3%, and most wrong predictions copy the lexical style of the surrounding code rather than improve on it. Probing the internal states of DiffuCoder-7B shows why: the signal that tells a bad name from a good one appears only in the last few layers and the last few denoising steps, after the unmasking schedule has already confirmed most of its predictions. Providing the rename positions as masks bypasses this timing problem, which is why dLLMs work as filling engines but not as standalone refactoring agents.

***Index Terms*—Identifier renaming, Diffusion large language models, Code refactoring, Large language models, Fill-in-the-middle**

I. Introduction

Identifiers are the linguistic backbone of source code: they account for up to 70% of the characters in a typical project [17] and largely determine whether a future reader can understand what the code does. Yet developers routinely encounter identifiers whose names no longer fit the role they play, such as a variable called `tmp` that survives into production, or a name that meant something to its author but nothing to anyone else. Therefore, renaming identifiers is one of the frequently mentioned tasks in the software development lifecycle [10].

A renaming refactoring has two steps: *deciding* what the new name should be and *applying* the rename consistently across every site where the identifier appears. Modern IDEs have solved the second step well by using grammar-based methods. However, the first step is left to the developer. Static-analysis tools can guarantee grammar safety but cannot infer a contextually appropriate name because they do not model what the surrounding code is doing [9]. The problem we study is therefore not how to apply a rename, but how to automatically propose a name that fits the surrounding code.

Recent work has begun to address this problem with language models. Autoregressive (AR) decoder-only models with Fill-in-the-Middle (FIM) training [4] can generate replacement identifiers conditioned on both left and right context, and they outperform earlier encoder-only and encoder-decoder approaches on renaming benchmarks [7]. However, AR models have a structural limitation which will become visible when a refactoring touches more than one position. FIM resolves masked positions one at a time, inserting each prediction into the prefix before the next decision is made. A name chosen at one site therefore cannot inform the decision at another site that will be predicted later, and the decisions at sites predicted earlier cannot be revised based on what came later.

For identifier renaming this matters in practice, because a typical rename touches many co-occurring sites that all need to agree on one name. We are therefore interested in models whose inference protocol can resolve all sites jointly. Diffusion Large Language Models (dLLMs) [19], [40] propose a new structure that generate by iteratively denoising a fully masked sequence under bidirectional attention, which allows the model to commit to high-confidence positions first and propagate that information to other positions in the same forward pass. If a poorly named identifier is viewed as semantic noise overlaid on otherwise correct code [24], then renaming becomes a targeted denoising task. We instantiate this view as **CoReFusion** (**Co**de **Re**factoring with Dif**Fusion** language models).

As the first study to use dLLMs for identifier renaming, we are interested both in what dLLMs can do on this task and in how they do it. We ask three research questions:

- **RQ1:** How do dLLMs perform on multi-site renaming of a single identifier, compared against autoregressive and

sequence-to-sequence baselines?

- **RQ2:** Does the same advantage extend to multi-site renaming of multiple distinct identifiers at once?
- **RQ3:** What, if anything, distinguishes the internal state of a dLLM when it processes a poorly named identifier from when it processes a well-named one?

Our findings are threefold. On RQ1, two fixed-canvas dLLMs (DreamCoder-7B and DiffuCoder-7B) reach 33.2% and 31.1% Exact Match on the RefineID test split, against 23.5% for the strongest of seventeen non-dLLM baselines and despite being roughly $9\times$ smaller than the largest baseline. The gap widens as the number of co-referring sites grows: FIM-AR baselines win at one site but fall to single digits past eleven, while the dLLMs hold a 26.8%-36.4% band across the entire multi-site range. On RQ2, the dLLM advantage does not extend to multi-identifier renaming under aggressive obfuscation: target-EM falls to single digits and 71.3% of wrong predictions copy the lexical style of the obfuscated neighbours, sharpening rather than overturning the RQ1 result. On RQ3, four probes of DiffuCoder-7B's internal state show that a representation distinguishing well-named from poorly-named identifiers exists, but it is concentrated in the final third of the transformer stack and the final third of the denoising trajectory, after the model's confidence-ordered unmasking schedule has typically confirmed most of its decisions.

II. Background and Related Work

A. Identifier Renaming in software refactoring

Software refactoring aims to improve the internal structure of a codebase without altering its external behavior [10]. Among refactoring operations, identifier renaming is one of the most frequent. Also, it is shown to be hard to find a good name. A good renaming requires understanding the scope of the variable, the intent, and how the identifier is used in multiple functions and files.

Early tool support for renaming came from rule-based or grammar-driven features built into IDEs such as Eclipse and IntelliJ. These features use static scoping rules derived from language grammars and AST analysis to locate all usage sites of an identifier and rename them consistently [23]. These tools guarantee that the rename is safe under the rules of the grammar, but the matching itself is purely mechanical: they can propagate a renaming the developer has already chosen, but cannot suggest what the new name should be. They are mature, well-supported, and widely deployed in production IDEs, and our work does not aim to replace the safe-propagation mechanism they already provide. We instead focus on the complementary problem they leave unaddressed, which is deciding which name should replace the existing one in the first place, and this decision requires semantic reasoning.

To address this gap, machine learning approaches were introduced to recommend contextually appropriate names. Early work applied statistical language models and n-gram methods to learn naming conventions from corpora [29]. Subsequent representation-learning methods such as CodeBERT [11] and RefBERT [21] learned to encode identifier context for name recommendation. These methods treat renaming as a retrieval or classification problem, as machine learning works like a prediction of suitable name based on the given surrounding code. However, empirical studies show that they still struggle when a single identifier has many usage sites throughout the code, so manual renaming remains tedious and prone to error in practice [23].

More recently, AR LLMs have been applied to refactoring, using their semantic reasoning to produce names that reflect developer intent more accurately than rule-based predecessors [7], [12]. However, they still not escape the architectural limitation discussed. A renaming must stay consistent across many usage sites, but AR models decide each site sequentially one at a time.

B. Language Models: Autoregressive, Seq2Seq, and Diffusion

1) Autoregressive Language Models: AR language models generate text by factorizing the joint probability of a sequence into a product of conditionals. Given a sequence $\mathbf{x} = (x_1, x_2, \ldots, x_T)$,

$$p(\mathbf{x}) = \prod_{t=1}^{T} p(x_t \mid x_1, x_2, \ldots, x_{t-1}) \tag{1}$$

predicting each token given all preceding tokens [5]. This left-to-right (L2R) paradigm underpins models such as the GPT family [25], CodeLlama [26], and DeepSeek-Coder [14]. However, there is a fundamental constraint that each token is generated conditioned only on its left context, so the model cannot directly attend to tokens that appear later in the sequence [2].

For variable renaming this constraint matters. A good renaming requires looking at the identifier's declaration and all of its usage sites. This is a pattern of access that strict L2R generation does not natively support [23]. FIM was introduced as a practical workaround [4]. Given a sequence split into a prefix P, a middle span M. and a suffix S, FIM rearranges the training example into $\langle \texttt{PRE} \rangle \, P \, \langle \texttt{SUF} \rangle \, S \, \langle \texttt{MID} \rangle \, M$, so that L2R training on the rearranged sequence implicitly teaches the model to fill M conditioned on both surrounding contexts. Models trained with FIM objectives, such as StarCoder [20] and CodeLlama [26], perform well on infilling benchmarks, but still lack consistency. When several positions need to be filled, each position is still resolved independently. The renaming decision at one site cannot inform a decision at another, and earlier predictions cannot be revised in light of later context [41].

2) Sequence-to-Sequence Models: Seq2Seq models factor generation into two stages. A bidirectional encoder maps the full input sequence $\mathbf{x}$ into a sequence of contextual representations, and an autoregressive decoder then produces the output sequence $\mathbf{y} = (y_1, \ldots, y_{T'})$ token by token under

$$p(\mathbf{y} \mid \mathbf{x}) = \prod_{t=1}^{T'} p(y_t \mid y_1, \ldots, y_{t-1}, \mathrm{enc}(\mathbf{x})), \tag{2}$$

Fig. 1. Iterative denoising in a diffusion LLM applied to multi-site variable renaming.

where $\text{enc}(\mathbf{x})$ is the encoder output [31]. This architecture underpins the original Transformer [33], BART [18], and code-specific variants such as CodeT5 [36] and CodeT5+ [35], which are pretrained with span-denoising objectives that directly match identifier infilling: a sentinel token replaces the target span in the input, and the decoder is trained to emit the original span as the output.

For identifier renaming, this architecture removes one of the limitations of AR generation. The encoder reads the entire source file in a single forward pass under full bidirectional attention, so the representation of a target position attends to both its declaration and all of its usage sites simultaneously [36]. Decoding the replacement name therefore conditions on the full context, not just the left side.

The remaining limitation is that, like FIM-AR, Seq2Seq models resolve one masked position at a time. Each masked position is decoded independently, and earlier predictions are inserted into the input before the next one is queried. When several identifier sites must be renamed in the same file, the model still cannot make the renaming decision at one site condition directly on the decision at another. The bidirectional encoder gives Seq2Seq a stronger context view than FIM-AR for any single span, but joint multi-site resolution remains outside the architecture.

3) Diffusion Large Language Models: dLLMs approach text generation through a different paradigm. They define a forward process that progressively replaces tokens in a target sequence with a special `<|mask|>` symbol, and train a neural network to reverse this process through iterative denoising [1], [28]. At each denoising step t, the model conditions on the full corrupted sequence and predicts the original tokens at all masked positions simultaneously:

$$p_\theta(\mathbf{x}_0 \mid \mathbf{x}_t) = \prod_{i \in \mathcal{M}_t} p_\theta(x_i \mid \mathbf{x}_t) \tag{3}$$

where $\mathcal{M}_t$ is the set of masked positions at step t. In practice, tokens are unmasked in order of decreasing confidence. The model scores all masked positions, commits the highest-confidence predictions, and leaves the rest for subsequent steps [6], [27].

The forward process above is similar to the objectives of encoder-only and encoder-decoder models such as BERT [8] and BART [18]. However, three properties distinguish dLLMs from BERT-style models in ways that matter for multi-site infilling. First, BERT models are trained with a small

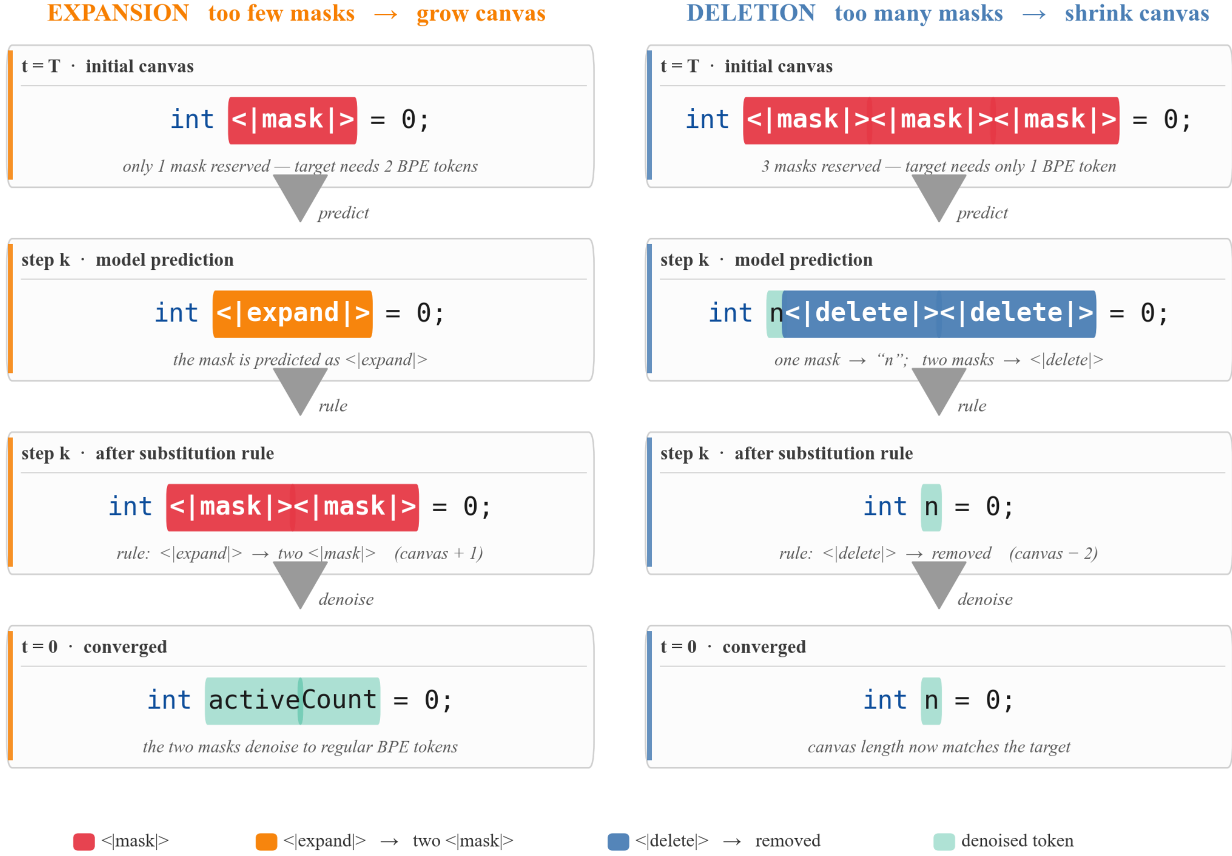


Fig. 2. Variable-canvas denoising in DreamOn.

fixed corruption ratio, whereas dLLMs are trained over the full range of corruption ratios and remain calibrated when most positions are masked. Second, BERT models assume the masked positions are conditionally independent and predict them all in a single forward pass, while dLLMs factor the joint distribution into many sequential steps and commit only the predictions with highest confidence at each step. Each subsequent prediction conditions on the tokens filled in earlier. Third, BERT inference has no equivalent to this confidence-ordered unmasking, and it is the precise mechanism that lets a dLLM resolve many masked sites in the same trajectory.

