DuET: Dual Execution for Test Output Prediction with Generated Code and Pseudocode

Hojae Han[1] **Jaejin Kim**[2,3] **Seung-won Hwang**[2,3]* **Yu Jin Kim**[4] **Moontae Lee**[4]

[1]ETRI, [2]Seoul National University, [3]SNU-LG AI Research Center, [4]LG AI Research

hojae.han@etri.re.kr {jaejin.kim,seungwonh}@snu.ac.kr

{yujin.kim,moontae.lee}@lgresearch.ai

Abstract

This work addresses test output prediction, a key challenge in test case generation. To improve the reliability of predicted outputs by LLMs, prior approaches generate code first to ground predictions. One grounding strategy is direct execution of generated code, but even minor errors can cause failures. To address this, we introduce *LLM-based pseudocode execution*, which grounds prediction on more error-resilient pseudocode and simulates execution via LLM reasoning. We further propose DuET, a dual-execution framework that combines both approaches by functional majority voting. Our analysis shows the two approaches are complementary in overcoming the limitations of direct execution suffering from code errors, and pseudocode reasoning from hallucination. On LiveCodeBench, DuET achieves the state-of-the-art performance, improving Pass@1 by 13.6 pp. For filtering candidates in code generation, DuET shows the best Pass@1 on LiveCodeBench-Easy, BigCodeBench-Hard, DevEval and HumanEval(+).[1]

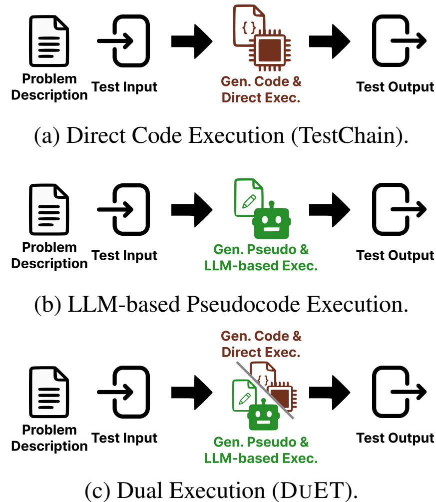


Figure 1: Comparison of test output prediction methods: (a) **TestChain** (Li and Yuan, 2024) grounds generated code and executes it directly. (b) **LLM-based Pseudocode Execution** grounds generated pseudocode and simulates its execution via an LLM. (c) **DuET** combines both code and pseudocode grounding, executing them via direct and LLM-based paths, respectively.

1 Introduction

Test cases play a critical role in code generation, serving as tools to evaluate and refine model outputs (Chen et al., 2023; Shinn et al., 2023; He et al., 2024; Han et al., 2024; Huang et al., 2024). For instance, integrating test case generation into the code generation pipeline enables automatic evaluation, allowing LLMs to generate multiple code candidates and filter them based on their predicted test outputs (Chen et al., 2023; Han et al., 2024).

Test case generation can be categorized into input generation and output prediction (Li and Yuan, 2024). In this work, we mainly focus on the output prediction, which requires precise program reasoning (Frieder et al., 2023; Liu et al., 2023b; Brown et al., 2024), considered difficult despite recent LLM advances (Barr et al., 2014; Li and Yuan, 2024).[2] Recent studies like TestChain (Li and Yuan, 2024) improve prediction performance by grounding it on generated code (Figure 1a); however, their reliability remains brittle, as models may produce the correct logic (e.g., correct pseudocode) but fail to realize it due to minor implementation errors (Zhong et al., 2020; Jiang et al., 2024; Figure 2a).

Our first contribution is to deconfound correct logic from implementation errors in generated code. We propose *LLM-based pseudocode execution* that

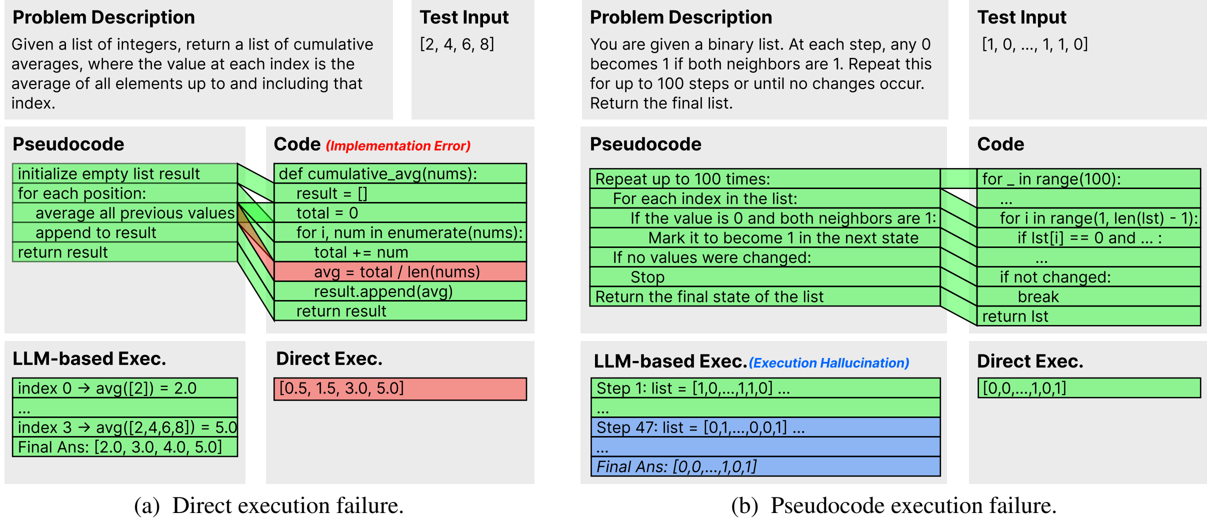
* Corresponding author.

[1]https://github.com/ldilab/DuET

[2]We note that test output prediction is orthogonal to input generation and valuable in its own right: for instance, while AlphaCode (Li et al., 2022) focuses solely on test input generation, CodeT (Chen et al., 2023) additionally predicts test outputs and achieves stronger empirical performance on end-to-end code generation.

Findings of the Association for Computational Linguistics: ACL 2026, pages 43221–43243

July 2-7, 2026 ©2026 Association for Computational Linguistics



(a) Direct execution failure. (b) Pseudocode execution failure.

Figure 2: (a) The problem involves complex functional logic that is accurately captured in the generated pseudocode but mistranslated into executable code (i.e., implementation errors). (b) Both the generated code and pseudocode are correct, but simulating the pseudocode (e.g., involving deeply nested loops) introduces reasoning challenges that lead to LLM prediction errors (i.e., execution hallucinations).

employs pseudocode in place of generated code then simulates execution by LLM reasoning (Figure 1b; §3.2.1). By expressing the algorithmic intent at a higher level of abstraction, pseudocode bypasses minor implementation errors. While this approach offsets the weaknesses of direct execution, LLM-based execution may generate hallucinated reasoning steps (an execution hallucination) that diverge from the intended logic (Kojima et al., 2022; Luo et al., 2024; Figure 2b).

Our second contribution is to propose DuET that integrates direct code execution and LLM-based pseudocode execution using functional majority voting (Li et al., 2022; Chen et al., 2023; Launer et al., 2026) (Figure 1c; §3.2.2). Figure 3 illustrates why the two approaches are effective in combination: (1) For problems that are susceptible to implementation errors (Figure 2a), pseudocode offers a more reliable grounding by capturing the algorithmic intent without being constrained by syntactic or low-level details (§6.2). (2) When execution hallucinations are more likely (Figure 2b), direct execution offers more dependable results due to its deterministic nature. It computes the output exactly as specified, without relying on LLM inference (§6.3).

We evaluate DuET on the LiveCodeBench test output prediction benchmark (Jain et al., 2024; §5.1). DuET achieves 13.6 pp gain over the previous state-of-the art and 5.6 pp gain over TestChain of using direct code execution only.

For end-to-end code generation (§5.2), integrating DuET into CodeT (Chen et al., 2023) improves the Pass@1 code generation performance of `Llama-3.1-8B-Instruct` (Grattafiori et al., 2024) by 3.2 pp. Notably, replacing it with TestChain hurts the performance by 5.6 pp. This is because direct code execution suffers from the zero-advantage problem (Yu et al., 2025; Le et al., 2026; §6.1): it can yield incorrect outputs when the generated code is flawed, reducing its effectiveness for filtering candidate programs based on predicted outputs. In contrast, LLM-based pseudocode execution is decoupled from code correctness and thus more robust in this setting.

Our key contributions are as follows:

- **First use of LLM-based pseudocode execution** for test-output prediction.
- **A novel dual-path framework (DuET)** that leverages the complementary strengths of direct and pseudocode execution paths.
- **State-of-the-art** performance in **test output prediction** on LiveCodeBench and **best** performance in **end-to-end code generation** across LiveCodeBench-Easy, BigCodeBench-Hard (Zhuo et al., 2024), and HumanEval(+) (Chen et al., 2021; Liu et al.).
- **Identification of the zero-advantage problem** in end-to-end code generation.

