DiffuTester: Accelerating Unit Test Generation for Diffusion LLMs via Mining Structural Pattern

Lekang Yang, Yuetong Liu, Yitong Zhang, Jia Li*
College of AI, Tsinghua University, Beijing, China
ylk22@mails.tsinghua.edu.cn, 23371522@buaa.edu.cn,
zhangyt26@mails.tsinghua.edu.cn, jia_li@mail.tsinghua.edu.cn

Abstract

Diffusion large language models (dLLMs) enable parallel generation and are promising for unit test generation (UTG), where efficient and large-scale automated testing is essential in software development. Despite this advantage, their application to UTG is still constrained by a clear trade-off between efficiency and test quality, since increasing the number of tokens generated in each step often causes a sharp decline in the quality of test cases. To overcome this limitation, we present DIFFUTESTER, an acceleration framework specifically tailored for dLLMs in UTG. The motivation of DIFFUTESTER is that unit tests targeting the same focal method often share structural patterns. DIFFUTESTER employs a novel structural pattern based decoding approach, which dynamically identifies structural patterns across unit tests through their abstract syntax trees and additionally decodes the corresponding tokens, thereby achieving acceleration without compromising the quality of the output. To enable comprehensive evaluation, we extend the original TestEval benchmark to three programming languages. Extensive experiments on three benchmarks with two representative models show that DIFFUTESTER delivers significant acceleration while preserving test coverage. Moreover, DIFFUTESTER generalizes well across different dLLMs and programming languages, providing a practical and scalable solution for efficient UTG in software development. Code and data are publicly available at https://github.com/THU-Agent/DiffuTester.

1 Introduction

Recently, Diffusion Large Language Models (dLLMs) (Nie et al., 2025b; Ye et al., 2025) are emerging as a promising paradigm because of their unique generation paradigm, which naturally supports multi-token prediction and enables faster generation. An increasing number of dLLM models (Xie et al., 2025; Gong et al., 2025) for code generation are emerging as a result of the high efficiency of dLLMs.

Unit testing plays a vital role in software development, ensuring that a functionally discrete program unit (*e.g.,*a method) behaves correctly and meets the intended design expectations (Olan, 2003; Gren and Antinyan, 2017). Real-world code repositories are often large, requiring developers to write a substantial number of unit tests (UTs) to cover each statement and branch (Robinson et al., 2011; Li et al., 2006), which is extremely time-consuming and labor-intensive (Runeson, 2006; Shang et al., 2025). In this context, efficiently generating unit tests becomes critically important, and dLLMs are particularly well-suited to this requirement.

Although dLLMs are theoretically capable of generating multiple tokens per denoising step, existing studies (Zeng et al., 2025; Barr et al., 2014) typically retain only one or two tokens per step to maintain generation quality. Our preliminary experiments show that naively increasing the number of retained tokens substantially accelerates decoding but sharply degrades the quality of generated unit tests (see Appendix C.2.5). This speed-quality trade-off motivates the need for a UTG-specific acceleration method that can decode more tokens per step while preserving test quality.

To address this limitation, we propose DIFFUTESTER to accelerate UTG for dLLMs. Our method is inspired by an observation that test cases targeting the same focal method often share similar structural patterns. In Figure 1(a), two mocking-based test cases exhibit nearly identical syntactic structures, while differing only in values and constants. Figure 1(b) further shows that even within a single complex test case with intricate setup sequences, repeated structural patterns frequently appear in setup and execution logic. To further validate this motivation, we randomly sample 100

*Corresponding author

Proceedings of the 2026 Conference on Empirical Methods in Natural Language Processing, pages 24029–24048
October 24-29, 2026 ©2026 Association for Computational Linguistics

(a) Unit test case with mocking.

```
def test_wrapper_invoked(self):
    wrapper = self.mock_wrapper()
    with connection.exe_wrapper(wrapper):
        self.call_execute(connection)
    self.assertTrue(wrapper.called)
    (_, sql, params, many, context), _ =
wrapper.call_args
    self.assertIn("SELECT", sql)
    self.assertIsNone(params)
    self.assertIs(many, False)
    self.assertEqual(context["connection"],
connection)
```

```
def test_wrapper_invoked_many(self):
    wrapper = self.mock_wrapper()
    with connection.exe_wrapper(wrapper):
        self.call_executemany(connection)
    self.assertTrue(wrapper.called)
    (_, sql, param_list, many, context), _
= wrapper.call_args
    self.assertIn("DELETE", sql)
    self.assertIsInstance(param_list,
(list, tuple))
    self.assertIs(many, True)
    self.assertEqual(context["connection"],
connection)
```

(b) Unit test case with intricate setup sequences.

```
def test_minimize_rollbacks_branchy(self):
    a1_impl = FakeMigration('a1'); a1 = ('a', '1')
    a2_impl = FakeMigration('a2'); a2 = ('a', '2')
    a3_impl = FakeMigration('a3'); a3 = ('a', '3')
    a4_impl = FakeMigration('a4'); a4 = ('a', '4')
...
    graph = MigrationGraph()
    graph.add_node(a1, a1_impl)
    graph.add_node(a2, a2_impl)
    graph.add_node(a3, a3_impl)
    graph.add_node(a4, a4_impl)
    graph.add_dependency(None, a2, a1)
    graph.add_dependency(None, a3, a1)
    graph.add_dependency(None, a4, a2)
...
```

(c) Structural prevalence in real_world repos.

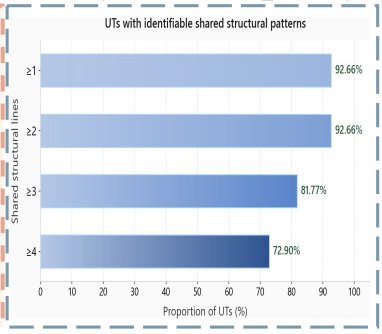

Figure 1: Real-world unit tests exhibit recurring structural patterns that can be exploited by DIFFUTESTER.

test files from real-world repositories, containing 4,646 unit tests in total. Using the AST-merge algorithm in DIFFUTESTER, we find that 92.66% of UTs contain at least two shared structural lines (as illustrated in Figure 1(c)). These results indicate that structural patterns are highly prevalent in real-world unit tests and can be effectively captured by DIFFUTESTER, thereby supporting our motivation.

To exploit structural patterns, DIFFUTESTER dynamically mines such patterns at each intermediate step of the diffusion process and additionally decodes the tokens corresponding to these patterns. Therefore it reduces the number of denoising steps by retaining more reliable tokens at each step, which leads to the core speedup of DIFFUTESTER. To enable this acceleration, DIFFUTESTER must first solve the problem of identifying structurally reliable tokens from incomplete test outputs. Abstract syntax trees (ASTs) provide a natural representation for separating syntactic structure from instance-specific lexical content in code. Based on this property, DIFFUTESTER parses the generated code into multiple ASTs and extracts the common nodes across them, which we regard as indicative of structural patterns inherent to the test cases.

We perform extensive experiments to validate the effectiveness of our proposed DIFFUTESTER on the `TestEval` benchmark (Wang et al., 2025) (whose focal methods are all implemented in Python) and new benchmarks collected by us (`TestEval-C++` and `TestEval-Java`) with two representative dLLMs, including `Dream` (Ye et al., 2025) and `DiffuCoder` (Gong et al., 2025). The results show that: ❶ DIFFUTESTER accelerates UTG by up to 2×–3× and effectively reduces computational cost. ❷ DIFFUTESTER preserves the maximum achievable test coverage while improving decoding efficiency. ❸ DIFFUTESTER is training-free and generalizes across different dLLMs, programming languages, code generation tasks, and real-world repository-level UTG scenarios, achieving about 1.9× speedup on `Tests4Py`.

2 Related Work

Diffusion language models have recently become a focal point in AI research (Sahoo et al., 2024; Liang et al., 2025; Zhang et al., 2026; Li et al., 2026), especially discrete diffusion language models (Nie et al., 2025b; DeepMind, 2025; Ye et al., 2025; Cheng et al., 2025). More and more dLLMs are also applied to code generation as a result of their high generation efficiency (Gong et al., 2025; Xie et al., 2025).

Various training-free strategies have been developed to accelerate the dLLMs generation process. One major class of acceleration methods is

based on KV-cache mechanisms, such as DLLM-Cache (Liu et al., 2025b) and Sparse-dLLM (Song et al., 2025a). Another class of acceleration methods is focused on advanced sampling techniques (Wei et al., 2025; Ben-Hamu et al., 2025; Israel et al., 2025; Yang et al., 2026). However, these general-purpose acceleration techniques are not tailored to UTG and may result in degradation of generation quality, whereas DIFFUTESTER leverages UTG-specific structural regularities to improve decoding efficiency while maintaining test quality.

3 DiffuTester

We propose DIFFUTESTER, an approach designed to accelerate dLLMs in unit test generation. The key idea is to mine shared structural patterns observed in multiple unit test cases generated at intermediate inference steps and then leverage them to guide token generation. We first provide a brief overview of the overall acceleration procedure (Section 3.1). Then we provide an in-depth explanation of how structural patterns are mined, which forms the core of our approach (Section 3.2). Finally, we introduce several additional techniques to further accelerate inference and ensure the quality of the generated test cases (Section 3.3). An overview of DIFFUTESTER is presented in Figure 2.

3.1 Overview of DIFFUTESTER

We detail the overview of DIFFUTESTER in this section. For a given focal method, we set the batch size to n (*e.g.,* $n = 3, 5, 7$) and prompt the dLLM to generate one unit test case in each instance, resulting in n unit test cases simultaneously. An illustration is shown in the left part of Figure 2

At one step t, for the k-th instance in the batch, the model first predicts tokens for all `[MASK]` positions, producing an intermediate response $\mathcal{Y}^{t,k} = (\hat{y}_i^{t,k})_{i=1}^{L}$. We then follow the standard confidence based decoding defined in existing dLLMs (*e.g.,* selecting the top-k tokens according to their confidence scores) to determine which tokens will be decoded (Ye et al., 2025; Xie et al., 2025; Gong et al., 2025). Subsequently and most importantly, we perform structural pattern based decoding. That's to say we extract structural patterns across the generated unit test cases and decode additional tokens corresponding to these patterns, with the detailed procedure presented in Section 3.2. After completing the procedure described above, we decode all selected tokens, while the remaining tokens at `[MASK]` positions are reverted back to `[MASK]`, and the process proceeds to the next step.

3.2 Structural Pattern Based Decoding

After the standard confidence based decoding, we perform structural pattern based decoding. The core is to mine structural patterns across multiple test cases. In this section, we detail how to mine these patterns through analysis of ASTs.