Figure 1 illustrates this process applied to a variable renaming task. All tokens to be renamed are initialized as `<|mask|>` at $t = T$. The model resolves high-confidence positions first until all positions are resolved at $t = 0$. This confidence-ordered unmasking is a defining property of dLLMs and forms the basis of the analysis in RQ3 (Section VI).

One practical constraint of the masked diffusion paradigm described so far is that the canvas size at each identifier site, namely the number k of `<|mask|>` tokens allocated for that single name as shown in Figure 3, must be fixed before inference begins [39]. For identifier renaming this matters in practice: a developer-chosen name may tokenize into anywhere from one to several sub-words, so a canvas that is too short cannot represent multi-token names, while one that is too long forces the model to commit to filler tokens that do not exist in the ground truth. We characterize this trade-off empirically in Section IV-C.

DreamOn [39], a follow-up to DreamCoder, removes the fixed-canvas constraint without changing the underlying diffusion objective. It adds two sentinel tokens to the vocabulary, `<|expand|>` and `<|delete|>`, which the model is trained to predict alongside regular tokens. At inference time, predicting `<|expand|>` at a masked position deterministically replaces it with two new `<|mask|>` tokens, allowing the canvas to grow when more room is needed; predicting `<|delete|>` removes the position from the sequence, allowing the canvas to shrink when fewer tokens are needed than originally reserved. Both rules are applied within the same iterative denoising loop, so length adjustment and content filling happen together rather than in separate stages. Figure 2 shows the two cases side by side on a minimal renaming example. On the left, the canvas starts with one `<|mask|>` but the target identifier `activeCount` tokenizes into two sub-words, so the model predicts `<|expand|>` and the substitution rule grows the canvas before the final denoising step. On the right, the canvas starts with three `<|mask|>` tokens but the target identifier `n`

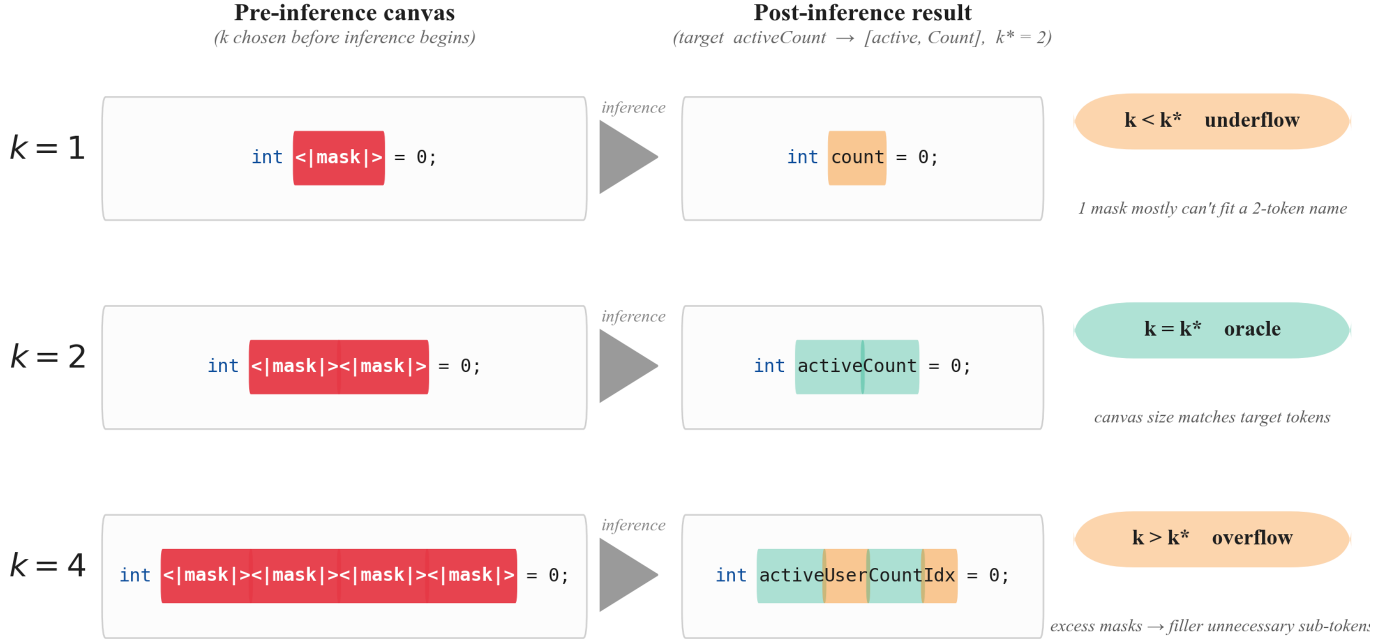


Fig. 3. Fixed-canvas constraint in standard masked diffusion large language models.

is a single sub-word, so the model predicts two `<|delete|>` tokens and the substitution rule shrinks the canvas to match. In both cases the final converged sequence matches the developer-chosen name exactly, without the caller having to guess the right number of mask slots in advance.

DreamOn therefore provides a useful control on the canvas-size axis. If the fixed-k choice required by DreamCoder and DiffuCoder were the operative bottleneck on our task, a variable-canvas model should close the gap to the FIM-AR and Seq2Seq baselines. We test this prediction directly in Section IV-A by including DreamOn-7B in the benchmark.

III. Methodology

A. Problem Formalization

We formalize identifier renaming as a targeted denoising problem. Let $\mathbf{x} = (x_1, \ldots, x_N)$ be a tokenized code sequence of length N, and let $\mathcal{S} \subseteq \{1, \ldots, N\}$ with $|\mathcal{S}| \ll N$ denote the subset of positions occupied by poorly named identifiers. We call $\mathcal{S}$ the *smell-position set*.

Under the denoising view, we treat $\mathbf{x}$ as a corrupted version of a clean sequence $\mathbf{x}^*$ in which the only corruption sits at the positions in $\mathcal{S}$. Outside those positions the two sequences are identical, and at every position in $\mathcal{S}$ the smelly name in $\mathbf{x}$ is replaced in $\mathbf{x}^*$ by a semantically appropriate name. The replacement is not constrained to the token length of the original name, a property we revisit in Section IV-C when choosing the number of `<|mask|>` slots per identifier.

In RefineID (Section III-C), every position in $\mathcal{S}$ belongs to the same identifier, so the model must propagate one renaming decision consistently across $|\mathcal{S}|$ sites within the same file. The number of sites ranges from one to 175 across the test set, which gives a natural axis along which to study how dLLMs scale with the size of $\mathcal{S}$. A renaming task that touches several distinct identifiers in the same file is a strict generalization of this regime: the model now has to partition $\mathcal{S}$ into groups that share a name and produce a different replacement per group.

This distinction organizes our three experimental conditions, which differ in the structure of $\mathcal{S}$ and in what the model is asked to look at.

- **RQ1 — Single identifier, $|\mathcal{S}|$ sites.** $\mathcal{S}$ is provided as explicit `<|mask|>` tokens, and every position in $\mathcal{S}$ belongs to the same identifier. This is the regime in which we benchmark dLLMs against FIM-AR and Seq2Seq baselines (Section IV-A) and stratify performance by $|\mathcal{S}|$ (Section IV-B).
- **RQ2 — Multiple identifiers, multiple sites.** We construct a harder regime by merging $\mathcal{S}$ sets from several RefineID samples into a single file, so that the model must rename more than one identifier in the same forward pass. The masks are still provided, but the model must keep separate identifiers separate rather than collapse them onto a single name.
- **RQ3 — Internal-state comparison.** We run the model twice on each sample, once on the smelly input $\mathbf{x}$ and once on the clean reference $\mathbf{x}^*$, and compare per-layer hidden states, per-step token confidence, and the unmasking order at the positions in $\mathcal{S}$ between the two runs.

B. Models Under Study

We evaluated three open-source masked dLLMs as two primary systems. **DreamCoder-7B** [40] and **DiffuCoder-7B** [12], both models are 7B parameter diffusion large language model trained on code. At the same time, we also evaluated **DreamOn** [39] as a representative of different implementations of the diffusion model (variable canvas).

For **decoder-only (FIM)** baselines, we include twelve causal-LM models trained with the fill-in-the-middle objective, spanning four model families and parameter scales from 1.3B to 15B: CodeLlama (7B, 13B) [26], StarCoder2 (3B, 7B, 15B) [22], DeepSeek-Coder (1.3B-Base, 6.7B-Base) [14], Qwen2.5-Coder (1.5B, 3B, 7B, 14B) [15], and CodeGemma (2B, 7B) [32]. All AR models use their native FIM format $\langle \texttt{PRE} \rangle\, P\, \langle \texttt{SUF} \rangle\, S\, \langle \texttt{MID} \rangle$ for inference.

For **encoder-decoder (Seq2Seq)** baselines, we additionally evaluated three models from the CodeT5 family: CodeT5-small, CodeT5-base, CodeT5-large [36], CodeT5p-2B, CodeT5p-6B, and CodeT5p-16b [35]. Unlike the decoder-only models, CodeT5 has a bidirectional encoder by construction. We query CodeT5 through its native span-denoising interface by replacing the masked identifier with the T5 sentinel `<extra_id_0>` and decoding the span.

C. Dataset Construction

We evaluate on the identifier-renaming dataset introduced by Vijayvargiya et al. [34] for masked-language-model based identifier renaming.[1] The dataset consists of identifier-renaming refactorings mined from real Java commits on public GitHub repositories. Each sample is built from a single commit in which an identifier was renamed. The file is reconstructed at the pre-rename revision, every occurrence of the renamed identifier is replaced with a `[MASK]` placeholder, and the post-rename name is recorded as the ground truth. The full mining procedure, the set of source repositories, and the commit time range are described in the original paper [34]. A detailed description of the dataset can be found in Table I.

D. Experiment Setup

1) RQ1: Single-Identifier Multi-Site Renaming: RQ1 evaluates how well each model fills the smell set $\mathcal{S}$ when every position in $\mathcal{S}$ belongs to the same identifier and all positions are given as explicit masks. All models see the same 1000 RefineID samples and the same $|\mathcal{S}|$ positions per sample, so any performance difference reflects how each architecture handles multi-site filling rather than differences in input.

dLLMs. The smelly code sequence $\mathbf{x}$ is preprocessed by replacing each token at the positions in $\mathcal{S}$ with the `<|mask|>` token, and the model produces predictions over all masked positions in a single bidirectional forward pass. The headline benchmark in Table II uses $T{=}64$ denoising steps to match the configurations reported in the original DiffuCoder [13] and DreamCoder [40] papers, with $k{=}2$ `<|mask|>` tokens per identifier site. Two ablations justify these settings: a static mask-token sweep over $k \in \{1,2,3,4,5\}$ on the full test set (Section IV-C) and a denoising-step sweep over $T \in \{1,2,4,8,16,32,64\}$ on the same test set (Section IV-D). The mask-token sweep identifies $k{=}2$ as the structural optimum for both dLLMs, because 76.8% of ground-truth identifiers tokenize into less than two sub-words. The step sweep shows that the efficiency–quality curve plateaus at $T{=}2$, with subsequent increases in T yielding sub-percentage-point improvements.

FIM-AR baselines. Each masked position is processed as an independent FIM query using the standard sentinel layout $\langle \texttt{PRE} \rangle\, P\, \langle \texttt{SUF} \rangle\, S\, \langle \texttt{MID} \rangle$. Positions are filled sequentially, with each prediction inserted into the sequence before the next query is issued, matching the standard FIM inference protocol. All AR models use greedy decoding (temperature = 0).

Seq2Seq baselines. Each masked position is similarly processed as an independent encoder-decoder query. We split the snippet at the first remaining `<|mask|>`, replace it with `<extra_id_0>`, and let the bidirectional encoder read the prefix, the sentinel, and the suffix in a single forward pass. The decoder then emits the predicted identifier as the span before `<extra_id_0>`. Other unfilled `<|mask|>` occurrences remain as literal text in the suffix, the same convention used for FIM-AR. The encoder context budget is split 60% for the prefix tail and 40% for the suffix head, and the outer iteration schedule is identical to the FIM-AR setup.