2 Preliminary and Related Work

In this section, we present the formal definition of test output prediction and its standard solution via direct code execution. Due to space limit, we leave an extensive survey in Appendix A.

2.1 Problem: Test Output Prediction

Test Case Generation Test case generation involves two stages: (1) input generation, which produces a valid test input based on the problem description; and (2) output prediction, which infers the corresponding output given the input and the problem.

While input generation has progressed through LLMs (Li et al., 2022) and fuzzing (Deng et al., 2023; Xia et al., 2024), output prediction remains difficult (Barr et al., 2014; Li and Yuan, 2024).

Test Output Prediction Formally, let $\mathcal{D}$, $\mathcal{I}$, and $\mathcal{O}$ denote the spaces of problem descriptions, test inputs, and outputs, respectively. We define output prediction as the function $f : \mathcal{D} \times \mathcal{I} \to \mathcal{O}$, which maps a problem description and test input to the corresponding output.

2.2 Baseline: Direct Code Execution

When correct code is available, test output prediction f reduces to a simple code execution function $e : \mathcal{C} \times \mathcal{I} \to \mathcal{O}$ that runs the code on the input and returns the result. The set $\mathcal{C}$ denotes the space of code implementations.

The goal of *direct code execution* baseline like TestChain (Li and Yuan, 2024) is to synthesize code that approximates the ground-truth implementation, such that its execution predicts test outcome.

Test output prediction f can thus be formulated via a code generator $g : \mathcal{D} \to \mathcal{C}$, followed by code execution e:

$$f(d, i) = e(g(d), i), \tag{1}$$

where $d \in \mathcal{D}$ is the problem description and $i \in \mathcal{I}$ denotes the test input.

3 Proposed Method

To improve test output prediction, we first deconfound direct execution baseline with pseudocode (§3.2.1). We then propose DuET (§3.2.2), a dual-execution strategy that combines direct and LLM-based execution via *path-weighted functional majority voting*, which accounts for intra-path agreement and adjusts voting weights accordingly.

3.1 Our Distinction: Pseudocode Grounding

Direct execution for test output prediction relies on a strong assumption that *the generated code is correct*. However, as illustrated in Figure 2a, even minor implementation errors introduced by the generator can significantly undermine prediction reliability (Zhong et al., 2020; Jiang et al., 2024).

Our design choice is to avoid relying on generated code, as prediction correctness becomes entangled with implementation quality. Instead, we ground test output prediction on pseudocode, which abstracts away low-level implementation details (see Appendix B for details).

Formally, we disentangle code generation g into two modules: $g = t \circ p$, where $p : \mathcal{D} \to \mathcal{P}$ generates high-level pseudocode and $t : \mathcal{P} \to \mathcal{C}$ translates it into executable code (Kulal et al., 2019; Huang et al., 2023; Jiang et al., 2024; Islam et al., 2024).[3]

Unlike direct execution using $g(d) = t \circ p(d)$, we bypass the translation step t and ground prediction directly on pseudocode $p(d)$. By doing so, we avoid errors introduced during translation, such as syntactic inaccuracies, execution failures, or misindexed operations, which are a major source of unreliability in g.

3.2 DuET

3.2.1 LLM-based Pseudocode Execution

Since pseudocode is not directly executable, we simulate its behavior using a language model, denoted as *LLM-based execution*. This yields the prediction function:

$$\hat{f}(d, i) = \hat{e}(d, p(d), i), \tag{2}$$

where $\hat{e}$ reasons over the pseudocode $p(d)$ and input i to produce an output.

While LLM-based pseudocode execution avoids implementation errors by not relying on executable code, it remains susceptible to *execution hallucinations*, i.e., failures that arise when the model produces plausible but incorrect intermediate reasoning steps that deviate from the intended logic (Kojima et al., 2022; Luo et al., 2024; Figure 2b). These are distinct from failures in direct code execution, where the model does not simulate execution through reasoning but instead directly runs the generated code. Such failures reflect implementation errors in the code itself.

[3] Prior work has shown that introducing this additional pseudocode step improves code generation quality compared to direct generation from descriptions (Jiang et al., 2024).

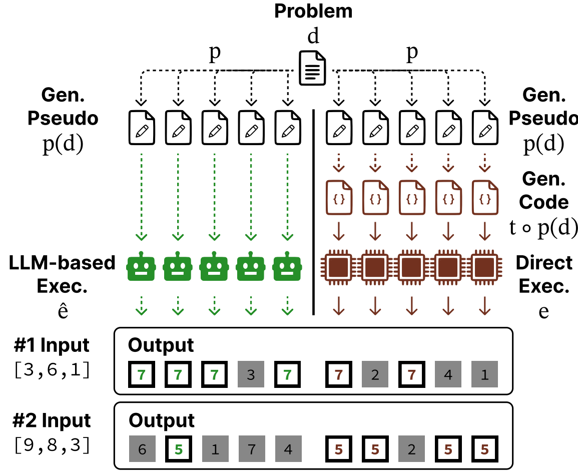


Figure 3: Overview of our dual-execution framework for test output prediction. Given a problem description and input, we perform two separate executions in parallel. In the left path, we generate pseudocode and use an LLM to reason over it to predict the output (i.e., LLM-based pseudocode execution). In the right path, we generate pseudocode independently, translate it into executable code, and run it on the input to obtain the output (i.e., direct code execution). Multiple outputs from both paths are aggregated via path-weighted functional majority voting to determine the final prediction.

3.2.2 Dual Execution

To overcome the limitations of relying on a single execution path, we propose DuET, a dual-execution framework that combines direct code execution and LLM-based pseudocode execution.

Our key contribution is to identify that these two approaches succeed under different conditions. Direct execution guarantees correctness when the generated code is correct, but suffers from brittle failure modes due to syntactic or semantic bugs (e.g., missing edge cases, off-by-one errors). Conversely, pseudocode execution is more resilient to such implementation issues by relying on high-level algorithmic intent, but can struggle when the reasoning chain is long or complex. As illustrated in Figure 2, pseudocode execution can outperform code execution when the generated code is difficult to synthesize correctly, while code execution becomes preferable when the reasoning burden becomes too high for LLMs.

As shown in Figure 3, we thus combine both execution methods and aggregate their outputs using functional majority voting (FMV; Li et al., 2022; Chen et al., 2023; Launer et al., 2026). This is well-suited for integrating heterogeneous results from both paths (see Table 5). This straightforward formulation also makes it easy to extend (see Appendix C for details). To further reflect confidence in each execution route, we propose a *path-weighted* variant of FMV that boosts the influence of outputs from routes with unanimous agreement during final aggregation.

Formally, given a problem description $d \in \mathcal{D}$ and a test input $i \in \mathcal{I}$, we collect l outputs from direct execution and m outputs from LLM-based pseudocode execution. Although we set $l{=}m$ in our experiments for simplicity, the framework allows them to differ in practice depending on resource allocation (see Appendix D for computational overhead analysis) or confidence in each execution method.

Path-Weighted FMV We extend FMV by introducing path-wise confidence weighting. Each path casts votes proportional to its matching outputs. If all valid outputs within a path unanimously agree on the same result, its votes receive a binary boost to w_{high} (otherwise w_{base}). Formally, let O and $\hat{O}$ be multisets of outputs from the direct (f) and LLM-based ($\hat{f}$) execution paths. The final output is selected as follows:

$$\begin{aligned} o &= \arg\max_{o \in O \cup \hat{O}} [w(o, O) \cdot V(o, O) \\ &\qquad + w(o, \hat{O}) \cdot V(o, \hat{O})], \\ w(o, O) &= \begin{cases} w_{high}, & \text{if } \forall o_i \in O, o_i = o, \\ w_{base}, & \text{otherwise}, \end{cases} \\ V(o, O) &= |\{i : o_i \in O, o_i = o\}|, \end{aligned} \tag{3}$$

where $w_{high} > w_{base}$.

This strategy introduces no additional cost and consistently yields slight performance gains without degradation, as shown in Appendix J.

4 Experimental Setup

4.1 Benchmark

Test Output Prediction We evaluate our method on LiveCodeBench (Jain et al., 2024), a benchmark covering four code-related tasks: test output prediction, code execution,[4] code generation, and self-repair. Our main focus is on the test output prediction task, which we evaluate under two knowledge cutoff settings to prevent data contamination: (i) **May 1, 2023–Apr 1, 2024 (442 problems)**: Larger

[4]Note that test output prediction differs from the code execution task, which is the primary focus of benchmarks like CRUXEval (Gu et al., 2024), as detailed in Appendix K.1.