We begin by analyzing the key challenges involved in leveraging such structural patterns, which lie in two main aspects.

❶ The first challenge is **how to flexibly and dynamically extract structural patterns across test cases**. As illustrated in the first example in Figure 1, although structural patterns in test cases are highly pronounced, they are flexible and diverse, making them difficult to capture only through templates.

❷ The second challenge is **how to ensure sufficient diversity in the generated test cases**. Achieving high test coverage relies heavily on diversity (Peacock et al., 2021; Yang, 2023), yet the process of extracting shared patterns may risk reducing it. For instance, the model might generate identical input data across multiple test cases. Designing a method that can exploit shared patterns while preserving the diversity of test data is therefore another important challenge.

We next provide a detailed description of how our approach addresses these challenges and ultimately enables the effective utilization of structural patterns for acceleration.

Extract Structural Patterns. Abstract Syntax Trees (ASTs) can intuitively decouple structural information from lexical information in code. Leaf nodes of an AST provide lexical information while non-leaf nodes provide structural information. Based on this property, our approach compares the ASTs of different test cases' code and attempts to merge them as much as possible, which means identifying the nodes that are shared across different ASTs. When two or more ASTs can be merged into a non-empty tree, this indicates the existence of a structural pattern. We then locate the tokens in the intermediate test cases that correspond to the merged AST.

In practice, we find that unit test cases generated at early steps often contain many syntax errors. Such errors propagate through the parsing process. To address this issue, we construct ASTs at the

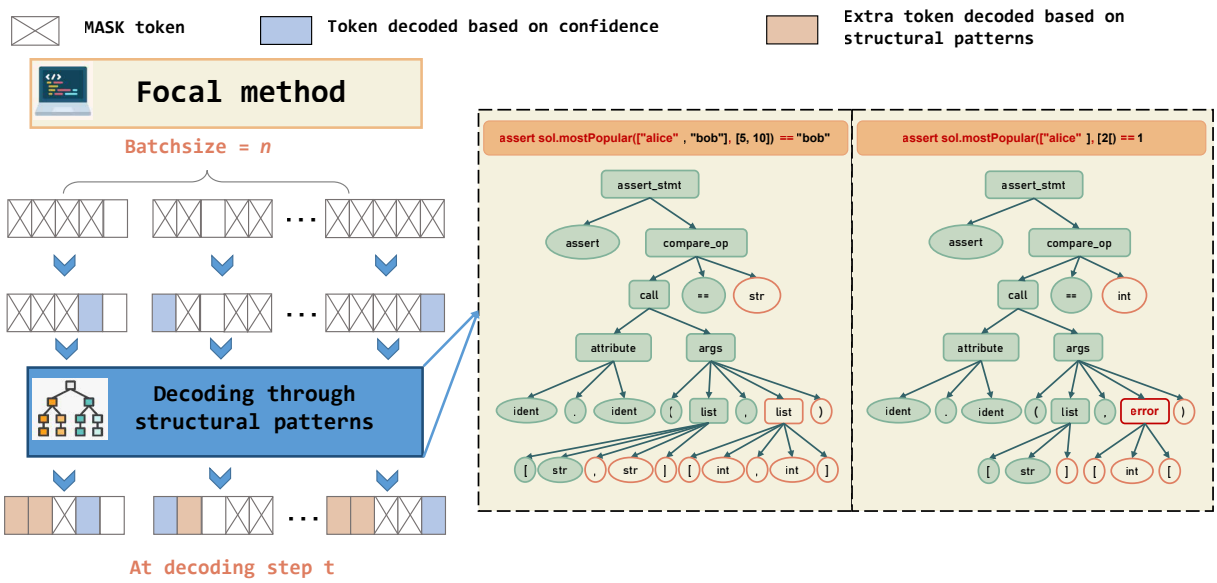

Figure 2: Overview of our proposed DIFFUTESTER.

granularity of individual code lines rather than entire test cases, as illustrated in Figure 2.

Ensure Sufficient Diversity. The diversity of unit test cases largely depends on the variability of the input data constructed for the focal method, which in turn is primarily determined by literal values such as integers or floats (Yang, 2023). To preserve this variability, we exclude the AST nodes corresponding to such literal values from the merging process. This design ensures that even if two test cases construct similar data at an intermediate step, the corresponding tokens are not retained in a single step but are instead remasked for subsequent refinement, allowing higher diversity.

After identifying the tokens that belong to the merged AST, we retain them in a single step, while the remaining tokens are remasked for refinement in the subsequent step. The details of the entire process are provided in Algorithm 1 and Algorithm 2 in the Appendix.

3.3 Additional Techniques of DIFFUTESTER

We additionally introduce two techniques to improve the quality of the generated test cases and to further accelerate the generation process.

❶ Our preliminary experiments show that directly decoding all tokens belonging to the merged AST can slightly reduce the syntactic correctness of the generated unit test cases. We hypothesize that this is mainly because tokens with very low confidence may occasionally be retained, while in practice, we observe that such cases are extremely rare. To address this issue, we retain only those tokens whose confidence exceeds a predefined threshold τ when guided by the merged AST, thereby ensuring that the retained tokens are more reliable for generation.

❷ Although parsing code to construct ASTs is computationally inexpensive, invoking the process at every step still introduces some overhead. Fortunately, we observe that the ASTs of code generated in consecutive steps change very little. Based on this observation, we choose not to apply DIFFUTESTER at every step, but instead to use it intermittently after several steps, which leads to faster acceleration. However, dLLM inference still accounts for the majority of the overall runtime. Based on a simple empirical analysis, the inference time of the dLLM is approximately 10× greater than the computational cost of the AST operations.

4 Experiment

In this section, we systematically evaluate the performance of DIFFUTESTER in accelerating unit test generation. Additional experimental results including ablation studies are provided in Appendix C.2.

4.1 Experiment setup

Models. We evaluate our approach using two representative dLLMs, namely DIFFUCODER-7B-CPGRPO (Gong et al., 2025) and DREAM-V0-INSTRUCT-7B (Ye et al., 2025). For brevity, we refer to them as `DiffuCoder` and `Dream` in the remainder of this paper.

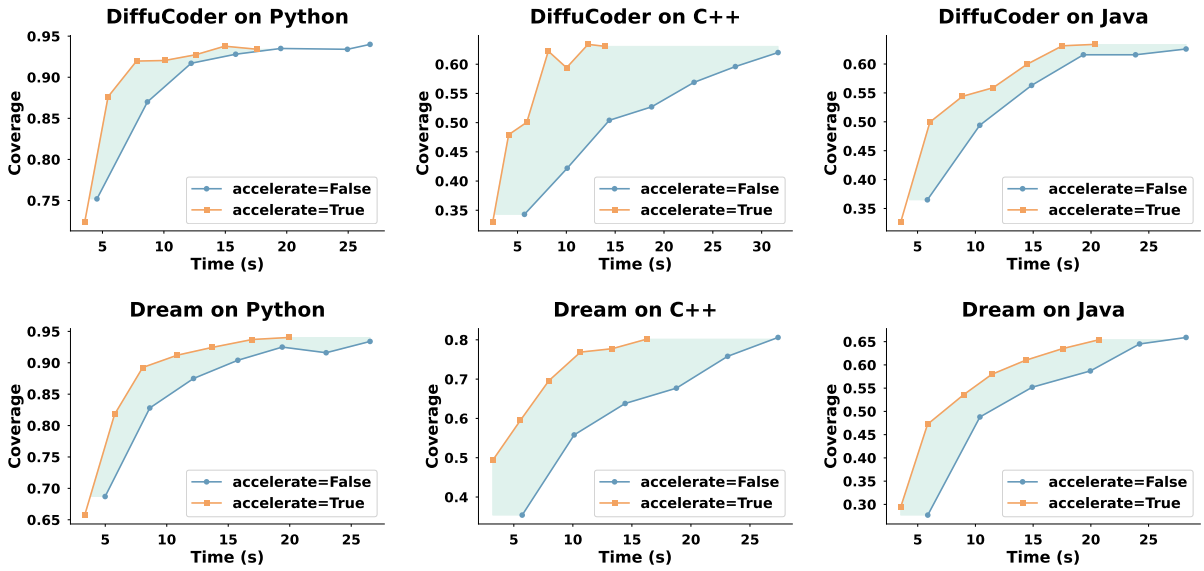

Figure 3: Comparison of line coverage with and without DIFFUTESTER at equal decoding time.

Benchmarks. To evaluate the performance of DIFFUTESTER, we conduct experiments on the `TestEval` benchmark (Wang et al., 2025). `TestEval` is specifically designed to assess the capability of models in unit test generation, and it comprises 210 Python programs collected from LeetCode. We denote this benchmark as `TestEval-Python`. To further validate the generalization ability of DIFFUTESTER across different programming languages and to avoid potential data contamination, we extend the `TestEval-Python` benchmark by incorporating two additional programming languages: C++ and Java. Following the construction methodology of `TestEval-Python`, we systematically gather corresponding C++ and Java implementations of the same set of 210 programs. We denote these benchmarks as `TestEval-C++` and `TestEval-Java`, respectively.

Evaluation Metrics. We adopt four widely used metrics. ❶ *Computational Cost (tflops)*: the average computation per batch. ❷ *Decoding Time (seconds, s)*: the average inference time per batch. ❸ *Throughput (tokens/s, tps)*: the average number of tokens generated per second per batch, excluding special tokens such as `[PAD]` and `[EOS]`. ❹ *Line Coverage*: the average ratio of the number of code lines covered by the generated test cases to the total number of code lines.

Baseline. Since there is currently no acceleration approach specifically tailored to dLLMs for UTG, we take dLLMs without applying DIFFUTESTER as the baseline.

Implementation Details. We set the predefined generation length L to 128 and the confidence threshold τ to 0.02. In addition, we apply DIFFUTESTER once every two steps. Further implementation details are provided in Appendix C.1.3.

4.2 Main Results

We conduct evaluations on the three benchmarks and two representative models described in Section 4.1. We use different batch sizes n, where n ranges from 1 to 7, corresponding to the number of test cases generated per focal method. Figure 3 presents the average test coverage achieved under varying time budgets, while Table 1 reports the efficiency comparison between DIFFUTESTER and the baseline when generating the same number of test cases. And qualitative case studies are provided in Appendix C.4.

DIFFUTESTER can effectively reduce the time needed to achieve a certain coverage rate. In the pipeline of most software development, the time available for unit testing is often limited, which makes achieving a certain coverage as fast as possible highly desirable. As shown in Figure 3 and Table 1, to achieve the same coverage, DIFFUTESTER can speedup up to $2\times - 3\times$ compared with the baseline. For example, with a batch size $n = 3$ on `TestEval-C++` using `DiffuCoder`, DIFFUTESTER reduces the decoding time from 14.4s to 6.0s, while increasing throughput from 9.7 TPS to 23.8 TPS. This demonstrates that the proposed DIFFUTESTER is highly competitive and

enables dLLMs to achieve higher test coverage more quickly in UTG.