Stratification by smell-set size. To see where the dLLM advantage arises, Section IV-B re-uses the per-sample predictions and groups them by the cardinality of $\mathcal{S}$ into five

TABLE I
REFINEID TEST SPLIT USED IN THIS PAPER. "SITES PER SAMPLE" COUNTS THE NUMBER OF `[MASK]` POSITIONS FILLED WITH THE SAME TARGET IDENTIFIER WITHIN ONE SAMPLE.

Property	Value
Total samples (renaming events)	1,000
Language	Java
Source corpus	GitHub Java OSS
Identifier kind (heuristic from declaration site)	
Local variable	84.9%
Formal parameter	10.3%
Class field (private / protected / public)	4.8%
Method name	0% (excluded)
Identifier surface form	
`camelCase`	59.9%
lowercase, single token	39.5%
`snake_case`	0.5%
Mean / median length (chars)	10.4 / 9
Sites per sample	
Mean / median	7.7 / 4
Range (min, max)	1, 175
Single-site ($\lvert\mathcal{S}\rvert$=1)	2.3%
2–5 sites	55.7%
6–10 sites	21.6%
11–20 sites	15.3%
21+ sites	5.1%
File context	
Mean / median file length (chars)	25,022 / 10,498

[1] The replication package is publicly available at https://huggingface.co/spaces/scam2024/ReIdentify. Following [16], who reuse this dataset to fine-tune their RefineID assistant, we refer to it as the RefineID test split for short throughout the paper.

buckets: $|\mathcal{S}|{=}1$ (23 samples), $|\mathcal{S}|{=}2$, $|\mathcal{S}|{\in}[3,5]$, $|\mathcal{S}|{\in}[6,10]$, and $|\mathcal{S}|{\geq}11$. The single-site bucket is small but informative, since it is the only regime in which FIM-AR's sequential factorization is on equal footing with dLLM's joint resolution.

2) RQ2: Multi-Identifier Multi-Site Renaming: RQ1 established that dLLMs can propagate one renaming decision across many sites belonging to the same identifier. RQ2 asks whether they can do the same when several distinct identifiers in the same file all need new names at once. This regime does not occur naturally in RefineID, so we construct it via obfuscation. Therefore, we construct a controlled stress test by running the model on obfuscated source files, where every local variable has been replaced with a semantically vacuous single-letter token and the smell set $\mathcal{S}$ potentially covers every local-variable position. We reuse RefineID rather than running a separate obfuscation experiment on arbitrary Java code so that the target identifier and the underlying file remain constant between RQ1 and RQ2, the developer-chosen name continues to serve as an objective Exact Match reference.

For each RefineID sample we reconstruct the clean reference $\mathbf{x}^*$ by substituting the ground-truth name into its mask positions. We then run a regex-based identifier extractor that skips comments, imports, annotations, qualified-name segments, method calls, keywords, built-in types, and single-character names. Every surviving identifier is renamed to a short alphabetic token (`a`, `b`, . . ., extending to `aa`, `ab`, . . . past 26 identifiers), one token per identifier, applied consistently to all occurrences. The original RefineID target receives its own alphabetic token like other identifiers. Types, method names, and package qualifiers are preserved, so control flow and library structure remain intact while meaningful local names are gone. The result is an obfuscated sequence $\tilde{\mathbf{x}}$ whose smell set $\mathcal{S}$ covers every position. Figure 4 shows the construction on a concrete sample.

We evaluate two conditions on $\tilde{\mathbf{x}}$, both using the RQ1-optimal configuration ($k{=}2$, $T{=}64$). In the *target-only* condition only the target's positions are masked; we score the target-position prediction against the developer-chosen name, giving a number directly comparable to the RQ1 EM on the same sample. In the *all-masked* condition every position in $\mathcal{S}$ is masked simultaneously; we then score at the identifier-group level across each identifier's positions. We report (i) target-position EM under the target-only condition and (ii) per-sample average identifier-group EM under the all-masked condition, stratified by the number of distinct identifiers in $\mathcal{S}$.

3) RQ3: Internal-State Differences Between Good and Bad Identifier Names: RQ3 asks how the internal state of a dLLM differs between a smelly identifier and a clean one. We design four experiments on DiffuCoder-7B-Base rather than DreamCoder-7B because its inference process exposes per-layer hidden states and the unmasking schedule through a stable interface. The two models share the same masked discrete diffusion architecture and confidence-ordered schedule, so we expect the observations to be transferable.

The four experiments look at the same comparison from four angles. The first two read the model from the outside,

Fig. 4. Construction of the RQ2 obfuscated sample on a concrete RefineID example.

comparing the output distribution at a smelly position to the distribution at a clean position. The third reads the model from the inside, comparing hidden states per-layer and per-step. The fourth looks at the unmasking schedule itself, asking whether smelly positions are confirmed at different denoising steps than clean ones.

The first experiment asks whether the output distribution distinguishes smelly tokens from clean ones at all. For 200 randomly sampled RefineID snippets, we score the ground-truth identifier at each target position under an unconditioned experiment and record its rank r^{gt}. Rank-of-ground-truth is a standard probing metric for masked language models, as it captures where the results sit in the preference order of the model over the entire vocabulary [30]. We stratify positions into three regimes by r^{gt}:

- HIGHCONFIDENT ($r^{\text{gt}} \leq 200$): the model ranks the developer-chosen name near the top of its distribution. Names in this regime are dominated by generic identifiers (`result`, `value`, `tmp`) that the model strongly expects in the surrounding context.
- UNCERTAIN ($200 < r^{\text{gt}} \leq 1000$): the model assigns the ground-truth name a middling rank. The regime is mixed in style, with no clear dominant category.
- RARECONFIDENT ($r^{\text{gt}} > 1000$): the model ranks the ground-truth name far down its distribution. Names in this regime are dominated by semantically specific, project-level identifiers (`decodedCapacity`, `firstEscapedByteIndex`) that the model has not learned to expect from the surrounding code alone.

At each position we then inject a known smell name drawn from a curated vocabulary (`tmp`, `x`, `foo`, single-letter aliases) and record the rank r^{smell} that the smell token achieves in the same output distribution. The gap $r^{\text{smell}} - r^{\text{gt}}$, reported separately by regime, describes whether the output distribution treats smelly names as out of place or as ordinary candidates.

The second experiment asks how strongly the prediction at each position depends on the surrounding code. For the

same 200 snippets, we randomly mask a fraction $\alpha \in \{0, 0.05, 0.1, \ldots, 1.0\}$ of the non-identifier context tokens, such as keywords, literals, operators, and declarations, and re-score the target position, recording the Shannon entropy $H(\alpha)$ of the resulting distribution. The context-sensitivity score $\Delta H = H(\alpha{=}0.8) - H(\alpha{=}0)$ describes how much the surrounding code shapes the prediction at each position. We report ΔH stratified by the same three confidence regimes, which lets us see whether some regimes are more context-sensitive than others [38].

Then we run DiffuCoder-7B-Base twice on each sample, once on the smelly input $\mathbf{x}$ and once on the clean reference $\mathbf{x}^*$, and extract the hidden state at the target position for every transformer layer and every denoising step. We compare the smelly and clean trajectories using cosine similarity and complement the cosine analysis with UMAP projections of the hidden-state clouds across layers and steps.

The fourth experiment looks at the unmasking schedule itself. For each token in 20 sampled snippets we label whether it belongs to a RefineID smell position (n=158) or to ordinary context (n=19,838) and record the first denoising step at which the model commits the position with confidence ≥ 0.8. A Mann-Whitney U test compares the two distributions and tells us whether smelly positions are confirmed earlier, later, or at the same denoising step as ordinary context tokens.

E. Evaluation Metrics

Exact Match (EM) measures the proportion of samples where the model's prediction $\hat{\mathbf{x}} = f_{\text{fix}}(\mathbf{x}, \mathcal{S})$ matches the ground truth at all positions in $\mathcal{S}$:

$$\text{EM} = \frac{1}{|\mathcal{D}|} \sum_{(\mathbf{x},\mathbf{x}^*)\in\mathcal{D}} \mathbf{1}[\hat{x}_i = x_i^* \;\; \forall\, i \in \mathcal{S}]\,. \qquad (4)$$

Since we are using Regex to extract the predictions, whitespace will not be considered in the match.

LLM-as-a-judge Result (LJ) Considering the Exact Match is strict while predictions of the identifiers can be lexically different but semantically similar, we adopt LLM-as-judge (LJ) accuracy. For LJ, we choose Qwen2.5-7B-Instruct [15] as the judge model to verify whether the prediction is suitable as an identifier naming here. This choice is based on the reproducibility and strong code understanding ability, which are the standards proposed in prior work of LJ [42]. During the LJ process, we show the smelly code snippet, the target position, the ground-truth name, and other model's prediction, to make sure the model can get a comprehensive overview of the task and make a careful judgment. The full judge setup, including prompt template, generation configuration, human-validation, known biases and scoring logic, is documented in the Appendix B, following the chatbot judgment guideline [42].

TABLE II
EXACT MATCH (EM) AND LLM-JUDGE ACCURACY (LJ) UNDER THE MASKED CONDITION (RQ1). MODELS GROUPED BY ARCHITECTURE AND SORTED BY EM WITHIN EACH GROUP.

Model	Architecture	Params	EM (%)	LJ (%)
DreamCoder-7B	dLLM (fixed-canvas)	7B	33.2	66.2
DiffuCoder-7B	dLLM (fixed-canvas)	7B	31.1	64.1
DreamOn-7B	dLLM (variable canvas)	7B	16.0	55.9
CodeLlama-13B	Decoder-only	13B	20.2	32.1
CodeLlama-7B	Decoder-only	7B	20.2	33.5
StarCoder2-15B	Decoder-only	15B	18.6	33.5
StarCoder2-7B	Decoder-only	7B	11.7	29.7
StarCoder2-3B	Decoder-only	3B	12.4	30.6
DeepSeek-Coder-6.7B	Decoder-only	6.7B	15.6	24.2
DeepSeek-Coder-1.3B	Decoder-only	1.3B	17.7	29.2
Qwen2.5-Coder-14B	Decoder-only	14B	13.7	22.3
Qwen2.5-Coder-7B	Decoder-only	7B	18.1	29.3
Qwen2.5-Coder-3B	Decoder-only	3B	14.4	24.9
Qwen2.5-Coder-1.5B	Decoder-only	1.5B	13.2	25.0
CodeGemma-7B	Decoder-only	7B	9.8	18.2
CodeGemma-2B	Decoder-only	2B	7.1	13.9
CodeT5+ 16B	Encoder-decoder	16B	23.5	26.6
CodeT5+ 6B	Encoder-decoder	6B	21.1	24.0
CodeT5+ 2B	Encoder-decoder	2B	20.5	22.9
CodeT5-large	Encoder-decoder	770M	20.6	22.5
CodeT5-base	Encoder-decoder	220M	17.7	19.0
CodeT5-small	Encoder-decoder	60M	13.2	15.1

IV. RQ1: BENCHMARKING IDENTIFIER RENAMING PERFORMANCE

A. Benchmarking Results

Table II reports Exact Match (EM) and LLM-as-a-judge accuracy (LJ) for all twenty-one models on the RefineID test set under the single-identifier multi-site condition. We discuss what the table shows in this section and leave the analysis of why these patterns appear to Section IV-E.

The two fixed-canvas dLLMs sit at the top of the table, with DreamCoder-7B and DiffuCoder-7B at 33.2% and 31.1% EM. The strongest non-dLLM baseline, CodeT5+ 16B, reaches 23.5% EM, and the strongest FIM-AR baseline, CodeLlama-7B/13B, reaches 20.2%. The gap between fixed-canvas dLLMs and the rest of the table is therefore roughly 10 percentage points on EM against the best Seq2Seq baseline and 13 percentage points against the best FIM-AR baseline. The same gap on LJ is 32.7 percentage points against FIM-AR. The dLLMs achieve these numbers at 7B parameters, less than half the size of either the largest FIM-AR baseline (StarCoder2-15B, 15B) or the largest Seq2Seq baseline (CodeT5+ 16B). Scale is therefore not what separates the top of the table from the rest.

Within each non-dLLM family the picture is more uneven. The twelve FIM-AR baselines span 7.1% to 20.2% EM, but the ordering does not track parameter count: CodeLlama-13B matches CodeLlama-7B, Qwen2.5-Coder-14B sits below Qwen2.5-Coder-7B, and DeepSeek-Coder-1.3B sits above DeepSeek-Coder-6.7B. The six Seq2Seq models cluster more tightly, in a 13.2% to 23.5% EM range, with the four larger variants (CodeT5+ 2B/6B/16B and CodeT5-large) all within 3 percentage points of one another. Both families therefore

lag the fixed-canvas dLLMs by a similar margin despite their architectural differences, and neither shows the smooth size-driven scaling one might expect.