Method	Pass@1
Llama-3-8B-Instruct†	27.2
Mixtral-8x22B-Instruct†	45.6
Mistral-Large†	48.6
Gemini-Pro-1.5-May†	48.6
Dracarys-Llama-3.1-70B-Instruct†	49.1
Dracarys2-Llama-3.1-70B-Instruct†	52.1
GPT-4-0613†	54.4
Claude-3-Opus†	57.8
GPT-4-Turbo-1106†	58.4
Dracarys-72B-Instruct†	58.9
Dracarys2-72B-Instruct†	59.6
GPT-4-Turbo-2024-04-09†	**67.5**
GPT-4o-2024-05-13†	73.5
GPT-4-Turbo-2024-04-09	65.8
+ *functional majority voting*	74.2
TestChain	69.4
+ *functional majority voting*	75.5
DUET	**81.1**

Table 1: Test output prediction results on LiveCodeBench (May 1, 2023–Apr 1, 2024). Red-colored models have a knowledge cutoff after this date, introducing the possibility of data contamination. Daggered (†) results are sourced from the leader board.[5][6]

and more representative, but some models may have exposure to the data; (ii) **Jan 1–Apr 1, 2024 (76 problems)**: Enables comparison across multiple models with minimal contamination risk.

Code Generation To apply our method to post-generation filtering (§5.2), we evaluate code generation on LiveCodeBench-Easy (554 problems, Jan 2024–Feb 2025) for competitive programming, and three realistic non-OJ scenarios: tool-integrated BigCodeBench-Hard (Zhuo et al., 2024) (§6.5), repo-level DevEval (Li et al., 2024) (§6.6), and HumanEval(+) (Appendix H).

4.2 Metric

We evaluate performance using Pass@k (Chen et al., 2021), adopting the *ranking-based formulation* (Chen et al., 2023) where the model selects the top-k candidates from generated samples. Regarding sample size, we set n=10 for baselines and l=m=10 in DUET for test output prediction, while setting l=m=5 per test input for end-to-end code generation.

4.3 Test Output Prediction Baselines

For all implementation details, refer to Appendix E.

[5] https://livecodebench.github.io/leaderboard.html

[6] We treat unbiased Pass@k as random selection (see Appendix K.3).

Method	Ground.	Exec.	Pass@1
GPT-4-Turbo-2024-04-09	-	-	72.4
TestChain	C	D	74.3
DUET	C & P	D & L	**81.6**
- *direct code exec.*	P	L	69.1
Llama-3.1-8B-Inst	-	-	34.0
TestChain	C	D	46.3
DUET	C & P	D & L	**53.3**
- *direct code exec.*	P	L	34.4
Llama-3.1-70B-Inst	-	-	64.5
TestChain	C	D	71.5
DUET	C & P	D & L	**78.3**
- *direct code exec.*	P	L	66.7
Mistral-Large	-	-	59.2
TestChain	C	D	55.7
DUET	C & P	D & L	**67.1**
- *direct code exec.*	P	L	46.3

Table 2: Pass@1 comparison on the LiveCodeBench test output prediction benchmark (Jan 1–Apr 1, 2024). Functional majority voting is applied to all baselines by default. Each method is labeled by its grounding type, code (C) or pseudocode (P), and execution type, direct (D) or LLM-based (L).

No Grounding The model predicts test outputs directly from the problem description and input, without grounding in any intermediate representations. This setting can be regarded as a variant of CODET (Chen et al., 2023) that uses test output prediction to rank candidate programs, isolating its output prediction component from the full pipeline.

TestChain (Li and Yuan, 2024) A grounded test output prediction baseline that predicts outputs by generating code from the input and executing it to obtain the final answer.

5 Experimental Results

5.1 Test Output Prediction

Main Leaderboard Results (Table 1) We report Pass@1 results on the full LiveCodeBench test output prediction benchmark (May 1, 2023–April 1, 2024). Among uncontaminated models, GPT-4-Turbo-2024-04-09 served as the strongest baseline. With this model as the backbone, DUET achieves a new state-of-the-art Pass@1 of 81.1, outperforming GPT-4-Turbo with FMV by 6.9 pp and TestChain with FMV by 5.6 pp. This result is consistent with models potentially suffering from data contamination (see Appendix I). These results confirm the benefit of combining both execution modes.

Expanded Comparison with Broader Model Set (Table 2) To compare methods across a wider

Method	Grounding	Execution	Pass@1	Pass@2	Pass@5	Pass@10
No Filtering	-	-	33.9	50.9	64.8	73.3
CodeT	-	-	54.2	62.8	71.3	74.3
TestChain	Code	Direct	48.6	54.9	61.4	67.6
DuET	Code & Pseudocode	Direct & LLM-based	**57.4**	**65.2**	**71.8**	**75.1**
- *direct code exec.*	Pseudocode	LLM-based	55.9	64.5	**71.8**	**75.1**

Table 3: End-to-end code generation performance of `Llama-3.1-8B-Instruct` under different test output prediction methods, evaluated on LiveCodeBench-Easy (Jan 1, 2024–Feb 1, 2025).

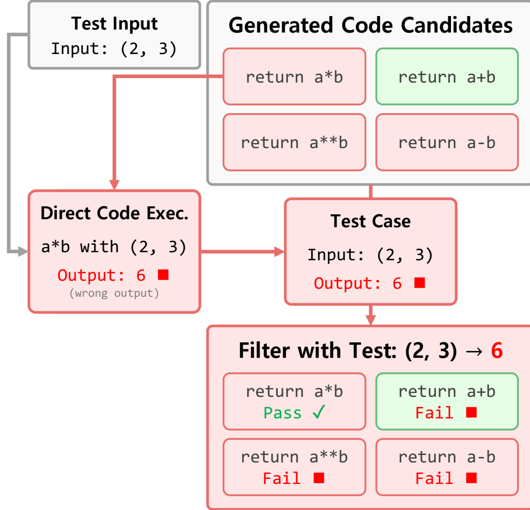


Figure 4: Illustrative example of TestChain's zero-advantage problem in end-to-end code generation, where test input (`Input: (2, 3)`) is generated by CodeT.

range of LLMs, we apply them to a contamination-free subset of LiveCodeBench (January 1–April 1, 2024). DuET achieves the highest Pass@1 among grounded methods across all LLM backbones, confirming the benefit of combining code and pseudocode execution.[7]

5.2 Impact on Code Generation

We evaluate whether improved test output prediction transfers into better code generation when applied to the code filtering stage (Chen et al., 2023; Han et al., 2024). For each problem, five test inputs are generated using CodeT (Chen et al., 2023) and 10 output predictions per input are aggregated via path-weighted FMV. Then the generated 5 test input output pairs are used to filter 20 candidate code snippets.

Table 3 shows that performance depends on how test outputs are obtained. TestChain performs 5.6 pp worse than CodeT, possibly due to the zero-advantage problem in direct code execution (§6.1; Appendix H). In contrast, LLM-based pseudocode execution is free from such problem and improves over CodeT by 1.7 pp in Pass@1 for end-to-end code generation. DuET achieves the best performance for all Pass@k metrics.

6 Analysis

6.1 Zero-Advantage Problem in TestChain

As shown in Table 3, TestChain (direct code execution only) hurts final Pass@1 in end-to-end code generation. We hypothesize that inferring test outputs from the candidate programs themselves and then reusing them to rank the same pool is structurally limited (Figure 4). Analogous to the zero-advantage problem in GRPO-style RL (Yu et al., 2025; Le et al., 2026), the induced outputs provide little signal when most candidates are correct and become unreliable when most are wrong. As a result, filtering is least informative when it is most needed. DuET mitigates this issue by introducing an orthogonal pseudocode-based execution path whose predictions are not directly tied to the candidate code being ranked, consistent with the gains in Table 3.

6.2 Impact of Generated Code Correctness

Execution accuracy is highly sensitive to implementation errors in generated code. To analyze this effect, we group problems in Figure 5 based on whether the generated code passes all test inputs (*Correct*) or fails at least one (*Wrong*).

In the *Correct* group, direct code execution naturally achieves 100, while LLM-based pseudocode execution reaches 86.7, limited by occasional execution hallucinations. DuET achieves 97.3, primarily by leveraging correct outputs from direct execution while mitigating hallucinations through path-weighted FMV.

In the *Wrong* group, direct code execution drops sharply to 39.4 due to implementation errors, while

[7] We further analyze test output prediction results by problem and input difficulty in Appendix G.

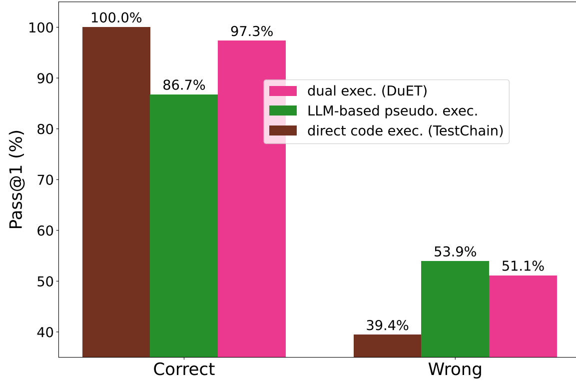


Figure 5: Pass@1 scores of execution methods on LiveCodeBench test output prediction (Jan 1–Apr 1, 2024) by `GPT-4-Turbo-2024-04-09`, across problem subsets based on the generated code correctness.

Grounding	Direct Exec.	LLM-based Exec.
Code	TestChain	LLM-based Code Exec.
Pseudocode	-	LLM-based Pseudo. Exec.