DIFFUTESTER can effectively reduce the computational cost needed to generate test cases. Across nearly all settings, dLLMs achieve more than a 1.6× improvement in computational cost when equipped with DIFFUTESTER. For example, with a batch size $n = 3$ on `TestEval-C++` using `DiffuCoder`, DIFFUTESTER reduces the computational cost from 1217 TFLOPs to 430 TFLOPs. These results demonstrate that DIFFUTESTER can substantially save computational resources.

DIFFUTESTER does not compromise the maximum achievable test coverage. The highest test coverage attainable without time constraints is another important optimization target, particularly for scenarios such as vehicle control systems (Conrad and Fey, 2017; Zhang et al., 2024). We find that DIFFUTESTER has little to no negative impact on the maximum test coverage and in some cases even improves it. For example, as shown in Figure 3, on the `TestEval-Python` benchmark with the `Dream` model, applying DIFFUTESTER leads to a slight increase in max test coverage. We further evaluate branch coverage in Appendix C.2.3, where the trends are consistent with line coverage.

DIFFUTESTER generalizes well across different models and programming languages. Across various models and programming languages, DIFFUTESTER exhibits consistent acceleration trends and similar effects on coverage. Notably, the acceleration effect of DIFFUTESTER is more pronounced on `TestEval-C++` compared to the other two benchmarks. For example, with a batch size of 5 using `DiffuCoder`, throughput improves by 1.55× on `TestEval-Python` and 1.71× on `TestEval-Java`, whereas on `TestEval-C++` it increases by 2.45×. We attribute this to the fact that commonly used syntactic structures in C++ exhibit greater structural similarity than those in Python and Java, which makes it easier to exploit structural patterns for acceleration.

Comparison with an AR baseline: We also conducted an experiment comparing dLLM equipped with DIFFUTESTER with an auto-regressive model for reference. Results show that dLLMs equipped with DIFFUTESTER require less time and computational cost to achieve the same coverage in most cases compared to auto-regressive models of similar scale. The details of the experiment are provided in Appendix C.2.2.

5 Discussion

5.1 Applicability to Real-World Repositories

To examine whether DIFFUTESTER also applies to realistic UTG scenarios, we conduct an additional evaluation on 135 samples from **Tests4Py** (Smytzek et al., 2023), which contains real bugs from popular open-source repositories such as FastAPI (Ramírez), Keras (Keras team), and Scrapy (Scrapy contributors). Using `DiffuCoder` as the base dLLM, we generate 2–5 unit tests for each prompt and report decoding time and line coverage in Table 2. DIFFUTESTER achieves about 1.9× speedup on average while maintaining comparable line coverage, suggesting that its structural-pattern-based acceleration can generalize beyond algorithmic benchmarks to real-world repositories. These results suggest that the structural-pattern-based acceleration of DIFFUTESTER is not limited to algorithmic benchmarks, but can also provide practical efficiency gains in real-world repository-level UTG scenarios without sacrificing test coverage. More details are provided in Appendix C.2.1.

5.2 Comparison with dLLM Acceleration for General Task

Training-free dLLM acceleration methods mainly include cache-based and sampling-based techniques, while DIFFUTESTER belongs to the latter and remains compatible with cache-based methods. We therefore compare DIFFUTESTER with two representative sampling-based approaches, EB-SAMPLER (Ben-Hamu et al., 2025) and SLOW-FAST SAMPLING (Wei et al., 2025), which are designed for general-purpose decoding rather than UTG. For experiments validating the compatibility between DIFFUTESTER and cache-based methods, refer to Appendix C.2.4

EB-SAMPLER accelerates dLLM inference through an Entropy-Bounded unmasking procedure, which dynamically unmasks multiple tokens under a predefined approximate error tolerance. We conduct experiments on the `TestEval-Python`, testing batch sizes of $n = 3, 5, 7$, and report both average decoding time and line coverage in Table 3.

❶ Compared with EB-SAMPLER, DIFFUTESTER achieves more effective acceleration in generating unit tests. For example, when $n = 7$, DIFFUTESTER reduces the average decoding time to 17.6s, whereas EB-SAMPLER reduces it only to 19.8s.

Table 1: Efficiency comparison with and without DIFFUTESTER across different batch sizes n on `TestEval-Python`, `TestEval-C++`, and `TestEval-Java`, with `Dream` and `DiffuCoder`.

Method	$n = 3$			$n = 5$			$n = 7$		
	Computational Cost (tflops)	Decoding Time (s)	Throughput (tps)	Computational Cost (tflops)	Decoding Time (s)	Throughput (tps)	Computational Cost (tflops)	Decoding Time (s)	Throughput (tps)
				`TestEval-Python`					
`DiffuCoder`	1015.59	12.22	16.97	1692.65	19.51	17.69	2369.72	26.78	18.09
+ DIFFUTESTER	580.36	7.77	26.86	997.16	12.59	27.45	1432.00	17.57	27.42
speedup	×1.75	×1.57	×1.58	×1.70	×1.55	×1.55	×1.65	×1.52	×1.52
`Dream`	1015.59	12.18	15.61	1692.66	19.39	15.96	2369.72	26.53	16.68
+ DIFFUTESTER	605.05	8.04	23.59	1093.38	13.68	23.09	1644.05	19.95	22.45
speedup	×1.68	×1.51	×1.51	×1.55	×1.42	×1.45	×1.44	×1.33	×1.35
				`TestEval-C++`					
`DiffuCoder`	1216.98	14.40	9.73	2098.93	23.08	9.62	2924.4	31.68	9.58
+ DIFFUTESTER	429.82	5.95	23.81	777.32	10.02	23.60	1121.37	14.00	23.20
speedup	×2.83	×2.42	×2.45	×2.70	×2.30	×2.45	×2.61	×2.26	×2.42
`Dream`	1216.98	14.43	13.11	2028.30	23.11	13.45	2839.62	31.60	13.88
+ DIFFUTESTER	588.17	7.96	25.93	1041.20	13.33	25.45	1614.85	19.97	24.82
speedup	×2.07	×1.81	×1.98	×1.95	×1.73	×1.89	×1.76	×1.58	×1.79
				`TestEval-Java`					
`DiffuCoder`	1259.36	14.86	16.10	2098.93	23.85	16.69	2924.41	32.59	16.71
+ DIFFUTESTER	667.65	8.88	29.17	1143.00	14.46	28.58	1649.17	20.33	28.20
speedup	×1.89	×1.67	×1.81	×1.84	×1.65	×1.71	×1.77	×1.60	×1.69
`Dream`	1259.36	14.92	15.75	2098.93	24.22	15.79	2938.51	32.64	13.58
+ DIFFUTESTER	673.40	8.97	27.76	1130.74	14.34	28.12	1678.98	20.72	27.35
speedup	×1.87	×1.66	×1.76	×1.86	×1.69	×1.78	×1.75	×1.58	×2.01

Table 2: Decoding time and line coverage on Tests4Py, comparing the baseline `DiffuCoder` and DIFFUTESTER.

Method	Decoding Time (s)				Line Coverage (%)			
	$n = 2$	$n = 3$	$n = 4$	$n = 5$	$n = 2$	$n = 3$	$n = 4$	$n = 5$
`DiffuCoder`	19.98	30.24	40.46	44.79	26.0	**26.9**	**30.0**	29.2
+ DIFFUTESTER	**10.27** (×1.95)	**15.86** (×1.91)	**21.18** (×1.91)	**24.18** (×1.85)	**26.3**	26.3	28.7	**30.5**

Table 3: Decoding time and line coverage on `TestEval-Python` with batch sizes $n = 3, 5, 7$, comparing the baseline `DiffuCoder`, EB-SAMPLER, and DIFFUTESTER.

Method	Decoding Time (s)			Line Coverage (%)		
	$n = 3$	$n = 5$	$n = 7$	$n = 3$	$n = 5$	$n = 7$
`DiffuCoder`	12.2	19.5	26.8	91	**94**	94
+ EB-SAMPLER	8.9	14.3	19.8	86	88	88
+ DIFFUTESTER	**7.8** (×1.56)	**12.6** (×1.54)	**17.6** (×1.52)	**92**	93	**94**

❷ In addition, the quality of the test cases generated by DIFFUTESTER is substantially higher. With DIFFUTESTER, the line coverage remains almost identical to the baseline without any acceleration, while EB-SAMPLER leads to a significant drop in coverage.

SLOWFAST SAMPLING is a novel dynamic sampling strategy that adaptively alternates between exploratory and accelerated decoding stages. We conduct experiments on the `TestEval-Python`, testing batch sizes of $n = 7, 10$ (for SLOWFAST SAMPLING) and report the average decoding time and line coverage in Table 4.

❶ Although SLOWFAST SAMPLING is the fastest at the same testing batch size, it substantially degrades generation quality, achieving only 77% line coverage even when $n = 10$, underperforming even the baseline configuration with $n = 3$.

Table 4: Decoding time and line coverage on `TestEval-Python`, comparing the baseline `DiffuCoder`, SLOWFAST SAMPLING, and DIFFUTESTER.

Method	Decoding Time (s)			Line Coverage (%)		
	$n = 3$	$n = 7$	$n = 10$	$n = 3$	$n = 7$	$n = 10$
`DiffuCoder`	12.2	26.8	36.1	91	94	94
+ SLOWFAST SAMPLING	**2.4**	**5.4**	**7.7**	73	77	77
+ DIFFUTESTER	7.8 (×1.56)	17.6 (×1.52)	24.5 (×1.47)	**92**	**94**	**94**

❷ Moreover, SLOWFAST SAMPLING attains identical line coverage at $n = 7$ and $n = 10$, indicating that its peak coverage (77%) is significantly reduced by the acceleration strategy. In contrast, DIFFUTESTER accelerates generation while preserving generation quality.

Compared to the two acceleration methods discussed earlier, DIFFUTESTER can achieve good speedup without sacrificing generation quality. We attribute the superior performance of DIFFUTESTER in terms of both efficiency and quality to its ability to exploit structural patterns that are inherent to the UTG task. This task-specific insight allows DIFFUTESTER to accelerate generation without compromising the quality of the resulting test cases.