The third dLLM, DreamOn-7B, behaves unlike the other two. Its EM (16.0%) sits in the middle of the FIM-AR group, but its LJ (55.9%) is higher than every FIM-AR and Seq2Seq baseline, and second only to the two fixed-canvas dLLMs. This is the only model exhibiting a huge difference between EM and LJ.

B. Refactoring Consistency Experiment

Section IV-A reported an EM gap of roughly 10 percentage points between the best dLLM and the best non-dLLM baseline, but the result does not show where in the input distribution this gap lives. To locate it, we re-use the per-sample predictions from Table II and group them by the smell set $|\mathcal{S}|$.

We group the 1000 samples into five groups: $|\mathcal{S}|=1$ (23 samples), $|\mathcal{S}|=2$, $|\mathcal{S}|\in\{3,4,5\}$, $|\mathcal{S}|\in\{6,\ldots,10\}$, and $|\mathcal{S}|\geq 11$. We report the two fixed-canvas dLLMs (DreamCoder-7B, DiffuCoder-7B) against the four FIM-AR baselines with the highest aggregate EM (CodeLlama-7B, CodeLlama-13B, StarCoder2-15B, Qwen2.5-Coder-7B). The comparison is restricted to these two families because the question this experiment is designed to answer concerns joint versus sequential resolution of multiple masks.

The two families trace opposite shapes across $|\mathcal{S}|$. On the single-site bucket (Table III), the four FIM-AR baselines reach 69.6%-73.9% EM while the two dLLMs reach 56.5% and 65.2%. FIM-AR leads by 5 to 17 percentage points on the smallest bucket of the dataset. The two families come within 7.5% of each other at $|\mathcal{S}|=2$, and the ranking flips at $|\mathcal{S}|\in\{3,4,5\}$, where both dLLMs move above every FIM-AR baseline. Past this crossing point the divergence widens: by $|\mathcal{S}|\geq 11$ the two dLLMs sit at 33.8% and 36.4% while the four FIM-AR baselines have fallen to 7.8%–8.8%, a gap of more than 25 percentage points in favor of the dLLMs.

Across the rows of Table III, the two shapes are different in kind, not just in level. dLLM EM stays inside a $\sim$10-percentage-point band (26.8%–36.4%) across all four multi-site buckets, with no monotone trend in $|\mathcal{S}|$. FIM-AR EM falls monotonically across the same axis, from the 69.6%–73.9% range at $|\mathcal{S}|=1$ down to 7.8%–8.8% at $|\mathcal{S}|\geq 11$—a 60-percentage-point collapse over the same span on which dLLMs barely move. The aggregate $\sim$10% gap reported in Section IV-A therefore averages over two completely different groups. For single-site cases dLLMs lose while they get better results than AR models in multi-site cases.

C. Ablation Experiment for Token Length Hyperparameter

Regular dLLM filling requires a design choice that is not in FIM-AR inference which is choosing the number of `<|mask|>` tokens k to place at each identifier site. Identifier names are tokenized into a variable number of sub-words, so the choice of k trades off the risk of truncating long names (k too small) against the risk of forcing the model to fill positions that have no ground-truth content (k too large). The right k is therefore not given by the task; it has to be measured. We sweep $k \in \{1,2,3,4,5\}$ on both DiffuCoder-7B and DreamCoder-7B over the 1000 RefineID samples, holding every other setting fixed at T=32.

TABLE III
EXACT MATCH STRATIFIED BY SMELL-SET CARDINALITY $|\mathcal{S}|$ ON THE REFINEID BENCHMARK. THE SINGLE-SITE BUCKET ($|\mathcal{S}|=1$) CONTAINS 23 SAMPLES; THE OTHER FOUR BUCKETS CONTAIN THE REMAINING 976 SAMPLES. BOLD: BEST PER COLUMN.

Model	**EM by smell-set cardinality $|\mathcal{S}|$**				
	$=1$	$=2$	3–5	6–10	≥ 11
Diffusion LLMs					
DreamCoder-7B	65.2	32.5	**33.2**	**35.1**	**33.8**
DiffuCoder-7B	56.5	29.9	26.8	35.1	**36.4**
FIM autoregressive baselines					
CodeLlama-13B	**73.9**	**37.4**	18.9	11.6	8.3
CodeLlama-7B	69.6	34.5	20.9	12.0	7.8
StarCoder2-15B	69.6	30.0	19.5	10.6	8.3
Qwen2.5-Coder-7B	**73.9**	31.5	16.9	10.2	8.8

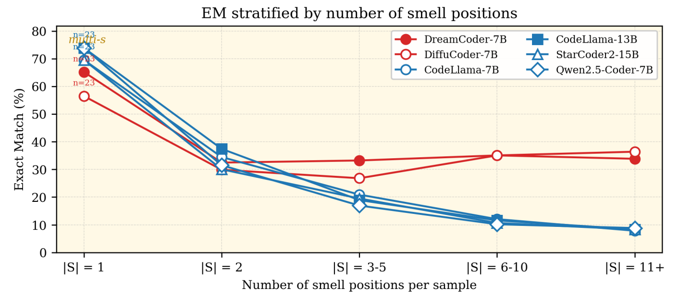


Fig. 5. Exact Match as a function of the smell-set cardinality $|\mathcal{S}|$.

Figure 6(b) reports the tokenizer's behavior on the ground-truth identifiers. Under the shared tokenizer, 76.7% of the identifiers tokenize into one or two sub-words, with a mean length of 1.97 and a median of 2. Three or more sub-words account for the remaining 23.3%. Figure 6(a) reports EM as a function of k. Both DiffuCoder-7B and DreamCoder-7B peak at k=2, which is consistent with k=2 being the modal token length in panel (b). Since the two models share a tokenizer, it is expected the peak being at the same k. EM degrades on either side of the peak but not symmetrically: at k=1, the model cannot represent the 38.6% of identifiers whose tokenization requires two sub-words, while at $k\geq 3$, the model must commit to filler tokens that do not exist in any ground-truth name. The cost of over-allocating the canvas grows with k: by k=5, DreamCoder drops to 15.60% and DiffuCoder to 13.60%, less than half of their respective peaks. The shape of the EM curve in panel (a) therefore tracks the shape of the token-length distribution in panel (b), as we would expect from the fixed-canvas constraint described in Section II-B.

Therefore, we use k=2 for both dLLMs in the main benchmark of Table II and throughout the rest of the paper.

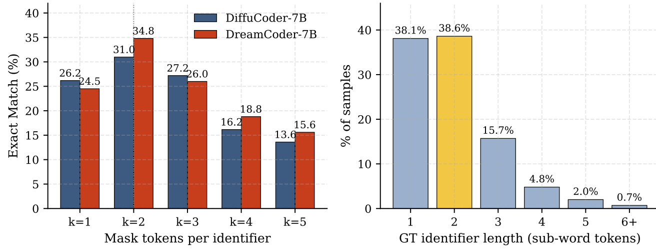


Fig. 6. Mask-token count ablation. (a) Exact Match on 1000 RefineID samples for $k \in \{1, \ldots, 5\}$. (b) Sub-word token-length distribution of ground-truth identifier names under the shared DiffuCoder/DreamCoder tokenizer.

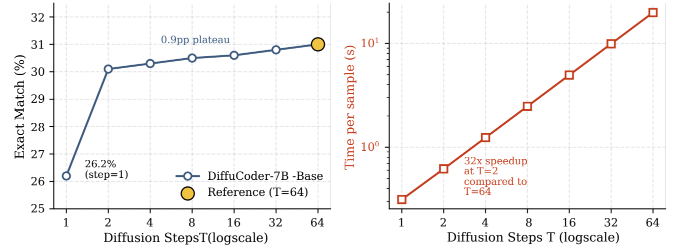


Fig. 7. Diffusion step sensitivity of DiffuCoder-7B-Base on the RefineID test set. (a) EM as a function of T. (b) Per-sample latency as a function of T on a log-log scale.

TABLE IV
DIFFUSION STEP SENSITIVITY OF DIFFUCODER-7B-BASE ON REFINEID (N=1000). SPEEDUP AND EM GAP ARE MEASURED RELATIVE TO T=64.

Steps	EM (%)	Time / sample	Speedup	EM Gap
1	26.20	0.313 s	63.3×	−4.80%
2	30.10	0.618 s	32.1×	−0.90%
4	30.30	1.238 s	16.0×	−0.70%
8	30.50	2.477 s	8.0×	−0.50%
16	30.60	4.957 s	4.0×	−0.40%
32	30.80	9.916 s	2.0×	−0.20%
64	**31.00**	19.831 s	1.0×	0.00%

D. Diffusion Step Robustness Experiment

A practical concern for deploying dLLMs is how many denoising steps T are actually needed, since each extra step costs an extra forward pass. We sweep $T \in \{1, 2, 4, 8, 16, 32, 64\}$ on DiffuCoder-7B-Base over the 1000 RefineID samples, holding all other settings fixed. A trial run at T=128 takes roughly 33 seconds per sample, which exceeds the runtime upper limit of 12 hours. We then cap the sweep at T=64 to make the experiments more feasible.

The EM curve (Table IV, Figure 7(a)) has two regimes. Between T=1 and T=2, EM rises by 3.9%, from 26.20% to 30.10%. Past T=2, the curve plateaus: every subsequent doubling adds at most 0.2%, and the total spread from T=2 to T=64 is 0.9% against a 32× increase in compute. Doubling T once, from 1 to 2, therefore buys 4.3× more EM (3.9%) than the next five doublings combined (0.9% from T=2 to T=64). Per-sample latency scales as a power of T (Figure 7(b)), each additional denoising step adds one forward pass, so latency grows essentially linearly in T. Combined with the flat EM curve past T=2, this means that the only step count that pays a meaningful EM return on its compute cost is the one between T=1 and T=2.

For 55.2% of the test samples, the prediction is identical across all seven values of T. Among the 310 samples that DiffuCoder-7B-Base gets correct at T=64, 85.5% are also correct at T=2 and 96.8% at T=16. The vast majority of correct predictions are confirmed early and do not change with additional denoising steps.

The headline benchmark in Table II uses T=64 to match the configurations reported in the original dLLM papers [13], [40], and we choose to report T=32 numbers throughout the paper for comparability. For deployment or reproduction under a compute constraint, T=2 recovers 97.1% of the peak EM at 16× the throughput.

E. Analysis

dLLMs resolve every [MASK] position in a single forward pass under confidence-ordered unmasking, so a high-confidence prediction at one site propagates to the other sites through bidirectional attention. FIM-AR has to predict one position at a time, inserting each prediction into the prefix before the next, and so a name chosen at one site cannot inform the choice at another. This is why FIM-AR wins at $|\mathcal{S}|$=1 (one site, no consistency to enforce) and loses progressively as $|\mathcal{S}|$ grows: the bottleneck is not per-site prediction quality but consistency across sites, which the FIM-AR factorization simply does not model. Because 976 of the 1,000 RefineID samples are multi-site, the same bottleneck also explains why scaling does not help inside the FIM-AR group. CodeLlama-13B matches CodeLlama-7B, Qwen2.5-Coder-14B falls below Qwen2.5-Coder-7B, since adding parameters improves per-site quality without removing the consistency ceiling.

Compared with fixed-canvas implementations, DreamOn-7B shows a different result. Its 16.0% EM lands in the area the other two dLLMs occupy only when forced above their structural optimum at $k \geq 4$, which is where DreamOn's training distribution actually sits. DreamOn was trained on function-level infilling, and its <|expand|> mechanism grows a slot in increments of two sub-words, which is the improper for identifiers that fit into one or two sub-words. However, it has 55.9% LJ, second only to the two fixed-canvas dLLMs, confirms that the names DreamOn produces are semantically reasonable. EM penalizes the extra tokens it leaves behind, the semantic judge does not. Fine-tuning DreamOn at identifier scale is a natural next step.

To answer RQ1, two of the three fixed-canvas dLLMs evaluated (DreamCoder-7B and DiffuCoder-7B) outperform every AR and sequence-to-sequence baseline on multi-site renaming of a single identifier, reaching 33.2% and 31.1% EM against 23.5% for the strongest non-dLLM baseline. The gap also widens as the number of identifier sites to be renamed grows.

TABLE V
TARGET-POSITION EM UNDER RQ1 (CLEAN CONTEXT) AND THE TWO RQ2 EXPERIMENTS. BOTH RQ2 COLUMNS REPORT EM EVALUATED AT THE SAME REFINEID TARGET POSITION FOR DIRECT COMPARABILITY WITH RQ1.