Table 4: Design space taxonomy, categorized by execution target (code vs. pseudocode) and an execution mode (direct vs. LLM-based). DuET, marked as a pink diagonal box, chooses the best combination of direct code execution and LLM-based pseudocode execution.

LLM-based pseudocode execution is more robust at 53.9. DuET achieves 51.1, generally benefiting from pseudocode over faulty code.

These results suggest that DuET benefits from pseudocode execution when generated code is faulty, and from direct execution when code is correct.

6.3 Robustness Across Execution Trace Lengths

Execution trace length, defined as the number of execution steps (Liu et al., 2023a) taken by generated code, reflects the complexity of reasoning required to compute outputs. To assess its impact on prediction accuracy, we group samples by trace length and measure Pass@1 for each group.

As shown in Figure 6, we observe a clear trend: Across **all trace lengths**, DuET of dual execution shows consistently strong performance, matching or surpassing competing methods. DuET of hybrid execution is specifically designed to mitigate errors in both paths: up to around 100 trace steps, LLM-based pseudocode execution often outperforms direct code execution by leveraging the LLM's reasoning capabilities. For longer traces, however, its performance degrades as multi-step reasoning becomes error-prone. Here, direct execution proves more reliable as it is immune to such brittleness.

We note that the decline in LLM-based execution with longer traces does not indicate a fundamental limitation of the LLM itself. Rather, it reflects a natural characteristic of multi-step reasoning: as the number of steps increases, even a single mistake can derail the entire computation, making long sequences inherently more error-prone (Wu et al., 2025; Patel et al., 2024). Nonetheless, LLM-based pseudocode execution remains effective for traces of up to ~100 steps, while the full DuET system combines both execution modes to adapt to trace complexity and maintain high accuracy.

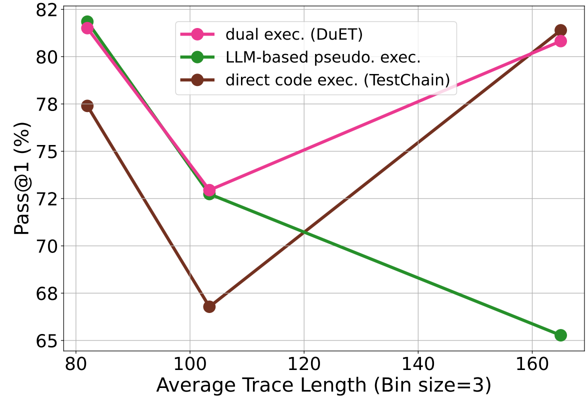


Figure 6: Pass@1 across bins of generated code execution trace lengths on LiveCodeBench test output prediction (Jan 1–Apr 1, 2024) by `GPT-4-Turbo-2024-04-09`. Each execution step corresponds to a single line of code executed during code execution.

Method	Ground.	Exec.	Pass@1
no grounding	-	-	74.2
direct code exec. (TestChain)	C	D	75.5
LLM-based code exec.	C	L	72.2
LLM-based pseudocode exec.	P	L	74.1
direct & LLM-based code exec.	C	D & L	77.4
LLM-based code & pseudo. exec.	C & P	L	73.6
***dual exec.* (DuET - *path-weighted FMV*)**	C & P	D & L	**80.8**

Table 5: Test output prediction results of `GPT-4-Turbo-2024-04-09` across grounding combinations and execution methods on LiveCodeBench (May 1, 2023–Apr 1, 2024). Functional majority voting is applied across methods. We highlight representative baselines and our methods using distinct colors: TestChain, LLM-based pseudocode execution, and DuET. Variants that use LLM-based code execution (alone or in combination) are marked in blue. Each method is labeled by its grounding type, code (C) or pseudocode (P), and execution type, direct (D) or LLM-based (L).

Method	Ground.	Exec.	Pass@1
No Filtering	-	-	24.0
CodeT	-	-	22.6
TestChain	C	D	24.0
DuET	C & P	D & L	**25.3**
- *direct code exec.*	P	L	24.0

Table 6: End-to-end code generation performance of QwQ-32B under different test output prediction methods, evaluated on BigCodeBench-Hard.

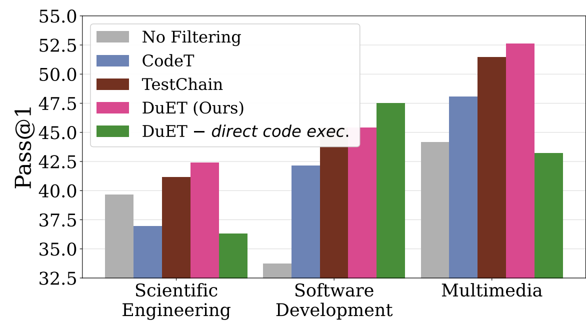


Figure 7: End-to-end code generation performance of Llama-3.1-8B-Instruct evaluated on DevEval (See Appendix F for the full results).

6.4 Design Space for Grounded Prediction

We explore the design space of grounded test output prediction (summarized in Table 4), defined by two orthogonal choices: the execution target (code vs. pseudocode) and the execution mode (direct vs. LLM-based). Table 5 reports Pass@1 results for all three feasible combinations, allowing us to compare the relative strengths of each strategy.

Code: Direct vs. LLM-based Execution Direct code execution outperforms LLM-based code execution (75.5 vs. 72.2) by faithfully reflecting actual runtime behavior, whereas the latter remains prone to reasoning failures. Consistently, direct execution achieves superior performance over LLM-based code execution when combined with LLM-based pseudocode execution (80.8 vs. 73.6).

LLM-based Execution: Pseudocode vs. Code Pseudocode offers a more reliable grounding than code for LLM-based execution (74.1 vs. 72.2), as it enables the model to focus on high-level logic. When paired with direct execution, pseudocode again yields better performance than code in LLM-based execution, helping reduce implementation errors (80.8 vs. 77.4).

These results support the combination of direct code execution and LLM-based pseudocode execution for the best performance.

6.5 On Realistic Benchmark with Reasoning-Enhanced LLM

We further conducted experiments with the reasoning model QwQ-32B (Team, 2025) on the more challenging BigCodeBench-Hard, which involves diverse function calls as tools for tasks such as data analysis and web development. In Table 6, DuET is the only approach that outperforms *No Filtering*, achieving the best performance (25.3 Pass@1). This shows that DuET generalizes to reasoning-based LLMs and remains effective even when test outputs involve arbitrary external library objects.

6.6 Repo-level Code Generation

We further validated DuET on the repository-level benchmark, DevEval (Li et al., 2024); detailed experimental setups are provided in Appendix F. As shown in Figure 7, the results align with the observations in Figure 5: LLM-based pseudocode execution outperforms direct execution (TestChain) in challenging domains where baseline generation quality (*No Filtering*) is low, whereas direct execution dominates in easier domains. By combining both paths, DuET consistently achieves top-tier performance, ranking first or second across all domains.

7 Conclusion

We proposed DuET, a dual-execution strategy that combines direct code execution and LLM-based pseudocode execution to enhance test output prediction. By integrating two execution paths into a single decision framework, DuET mitigates both implementation errors and execution hallucinations. Experiments on LiveCodeBench, BigCodeBench-Hard, and DevEval show that DuET sets a new state-of-the-art in test output prediction and enhances code generation via candidate filtering.

Acknowledgment

This work was supported by the Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government(MSIT) (No.2022-0-00995, Automated reliable source code generation from natural language descriptions).

Limitations

Our analysis indicates that the effectiveness of direct vs. LLM-based execution varies with trace

length, suggesting potential for adaptive strategies that predict the more reliable method per instance and adjust path weights in functional majority voting accordingly. Our test input generation is intentionally simple; incorporating more diverse or diagnostic inputs such as edge cases may further benefit downstream applications like code generation. The evaluation of newer models was limited by the current LiveCodeBench setup, which has not yet been updated to support recent versions for the test output prediction task.

References

Earl T 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(5):507–525.

Bradley Brown, Jordan Juravsky, Ryan Ehrlich, Ronald Clark, Quoc V. Le, Christopher Ré, and Azalia Mirhoseini. 2024. Large language monkeys: Scaling inference compute with repeated sampling.

Harrison Chase. 2022. Langchain: Building applications with llms through composability. https://github.com/langchain-ai/langchain. Accessed: 2025-05-18.

Harrison Chase. 2023. Langgraph: A graph-based framework for building agentic workflows. https://github.com/langchain-ai/langgraph. Accessed: 2025-05-18.

Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. 2023. Codet: Code generation with generated tests. In The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023. OpenReview.net.

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

Yinlin Deng, Chunqiu Steven Xia, Haoran Peng, Chenyuan Yang, and Lingming Zhang. 2023. Large language models are zero-shot fuzzers: Fuzzing deep-learning libraries via large language models. In Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, pages 423–435.