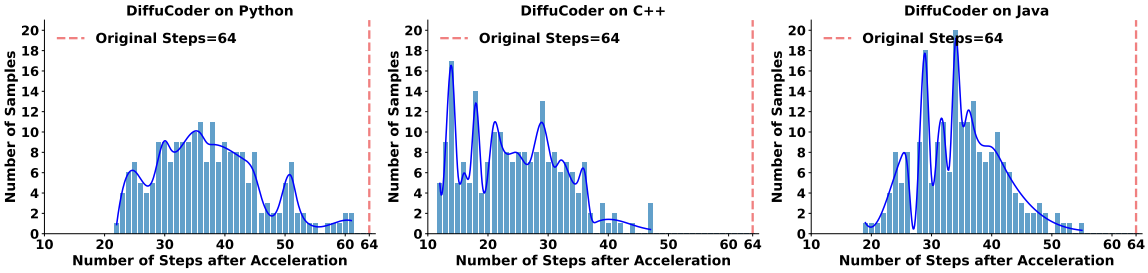

Figure 4: Distribution of the number of inference steps after applying DIFFUTESTER on DiffuCoder. The red dashed line indicates the original fixed step number without acceleration.

Table 5: Decoding time (s) and pass@1 (%) on other code generation tasks using DiffuCoder.

General Code Generation	Python		C++		Java	
	Time	Pass@1	Time	Pass@1	Time	Pass@1
DiffuCoder	5.68	**50**	5.73	11	6.32	11
+ DIFFUTESTER	**3.87** (×1.47)	49	**4.21** (×1.36)	**13**	**5.36** (×1.18)	**11**
Code Translation	**Python to C++**		**C++ to Java**		**Java to Python**	
	Time	Pass@1	Time	Pass@1	Time	Pass@1
DiffuCoder	11.75	42.9	11.97	39.0	12.29	95.8
+ DIFFUTESTER	**10.34** (×1.14)	**46.2**	**10.06** (×1.19)	**40.1**	**10.00** (×1.18)	**97.5**
Code Translation	**C++ to Python**		**Java to C++**		**Python to Java**	
	Time	Pass@1	Time	Pass@1	Time	Pass@1
DiffuCoder	11.98	**96.4**	12.31	**57.0**	11.76	35.2
+ DIFFUTESTER	**9.59** (×1.25)	95.5	**10.57** (×1.16)	56.1	**10.26** (×1.15)	**36.2**

5.3 Applying DIFFUTESTER to Other Code Generation Tasks

We also evaluate our method on other code generation tasks. Although these tasks may not exhibit repeated patterns as prominently as UTG, when multiple samples are generated, repeated patterns across samples can still be captured by DIFFUTESTER and exploited to accelerate generation. For general code generation, we use the Python, C++ and Java subsets of the **HumanEval-X** benchmark (Zheng et al., 2023). For code translation, we adopt datasets from **UniTrans** (Yang et al., 2024b). The experimental results are summarized in Table 5.

We conduct the experiments using DiffuCoder to generate 4 samples each prompt and report the average decoding time and pass@1[1]. DIFFUTESTER achieves a modest acceleration over the baseline. The acceleration is less pronounced than in UTG because repeated structures in these code generation tasks are not as salient as those in UTG. Nevertheless, DIFFUTESTER generally does not degrade generation quality and, in some cases, even yields higher-quality outputs.

5.4 Distribution of Inference Steps with DIFFUTESTER

In this section, we analyze the distribution of the number of inference steps after applying DIFFUTESTER. The experiments are conducted on DiffuCoder across three benchmarks, with the predefined generation length set to 128. Under the default remasking strategy, the model decodes two tokens at each step, which results in 64 inference steps without DIFFUTESTER. The distribution of inference steps after applying DIFFUTESTER is shown in Figure 4. We observe that the number of steps follows an approximately normal distribution, with most cases concentrated at values significantly smaller than 64. Moreover, the number of inference steps adapts to the characteristics of each focal method, making the approach well-suited for diverse testing scenarios.

6 Conclusion

In this paper, we present an in-depth analysis of the pervasive structural patterns in unit test generation and the characteristics of dLLMs' inference. Building on this analysis, we propose DIFFUTESTER, the first acceleration framework specifically designed for dLLMs in unit test generation. The central idea of DIFFUTESTER is to extract structural patterns in intermediate unit test cases and leverage them to decode a larger number of tokens per inference step. Extensive experimental results demonstrate that DIFFUTESTER can substantially accelerate dLLMs in unit test generation while preserving test coverage, and that it generalizes effectively across different dLLMs and programming languages. We believe that this work provides a practical and scalable solution for improving the efficiency of unit test generation in software development.

[1]Average pass rate of 4 samples

7 Limitations

Due to limited time and computational resources, we did not evaluate our method on additional models or code generation tasks. However, we believe that our approach is generally applicable to other dLLMs and code generation tasks.

8 Ethical Considerations

Our work focuses on improving software quality assurance and does not involve any foreseeable ethical risks. The benchmark used in our experiments is constructed entirely from publicly available sources, without the use of private or otherwise sensitive information. No human subjects, personally identifiable data, or sensitive content are involved in this research.

Acknowledgments

We sincerely thank the anonymous reviewers for their valuable comments and constructive feedback, which helped improve this work. We also thank the authors and developers of the models, datasets, and open-source resources used in this study for making their work publicly available and facilitating research in this area. This work was supported by the National Natural Science Foundation of China (No. 62602364), the Beijing Natural Science Foundation (No. 4264107), and the Tsinghua University–Keystone Electrical (Zhejiang) Co., Ltd. Joint Research Center for Embodied Multimodal Artificial Intelligence (JCEMAI).

References

Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, and 1 others. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*.

Maurício Aniche. 2022. *Effective Software Testing: A developer's guide*. Simon and Schuster.

Marianne Arriola, Subham Sekhar Sahoo, Aaron Gokaslan, Zhihan Yang, Zhixuan Qi, Jiaqi Han, Justin T Chiu, and Volodymyr Kuleshov. 2025. Block diffusion: Interpolating between autoregressive and diffusion language models. In *The Thirteenth International Conference on Learning Representations*.

Roberto Baldoni, Emilio Coppa, Daniele Cono D'Elia, Camil Demetrescu, and Irene Finocchi. 2016. A survey of symbolic execution techniques.

Earl Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. 2014. The oracle problem in software testing: A survey. *IEEE Transactions on Software Engineering*, 41:1–1.

Heli Ben-Hamu, Itai Gat, Daniel Severo, Niklas Nolte, and Brian Karrer. 2025. Accelerated sampling from masked diffusion models via entropy bounded unmasking. *arXiv preprint arXiv:2505.24857*.

Yoshua Bengio and Yann LeCun. 2007. Scaling learning algorithms towards AI. In *Large Scale Kernel Machines*. MIT Press.

Shreya Bhatia, Tarushi Gandhi, Dhruv Kumar, and Pankaj Jalote. 2024. Unit test generation using generative ai: A comparative performance analysis of autogeneration tools. In *Proceedings of the 1st International Workshop on Large Language Models for Code*, pages 54–61.

Arianna Blasi, Alessandra Gorla, Michael Ernst, and Mauro Pezzè. 2023. Call me maybe: Using nlp to automatically generate unit test cases respecting temporal constraints. pages 1–11.

Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. volume 8, pages 209–224.

Charlie Chen, Sebastian Borgeaud, Geoffrey Irving, Jean-Baptiste Lespiau, Laurent Sifre, and John Jumper. 2023. Accelerating large language model decoding with speculative sampling. *Preprint*, arXiv:2302.01318.

Haorui Chen, Yuancheng Zhu, Yitong Zhang, and Jia Li. 2026. Coact: Action-preserving observation compression for coding agents. *Preprint*, arXiv:2607.02911.

Mouxiang Chen, Zhongxin Liu, He Tao, Yusu Hong, David Lo, Xin Xia, and Jianling Sun. 2024. B4: Towards optimal assessment of plausible code solutions with plausible tests. ASE '24, page 1693–1705, New York, NY, USA. Association for Computing Machinery.

Shuang Cheng, Yihan Bian, Dawei Liu, Linfeng Zhang, Qian Yao, Zhongbo Tian, Wenhai Wang, Qipeng Guo, Kai Chen, Biqing Qi, and Bowen Zhou. 2025. Sdar: A synergistic diffusion-autoregression paradigm for scalable sequence generation. *Preprint*, arXiv:2510.06303.

Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. 2011. S2e: A platform for in-vivo multi-path analysis of software systems. *Acm Sigplan Notices*, 46(3):265–278.

Mirko Conrad and Ines Fey. 2017. Testing automotive control software. In *Automotive Embedded Systems Handbook*, pages 11–1. CRC Press.

Christoph Csallner, Nikolai Tillmann, and Yannis Smaragdakis. 2008. Dysy: dynamic symbolic execution for invariant inference. pages 281–290.

Ermira Daka and Gordon Fraser. 2014. A survey on unit testing practices and problems. In *2014 IEEE 25th International Symposium on Software Reliability Engineering*, pages 201–211.

Arghavan Moradi Dakhel, Amin Nikanjam, Vahid Majdinasab, Foutse Khomh, and Michel C Desmarais. 2024. Effective test generation using pre-trained large language models and mutation testing. *Information and Software Technology*, 171:107468.

Google DeepMind. 2025. Gemini diffusion.

Pedro Delgado-Pérez, Aurora Ramírez, Kevin Valle-Gómez, Inmaculada Medina-Bulo, and José Raúl Romero. 2022. Interevo-tr: Interactive evolutionary test generation with readability assessment. *IEEE Transactions on Software Engineering*, PP:1–17.

Chunyuan Deng, Yilun Zhao, Xiangru Tang, Mark Gerstein, and Arman Cohan. 2023. Investigating data contamination in modern benchmarks for large language models. *arXiv preprint arXiv:2311.09783*.

Gordon Fraser and Andrea Arcuri. 2011. Evosuite: Automatic test suite generation for object-oriented software. pages 416–419.

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

Ian Goodfellow, Yoshua Bengio, Aaron Courville, and Yoshua Bengio. 2016. *Deep learning*, volume 1. MIT Press.

Lucas Gren and Vard Antinyan. 2017. On the relation between unit testing and code quality. In *2017 43rd Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pages 52–56. IEEE.

Marton Havasi, Brian Karrer, Itai Gat, and Ricky Chen. 2025. Edit flows: Flow matching with edit operations.

Geoffrey E. Hinton, Simon Osindero, and Yee Whye Teh. 2006. A fast learning algorithm for deep belief nets. *Neural Computation*, 18:1527–1554.

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

Daniel Israel, Guy Van den Broeck, and Aditya Grover. 2025. Accelerating diffusion llms via adaptive parallel decoding. *arXiv preprint arXiv:2506.00413*.