Model	RQ1 clean ctx.	RQ2 target-EM all-masked	target-only
DiffuCoder-7B	31.1	12.2	3.1
DreamCoder-7B	33.2	13.9	4.1

V. RQ2: MULTI-IDENTIFIER MULTI-SITE RENAMING

RQ1 showed that dLLMs can propagate a single renaming decision across many co-referrin sites in one bidirectional forward pass, and that the advantage over FIM-AR baselines widens as the number of sites grows. The natural next question is whether the same advantage extends from one identifier to many: can a dLLM produce distinct names for several distinct identifiers in the same file simultaneously? RefineID does not ask this question directly, since each sample annotates exactly one identifier as the target.

The experiment setup in Section III-D2 enables us to decompose the gap between RQ1 and the full multi-identifier multi-site task into two parts that can be measured separately. We report aggregate Exact Match in Section V-D, examine the qualitative failure mode of wrong predictions in Section V-C, and discuss what the results say about the scope of the joint multi-site advantage demonstrated in RQ1 in Section V-D.

A. All-masked Experiment

To explore the ability of renaming with multiple sites and multiple identifiers, we designed the all-masked experiment as described in Section III-D2.

Under this condition, DiffuCoder-7B reaches 12.2% target-EM and DreamCoder-7B reaches 13.9% (Table V). Both numbers are below their RQ1 figures of 31.1% and 33.2%. Figure 8(b) reports how this number varies with the number of distinct identifiers in the smell set $\mathcal{S}$. EM declines as the identifier count grows, which contrasts with the RQ1 result on the same models (Figure 5), where the EM band of 26.8%–36.4% was held across single-identifier multi-site samples regardless of site count. The joint-resolution advantage is therefore robust to growth in the number of co-referring sites of one identifier, but degrades as the number of distinct identifier groups in $\mathcal{S}$ grows.

B. Target-only Experiment

The second experiment keep them in place but rewrite each as a single-letter token (`a`, `b`, `c`, ...), so the model sees concrete identifier-shaped tokens that carry no information about what those identifiers mean.

Under this condition, EM falls to 3.1% (DiffuCoder-7B) and 4.1% (DreamCoder-7B), 9–10% below the all-masked result on the same target positions. The two experiments differ only in whether the surrounding identifiers are absent (all-masked) or actively misleading (target-only), so the gap between them measures the cost of adversarial neighbors over and above the cost of absent neighbors.

TABLE VI
DIFFUCODER-7B'S WRONG PREDICTIONS UNDER THE TARGET-ONLY CONDITION. "OBF." IS THE SINGLE-LETTER TOKEN ASSIGNED TO THE TARGET DURING OBFUSCATION. ROWS MARKED † ARE CASES WHERE THE MODEL'S PREDICTION MATCHES THE OBFUSCATED LABEL OF THE TARGET ITSELF.

ID	Target	Obf.	Prediction
0	`decodedCapacity`	`ar`	`aq`
2	`expectedIndexName`	`n`	`r`
3	`style`	`e`	`e`†
6	`status`	`r`	`a`
9	`bytes`	`j`	`b`
11	`context`	`c`	`a`
12	`cache`	`a`	`a`†
13	`relativePath`	`j`	`j`†
15	`printer`	`n`	`p`

C. Qualitative Analysis of Wrong Predictions

The 28-29% percentage points lost in the first decomposition step leave the question of what DiffuCoder-7B actually produces in the 96.9% of target-only wrong samples. Compared with EM result, we want to explore what guesses the model is making. Therefore, we inspect the wrong predictions.

Among the 928 wrong predictions, 662 (71.3%) are one-or-two-letter lowercase strings that share the lexical form of the obfuscated single-letter tokens that surround them. Most of the remainder split between longer meaningful identifiers (189, 20.4%, such as `parseInt` and `fConfig`) and empty strings from which no identifier can be extracted (77, 8.3%). Two-thirds of the failures are not informed guesses that happen to disagree with the choice of developers. These results are predictions in the wrong shape. The general ability of the model on identifier renaming fails in the target-only condition when there is surrounding obfuscation.

In three of the nine rows (†), the prediction coincides with the single-letter token the obfuscator had assigned to the target itself. This is a selected sample, not recovery. The mapping is one-to-one and the target's positions are all `<|mask|>`, so the assigned letter never appears in the input. The bias toward frequent single-letter tokens under an obfuscated context produces a small number of chance collisions with the alphabet.

This bias is the qualitative companion of the second decomposition step. If the model in the target-only condition is following its prediction to the lexical style of the obfuscated context rather than to the structural cues that survive obfuscation, then masking those tokens away should remove the anchor and improve the result.

D. Analysis

The multi-site advantage observed in RQ1 was conditional on the model having semantically informative neighbours to attend to. When the neighbours carry the developer's chosen names, bidirectional attention can propagate one renaming decision across many sites in a single pass and the advantage scales with the number of sites. When the same model is given concrete-but-wrong neighbours (target-only), the EM advantage all but disappears, falling to 3–4%. When the

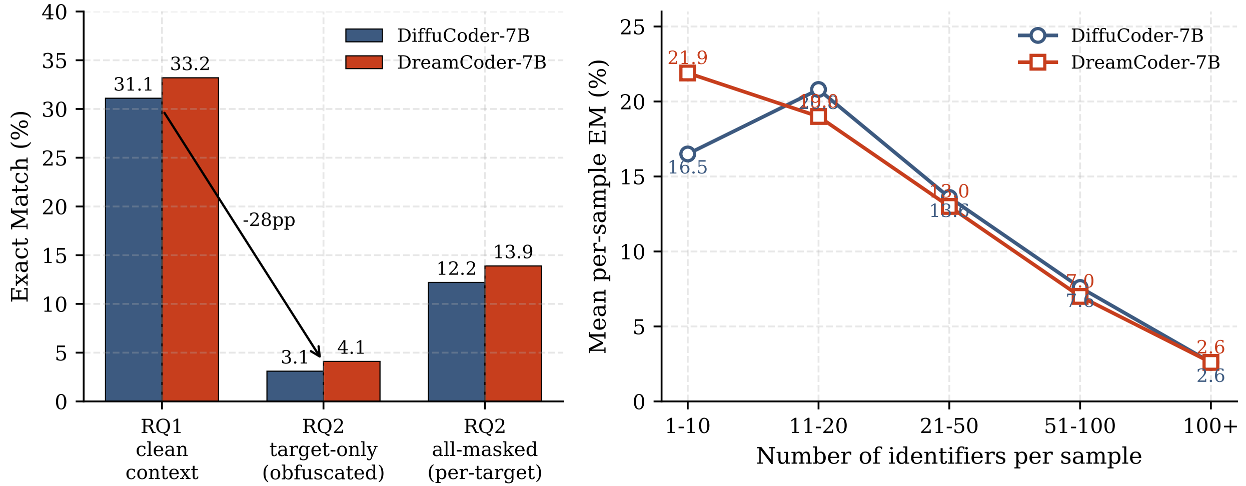


Fig. 8. RQ2 results. (a) Target-position Exact Match for DiffuCoder-7B and DreamCoder-7B under RQ1 (clean context), the all-masked experiment, and the target-only experiment. (b) Mean per-sample Exact Match across identifier-count buckets in the all-masked experiment.

neighbours are absent rather than misleading (all-masked), target-EM stays low, but still above the target-only result while below RQ1. Concrete-but-wrong neighbours are therefore more costly than absent ones. 71.3% of wrong predictions under target-only adopt the lexical form of the obfuscated neighbours, so the obfuscated context anchors the prediction to its own shape rather than letting the model use the structural cues that survive obfuscation.

Figure 8(b) reports how the all-masked target-EM varies with the number of distinct identifiers in $\mathcal{S}$. EM declines monotonically as the identifier count grows, which contrasts with the RQ1 result (Figure 5) where the same dLLMs held an EM band of 26.8%–36.4% across single-identifier multi-site samples regardless of site count. It is robust to growth in the number of multiple sites of one identifier, but degrades as the number of distinct identifier groups grows.

Therefore, RQ2 actually supports RQ1 rather than denies it. The joint multi-site advantage holds when the surrounding identifier names are semantically informative and when $\mathcal{S}$ resolves into a small number of co-reference groups, while it will be weakened when either condition fails.

VI. RQ3: Internal-State Differences Between Good and Bad Identifier Names

RQ1 showed that fixed-canvas dLLMs handle single-identifier multi-site renaming substantially better than FIM-AR and Seq2Seq baselines, and RQ2 showed that this advantage holds within a specific scope. Both results characterize the model from the outside, by what it produces at masked positions. A model that recovers the developer's exact identifier 33% of the time, and a semantically appropriate identifier 66% of the time, must compute something internally that distinguishes the names it accepts from the names it would replace. RQ3 turns the question inward and asks what this internal correlate state looks like.

We design four experiments on DiffuCoder-7B-Base that approach the question from four angles. The first two read the model from the outside, by querying the output distribution at a target position and asking whether a known smelly name lands in the same place as a well-formed one. The third reads the model from the inside, by extracting hidden states at every transformer layer and every denoising step under a smelly input and a clean reference and comparing the two trajectories directly. The fourth looks at the confidence-ordered unmasking schedule and asks whether smelly and clean positions are confirmed at the same denoising step. We use DiffuCoder-7B-Base rather than DreamCoder-7B because its inference engine exposes per-layer hidden states and the unmasking schedule through a stable interface; the two models share the same masked discrete diffusion architecture, so we expect the qualitative observations to transfer.

A. Output-Distribution Rank Comparison

The first experiment compares where good and bad identifier names land in the model's output distribution at the same target position. As described in Section III-D3, we score the ground-truth identifier at each of the 7,800 target positions in 200 RefineID snippets, record its rank r^{gt}, and stratify positions into three regimes:

- HIGHCONFIDENCE ($r^{\text{gt}} \leq 200$), where the ground-truth name is among the top 200 guesses; these positions are dominated by generic names such as `result`, `value`, `name`.
- UNCERTAIN ($200 < r^{\text{gt}} \leq 1000$), where the ground-truth name sits in the mid-range.
- RARECONFIDENT ($r^{\text{gt}} > 1000$), where the ground-truth name is project-specific (e.g., `decodedCapacity`, `expectedIndexNumber`) and far down the model's list.

We then inject a known smelly name drawn from a curated vocabulary (`tmp`, `x`, `foo`, single-letter aliases) at the same

TABLE VII
MEDIAN GROUND-TRUTH RANK r^{GT} AND MEDIAN INJECTED-SMELL RANK r^{SMELL} AT THE TARGET POSITION, STRATIFIED BY CONFIDENCE REGIME ON 200 REFINEID SNIPPETS.

Regime	|pos|	median r^{gt}	median r^{smell}
HIGHCONFIDENT	3,744	77	492
UNCERTAIN	1,755	524	676
RARECONFIDENT	2,301	10,989	733

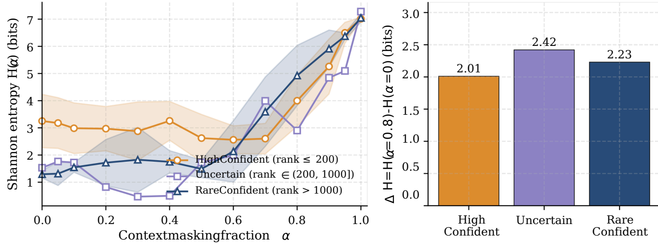


Fig. 9. Context sensitivity under progressive context masking on 200 RefineID snippets. (a) Mean entropy $H(\alpha)$ at the target position as a function of the masked-context fraction α, stratified by confidence regime. (b) ΔH stratified by regime.

position and record the rank r^{smell} the smelly token achieves in the same distribution.

Table VII reports the two median ranks per regime. In the HIGHCONFIDENT regime, the ground-truth name sits at median rank 77 and the injected smelly name sits at median rank 492, a gap of 415 positions in favour of the ground truth. In the UNCERTAIN regime, the two ranks are closer: 524 for the ground truth and 676 for the smelly probe, a gap of 152 positions still in favour of the ground truth. In the RARECONFIDENT regime, the ordering inverts: the ground truth sits at median rank 10,989 and the smelly probe at median rank 733, so the smelly probe is ranked roughly 15× higher than the developer's chosen name. The relative position of r^{gt} and r^{smell} therefore depends on the regime, not on a fixed property of the smelly vocabulary: smelly names are pushed down only when the model already places the ground-truth name near the top of its distribution.

B. Context Dependence of the Prediction

The second experiment measures how strongly the model's prediction at the target position depends on the surrounding code, separately for each confidence regime defined in Section VI-A. As described in Section III-D3, we corrupt a fraction $\alpha \in \{0, 0.05, 0.1, \ldots, 1.0\}$ of the non-identifier context tokens and record the Shannon entropy $H(\alpha)$ of the resulting prediction distribution at the target position. The context-sensitivity score $\Delta H = H(\alpha{=}0.8) - H(\alpha{=}0)$ summarizes how much the entropy moves when most of the surrounding code is corrupted.