Simon Frieder, Luca Pinchetti, , Ryan-Rhys Griffiths, Tommaso Salvatori, Thomas Lukasiewicz, Philipp Petersen, and Julius Berner. 2023. Mathematical capabilities of chatgpt. In Advances in Neural Information Processing Systems, volume 36, pages 27699–27744. Curran Associates, Inc.

Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, et al. 2024. The llama 3 herd of models. arXiv preprint arXiv:2407.21783.

Alex Gu, Baptiste Roziere, Hugh James Leather, Armando Solar-Lezama, Gabriel Synnaeve, and Sida Wang. 2024. CRUXEval: A benchmark for code reasoning, understanding and execution. In Proceedings of the 41st International Conference on Machine Learning, volume 235 of Proceedings of Machine Learning Research, pages 16568–16621. PMLR.

Hojae Han, Jaejin Kim, Jaeseok Yoo, Youngwon Lee, and Seung-won Hwang. 2024. ArchCode: Incorporating software requirements in code generation with large language models. In Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pages 13520–13552, Bangkok, Thailand. Association for Computational Linguistics.

Xinyi He, Jiaru Zou, Yun Lin, Mengyu Zhou, Shi Han, Zejian Yuan, and Dongmei Zhang. 2024. CoCoST: Automatic complex code generation with online searching and correctness testing. In Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing, pages 19433–19451, Miami, Florida, USA. Association for Computational Linguistics.

Baizhou Huang, Shuai Lu, Xiaojun Wan, and Nan Duan. 2024. Enhancing large language models in coding through multi-perspective self-consistency. In Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pages 1429–1450, Bangkok, Thailand. Association for Computational Linguistics.

Dong Huang, Qingwen Bu, Yuhao Qing, and Heming Cui. 2023. Codecot: Tackling code syntax errors in cot reasoning for code generation. arXiv preprint arXiv:2308.08784.

Jeevana Priya Inala, Chenglong Wang, Mei Yang, Andres Codas, Mark Encarnación, Shuvendu K Lahiri, Madanlal Musuvathi, and Jianfeng Gao. 2022. Fault-aware neural code rankers. In Advances in Neural Information Processing Systems.

Md Ashraful Islam, Mohammed Eunus Ali, and Md Rizwan Parvez. 2024. Mapcoder: Multi-agent code generation for competitive problem solving. arXiv preprint arXiv:2405.11403.

Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. 2024. Livecodebench: Holistic and contamination free evaluation of large language models for code.

Xue Jiang, Yihong Dong, Lecheng Wang, Zheng Fang, Qiwei Shang, Ge Li, Zhi Jin, and Wenpin Jiao. 2024. Self-planning code generation with large language models. ACM Transactions on Software Engineering and Methodology, 33(7):1–30.

Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. 2022. Large language models are zero-shot reasoners. Advances in neural information processing systems, 35:22199–22213.

Sumith Kulal, Panupong Pasupat, Kartik Chandra, Mina Lee, Oded Padon, Alex Aiken, and Percy S Liang. 2019. Spoc: Search-based pseudocode to code. Advances in Neural Information Processing Systems, 32.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient memory management for large language model serving with pagedattention. In Proceedings of the 29th Symposium on Operating Systems Principles, pages 611–626.

Shuvendu K Lahiri, Sarah Fakhoury, Aaditya Naik, Georgios Sakkas, Saikat Chakraborty, Madanlal Musuvathi, Piali Choudhury, Curtis von Veh, Jeevana Priya Inala, Chenglong Wang, et al. 2022. Interactive code generation via test-driven user-intent formalization. arXiv preprint arXiv:2208.05950.

Tim Launer, Jonas Hübotter, Marco Bagatella, Ido Hakimi, and Andreas Krause. 2026. Majority voting for code generation. In Third Workshop on Test-Time Updates (Main Track).

Hung Le, Hailin Chen, Amrita Saha, Akash Gokul, Doyen Sahoo, and Shafiq Joty. 2023. Codechain: Towards modular code generation through chain of self-revisions with representative sub-modules. arXiv preprint arXiv:2310.08992.

Thanh-Long V. Le, Myeongho Jeon, Kim Vu, Viet Dac Lai, and Eunho Yang. 2026. No prompt left behind: Exploiting zero-variance prompts in LLM reinforcement learning via entropy-guided advantage shaping. In The Fourteenth International Conference on Learning Representations.

Jia Li, Ge Li, Yongmin Li, and Zhi Jin. 2025a. Structured chain-of-thought prompting for code generation. ACM Transactions on Software Engineering and Methodology, 34(2):1–23.

Jia Li, Ge Li, Yunfei Zhao, Yongmin Li, Huanyu Liu, Hao Zhu, Lecheng Wang, Kaibo Liu, Zheng Fang, Lanshen Wang, Jiazheng Ding, Xuanming Zhang, Yuqi Zhu, Yihong Dong, Zhi Jin, Binhua Li, Fei Huang, Yongbin Li, Bin Gu, and Mengfei Yang. 2024. DevEval: A manually-annotated code generation benchmark aligned with real-world code repositories. In Findings of the Association for Computational Linguistics: ACL 2024, pages 3603–3614, Bangkok, Thailand. Association for Computational Linguistics.

Kefan Li and Yuan Yuan. 2024. Large language models as test case generators: Performance evaluation and enhancement.

Wen-Ding Li, Darren Yan Key, and Kevin Ellis. 2025b. Toward trustworthy neural program synthesis. In ICLR 2025 Third Workshop on Deep Learning for Code.

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

Chenxiao Liu, Shuai Lu, Weizhu Chen, Daxin Jiang, Alexey Svyatkovskiy, Shengyu Fu, Neel Sundaresan, and Nan Duan. 2023a. Code execution with pre-trained language models. In Findings of the Association for Computational Linguistics: ACL 2023, pages 4984–4999, Toronto, Canada. Association for Computational Linguistics.

Hanmeng Liu, Ruoxi Ning, Zhiyang Teng, Jian Liu, Qiji Zhou, and Yue Zhang. 2023b. Evaluating the logical reasoning ability of chatgpt and gpt-4.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. Is Your Code Generated by ChatGPT Really Correct?

Liangchen Luo, Yinxiao Liu, Rosanne Liu, Samrat Phatale, Meiqi Guo, Harsh Lara, Yunxuan Li, Lei Shu, Yun Zhu, Lei Meng, et al. 2024. Improve mathematical reasoning in language models by automated process supervision. arXiv preprint arXiv:2406.06592.

Ollama Inc. 2025. Ollama documentation: Getting started with local LLM inference.

OpenAI. 2023. Openai API reference.

Nisarg Patel, Mohith Kulkarni, Mihir Parmar, Aashna Budhiraja, Mutsumi Nakamura, Neeraj Varshney, and Chitta Baral. 2024. Multi-logieval: Towards evaluating multi-step logical reasoning ability of large language models. arXiv preprint arXiv:2406.17169.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. Reflexion: Language agents with verbal reinforcement learning. In Advances in Neural Information Processing Systems.

Snowflake Inc. 2024. Snowflake cortex LLM functions documentation.

Qwen Team. 2025. Qwq-32b: Embracing the power of reinforcement learning.

Yuyang Wu, Yifei Wang, Tianqi Du, Stefanie Jegelka, and Yisen Wang. 2025. When more is less: Understanding chain-of-thought length in llms. arXiv preprint arXiv:2502.07266.

Chunqiu Steven Xia, Matteo Paltenghi, Jia Le Tian, Michael Pradel, and Lingming Zhang. 2024. Fuzz4all: Universal fuzzing with large language models. In Proceedings of the IEEE/ACM 46th International Conference on Software Engineering, pages 1–13.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R. Narasimhan, and Yuan Cao. 2023. React: Synergizing reasoning and acting in language models. In The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023. OpenReview.net.

Qiying Yu, Zheng Zhang, Ruofei Zhu, Yufeng Yuan, Xiaochen Zuo, YuYue, Weinan Dai, Tiantian Fan, Gaohong Liu, Juncai Liu, LingJun Liu, Xin Liu, Haibin Lin, Zhiqi Lin, Bole Ma, Guangming Sheng, Yuxuan Tong, Chi Zhang, Mofan Zhang, Ru Zhang, Wang Zhang, Hang Zhu, Jinhua Zhu, Jiaze Chen, Jiangjie Chen, Chengyi Wang, Hongli Yu, Yuxuan Song, Xiangpeng Wei, Hao Zhou, Jingjing Liu, Wei-Ying Ma, Ya-Qin Zhang, Lin Yan, Yonghui Wu, and Mingxuan Wang. 2025. DAPO: An open-source LLM reinforcement learning system at scale. In The Thirty-ninth Annual Conference on Neural Information Processing Systems.

Ruiqi Zhong, Mitchell Stern, and Dan Klein. 2020. Semantic scaffolds for pseudocode-to-code generation. In Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, pages 2283–2295, Online. Association for Computational Linguistics.

Terry Yue Zhuo, Minh Chien Vu, Jenny Chim, Han Hu, Wenhao Yu, Ratnadira Widyasari, Imam Nur Bani Yusuf, Haolan Zhan, Junda He, Indraneil Paul, et al. 2024. Bigcodebench: Benchmarking code generation with diverse function calls and complex instructions. arXiv preprint arXiv:2406.15877.