Kush Jain, Gabriel Synnaeve, and Baptiste Rozière. 2025. Testgeneval: A real world unit test generation and test completion benchmark. *Preprint*, arXiv:2410.00752.

Keras team. Keras.

Samar Khanna, Siddhant Kharbanda, Shufan Li, Harshit Varma, Eric Wang, Sawyer Birnbaum, Ziyang Luo, Yanis Miraoui, Akash Palrecha, Stefano Ermon, and 1 others. 2025. Mercury: Ultra-fast language models based on diffusion. *arXiv preprint arXiv:2506.17298*.

Jaeyeon Kim, Lee Cheuk-Kit, Carles Domingo-Enrich, Yilun Du, Sham Kakade, Timothy Ngotiaoco, Sitan Chen, and Michael Albergo. 2025a. Any-order flexible length masked diffusion.

Jaeyeon Kim, Kulin Shah, Vasilis Kontonis, Sham M. Kakade, and Sitan Chen. 2025b. Train for the worst, plan for the best: Understanding token ordering in masked diffusions. In *Forty-second International Conference on Machine Learning*.

Caroline Lemieux, Jeevana Priya Inala, Shuvendu K Lahiri, and Siddhartha Sen. 2023. Codamosa: Escaping coverage plateaus in test generation with pre-trained large language models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 919–931. IEEE.

Yaniv Leviathan, Matan Kalman, and Yossi Matias. 2023. Fast inference from transformers via speculative decoding. *Preprint*, arXiv:2211.17192.

Chengze Li, Yitong Zhang, Jia Li, Liyi Cai, Ge Li, and 1 others. 2025a. Beyond autoregression: An empirical study of diffusion large language models for code generation. *arXiv preprint arXiv:2509.11252*.

Jinsong Li, Xiaoyi Dong, Yuhang Zang, Yuhang Cao, Jiaqi Wang, and Dahua Lin. 2025b. Beyond fixed: Training-free variable-length denoising for diffusion large language models. *Preprint*, arXiv:2508.00819.

Tianyi Li, Mingda Chen, Bowei Guo, and Zhiqiang Shen. 2025c. A survey on diffusion language models. *arXiv preprint arXiv:2508.10875*.

Yaohang Li, Tao Dong, Xinyu Zhang, Yong-duan Song, and Xiaohong Yuan. 2006. Large-scale software unit testing on the grid. In *GrC*, pages 596–599.

Zherui Li, Zheng Nie, Zhenhong Zhou, Yue Liu, Yitong Zhang, Yu Cheng, Qingsong Wen, Kun Wang, Yufei Guo, and Jiaheng Zhang. 2026. Diffuguard: How intrinsic safety is lost and found in diffusion large language models. *Preprint*, arXiv:2509.24296.

Zhixuan Liang, Yizhuo Li, Tianshuo Yang, Chengyue Wu, Sitong Mao, Liuao Pei, Xiaokang Yang, Jiangmiao Pang, Yao Mu, and Ping Luo. 2025. Discrete diffusion vla: Bringing discrete diffusion to action decoding in vision-language-action policies. *arXiv preprint arXiv:2508.20072*.

Xiaoran Liu, Zhigeng Liu, Zengfeng Huang, Qipeng Guo, Ziwei He, and Xipeng Qiu. 2025a. Longllada: Unlocking long context capabilities in diffusion llms. *Preprint*, arXiv:2506.14429.

Zhiyuan Liu, Yicun Yang, Yaojie Zhang, Junjie Chen, Chang Zou, Qingyan Wei, Shaobo Wang, and Linfeng Zhang. 2025b. dllm-cache: Accelerating diffusion large language models with adaptive caching.

Stephan Lukasczyk and Gordon Fraser. 2022. Pynguin: Automated unit test generation for python.

Phil McMinn. 2004. Search-based software test data generation: a survey: Research articles. *Softw. Test., Verif. Reliab.*, 14:105–156.

Niels Mündler, Jasper Dekoninck, and Martin Vechev. 2025. Constrained decoding of diffusion llms with context-free grammars. *arXiv preprint arXiv:2508.10111*.

Shen Nie, Fengqi Zhu, Chao Du, Tianyu Pang, Qian Liu, Guangtao Zeng, Min Lin, and Chongxuan Li. 2025a. Scaling up masked diffusion models on text. *Preprint*, arXiv:2410.18514.

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

Michael Olan. 2003. Unit testing: Test early, test often. *Journal of Computing Sciences in Colleges - JCSC*, 19.

Carlos Pacheco, Shuvendu Lahiri, Michael Ernst, and Thomas Ball. 2007. Feedback-directed random test generation. pages 75–84.

Samuel Peacock, Lin Deng, Josh Dehlinger, and Suranjan Chakraborty. 2021. Automatic equivalent mutants classification using abstract syntax tree neural networks. In *2021 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 13–18. IEEE.

Mihir Prabhudesai, Mengning Wu, Amir Zadeh, Katerina Fragkiadaki, and Deepak Pathak. 2025. Diffusion beats autoregressive in data-constrained settings. *Preprint*, arXiv:2507.15857.

Sebastián Ramírez. FastAPI.

Brian Robinson, Michael D Ernst, Jeff H Perkins, Vinay Augustine, and Nuo Li. 2011. Scaling up automated test generation: Automatically generating maintainable regression unit tests for programs. In *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*, pages 23–32. IEEE.

Per Runeson. 2006. A survey of unit testing practices. *IEEE Software*, 23.

Subham Sahoo, Marianne Arriola, Yair Schiff, Aaron Gokaslan, Edgar Marroquin, Justin Chiu, Alexander Rush, and Volodymyr Kuleshov. 2024. Simple and effective masked diffusion language models. *Advances in Neural Information Processing Systems*, 37:130136–130184.

Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. 2023. An empirical evaluation of using large language models for automated unit test generation. *IEEE Transactions on Software Engineering*, PP:1–21.

Scrapy contributors. Scrapy.

Ye Shang, Quanjun Zhang, Chunrong Fang, Siqi Gu, Jianyi Zhou, and Zhenyu Chen. 2025. A large-scale empirical study on fine-tuning large language models for unit testing. *Proceedings of the ACM on Software Engineering*, 2(ISSTA):1678–1700.

Marius Smytzek, Martin Eberlein, Batuhan Serce, Lars Grunske, and Andreas Zeller. 2023. Tests4py: A benchmark for system testing. *Preprint*, arXiv:2307.05147.

Yuerong Song, Xiaoran Liu, Ruixiao Li, Zhigeng Liu, Zengfeng Huang, Qipeng Guo, Ziwei He, and Xipeng Qiu. 2025a. Sparse-dllm: Accelerating diffusion llms with dynamic cache eviction. *arXiv preprint arXiv:2508.02558*.

Yuxuan Song, Zheng Zhang, Cheng Luo, Pengyang Gao, Fan Xia, Hao Luo, Zheng Li, Yuehang Yang, Hongli Yu, Xingwei Qu, Yuwei Fu, Jing Su, Ge Zhang, Wenhao Huang, Mingxuan Wang, Lin Yan, Xiaoying Jia, Jingjing Liu, Wei-Ying Ma, and 3 others. 2025b. Seed diffusion: A large-scale diffusion language model with high-speed inference. *Preprint*, arXiv:2508.02193.

Weisong Sun, Chunrong Fang, Yun Miao, Yudu You, Mengzhe Yuan, Yuchen Chen, Quanjun Zhang, An Guo, Xiang Chen, Yang Liu, and 1 others. 2023. Abstract syntax tree for programming language understanding and representation: How far are we? *arXiv preprint arXiv:2312.00413*.

Vasin Suttichaya, Niracha Eakvorachai, and Tunchanok Lurkraisit. 2022. Source code plagiarism detection based on abstract syntax tree fingerprintings. In *2022 17th International Joint Symposium on Artificial Intelligence and Natural Language Processing (iSAI-NLP)*, pages 1–6. IEEE.

Michele Tufano, Dawn Drain, Alexey Svyatkovskiy, Shao Kun Deng, and Neel Sundaresan. 2020. Unit test case generation with transformers and focal context. *arXiv preprint arXiv:2009.05617*.

Junjie Wang, Yuchao Huang, Chunyang Chen, Zhe Liu, Song Wang, and Qing Wang. 2024. Software testing with large language models: Survey, landscape, and vision. *IEEE Transactions on Software Engineering*, 50(4):911–936.

Wenhan Wang, Chenyuan Yang, Zhijie Wang, Yuheng Huang, Zhaoyang Chu, Da Song, Lingming Zhang, An Ran Chen, and Lei Ma. 2025. Testeval: Benchmarking large language models for test case generation. *Preprint*, arXiv:2406.04531.

Qingyan Wei, Yaojie Zhang, Zhiyuan Liu, Dongrui Liu, and Linfeng Zhang. 2025. Accelerating diffusion large language models with slowfast: The three golden principles. *arXiv preprint arXiv:2506.10848*.

Chengyue Wu, Hao Zhang, Shuchen Xue, Zhijian Liu, Shizhe Diao, Ligeng Zhu, Ping Luo, Song Han, and Enze Xie. 2025a. Fast-dllm: Training-free acceleration of diffusion llm by enabling kv cache and parallel decoding.

Zirui Wu, Lin Zheng, Zhihui Xie, Jiacheng Ye, Jiahui Gao, Yansong Feng, Zhenguo Li, Victoria W., Guorui Zhou, and Lingpeng Kong. 2025b. Dreamon: Diffusion language models for code infilling beyond fixed-size canvas.

Zhihui Xie, Jiacheng Ye, Lin Zheng, Jiahui Gao, Jingwei Dong, Zirui Wu, Xueliang Zhao, Shansan Gong, Xin Jiang, Zhenguo Li, and Lingpeng Kong. 2025. Dream-coder 7b.

Kaisen Yang, Jayden Teoh, Kaicheng Yang, Yitong Zhang, and Alex Lamb. 2026. Improving sampling for masked diffusion models via information gain. *Preprint*, arXiv:2602.18176.

Lin Yang, Chen Yang, Shutao Gao, Weijing Wang, Bo Wang, Qihao Zhu, Xiao Chu, Jianyi Zhou, Guangtai Liang, Qianxiang Wang, and 1 others. 2024a. On the evaluation of large language models in unit test generation. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, pages 1607–1619.

Ling Yang, Ye Tian, Bowen Li, Xinchen Zhang, Ke Shen, Yunhai Tong, and Mengdi Wang. 2025. Mmada: Multimodal large diffusion language models. *arXiv preprint arXiv:2505.15809*.

Yixiao Yang. 2023. Improve model testing by integrating bounded model checking and coverage guided fuzzing. *Electronics*, 12(7):1573.