Figure 9(a) plots the mean entropy at the target position as a function of α. All three regimes show monotonic entropy growth as the context is corrupted, rising from $H(0) \approx 1.5$ bits to $H(1) \approx 7.3$ bits, close to the full-vocabulary entropy under a uniform prior. Figure 9(b) reports ΔH per regime: HIGHCONFIDENT positions have $\Delta H = 2.01$, RARECONFIDENT positions have $\Delta H = 2.23$, and UNCERTAIN positions have $\Delta H = 2.42$. The ordering puts HIGHCONFIDENT at the bottom and UNCERTAIN at the top, with a spread of 0.41 bits across the three regimes.

Alongside the rank gaps reported in Section VI-A, the two output-side experiments paint a consistent picture of the regimes. HIGHCONFIDENT positions show a large r^{gt}–r^{smell} gap and the lowest sensitivity to surrounding code, so the prediction at these positions is anchored by the unconditioned prior. RARECONFIDENT positions show an inverted rank gap and an intermediate context sensitivity, so the prediction at these positions does respond to context but does not respond enough to lift the rare ground truth above the smelly probe. UNCERTAIN positions show a small rank gap and the highest context sensitivity, so the prediction at these positions is the most fluid of the three.

C. Hidden-State Trajectory

The first two experiments looked at what the model produces at the output layer. The results show that smelly and clean names sit in distinguishable but regime-dependent positions of the output distribution, and that the gap shrinks as the human-decided name becomes less generic. The third experiment moves the comparison inside the model and asks whether the internal state at the target position carries a clearer separation than the output distribution does. We run DiffuCoder-7B-Base twice on each of 926 RefineID samples, once on the smelly input $\mathbf{x}$ and once on the clean reference $\mathbf{x}^*$, and compare the smelly and clean trajectories using cosine similarity on the hidden-state vectors at every layer and every denoising step.

Two independent axes of the trajectory show the same shape. Along the depth axis (Figure 10(a)), the cosine starts at 0.09 at layer 0, where the two runs differ only at the target token by construction. It rises rapidly under self-attention to $\sim$0.40 by layer 5 and stabilizes near 0.50 through the middle of the stack. Only in the final third (layers 19–28) does the cosine rise a second time, reaching 0.61 at the output layer. Along the time axis (Figure 10(b)), the cosine stays close to 0.35 for the first 21 of the 32 denoising steps, then climbs sharply over the final 11 steps to $\sim$0.60 at $t{=}0$. The smelly and clean trajectories therefore sit at intermediate cosine values for the first two-thirds of the computation on either axis and only converge, in the sense of the smelly representation moving toward the clean one—in the final third of each.

The UMAP projections in Figure 11 give a visual check on what the cosine numbers describe. Along both the layer and step axes, smelly representations (red) and clean representations (blue) occupy partially overlapping regions in two dimensions: they form distinguishable clusters but not cleanly separated ones. The visual overlap matches the cosine values around 0.6–0.65 in the final layers and steps.

Cosine and UMAP both describe the geometry of the hidden state. The trajectory also represents how much the prediction distribution at a position moves as denoising progresses. For

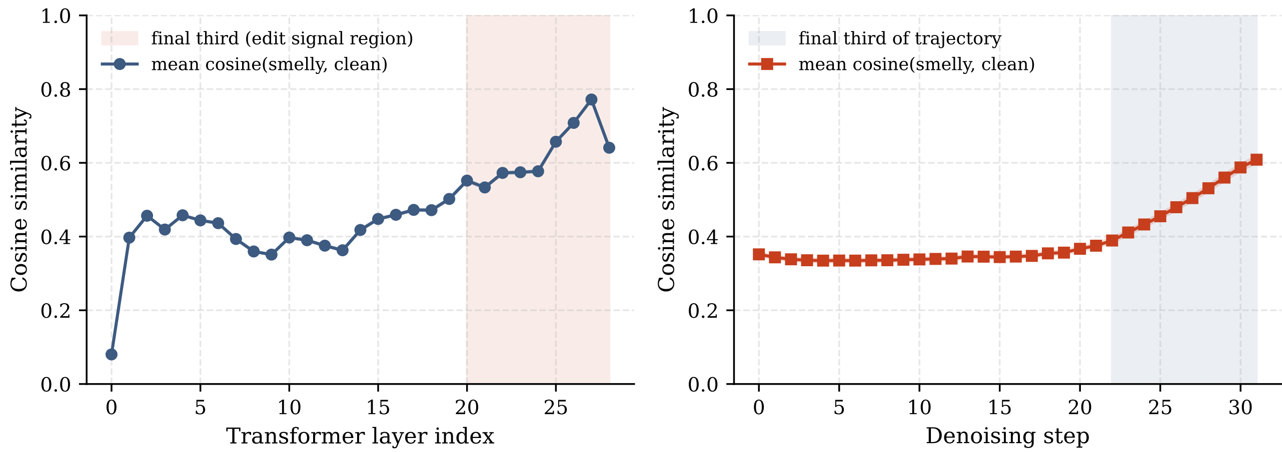


Fig. 10. Cosine similarity between smelly and clean hidden states at the target position, averaged over 926 RefineID samples. (a) Across the 29 transformer layers. (b) Across the 32 denoising steps. Shaded regions mark the final third of each axis.

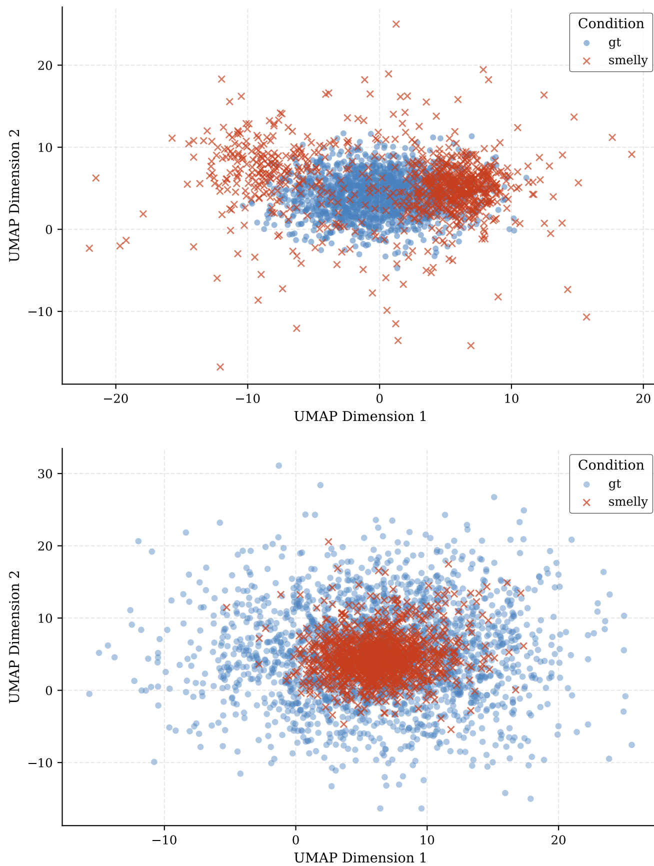


Fig. 11. UMAP projections of smell-position hidden states. Top: across transformer layers. Bottom: across denoising steps. Smelly representations (red ×) and clean representations (blue •).

each token in the trajectory we compute the mean entropy change across denoising steps, and compare smell positions against ordinary context positions. Smell positions move substantially more as shown in Figure 12. Smell positions look

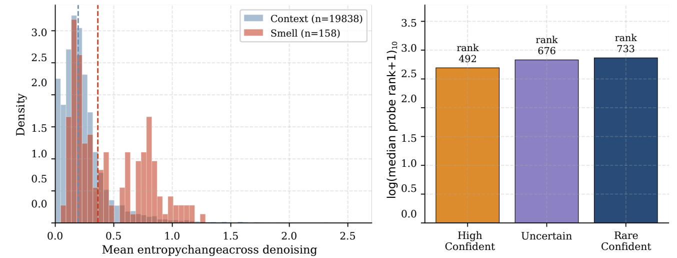


Fig. 12. Distribution of mean entropy change ΔH across the denoising trajectory for smell tokens (n=158) vs. context tokens (n=19,838).

different from clean ones along the smelly-clean axis (cosine, UMAP), and they also behave differently from ordinary context positions during denoising (entropy change).

D. Commitment Step Under the Unmasking Schedule

The fourth experiment looks at the confidence-ordered unmasking schedule itself. As described in Section III-D3, we sample 20 snippets and label every token as either a smell token (n=158, RefineID smell positions) or an ordinary context token (n=19,838). For each token we record the first denoising step at which the model commits the position with confidence ≥ 0.8.

Figure 13(a) reports the commitment step distributions. Smell tokens reach confidence ≥ 0.8 at median step 13.6, ordinary context tokens at median step 11.0, a difference of 2.6 steps out of the 32-step trajectory. A Mann–Whitney U test gives $p < 10^{-3}$ for the hypothesis "smell commits later than context" and $p \approx 1$ for the opposite direction. Figure 13(b) reports the average flip step across all token occurrences: ordinary context tokens cluster near step 32 (confirmed early and rarely revised), while smell tokens spread more evenly across the trajectory.

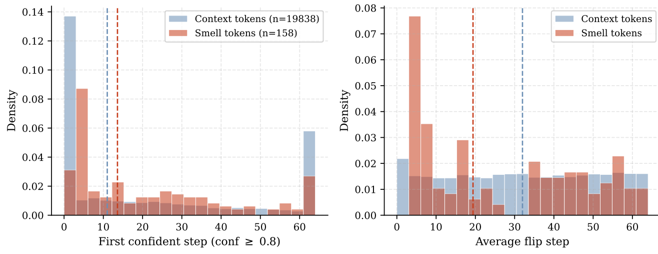


Fig. 13. (a) First denoising step at which a token reaches confidence ≥ 0.8, for smell tokens (n=158) vs. context tokens (n=19,838). (b) Average flip step across all token occurrences.

E. Analysis

The four experiments characterize the internal difference between good and bad identifier names from four angles.

At the level of the output distribution, where good and bad names land depends sharply on which confidence regime the position belongs to. In the HIGHCONFIDENT regime the ground-truth name sits at median rank 77 and the injected smelly name at rank 492, a gap of 415 positions. In the UNCERTAIN regime the gap narrows to 152 positions. In the RARECONFIDENT regime the ordering inverts: the smelly probe sits at rank 733 while the developer's project-specific name sits at rank 10,989. The model therefore distinguishes good from bad names at the output, but the magnitude and even the sign of the distinction varies with how generic the developer's name is.

At the level of context dependence, the same three regimes order monotonically by sensitivity. HIGHCONFIDENT positions have the smallest ΔH (2.01), RARECONFIDENT positions are intermediate (2.23), and UNCERTAIN positions are the most fluid (2.42). The regime where good and bad names are most cleanly separated at the output is also the regime whose prediction is least pulled by surrounding code; the regime where the rank ordering inverts responds to context only at an intermediate level.

At the level of the hidden state, smelly and clean inputs at the same position trace different trajectories through the network, but the difference is small for most of the trajectory. Cosine similarity sits between 0.35 and 0.50 for the first two-thirds of the layer stack and the first two-thirds of the denoising schedule, and rises to $\sim$0.60–0.65 only in the final third of each axis. The UMAP projections show partially overlapping smelly and clean clouds in the final layers and steps, separable in 2D but not cleanly. Per-token entropy change across the trajectory is 86% higher at smell positions than at ordinary context positions.

At the level of the unmasking schedule, smell positions reach the 0.8 confidence threshold at median step 13.6, while ordinary context positions reach it at median step 11.0. The gap is statistically separated but small in absolute terms: 2.6 steps out of a 32-step trajectory.

The four findings share a common shape. However, on every axis the separation is partial, regime-dependent, or concentrated in the late portion of the computation. The output distribution distinguishes good from bad names only in the HIGHCONFIDENT regime and inverts the ordering in RARECONFIDENT; the hidden-state divergence appears only after the first two-thirds of the network and the schedule have already passed; and smell positions commit slightly later than context positions, but not dramatically so. The internal correlate of the RQ1 advantage is therefore not a clean, early, or uniform signal throughout the network. It is a layered set of small differences that the model accumulates as it processes the input, becoming most distinct precisely in the part of the trajectory that produces the final commitment.

This shape also offers a reading of the scope boundary identified in RQ2. The cases in which the model handles well in RQ1, where the smell set contains a single identifier with semantic neighbors, are precisely the cases in which late-arriving internal differences have time to influence the final fill before the schedule locks in. In the multi-identifier obfuscated regime of RQ2, where surrounding identifiers carry no semantic content and the model must produce many distinct names in one pass, the same late-arriving differences are still computed but enter the trajectory at a point where most commitments have already been made.

VII. THREATS TO VALIDITY

We discuss threats to validity along the four categories established for empirical software engineering research [37]: internal, external, conclusion, and construct validity.

Internal validity concerns factors that may affect the dependent variables without the researcher's knowledge. The mask-token count k=2, the RQ3 confidence regime thresholds ($r^{\text{gt}} \leq 200$, $r^{\text{gt}} > 1000$), and the single-letter obfuscation alphabet are design choices made empirically rather than from theory; different choices would shift the absolute numbers, though the qualitative orderings we report are stable to moderate variation.