A Related Work

Grounded Test Output Prediction Test case generation provides input-output pairs to verify code correctness (Chen et al., 2023; Huang et al., 2024; Han et al., 2024), while also enabling feedback-driven refinement (Shinn et al., 2023; He et al., 2024). Recent methods (Jain et al., 2024; Li and Yuan, 2024) emphasize test output prediction as a distinct subtask, where the goal is to infer the correct output given a problem and test input, often requiring nontrivial program reasoning. Whereas CODET (Chen et al., 2023) couples input and output generation, TestChain (Li and Yuan, 2024) explicitly decouples them and performs grounded prediction by executing code generated from the problem description, highlighting the need for reliable grounding to support output prediction.

Intermediate Representations for Code Reasoning A parallel line of work explores using **intermediate representations**, such as pseudocode or high-level plans, to bridge natural language and executable code. Plan-first methods like Huang et al. (2023), Jiang et al. (2024), and Islam et al. (2024) generate action sketches before code, improving alignment with the problem intent. SCoT (Li et al., 2025a) introduces structured intermediate forms, and CodeChain (Le et al., 2023) further modularizes the process into reusable components. By abstracting away low-level syntax, these works suggest that intermediate representations allow models to focus on semantic intent, facilitating more accurate and flexible reasoning.

Our Distinction While DUET tackles test output prediction by grounding as TestChain does, it also overcomes the brittleness of direct execution by leveraging pseudocode as an intermediate representation. By combining both paths, DUET pairs the symbolic precision of code (§6.3) with the abstraction of pseudocode (§6.2), covering failure modes that neither path handles well alone.

B Analysis on Grounding Design and Reasoning Steps

Beyond pseudocode design, we also considered alternative intermediate representations such as abstract syntax trees (ASTs), control flow graphs (CFGs), and type-based analyses. ASTs encode fine-grained syntactic details that obscure semantic intent, while type-based analyses offer limited structural context. CFGs capture execution flow but impose a nontrivial burden on LLMs to linearize and interpret graph structures. In contrast, pseudocode expressed in natural language offers a more accessible abstraction for reasoning-oriented models, while retaining sufficient alignment with program semantics.

Pseudocode Granularity	LLM-based Execution	Pass@1
Low	Step-by-step (default)	31.1
Low	3-step	31.3
Mid (default)	Step-by-step (default)	34.4
Mid (default)	3-step	32.5
High	Step-by-step (default)	25.1
High	3-step	35.0

Table 7: Test output prediction performance comparison of `Llama-3.1-8B-Instruct` with LLM-based pseudocode execution over prompt design choices on LiveCodeBench (May 1, 2023–Apr 1, 2024). Rows vary the granularity of the generated pseudocode (low, mid, high), while columns compare two LLM-based execution prompting strategies: a step-by-step simulation and a condensed 3-step reasoning format. Default settings are indicated in parentheses. Functional majority voting is applied with 10 independent samples per output.

As shown in Table 7, this design choice is empirically supported: moderately abstract pseudocode (Mid- or High-level) consistently outperforms overly concrete (Low-level) forms, suggesting that abstraction helps LLMs focus on semantic reasoning rather than token-level execution details. Regarding reasoning steps, step-by-step and 3-step executions exhibit no consistent superiority across tasks, implying that the structure of reasoning matters less than the grounding it operates on. Our default configuration (Mid-level pseudocode with step-by-step reasoning) achieves strong overall performance, while the best-performing setting (High-level + 3-step) provides only a marginal gain of 0.6 pp. These observations suggest that grounding design plays a key role in stabilizing execution-based reasoning, while exploring richer graph- or type-aware forms remains an open direction.

C On the Simplicity of the Method

Prior execution-based approaches such as TestChain rely solely on direct code execution, making them sensitive to code generation quality and prone to failure in end-to-end scenarios (§6.1). DUET addresses this by adding an LLM-based pseudocode path, requiring no architectural changes or training overhead. Its simplicity is

a practical advantage: the framework is easy to implement and extend with additional paths or weighting schemes.

D Computational Overhead Analysis

Under the functional majority voting with n test outputs per input, the three execution paths differ mainly in the number of LLM calls and code executions. First, direct code execution initially generates n pseudocode candidates and their corresponding programs then executes each once, requiring $2n$ LLM calls and n code executions. Second, LLM-based pseudocode execution generates n pseudocode candidates and evaluates them via the LLM, leading to $2n$ LLM calls without external execution. Last, DuET performs both executions for the same n candidates,[8] totaling $4n$ LLM calls and n code executions.

Despite higher nominal cost, all methods can batch their operations: direct and LLM-based paths handle n items per forward pass, while DuET processes $2n$. Hence, with full parallelization, the effective runtime reduces to two forward passes for all methods.

E Implementation Details

Temperature For LLM inference, we use two different decoding settings: When using greedy decoding, we set `temperature=0.0` and `top_p=1.0`. For nucleus sampling, we set `temperature=0.8` and `top_p=0.95`. Specifically, pseudocode generation is performed using the nucleus sampling, while code generation and LLM-based Execution uses greedy decoding. For **No Grounding**, outputs are directly predicted by nucleus sampling.

Path-Weighted FMV To reflect path-level confidence, we assign a higher weight $w_{high}=2$ to predictions when all the outputs of a given path agree unanimously; otherwise, we use the base weight $w_{base}=1$.

Fallback As aforementioned, LLM-based execution uses nucleus sampling with high temperature to encourage diversity. To avoid committing to uncertain predictions, we fall back to a default ungrounded prediction when the LLM-based execution outputs from this path are not unanimous.

[8] See Appendix K.2 for a discussion on common mode failure mitigation.

Method-Specific Hyperparameters

- **No Grounding**: Directly predicts $n=10$ outputs from the problem and input.
- **TestChain** (Li and Yuan, 2024): Generates $l=10$ outputs via direct code execution in Section 3.2.2.
- **LLM-based Pseudocode Execution**: Generates $m=10$ outputs via LLM-based pseudocode execution in Section 3.2.2.
- **DuET**: Combines both strategies, generating $l=10$ outputs via direct execution and $m=10$ via LLM-based execution.

For end-to-end code generation, we used $l=m=5$ in DuET while keeping the baseline settings unchanged.

Implementation of TestChain Given the absence of a public implementation for TestChain, we reproduced the method with a specific design choice to ensure a competitive baseline. The original TestChain relies on a ReAct (Yao et al., 2023)-style interaction loop for incremental code construction, where new code segments are appended without modifying previously generated parts. However, prior studies (Jiang et al., 2024) indicate that such multi-turn incremental strategies often underperform compared to single-pass generation. Consequently, we adopted a direct-execution formulation that synthesizes the complete solution in a single pass, avoiding the potential limitations of the incremental approach and providing a more robust standard for comparison.

LLM Inference Backends We utilize multiple inference backends depending on availability and deployment settings: (1) vLLM (Kwon et al., 2023): An optimized inference engine supporting fast batched decoding; (2) Ollama (Ollama Inc., 2025): A lightweight on-device LLM inference runtime; (3) Snowflake Cortex (Snowflake Inc., 2024): A commercial LLM inference platform; (4) OpenAI API (OpenAI, 2023): Used for models such as `GPT-4-Turbo-2024-04-09`. All models are used without quantization, employing full-precision FP16 weights when possible.

We run local inference with vLLM or Ollama on eight NVIDIA RTX 3090 GPUs. Under this setup, `Llama-3.1-8B-Instruct` processes the full LiveCodeBench benchmark ($\sim$ 22K tokens/min) in

Model	Grounding	Execution	Pass@1			
			All	Easy	Medium	Hard
`Llama-3.1-8B-Instruct`	-	-	34.0	35.3	40.9	23.7
TestChain	Code	Direct	46.3	55.2	51.5	23.7
DuET	Code & Pseudocode	Direct & LLM-based	**53.3**	**61.4**	**62.5**	27.6
- *direct code exec.*	Pseudocode	LLM-based	34.4	35.1	27.3	**41.2**

Table 8: Test output prediction results on LiveCodeBench (Jan 1–Apr 1, 2024), where difficulty levels of test inputs are estimated using the correctness rate of code generated by the best performing model (`GPT-4-Turbo-2024-04-09`). Functional majority voting is applied to all baselines by default.

Model	Grounding	Execution	Pass@1			
			All	Easy	Medium	Hard
`Llama-3.1-8B-Instruct`	-	-	34.0	39.0	35.3	18.2
TestChain	Code	Direct	46.3	58.0	39.6	**43.9**
DuET	Code & Pseudocode	Direct & LLM-based	**53.3**	**64.0**	**51.3**	36.4
- *direct code exec.*	Pseudocode	LLM-based	34.4	46.4	28.8	27.3

Table 9: Pass@1 performance on the LiveCodeBench test output prediction benchmark (Jan 1–Apr 1, 2024), where difficulty levels follow the original LiveCodeBench setting based on the problem's difficulty. Functional majority voting is applied to all baselines by default.

about one hour, while `Llama-3.1-70B-Instruct` achieves 4K tokens/min, taking roughly six hours.