Zhen Yang, Fang Liu, Zhongxing Yu, Jacky Wai Keung, Jia Li, Shuo Liu, Yifan Hong, Xiaoxue Ma, Zhi Jin, and Ge Li. 2024b. Exploring and unleashing the power of large language models in automated code translation. *Proc. ACM Softw. Eng.*, 1(FSE).

Jiacheng Ye, Zhihui Xie, Lin Zheng, Jiahui Gao, Zirui Wu, Xin Jiang, Zhenguo Li, and Lingpeng Kong. 2025. Dream 7b: Diffusion large language models. *arXiv preprint arXiv:2508.15487*.

Zebin You, Shen Nie, Xiaolu Zhang, Jun Hu, Jun Zhou, Zhiwu Lu, Ji-Rong Wen, and Chongxuan Li. 2025. Llada-v: Large language diffusion models with visual instruction tuning. *arXiv preprint arXiv:2505.16933*.

Zhiqiang Yuan, Mingwei Liu, Shiji Ding, Kaixin Wang, Yixuan Chen, Xin Peng, and Yiling Lou. 2024. Evaluating and improving chatgpt for unit test generation. *Proceedings of the ACM on Software Engineering*, 1:1703–1726.

Lucas Zamprogno, Braxton Hall, Reid Holmes, and Joanne M Atlee. 2022. Dynamic human-in-the-loop assertion generation. *IEEE Transactions on Software Engineering*, 49(4):2337–2351.

Yiming Zeng, Jinghan Cao, Zexin Li, Yiming Chen, Tao Ren, Dawei Xiang, Xidong Wu, Shangqian Gao, and Tingting Yu. 2025. Treediff: Ast-guided code generation with diffusion llms. *arXiv preprint arXiv:2508.01473*.

Tianyuan Zhang, Lu Wang, Xinwei Zhang, Yitong Zhang, Boyi Jia, Siyuan Liang, Shengshan Hu, Qiang Fu, Aishan Liu, and Xianglong Liu. 2024. Visual adversarial attack on vision-language models for autonomous driving. *arXiv preprint arXiv:2411.18275*.

Yitong Zhang, Yongmin Li, Yuetong Liu, Jia Li, Xiaoran Jia, Zherui Li, and Ge Li. 2026. Lookahead-then-verify: Reliable constrained decoding for diffusion llms under context-free grammars. *Preprint*, arXiv:2602.00612.

Siyan Zhao, Devaansh Gupta, Qinqing Zheng, and Aditya Grover. 2025. d1: Scaling reasoning in diffusion large language models via reinforcement learning. In *ES-FoMo III: 3rd Workshop on Efficient Systems for Foundation Models*.

Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Zihan Wang, Lei Shen, Andi Wang, Yang Li, Teng Su, Zhilin Yang, and Jie Tang. 2023. Codegeex: A pre-trained model for code generation with multilingual benchmarking on humaneval-x. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pages 5673–5684.

Fengqi Zhu, Rongzhen Wang, Shen Nie, Xiaolu Zhang, Chunwei Wu, Jun Hu, Jun Zhou, Jianfei Chen, Yankai Lin, Ji-Rong Wen, and Chongxuan Li. 2025. Llada 1.5: Variance-reduced preference optimization for large language diffusion models. *Preprint*, arXiv:2505.19223.

A LLM Usage Statement

We used Large Language Models solely as a tool to assist with the linguistic aspects of our manuscript. Specifically, the model was employed to help with translation, refine grammar, enhance clarity, improve conciseness, and optimize word choice. At no point did the model contribute any new ideas, original content, or substantive changes to the manuscript. Its function was limited strictly to editing and language optimization, ensuring that the scientific integrity and originality of our work remained entirely unaffected by the use of the model.

B Reproducibility statement

To ensure the reproducibility of our results, we have made all relevant code and datasets publicly available at https://github.com/THU-Agent/DiffuTester. The repository includes detailed instructions for setting up the environment, running experiments, and reproducing all results presented in the paper. We encourage the community to use these resources to verify our findings and to facilitate further research in this area.

All existing artifacts used in this work, including models, benchmarks, and source code repositories, were utilized in a manner consistent with their original licenses and intended research purposes. Our extended benchmark is constructed entirely from publicly available sources and is released solely for research and evaluation purposes. The source code and benchmark of this project are released under the MIT License.

C Additional Discussion

C.1 Detailed Setup

The results reported in Table 1 were collected on a server with 8 NVIDIA A100-SXM4-40G GPUs (PCIe between CPU and GPU), and two Intel Xeon Gold 6348 CPUs, with 28 cores per socket and 56 physical cores in total.

C.1.1 Models

We primarily conducted experiments using the dLLMs `DiffuCoder-7B-cpGRPO` (Gong et al., 2025) (`DiffuCoder` in this paper for short) and `Dream-v0-Instruct-7B` (Ye et al., 2025) (`Dream` in this paper for short).

DiffuCoder-7B-cpGRPO is a code generation dLLM developed by Apple. It is a refined variant of DiffuCoder-Instruct, further improved using Coupled-GRPO reinforcement learning. The model is initialized from DiffuCoder-7B-Instruct and post-trained on 21,000 code samples for one epoch.

Dream-v0-Instruct-7B is a variant of the Dream 7B model developed by the HKU NLP Group with 7 billion parameters. It is an instruction-tuned (SFT) large diffusion language model. It is trained on a mixture of text, math, and code data, leveraging weight initialization from auto-regressive models for more efficient learning, and it demonstrates strong performance on general, coding, and reasoning tasks.

C.1.2 Metrics

The following provides a detailed description of the metrics measured in our main experiments.

Coverage. A software testing metric indicating the percentage of source code lines (or branches) exercised by the generated test cases. Higher coverage generally means that the generated tests explore more of the code base, leading to better test quality. In our experiments, we use line coverage, calculated as the number of code lines covered by the test cases divided by the total number of code lines.

Computational Cost (tflops). The overall amount of computation required by the model during inference reflects the hardware resources needed to complete the task. In our experiments, we measure the average computational cost to generate a certain number of test cases consumed per problem across the entire benchmark.

Time (s). Time needed during inference to generate a certain number of test cases. In our experiments, we measure the average inference time consumed per problem across the entire benchmark.

Throughput (tps). The number of tokens generated per second (tokens per second, tps). It reflects how many outputs can be produced in parallel and is critical for large-scale deployment. In our experiments, throughput is calculated as the total number of tokens generated across the entire benchmark (excluding special tokens such as `[EOS]` and `[PAD]`) divided by the total GPU time used.

C.1.3 Hyperparameters

Hyperparameters we used in the main experiments are listed in Table 6:

In our experiments with the EB-SAMPLER method, we adopted the parameters recommended in the original paper (Ben-Hamu et al., 2025). We used confidence as the *error proxy function*. And

Table 6: Hyperparameters used in the main experiments.

Model	Language	Steps	Temperature	Max length	Alg. temp.[2]	Threshold[3]	Step size[4]
DiffuCoder	Python	64	1.5	128	0.2	0.02	2
	C++	64	1.5	192	0.2	0.02	2
	Java	64	1.5	192	0.2	0.02	2
Dream	Python	64	1.0	128	0.2	0.02	2
	C++	64	1.0	192	0.2	0.02	2
	Java	64	1.0	192	0.2	0.02	2

we used $\gamma = 0.1$.

In our experiments with the SLOWFAST SAMPLING (Wei et al., 2025) method, we adopted default parameters in the implementation by the authors.[5]

C.1.4 Benchmark

To evaluate the performance of our method, we experiment on the TestEval benchmark (Wang et al., 2025). TestEval is a benchmark for test case generation with LLMs. It includes 210 Python programs from an online programming platform, LeetCode. It has three metrics: overall coverage, targeted line/branch coverage, and targeted path coverage.

To further assess the generalizability of our method across multiple programming languages, we extend the TestEval benchmark by collecting corresponding C++ and Java implementations for the same set of 210 problems. These additional solutions are obtained from a publicly available GitHub repository licensed under the MIT license (see https://github.com/walkccc/LeetCode for details).

Building on TestEval, we also adapted the evaluation code for line coverage and branch coverage in Java and C++. For coverage measurement, we used pytest for Python, Maven for Java, and gcov for C++.

C.2 More Experiments

C.2.1 Details of the experiment on Tests4Py

In this section, we describe the evaluation pipeline for our experiments on Tests4Py.

We first collect the source code of all bugs included in Tests4Py[6] and, for simplicity, retain only bugs localized within a single function. We then build the execution environment for each bug in each repository. Bugs for which the environment cannot be successfully built are discarded, resulting in a final benchmark of 135 instances.

We then construct prompts using the code of the target function together with surrounding contextual code from the same file. The hyperparameters used in this experiment are identical to those used for `TestEval-Python`.

C.2.2 Comparison with AR model

In this section, we present a comparison between our method and an AR baseline model.

From Figure 5, we can see that with DIFFUTESTER, dLLMs require substantially less time and computational cost to achieve the same coverage in most cases compared to auto-regressive models of similar scale.

In this experiment, the AR baseline employs a strictly vanilla decoding procedure—without any acceleration techniques (e.g., speculative decoding (Leviathan et al., 2023; Chen et al., 2023)) or quality-enhancing strategies (e.g., codamosa (Lemieux et al., 2023)). The reason for not applying the acceleration techniques is that we primarily aim to explore the fundamental advantages of dLLMs over AR models at the foundation model level. Applying additional acceleration strategies to AR models would compromise the fairness of comparison, and moreover, these techniques could also be applied to dLLMs in the future. The reason for not applying the quality-enhancing strategies is that our goal is to accelerate UTG rather than to produce higher-quality unit tests. Such quality-enhancing techniques can likewise be incorporated to improve the test cases generated by dLLMs.

C.2.3 More Coverage Metrics

In this section, we present some experimental results using branch coverage.