External validity concerns the ability to generalize the results. RefineID is Java-only and inherits the conventions of its upstream GitHub projects; the three dLLMs evaluated all use masked discrete diffusion with confidence-ordered unmasking; and the internal-state experiments in Section VI use DiffuCoder-7B-Base alone. The findings characterize this specific scope; continuous or score-based diffusion models, other languages, or other obfuscation regimes may show different patterns.

Conclusion validity concerns the possibility to draw correct conclusions about the relationship between treatments and outcomes. The Welch t-test and Mann–Whitney U test in Section VI are robust to the unequal sample sizes (158 vs. 19,838) we use. EM and LJ percentages are reported as point estimates without confidence intervals; the headline gaps we discuss ($\geq$ 10%) are large enough that this does not affect rankings, but finer comparisons should be read as suggestive.

Construct validity concerns the relationship between the concepts behind the experiment and what is measured. Exact

Match treats semantically equivalent but lexically distinct predictions as wrong; we mitigate this by reporting LJ alongside EM, and the relative model ordering is preserved under both. LJ itself uses Qwen2.5-7B-Instruct as judge and inherits the biases of an LLM evaluator (Appendix B). The RQ3 *smelly* class is operationalized through a curated vocabulary (`tmp`, `x`, `foo`, single-letter aliases) and applies to the obvious-smell regime rather than to subtler real-world smells.

VIII. Future Work

The results presented here open several directions for further investigation. The most direct extension is fine-tuning on regimes the present study has only evaluated zero-shot. DreamOn-7B's variable-canvas mechanism was trained for function-level infilling and operates near the lower bound of its useful range on RefineID, yet its 55.9% LJ alongside a 16.0% EM suggests the architecture produces semantically appropriate names but is mismatched to the identifier-scale granularity; fine-tuning DreamOn on identifier-scale infilling, and fine-tuning fixed-canvas dLLMs explicitly on the multi-identifier multi-site task exposed in RQ2, would test how far the joint multi-site advantage extends once the training distribution matches the deployment task.

A second direction follows from the internal-state findings of RQ3. The representation that distinguishes well-named from poorly-named identifiers exists, but it emerges in the final third of both the layer stack and the denoising trajectory—after the confidence-ordered unmasking schedule has typically confirmed most of its decisions. Training objectives that propagate this signal to earlier layers, or auxiliary heads that read the late-layer hidden state and feed it back into the unmasking schedule, would let the model use information that is currently present but arrives too late to influence its commitments, without requiring changes to the underlying diffusion architecture.

The third direction concerns the scope under which our conclusions were established. The RQ2 copy-paste bias is observed under a single obfuscation regime (single-letter alphabet) on Java code from a single benchmark; replicating the result under typo-adjacent, synonym-substitution, and randomized-hash obfuscation, and across languages with different naming conventions, would test whether lexical-style anchoring is a general property of dLLMs or specific to the regime we used. Verifying the RQ3 internal-state findings directly on DreamCoder-7B and on other diffusion large language models—including continuous and score-based variants that do not use confidence-ordered unmasking—would similarly distinguish architecture-specific behaviour from properties of masked discrete diffusion in general.

Finally, the pattern that emerges across the three research questions maps cleanly onto how production IDEs already identify rename candidates through grammar-based static analysis. A practical follow-up is to build and evaluate a hybrid pipeline in which the IDE supplies candidate positions and a dLLM, at the efficient $T{=}2$, $k{=}2$ configuration identified in Sections IV-C and IV-D, supplies the replacement names. Comparing such a pipeline against existing rule-based renaming on developer-facing metrics would translate the architectural findings of this thesis into a deployment recipe that can be assessed in real refactoring workflows.

IX. Conclusion

We presented CoReFusion, a study of masked diffusion large language models on Java identifier renaming framed as a multi-site denoising problem with dLLMs. Across three research questions, fixed-canvas dLLMs substantially outperform every FIM-AR and Seq2Seq baseline on the single-identifier multi-site case (33.2% vs. 23.5% EM, with the gap widening as the number of co-referring sites grows, RQ1). The advantage holds within scope. Under aggressive multi-identifier obfuscation, target-EM drops to 3–4%, and 71.3% of wrong predictions copy the lexical style of the obfuscated neighbors, sharpening rather than overturning the RQ1 result (RQ2). Inside DiffuCoder-7B-Base, four experiments show that a representation distinguishing well-named from poorly-named identifiers exists but is partial, regime-dependent, and concentrated in the final third of both the layer stack and the denoising trajectory (RQ3). These results point to dLLMs as effective multi-site filling engines when rename positions and semantic context are given (with the efficient $T{=}2$, $k{=}2$ configuration recovering 97.1% of peak EM at $32\times$ throughput), and to surfacing RQ3's late-trajectory internal differences earlier in the computation as a natural direction for future training objectives.

Acknowledgment

Finally, we are approaching the end of master level. I want to express my appreciation to all those who helped me, collaborated with me and supported me during this trip. First of all, my mother, Dan Gao. Without her support, I will never have the opportunity to study in TUDelft, and achieve that many results. Then it comes to my excellent supervisors, from Chair Prof.Dr.Arie van Deursen, to Daily Supervisor Dr. Maliheh Izadi and Ir. Jonathan Katzy. Moreover, I want to thank my girlfriend Tianye Ren and all my other friends who provide emotional support and energy during the process.

Here is a short list of all of them (Ranked by Alphabet of Last Name): Yiming Chen, Shenfei Gu, Xingyu Han, Chen Hu, Jian Yu Kei Jane, Shaoqi Wang, Ye Zhao

After all, I want to cheer for my own effort. I did it. I hope this journey leads towards the stars.

References

[1] Austin, J., Johnson, D.D., Ho, J., Tarlow, D., Van Den Berg, R.: Structured denoising diffusion models in discrete state-spaces. Advances in neural information processing systems **34**, 17981–17993 (2021)

[2] Bachmann, G., Nagarajan, V.: The pitfalls of next-token prediction (2025), https://arxiv.org/abs/2403.06963

[3] Baltes, S., Angermeir, F., Arora, C., Barón, M.M., Chen, C., Böhme, L., Calefato, F., Ernst, N., Falessi, D., Fitzgerald, B., Fucci, D., He, J., Treude, C., Kalinowski, M., Lambiase, S., Russo, D., Lungu, M., Montes, C.M., Prechelt, L., Ralph, P., van Tonder, R., Wagner, S.: Guidelines for Empirical Studies in Software Engineering involving Large Language Models (2026), https://arxiv.org/abs/2508.15503

[4] Bavarian, M., Jun, H., Tezak, N., Schulman, J., McLeavey, C., Tworek, J., Chen, M.: Efficient training of language models to fill in the middle. arXiv preprint arXiv:2207.14255 (2022)

[5] Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J.D., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., et al.: Language models are few-shot learners. Advances in neural information processing systems **33**, 1877–1901 (2020)

[6] Chang, H., Zhang, H., Jiang, L., Liu, C., Freeman, W.T.: Maskgit: Masked generative image transformer. In: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR). pp. 11315–11325 (June 2022)

[7] Cordeiro, J., Noei, S., Zou, Y.: An empirical study on the code refactoring capability of large language models. ACM Transactions on Software Engineering and Methodology (2024)

[8] Devlin, J., Chang, M.W., Lee, K., Toutanova, K.: BERT: Pre-training of deep bidirectional transformers for language understanding. In: Burstein, J., Doran, C., Solorio, T. (eds.) 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). pp. 4171–4186. Association for Computational Linguistics, Minneapolis, Minnesota (Jun 2019). https://doi.org/10.18653/v1/N19-1423, https://aclanthology.org/N19-1423/

[9] Dong, C., Jiang, Y., Niu, N., Zhang, Y., Liu, H.: Context-aware name recommendation for field renaming. In: Proceedings of the IEEE/ACM 46th International Conference on Software Engineering. ICSE '24, Association for Computing Machinery, New York, NY, USA (2024). https://doi.org/10.1145/3597503.3639195, https://doi.org/10.1145/3597503.3639195

[10] Erdil, K., Finn, E., Keating, K., Meattle, J., Park, S., Yoon, D.: Software maintenance as part of the software life cycle. Comp180: Software Engineering Project **1**, 1–49 (2003)

[11] Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D., et al.: Codebert: A pre-trained model for programming and natural languages. In: Findings of the association for computational linguistics: EMNLP 2020. pp. 1536–1547 (2020)

[12] Gong, L.: Advancing Large Language Models for Code Using Code-Structure-Aware Methods. Phd thesis, University of California, Berkeley, Berkeley, CA (Spring 2025), https://escholarship.org/uc/item/94x2m8zp, advisor: Alvin Cheung. Committee: Alvin Cheung (Chair), Xiaodong Song, Sida Wang. Also available as UCB/EECS-2025-50 technical report: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2025/EECS-2025-50.html. ProQuest Dissertation ID: 32041464

[13] Gong, S., Zhang, R., Zheng, H., Gu, J., Jaitly, N., Kong, L., Zhang, Y.: Diffucoder: Understanding and improving masked diffusion models for code generation (2025), https://arxiv.org/abs/2506.20639

[14] Guo, D., Zhu, Q., Yang, D., Xie, Z., Dong, K., Zhang, W., Chen, G., Bi, X., Wu, Y., Li, Y., et al.: Deepseek-coder: when the large language model meets programming–the rise of code intelligence. arXiv preprint arXiv:2401.14196 (2024)

[15] Hui, B., Yang, J., Cui, Z., Yang, J., Liu, D., Zhang, L., Liu, T., Zhang, J., Yu, B., Lu, K., Dang, K., Fan, Y., Zhang, Y., Yang, A., Men, R., Huang, F., Zheng, B., Miao, Y., Quan, S., Feng, Y., Ren, X., Ren, X., Zhou, J., Lin, J.: Qwen2.5-coder technical report (2024), https://arxiv.org/abs/2409.12186

[16] Jeljli, E., Sharma, T.: Refineid: A developer-centric ide assistant for better identifiers. SANER (2026)

[17] Jiang, L., Liu, H., Jiang, H.: Machine learning based recommendation of method names: How far are we. In: 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE). pp. 602–614 (2019). https://doi.org/10.1109/ASE.2019.00062

[18] Lewis, M., Liu, Y., Goyal, N., Ghazvininejad, M., Mohamed, A., Levy, O., Stoyanov, V., Zettlemoyer, L.: BART: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. In: Jurafsky, D., Chai, J., Schluter, N., Tetreault, J. (eds.) Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics. pp. 7871–7880. Association for Computational Linguistics, Online (Jul 2020). https://doi.org/10.18653/v1/2020.acl-main.703, https://aclanthology.org/2020.acl-main.703/

[19] Li, C., Zhang, Y., Li, J., Cai, L., Li, G.: Beyond autoregression: An empirical study of diffusion large language models for code generation (2025), https://arxiv.org/abs/2509.11252

[20] Li, R., Allal, L.B., Zi, Y., Muennighoff, N., Kocetkov, D., Mou, C., Marone, M., Akiki, C., Li, J., Chim, J., et al.: Starcoder: may the source be with you! arXiv preprint arXiv:2305.06161 (2023)

[21] Liu, H., Wang, Y., Wei, Z., Xu, Y., Wang, J., Li, H., Ji, R.: Refbert: A two-stage pre-trained framework for automatic rename refactoring. In: Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis. pp. 740–752 (2023)

[22] Lozhkov, A., Li, R., Allal, L.B., Cassano, F., Lamy-Poirier, J., Tazi, N., Tang, A., Pykhtar, D., Liu, J., Wei, Y., Liu, T., Tian, M., Kocetkov, D., Zucker, A., Belkada, Y., Wang, Z., Liu, Q., Abulkhanov, D., Paul, I., Li, Z., Li, W.D., Risdal, M., Li, J., Zhu, J., Zhuo, T.Y., Zheltonozhskii, E., Dade, N.O.O., Yu, W., Krauß, L., Jain, N., Su, Y., He, X., Dey, M., Abati, E., Chai, Y., Muennighoff, N., Tang, X., Oblokulov, M., Akiki, C., Marone, M., Mou, C., Mishra, M., Gu, A., Hui, B., Dao, T., Zebaze, A., Dehaene, O., Patry, N., Xu, C., McAuley, J., Hu, H., Scholak, T., Paquet, S., Robinson, J., Anderson, C.J., Chapados, N., Patwary, M., Tajbakhsh, N., Jernite, Y., Ferrandis, C.M., Zhang, L., Hughes, S., Wolf, T., Guha, A., von Werra, L., de Vries, H.: Starcoder 2 and the stack v2: The next generation (2024), https://arxiv.org/abs/2402.19173

[23] Mastropaolo, A., Aghajani, E., Pascarella, L., Bavota, G.: Automated variable renaming: are we there yet? Empirical Software Engineering **28**(2), 45 (2023)