Prompt Orchestration We implement the system pipeline using the LangChain framework (Chase, 2022), which provides modular abstractions for prompt orchestration and LLM interaction. For managing multi-step workflows and maintaining intermediate states across executions, we adopt LangGraph (Chase, 2023), a directed-graph-based orchestration engine built on top of LangChain. This allows flexible integration of both code-based and pseudocode-based reasoning within a unified execution graph.

F Repository-level Code Generation

For the DevEval (Li et al., 2024) evaluation, we utilized Llama-3.1-8B-Instruct to generate 20 code candidates and test cases per problem. To strictly assess the discriminative power of code filtering, we excluded instances where all candidates were either completely correct or incorrect, as these scenarios do not differentiate filtering performance. Consequently, the evaluation was conducted on a subset of 367 problems selected from the original 1,825 problems across 10 domains. Other configurations follow the settings described in Section 5.2.

G Difficulty Categorization in Test Output Prediction

In terms of difficulty categorization, test output prediction is better characterized by the test input, since even problems with difficult reference solutions may present inputs that are easy to predict. Thus, we redefine difficulty for this task based on the test input rather than the problem: specifically, we generate 10 code samples per test input using `GPT-4-Turbo-2024-04-09` (the strongest baseline) and compute the proportion of samples that produce the correct output.

As shown in Table 8, direct code execution (TestChain) outperforms LLM-based pseudocode execution (DuET- *direct code exec.*) on the Easy and Medium subsets, likely due to its precision on straightforward inputs. Conversely, LLM-based pseudocode execution excels on the Hard subset, where more challenging inputs benefit from additional reasoning. The dual execution approach (DuET) achieves the best overall performance: it ranks highest on the Easy and Medium subsets, and second-best on the Hard subset.

These results confirm that the two paths cover each other's weaknesses. For Easy and Medium inputs, the direct-execution path remains effective because even imperfectly generated code often yields correct outputs for simpler cases. In contrast, Hard inputs require strict logical correctness, making the pseudocode-based simulation path more critical.

Method	Grounding	Execution	Pass@1	
			HumanEval	HumanEval+
No Filtering	-	-	79.2	69.9
CodeT	-	-	84.7	75.0
TestChain	Code	Direct	83.0	73.6
DuET	Code & Pseudocode	Direct & LLM-based	**85.1**	**75.4**
- *direct code exec.*	Pseudocode	LLM-based	84.8	75.2

Table 10: End-to-end code generation performance of `Llama-3.1-70B-Instruct` under different test output prediction methods, evaluated on HumanEval and HumanEval+.

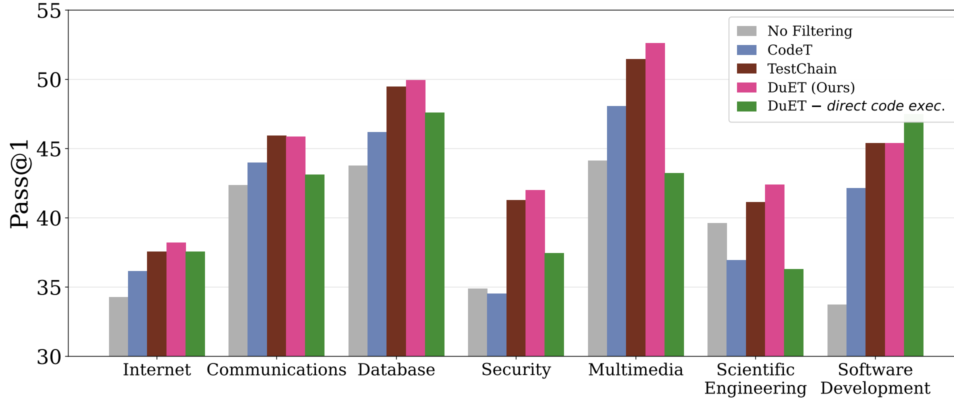


Figure 8: End-to-end code generation performance of `Llama-3.1-8B-Instruct` evaluated on DevEval.

DuET ranks 1st on the All, Easy, and Medium splits, and 2nd on Hard, while consistently outperforming the direct-execution baseline.

We also report the results under the original problem-based difficulty split in Table 9 for comparison. Interestingly, under this split, TestChain achieves the highest performance on the Hard subset, contrary to its underperformance on the Hard subset defined by input difficulty (Table 8). This discrepancy highlights the limitations of using problem-level metadata, such as reference code complexity, as a proxy for prediction difficulty. In contrast, input-based categorization more directly reflects the reasoning challenges encountered during test output prediction, better aligning with the actual performance characteristics of execution-based methods.

Meanwhile, Table 6 shows that with a stronger reasoning model QwQ-32B (Team, 2025), both code and pseudocode executions become more reliable, enabling DuET to realize its advantage even on inputs categorized as Hard. This suggests that input-based difficulty interacts with model capability: as reasoning strength increases, brittle failure modes in direct execution diminish, narrowing method gaps and amplifying the gains of dual execution.

H End2End Code Generation on HumanEval and HumanEval+

Table 10 reports end-to-end code generation results on HumanEval (Chen et al., 2021) and HumanEval+ (Liu et al.). DuET achieves the highest Pass@1 on both benchmarks, followed closely by LLM-based pseudocode execution (DuET – *direct code exec.*). These are the only two methods that outperform the base code filtering approach (CodeT), while TestChain again underperforms relative to CodeT, consistent with observations on Table 3. These results confirm that DuET improves performance not only on challenging tasks like LiveCodeBench but also on easier benchmarks such as HumanEval(+), where overall model accuracy is already high. The relatively small performance gains are likely due to the saturated nature of these benchmarks, where most candidate solutions are already correct, making the benefits of better test output prediction marginal.

I Test Output Prediction Results with Contamination Risk Models

Table 11 presents the performance of models with potential data contamination, excluded from the main text. The knowledge cutoff dates for all mod-

Method	Ground.	Exec.	Pass@1
GPT-4-Turbo-2024-04-09	-	-	72.4
TestChain	C	D	74.3
DuET	C & P	D & L	**81.6**
- direct code exec.	P	L	69.1
Llama-3.1-8B-Inst	-	-	34.0
TestChain	C	D	46.3
DuET	C & P	D & L	**53.3**
- direct code exec.	P	L	34.4
Llama-3.1-70B-Inst	-	-	64.5
TestChain	C	D	71.5
DuET	C & P	D & L	**78.3**
- direct code exec.	P	L	66.7
Mistral-Large	-	-	59.2
TestChain	C	D	55.7
DuET	C & P	D & L	**67.1**
- direct code exec.	P	L	46.3
EXAONE-3.5-7.8B-Inst	-	-	47.1
TestChain	C	D	45.7
DuET	C & P	D & L	**59.6**
- direct code exec.	P	L	45.6
EXAONE-3.5-32B-Inst	-	-	51.0
TestChain	C	D	59.9
DuET	C & P	D & L	**65.1**
- direct code exec.	P	L	51.1

Table 11: Pass@1 comparison on the LiveCodeBench test output prediction benchmark (Jan 1–Apr 1, 2024). Functional majority voting is applied to all baselines by default. Each method is labeled by its grounding type, code (C) or pseudocode (P), and execution type, direct (D) or LLM-based (L). Red-colored models have the possibility of data contamination.

Model	DuET Voting Method	Pass@1
GPT-4-Turbo-2024-04-09	*functional majority voting*	81.6
	+ path-weighted	81.6
Llama-3.1-8B-Inst	*functional majority voting*	53.3
	+ path-weighted	53.3
Llama-3.1-70B-Inst	*functional majority voting*	77.6
	+ path-weighted	**78.3**
Mistral-Large	*functional majority voting*	67.1
	+ path-weighted	67.1
EXAONE-3.5-7.8B-Inst	*functional majority voting*	59.6
	+ path-weighted	59.6
EXAONE-3.5-32B-Inst	*functional majority voting*	64.5
	+ path-weighted	**65.1**

Table 12: The effect of path-weighted FMV on the LiveCodeBench test output prediction benchmark (Jan 1–Apr 1, 2024). Red-colored models have the possibility of data contamination.

els, including these, are detailed in Table 13. Overall, the results show performance trends consistent with Table 2 in the main context.

J Path-Weighted FMV: Safe Gains at No Cost

Table 12 shows that path-weighted FMV consistently matches or slightly outperforms the standard FMV across all models on LiveCodeBench. While the gains are modest (up to 0.7 pp Pass@1), the method requires zero additional cost and ensures non-decreasing accuracy. We leave more advanced weighting mechanisms for future work (see Appendix L).

Model Name	Approximate Cutoff Date
GPT-4-Turbo-2024-04-09	2023-04-30
Llama-3.1-8B-Instruct	2023-12-31
Llama-3.1-70B-Instruct	2023-12-31
Mistral-Large	2023-01-01
EXAONE-3.5-7.8B-Instruct	2024-11-30
EXAONE-3.5-32B-Instruct	2024-11-30

Table 13: Cutoff dates for each LLM.