From Figure 6, we can see that using branch coverage as a metric yields results that are highly consistent with those of line coverage. That is the reason why we only reported line coverage in the

[5] https://github.com/LiangrunFlora/Slow-Fast-Sampling/blob/main/slow-fast-sampling/sampling_utils/dream_generation_utils.py

[6] https://github.com/smythi93/Tests4Py

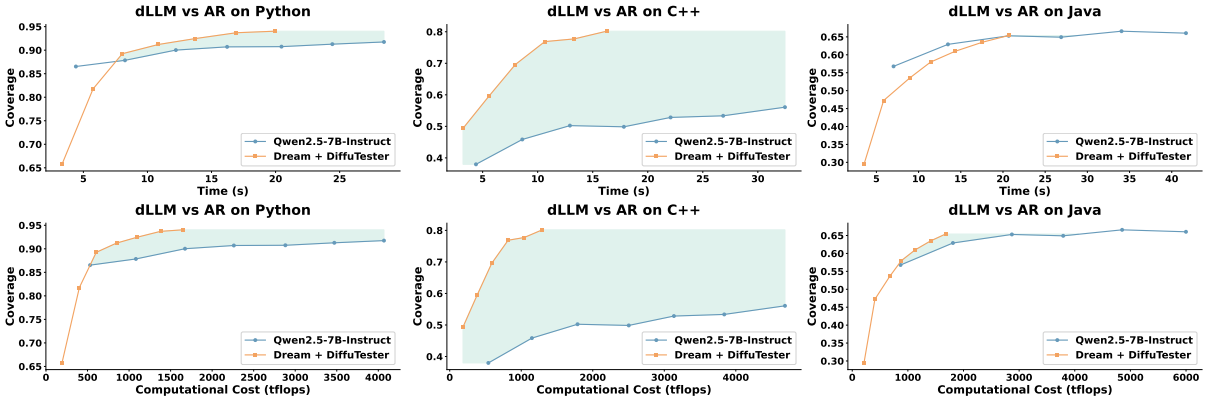

Figure 5: Dream + DIFFUTESTER vs Qwen-2.5-7B-Instruct

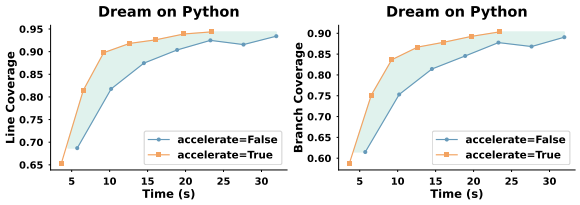

Figure 6: Comparison of line/branch coverage with and without DIFFUTESTER at equal decoding time.

Table 7: Comparison of decoding efficiency and generation quality on `TestEval-Python`.

Method	Decoding Time (s)	Throughput (TPS)	Line Coverage (%)
`DiffuCoder`	18.73	18.44	**93.6**
+ dLLMCACHE	8.26	41.62	91.9
+ dLLMCACHE & DIFFUTESTER	**5.82**	**58.67**	92.8

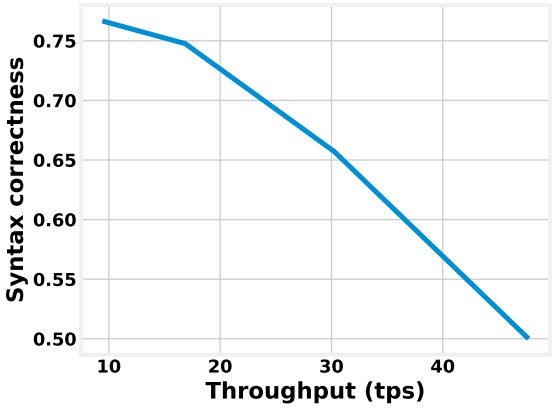

Figure 7: Syntactic correctness of the code generated with different generation speeds.

main results.

C.2.4 Comparison with KV-Cache Methods

DIFFUTESTER is a sampling-based technique and is therefore naturally compatible with cache-based acceleration methods. To further verify this compatibility, we evaluated `DiffuCoder` on `TestEval-Python` with $n = 5$, using the cache-based acceleration method dLLM-Cache (Liu et al., 2025b). The results are reported in Table 7.

The results show that dLLM-Cache substantially accelerates the generation process, while DIFFUTESTER can further improve generation efficiency on top of it without sacrificing generation quality.

C.2.5 Quality vs Speed

In this section, we demonstrate the syntactic correctness of the code generated with different dLLM generation speeds. Different speeds are caused by varying the number of tokens unmasked per step. The experiment is conducted on `DiffuCoder` model on Python language.

From Figure 7, we can see that unmasking more tokens at each step can obviously improve speed, but it also degrades generation quality. The results highlight a clear trade-off between generation speed and syntactic correctness. As the number of tokens unmasked per step increases, the model is able to generate outputs at a substantially higher throughput. However, this acceleration comes at the cost of syntactic quality, as evidenced by the marked decline in syntax correctness.

C.2.6 Ablation Study on Threshold

During the dLLM denoising process, each unmasked token is assigned a confidence score, which reflects how certain the model is about its prediction. This confidence can be measured in two common ways: (1) as the probability of the selected token according to the model's output distribution, or (2) as the negative entropy of the token distribution, which captures the overall uncertainty across all possible tokens. In all of our experiments, we simply use the first method, i.e., the probability of

the selected token, to represent confidence.

In our method, only tokens whose confidence is greater than the specified threshold are unmasked. In this section, we show the influence of this threshold.

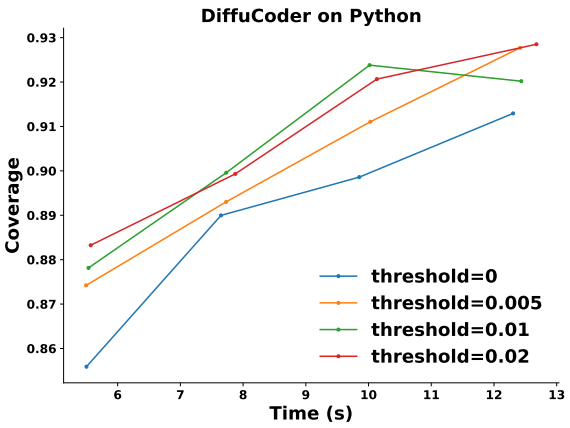

Figure 8: Coverage vs Time for different thresholds.

From Figure 8, we can know that accelerating without a threshold (threshold = 0) leads to a decrease in the final coverage. That is because the method potentially results in syntactically incorrect code, since it attempts to match even when syntax errors are present. A proper threshold (like 0.02) alleviates it.

C.2.7 Ablation Study on Step Size

We observed that applying the method at every denoising step is not necessary; using it only at certain steps has minimal impact on generation quality and the number of extra unmasked tokens, while significantly reducing the computational overhead introduced by the algorithm.

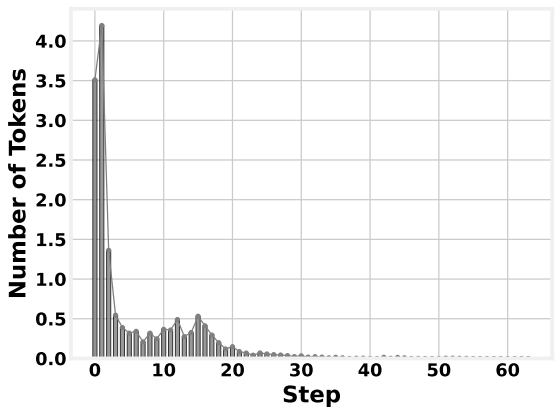

Figure 9: Number of Tokens Accelerated by Our Method for Every Step.

We conducted experiments utilizing both fixed-interval and dynamically adjusted step size strategies for applying our method. In the dynamically adjusted approach, the method is applied with greater frequency during the initial steps, while the interval between successive applications increases linearly in later steps. This design is motivated by our observation that the method is able to unmask a greater number of additional tokens during earlier steps compared to later ones, as illustrated in Figure 9. Empirically, we find that while the dynamically adjusted step size yields competitive results, a constant step size of 2 ultimately achieves the best overall performance.

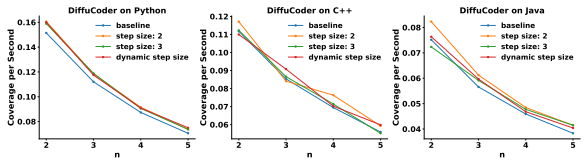

Figure 10: Results of different step sizes.

Figure 10 compares the results of different fixed step sizes and the dynamic step size strategy. The y-axis represents the average coverage divided by the average time, while the x-axis denotes the number of test cases generated simultaneously. The higher y-value means better performance. The curve labeled *baseline* is the case in which we apply our method every step. From the figure, we can know that not applying the method to every step produces better results. In general, a constant step size of 2 yields the best results.

C.2.8 Ablation Study on `[PAD]` Token

In this section, we examine the impact of a straightforward yet effective technique for further accelerating the generation process. During training, `[PAD]` tokens are appended to the ends of sequences to ensure uniform length, resulting in `[PAD]` tokens exclusively appearing at sequence termini in standard text. Building on this observation, we propose the following strategy: once a `[MASK]` token is decoded as a `[PAD]` token during inference, all subsequent positions in the sequence are immediately assigned as `[PAD]` tokens. This approach leverages the inherent structure of the training data to bypass unnecessary computations for positions that are, by construction, expected to be padding, thereby yielding additional improvements in generation efficiency.

Results are shown in Figure 11. The y-axis rep-

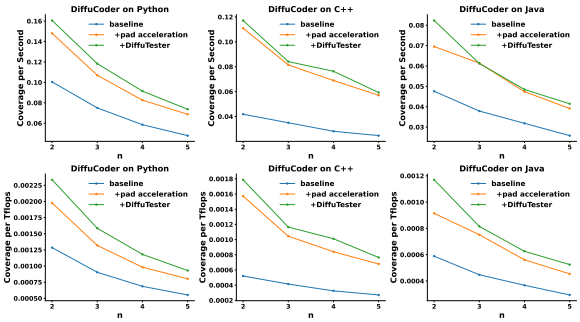

Figure 11: Acceleration result of pad acceleration and pattern acceleration.

resents the average coverage divided by the average time or average computational cost, while the x-axis denotes the number of test cases generated simultaneously. The higher y-value means better performance. The curve labeled *baseline* means no acceleration method applied. The curve labeled +*pad acceleration* means applying this trick on the baseline, while the curve labeled +DIFFUTESTER means applying both this trick and our method.

From the results, we can see that this simple trick can also accelerate the generation process, while our AST-based pattern mining method further accelerates the process.