[24] Onan, A., Alhumyani, H.A.: Codediffuse: A masked diffusion framework for structure-aware code completion and repair. Journal of King Saud University - Computer and Information Sciences **37**(8), 230 (2025). https://doi.org/10.1007/s44443-025-00237-6

[25] Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., Sutskever, I.: Language models are unsupervised multitask learners. OpenAI blog **1**(8), 9 (2019)

[26] Roziere, B., Gehring, J., Gloeckle, F., Sootla, S., Gat, I., Tan, X.E., Adi, Y., Liu, J., Sauvestre, R., Remez, T., et al.: Code llama: Open foundation models for code. arXiv preprint arXiv:2308.12950 (2023)

[27] von Rütte, D., Fluri, J., Pooladzandi, O., Schölkopf, B., Hofmann, T., Orvieto, A.: Scaling behavior of discrete diffusion language models (2026), https://arxiv.org/abs/2512.10858

[28] Sahoo, S.S., Arriola, M., Schiff, Y., Gokaslan, A., Marroquin, E., Chiu, J.T., Rush, A., Kuleshov, V.: Simple and effective masked diffusion language models. In: Globerson, A., Mackey, L., Belgrave, D., Fan, A., Paquet, U., Tomczak, J., Zhang, C. (eds.) Advances in Neural Information Processing Systems. vol. 37, pp. 130136–130184. Curran Associates, Inc. (2024). https://doi.org/10.52202/079017-4135, https://proceedings.neurips.cc/paper_files/paper/2024/file/eb0b13cc515724ab8015bc978fdde0ad-Paper-Conference.pdf

[29] Sevastjanova, R., Kalouli, A.L., Beck, C., Schäfer, H., El-Assady, M.: Explaining contextualization in language models using visual analytics. In: Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers). pp. 464–476 (2021)

[30] Shi, J., Yang, Z., He, J., Xu, B., Lo, D.: Can identifier splitting improve open-vocabulary language model of code? In: 2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER). pp. 1134–1138 (2022). https://doi.org/10.1109/SANER53432.2022.00130

[31] Sutskever, I., Vinyals, O., Le, Q.V.: Sequence to sequence learning with neural networks. In: Ghahramani, Z., Welling, M., Cortes, C., Lawrence, N., Weinberger, K. (eds.) Advances in Neural Information Processing Systems. vol. 27. Curran Associates, Inc. (2014), https://proceedings.neurips.cc/paper_files/paper/2014/file/5a18e133cbf9f257297f410bb7eca942-Paper.pdf

[32] Team, C., Zhao, H., Hui, J., Howland, J., Nguyen, N., Zuo, S., Hu, A., Choquette-Choo, C.A., Shen, J., Kelley, J., Bansal, K., Vilnis, L., Wirth, M., Michel, P., Choy, P., Joshi, P., Kumar, R., Hashmi, S., Agrawal, S., Gong, Z., Fine, J., Warkentin, T., Hartman, A.J., Ni, B., Korevec, K., Schaefer, K., Huffman, S.: Codegemma: Open code models based on gemma (2024), https://arxiv.org/abs/2406.11409

[33] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, Ł., Polosukhin, I.: Attention is all you need. Advances in neural information processing systems **30** (2017)

[34] Vijayvargiya, S., Saad, M., Sharma, T.: Enhancing identifier naming through multi-mask fine-tuning of language models of code. In: 2024 IEEE International Conference on Source Code Analysis and Manipulation (SCAM). pp. 71–82 (2024). https://doi.org/10.1109/SCAM63643.2024.00017

[35] Wang, Y., Le, H., Gotmare, A., Bui, N., Li, J., Hoi, S.: CodeT5+: Open code large language models for code understanding and generation. In: Bouamor, H., Pino, J., Bali, K. (eds.) Proceedings of the

2023 Conference on Empirical Methods in Natural Language Processing. pp. 1069–1088. Association for Computational Linguistics, Singapore (Dec 2023). https://doi.org/10.18653/v1/2023.emnlp-main.68, https://aclanthology.org/2023.emnlp-main.68/

[36] Wang, Y., Wang, W., Joty, S., Hoi, S.C.: CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In: Moens, M.F., Huang, X., Specia, L., Yih, S.W.t. (eds.) Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing. pp. 8696–8708. Association for Computational Linguistics, Online and Punta Cana, Dominican Republic (Nov 2021). https://doi.org/10.18653/v1/2021.emnlp-main.685, https://aclanthology.org/2021.emnlp-main.685/

[37] Wohlin, C., Runeson, P., H"ost, M., Ohlsson, M.C., Regnell, B., Wessl'en, A., et al.: Experimentation in software engineering, vol. 236. Springer (2012)

[38] Wu, Z., Chen, Y., Kao, B., Liu, Q.: Perturbed masking: Parameter-free probing for analyzing and interpreting BERT. In: Jurafsky, D., Chai, J., Schluter, N., Tetreault, J. (eds.) Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics. pp. 4166–4176. Association for Computational Linguistics, Online (Jul 2020). https://doi.org/10.18653/v1/2020.acl-main.383, https://aclanthology.org/2020.acl-main.383/

[39] Wu, Z., Zheng, L., Xie, Z., Ye, J., Gao, J., Gong, S., Feng, Y., Li, Z., Bi, W., Zhou, G., Kong, L.: Dreamon: Diffusion language models for code infilling beyond fixed-size canvas. In: The Fourteenth International Conference on Learning Representations (2026), https://openreview.net/forum?id=EQTPmqukiU

[40] Xie, Z., Ye, J., Zheng, L., Gao, J., Dong, J., Wu, Z., Zhao, X., Gong, S., Jiang, X., Li, Z., Kong, L.: Dream-coder 7b: An open diffusion language model for code (2025), https://arxiv.org/abs/2509.01142

[41] Zhang, J., Li, T., Zhang, X., Hu, Q., Shi, B.: Exploring the power of diffusion large language models for software engineering: An empirical investigation (2025), https://arxiv.org/abs/2510.04605

[42] Zheng, L., Chiang, W.L., Sheng, Y., Zhuang, S., Wu, Z., Zhuang, Y., Lin, Z., Li, Z., Li, D., Xing, E., Zhang, H., Gonzalez, J., Stoica, I.: Judging llm-as-a-judge with mt-bench and chatbot arena. In: Oh, A., Naumann, T., Globerson, A., Saenko, K., Hardt, M., Levine, S. (eds.) Advances in Neural Information Processing Systems. vol. 36, pp. 46595–46623. Curran Associates, Inc. (2023), https://proceedings.neurips.cc/paper_files/paper/2023/file/91f18a1287b398d378ef22505bf41832-Paper-Datasets_and_Benchmarks.pdf

Appendix A
AI Usage Disclosure

A. Tools Used

Two categories of AI tools were used during this research. Large language model assistants (ChatGPT and Claude) were used for learning English expressions, information retrieval, brainstorming, proofreading, and clarifying unfamiliar technical concepts during reading. AI-assisted coding tools (Codex and Claude Code) were used during the implementation of plotting utilities and data-processing pipelines.

B. Scope of Use in Writing

I used LLM assistants during the writing of this thesis in several distinct ways, none of which involved delegating the writing itself to the model. As a non-native English speaker, I used LLMs as a language-learning resource. When uncertain how to express a particular technical concept in English, I asked the assistant to suggest alternative phrasings, studied the options, and then decided how to apply what I had learned to the manuscript, with the goal of expanding my own range of expression rather than obtaining a finished sentence. I also used LLMs for information retrieval and conceptual brainstorming—identifying relevant prior work to read, gathering background on unfamiliar concepts encountered while reading the cited literature, and exploring framings for ideas before committing to one, where the output of these sessions informed my own reading and thinking rather than being transcribed into the manuscript. A further use was selective proofreading and structural review, including feedback on the framing of individual sections, the order in which results were presented, the coherence of connections between sections, and the removal of framing inconsistencies. I studied each suggestion and then decided whether to adopt, revise, or reject it. The remaining use was grammar and surface-level editions on finished text. In all of these uses, the research questions, the experimental designs, the interpretations of every numerical result, the choice of what to claim and what to leave open, and the final wording confirmed to the manuscript are mine. No paragraph was produced verbatim by an assistant, each was either written by me or rewritten by me from a suggestion that I studied and then incorporated.

C. Scope of Use in Code

AI assistance in code was used for boilerplate generation (inference loops, data loaders, plotting, CSV post-processing), suggesting candidate implementations for utility functions, and explaining error messages or library behavior during debugging. The experimental design, model configurations, evaluation protocols, statistical analyses, and the structure of the codebase were specified by the author. All AI-generated code was reviewed, executed, tested against expected outputs, and modified or rewritten before integration. No experimental results, numerical values, or figures were generated or modified by AI tools outside of plotting code that the author wrote or verified.

D. Excluded Uses

AI tools were not used to fabricate experimental results, numerical values, or statistical tests. They were not used to author the research questions or the conclusions of the thesis. They were not used as a replacement for reading the cited works: every reference cited in this thesis was identified, located, and read by the author.

E. Verification and Responsibility

All AI-assisted output, both prose and code, was reviewed and integrated by the author. Where an assistant proposed a framing or claim that did not match the evidence in the data, the author rejected it.

Appendix B
LLM-as-a-judge Setup

This appendix documents the LLM-as-a-judge setup used to compute LJ in the main text, following the reporting guidelines of [3]. The judge's raw output, the human-validation annotations, and the scripts that compute LJ from both are released with the supplementary material.

A. Judge model and generation configuration

The judge is Qwen2.5-7B-Instruct [15], loaded via the HuggingFace Transformers library, at `bfloat16` precision on a single NVIDIA A100 80GB GPU. The model is used off the shelf, with no fine-tuning or quantization setups. Generation uses `temperature`=0, `top_p`=1.0, and `max_new_tokens`=200. We chose an open-weights judge so that the entire evaluation can be replicated without access to a paid API.

B. Prompt Template

The template below is applied to every prediction from every model. The placeholder `ground_truth` denotes the original human-decided identifier in the dataset, and `prediction` is the predicted result from different models. The template is used unguided, which means there is no in-prompt demonstration provided during the process. We did not revise the template during the evaluation.

````
You are evaluating a code identifier-infilling
    task (RefineID). A specific identifier (
    variable, method, or type name) was **masked**
     in the original Java code, and a model
    predicted a replacement.
===================================================

GROUND TRUTH     : {ground_truth}
MODEL PREDICTION : {prediction}
===================================================

FULL CODE CONTEXT (the masked location is where
    the
identifier should appear):
```java
{full_code}
```
Evaluate the prediction by answering:
1. What is the relationship between the prediction
     and the
   ground truth in this exact context?
2. Would using the prediction instead of the
    ground truth
   preserve the program's correctness and
    readability?
Respond ONLY in this exact format (3 lines, no
    extra text):
Relationship: [Identical | Semantically Equivalent
     |
               Related but Different | Incorrect |
               Syntactically Invalid]
Explanation: [max 20 words]
Decision: [Accept | Reject]
````

The judge is asked for a relationship between the original ground truth and the generated result from the model, a short justification, and a binary decision. LJ is computed only from the `Decision` field result, any result with a decision other than Accept or Reject will be counted as a rejection. The labels and explanations are reserved for the disagreement analysis for the future but do not enter the score.

C. Human validation

To check that the judge's decisions track human judgment we manually checked random samples of the predictions drawn, so that each of the five relationships categories the judge produced is represented.

D. Known biases and limitations

LLM judges are known to exhibit positional and self-enhancement biases [42]. Positional bias does not apply here because the judge is asked about a single prediction. Self-enhancement bias is avoided by excluding every model under evaluation from the judge role, as Qwen2.5-7B-Instruct is not one of the models in the benchmark shown in Table II. We also report EM alongside LJ throughout the paper to make sure the EM is treated as additional evidence that LJ is not driving any conclusion on its own.

A separate limitation is that LJ measures agreement with the developer's chosen name rather than absolute naming quality. A prediction that is as good as the developer's choice but lexically different will often be rejected by the judge. LJ should therefore be read as an approximation of how often the model recovers the developer's intent, not as an estimate of how often the model produces a good name.

E. Scoring Logic

We map the judge's `Decision` field to a binary outcome per sample: `Accept` counts as 1, while `Reject` and any malformed output count as 0. LJ is the mean of these outcomes over the 1,000 test samples, expressed as a percentage:

$$\text{LJ} = \frac{100}{|\mathcal{D}|} \sum_{(\mathbf{x},\mathbf{x}^*)\in\mathcal{D}} \mathbf{1}\left[\texttt{Decision}_{\hat{\mathbf{x}}} = \texttt{Accept}\right] \ \%. \tag{5}$$

A model with LJ= 20.0% therefore corresponds to 200 `Accept` verdicts on the 1,000 samples.

Replication Package

The code, data, model configurations, prediction outputs, and analysis scripts used in this work are publicly available at https://github.com/d4vidhuang/corefusion.