C.3 Prompts

The system prompt template we used is as follows:

```
You are a professional who writes
{language} testcases. You always
respond without any natural
language descriptions.
Especially, your answer should
contain only one testcase.
```

The prompt template we used in different programming languages is as follows:

Python:

```
Please write a test method for
the function '{func_name}' given
the following program under test
and function description. Your
answer should only contain one
test input.

Program under test:
```python
{program}
```

Function description for
'{func_name}':
```txt
{description}
```

Your testcase should begin with:
```python
def test_{func_name}():
    solution = Solution()
```
```

Java:

```
Please write a test method for
the function '{func_name}' given
the following program under test
and function description. Your
answer should only contain one
test input.

Program under test:
```java
{program}
```

Function description for
'{func_name}':
```txt
{description}
```

You can directly use
`assertEquals` function for
assertion. Your testcase should
be formatted as:
```java
public class SolutionTest{
  @Test
  public void test_{func_name}(){

  }
}
```
```

C++:

```
Please write a test method for
the function '{func_name}' given
the following program under test
and function description. Your
answer should only contain one
test input.

Program under test:
```cpp
```
```

```
{program}
```

Function description for
'{func_name}':
```txt
{description}
```

You can directly use `assert` function for assertion. Your testcase should be formatted as:
```cpp
int main() {
    Solution solution;
}
```

C.4 More Cases and Data Visualization

C.4.1 Analysis of insignificant acceleration

Figure 13 illustrates situations where the speedup effect is less pronounced. Pattern matching becomes challenging when code spans multiple lines. Furthermore, when the data structure is highly complex and contains a large number of variables that need to be decoded, the model still requires a large number of steps to complete the decoding process. These observations highlight the limitations of pattern-based speedup in scenarios involving multi-line code grammatical structures or complex data representations.

C.4.2 More examples of step-by-step comparison generation

Figure 14 is another step-by-step comparison example of acceleration and non-acceleration. We clearly observe that, given the same number of decoding steps, our acceleration method produces a significantly greater number of decoded tokens compared to the unaccelerated baseline. This pronounced difference demonstrates that our method enables the generation of a more complete and diverse set of test cases within a shorter time frame, which directly contributes to higher test coverage. Therefore, this example also demonstrates that our method exhibits clear advantages in the domain of automated test case generation, providing robust technical support for improving both test coverage and software quality.

C.4.3 Generated examples of different models on different language tasks

Figure 15,16,17,18,19,20 are the generated examples of two models, `DiffuCoder` and `Dream`, on three language tasks–Python, Java and C++. The code segments highlighted in blue represent the structural patterns identified within the unit test cases. Notably, these structural patterns are consistently observed in each language, indicating their universality across diverse coding environments. This observation suggests that different dLLMs can effectively leverage these structural patterns to accelerate the decoding process. By exploiting the presence of such patterns, the models are able to generate test cases more efficiently. These more motivating examples underscore the potential for structural pattern utilization as a general mechanism for boosting decoding efficiency in unit test generation scenarios.

C.5 Algorithm

Algorithm 1: Algorithm to merge two ASTs

Input: Root nodes of two ASTs: $node_1$ and $node_2$

Output: Merged AST root node

$merged_node \leftarrow$ empty node;
if $node_1.type = node_2.type$ *and* $node_1$ *is not error node* **then**
  $merged_node.type \leftarrow node_1.type$;
  foreach $child_1$ *and* $child_2$ *from* $node_1.children$ *and* $node_2.children$ **do**
    $merged \leftarrow$ recursively merged root node of $child_1$ and $child_2$;
    if $merged$ *is not empty node* **then**
      add $merged$ to $merged_node.children$;
return $merged_node$;

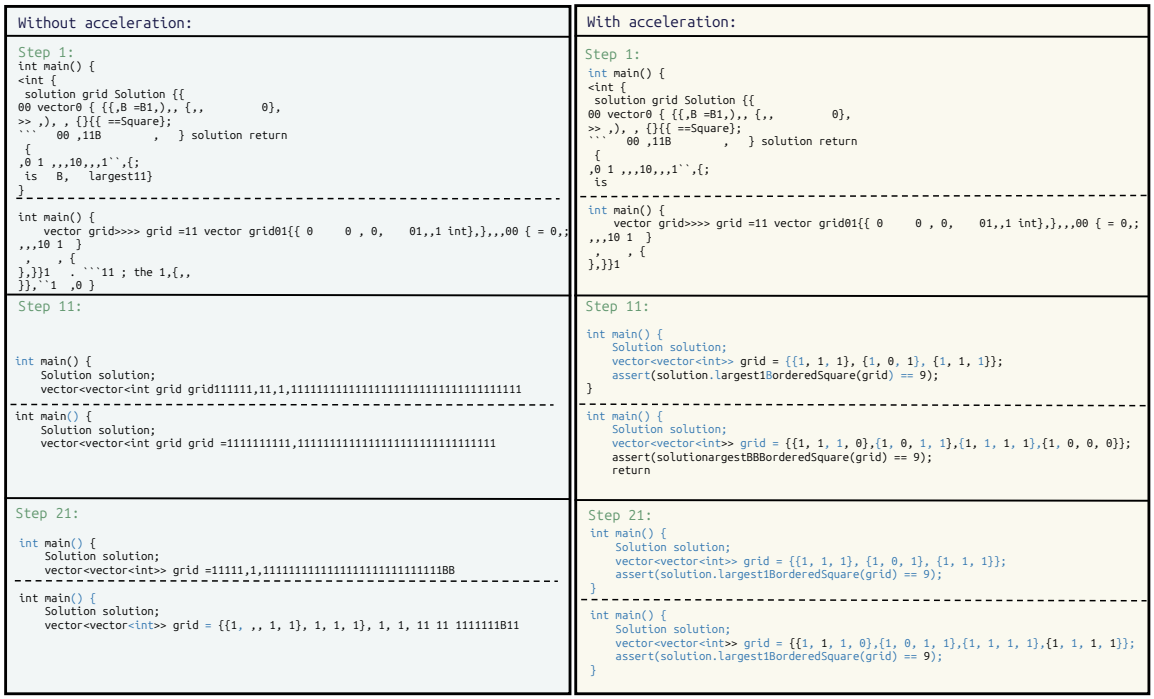

Figure 12: Case study on `Dream` illustrating the decoding process with and without DIFFUTESTER.

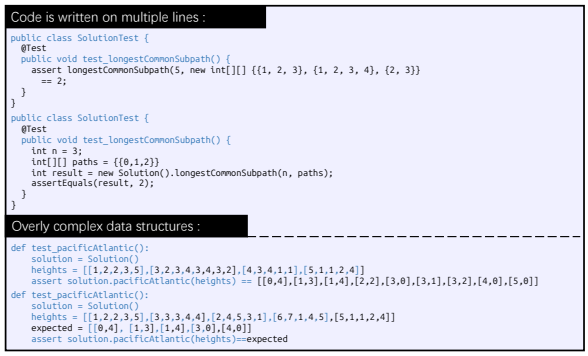

Figure 13: Examples of insignificant acceleration.

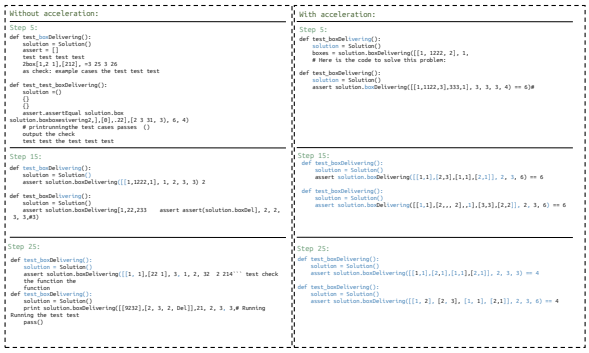

Figure 14: Another step-by-step comparison example of acceleration and non-acceleration.

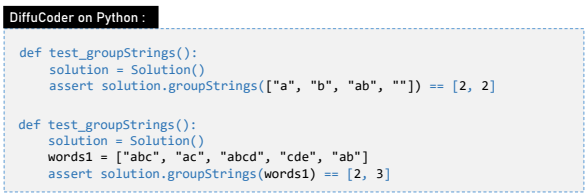

Figure 15: Example of DiffuCoder on Python.

Figure 16: Example of DiffuCoder on Java.

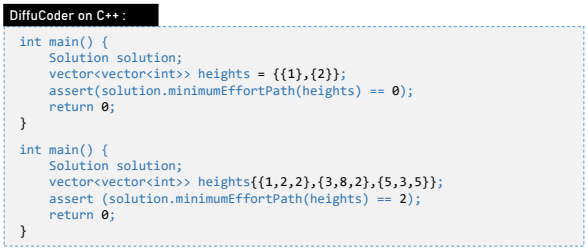

Figure 17: Example of DiffuCoder on C++.

Dream on Python :

```python
def test_longestCommonPrefix():
    solution = Solution()
    arr1 = [1234 , 123456]
    arr2 = [12345, 123456]
    assert solution.longestCommonPrefix(arr1, arr2) == 4

def test_longestCommonPrefix():
    solution = Solution()
    arr1 = [123, 456]
    arr2 = [1234, 45678]
    assert solution.longestCommonPrefix(arr1, arr2) == 3
```

Figure 18: Example of Dream on Python.

Dream on Java :

```java
public class SolutionTest {
  @Test
  public void test_maxStrength() {
    int[] nums = {1, -2, -3, 0};
    assertEquals(6, new Solution().maxStrength(nums));
  }
}

public class SolutionTest {
  @Test
  public void test_maxStrength() {
    var nums = new int[] { 3, -1, -5, 2, 4 };
    assertEquals(20, new Solution().maxStrength(nums));
  }
}
```

Figure 19: Example of Dream on Java.

Dream on C++ :

```cpp
int main() {
  Solution solution;
  vector<vector<int>> board1 = {{1, 0}, {0, 1}};
  assert(solution.movesToChessboard(board1) == 0);
}

int main() {
  Solution solution;
  vector<vector<int>> board = {{1, 0, 1, 0}, {0, 1, 0, 1}, {1, 0, 1, 0}};
  assert(solution.movesToChessboard(board) == 1);
}
```

Figure 20: Example of Dream on C++.

Algorithm 2: Unmasking extra tokens via structural pattern

Input: $codelines \leftarrow$ all lines of code

```
/* merged_list represents a
   list of merged nodes and
   their corresponding source
   code lines                 */
```

$merged_list \leftarrow$ empty list of (node, list) tuple;

for $i = 0$ **to** $len(codelines)$ **do**

  $current_node \leftarrow$ root node of AST of this code line;

  `/* The variable found indicates whether current AST has already been successfully merged with another AST, with the merge result being non-empty. */`

  $found \leftarrow False$;

  foreach $(node, lines) \in merged_list$ **do**

    $merged_node \leftarrow$ merged node of $node$ and $current_node$;

    if $merged_node$ *is not empty node* **then**

      change $node$ into $merged_node$;

      add i to $lines$;

      $found \leftarrow True$;

      break the loop;

  if $found$ *is False* **then**

    $merged_list.append((current_node, [i]))$;

foreach $(merged_ast, lines) \in merged_list$ **do**

  `/* Only those greater than 1 indicate the presence of a structural pattern. */`

  if $len(lines) > 1$ **then**

    foreach $i \in lines$ **do**

      unmask tokens in $codelines[i]$ that match the $merged_ast$ and have a confidence greater than the threshold τ;