K Discussion

K.1 Difference between Test Output Prediction and Code Execution

While both tasks involve predicting the result of a program, there is a fundamental distinction regarding the information provided to the model. Code execution tasks, exemplified by benchmarks such as CRUXEval (Gu et al., 2024), provide the ground-truth code and evaluate a model's ability to simulate the execution process (i.e., predicting outputs given specific code and inputs). In contrast, the test output prediction task in LiveCodeBench operates without access to ground-truth code. It requires the model to derive the intended program behavior solely from a natural language problem description. Consequently, this task evaluates the model's capability to reason about requirements and synthesize logic from scratch, mimicking the real-world software development process, rather than the narrower capability of mentally tracing provided code.

K.2 Common Mode Failure Mitigation

In our dual-path framework, common mode failure refers to the risk where a single upstream error in the LLM-generated pseudocode propagates to both execution paths, causing simultaneous failure. DuET mitigates this by diversifying the pseudocode space, generating multiple candidates rather than relying on a single interpretation. Because the two paths have different error profiles, a flawed pseudocode is unlikely to cause both paths to fail in the same way, preventing upstream errors from dictating the final prediction.

K.3 Metric Comparison Justification

In Table 1, we compare our ranking-based results against the official leaderboard's unbiased Pass@k. Since unbiased Pass@k represents the expected performance of a random selection baseline (i.e., uniform random sampling), this comparison quantifies the effectiveness of our *intelligent selection* over random selection, consistent with prior work on neural code rankers (Inala et al., 2022; Li et al., 2025b; Lahiri et al., 2022).

L Future Directions

A promising future extension of DuET is to develop an adaptive weighting mechanism that dynamically adjusts the contribution of each execution path based on its estimated reliability or difficulty. In particular, a lightweight model could be trained to predict which path (code or pseudocode) is more trustworthy for a given instance, using signals such as the log-likelihood of generated code, execution-trace length, or recent error patterns. Such a learned weighting scheme would generalize our current unanimous rule, allowing DuET to assign path weights more selectively based on task difficulty and reasoning complexity.

M Case Study (generated by `GPT-4-Turbo-2024-04-09`)

M.1 Example of Execution Hallucination (Figures 9-12)

In Figure 11, lines `4-6` of the LLM-based execution output reflect a hallucinated assumption about the input string "`2245047`": it incorrectly identifies two zeros at indices 5 and 6, while in fact, only a single '`0`' exists at index 5. As a result, forming a suffix ending in "`00`" is impossible. This leads to an invalid computation of the operation cost and ultimately an incorrect prediction that no deletions are required. By contrast, when directly executing the correctly generated code shown in Figure 12, the reasoning proceeds without such hallucinations and yields the correct output.

M.2 Example of Implementation Error (Figures 13-16)

Figure 16 illustrates a translation discrepancy. Line 6 of the pseudocode (Figure 14) implicitly requires sorting the indices of each connected component before reinsertion, but the implementation in lines 39–41 fails to do so. In the code, the list of indices is taken directly from the BFS discovery order (`indices = component`) and paired with sorted values using zip. However, since the BFS order is arbitrary, smaller values may be placed at larger indices, and the resulting array is not guaranteed to be lexicographically minimal. To align with the pseudocode's intent, the index list must be explicitly sorted (e.g., replacing line 39 with `indices = sorted(component)`) to ensure the smallest values occupy the leftmost positions and preserve lexicographic order. In contrast, as shown in Figure 15, LLM-based pseudocode execution correctly infers this ordering requirement and predicts the correct output while bypassing low-level implementation details.

N Prompt Templates

We adopt task-specific prompt templates for each stage of our pipeline. In Figures 17-22, we list the prompt formats used for each. All templates are written in Jinja2 format.

```
You are given a 0-indexed string num representing a non-negative integer.
In one operation, you can pick any digit of num and delete it. Note that if you delete all the digits of num, num becomes 0.
Return the minimum number of operations required to make num special.
An integer x is considered special if it is divisible by 25.
```

Figure 9: Problem description.

```
Algorithm minimumOperations(num):
    1. Initialize a list possible_ends with values ["00", "25", "50", "75"] which represent the numbers divisible by 25.
    2. Set min_operations to a large value (e.g., length of num + 1) to keep track of the minimum operations needed.
    3. Iterate over each possible_end in possible_ends:
        a. Initialize two pointers, i and j, to the end of the string num.
        b. Set found_digits to 0, representing how many digits of possible_end have been matched.
        c. While iterating from the end of num towards the beginning:
            - If found_digits is 0 and num[i] matches the second character of possible_end:
                * Increment found_digits to 1.
                * Move pointer j to i.
            - Else if found_digits is 1 and num[i] matches the first character of possible_end:
                * Increment found_digits to 2.
                * Break the loop.
            - Decrement i.
        d. If found_digits is 2:
            - Calculate the operations needed as (j - i - 1) + (length of num - j - 1).
            - Update min_operations with the minimum of itself and the calculated operations.
    4. If min_operations is still set to its initial large value, return the length of num - 1 (all digits removed except one).
    5. Return min_operations.
```

Figure 10: Generated Pseudocode (Correct).

````
To determine the minimum number of operations required to make the number "2245047" special (divisible by 25), we need to find the fewest deletions
↪ needed to end the number with "00", "25", "50", or "75".

1. For "00":
   - The last '0' is at index 6.
   - The second last '0' is at index 5.
   - Operations needed: (6 - 5 - 1) + (7 - 6 - 1) = 0 + 0 = 0.

2. For "25":
   - The last '5' is at index 4.
   - The nearest '2' before that '5' is at index 2.
   - Operations needed: (4 - 2 - 1) + (7 - 4 - 1) = 1 + 2 = 3.

3. For "50":
   - The last '0' is at index 6.
   - The nearest '5' before that '0' is at index 4.
   - Operations needed: (6 - 4 - 1) + (7 - 6 - 1) = 1 + 0 = 1.

4. For "75":
   - The last '5' is at index 4.
   - There is no '7' before that '5'.

The minimum operations among these options are 0 (for ending "00"). Thus, no deletions are needed to make the number special.

```
0
```
````

Figure 11: LLM-based Execution (Execution Hallucination).

```
def minimumOperations(num: str) -> int:
    possible_ends = ["00", "25", "50", "75"]
    min_operations = len(num) + 1

    for possible_end in possible_ends:
        i = len(num) - 1
        found_digits = 0
        j = 0

        while i >= 0:
            if found_digits == 0 and num[i] == possible_end[1]:
                found_digits = 1
                j = i
            elif found_digits == 1 and num[i] == possible_end[0]:
                found_digits = 2
                break
            i -= 1

        if found_digits == 2:
            operations = (j - i - 1) + (len(num) - j - 1)
            min_operations = min(min_operations, operations)

    if min_operations == len(num) + 1:
        return len(num) - 1  # All digits removed except one

    return min_operations
```

Figure 12: Generated Code (Correct).

```
You are given a 0-indexed array of positive integers nums and a positive integer limit.
In one operation, you can choose any two indices i and j and swap nums[i] and nums[j] if |nums[i] - nums[j]| <= limit.
Return the lexicographically smallest array that can be obtained by performing the operation any number of times.
An array a is lexicographically smaller than an array b if in the first position where a and b differ, array a has an element that is less than the corresponding element in b. For example, the array [2,10,3] is lexicographically smaller than the array [10,2,3] because they differ at index 0 and 2 < 10.
```

Figure 13: Problem Description.

```
Algorithm lexicographicallySmallestArray(nums, limit):
    1. Initialize a graph with nodes representing each index of the array nums.
    2. Connect two nodes with an edge if the difference between their corresponding values in nums is less than or equal to limit.
    3. Perform a Depth-First Search (DFS) or Breadth-First Search (BFS) to find all connected components of the graph.
    4. For each connected component:
       - Extract the subarray of nums corresponding to the indices in this component.
       - Sort this subarray to get the lexicographically smallest order.
    5. Replace the elements in the original nums array at the positions of the connected component with the sorted values.
    6. Return the modified nums array as the result.
```

Figure 14: Generated Pseudocode (Correct).

````
To solve the problem, we need to find the lexicographically smallest array by swapping elements within a certain limit of their differences. The approach involves creating a graph where each index in the array represents a node, and an edge exists between nodes if the absolute difference between their corresponding values is less than or equal to the given limit.

For the given input:
- nums = [1, 5, 3, 9, 8]
- limit = 2

We analyze the differences:
- |1 - 5| = 4 (greater than 2, no edge)
- |1 - 3| = 2 (equal to 2, edge between indices 0 and 2)
- |1 - 9| = 8 (greater than 2, no edge)
- |1 - 8| = 7 (greater than 2, no edge)
- |5 - 3| = 2 (equal to 2, edge between indices 1 and 2)
- |5 - 9| = 4 (greater than 2, no edge)
- |5 - 8| = 3 (greater than 2, no edge)
- |3 - 9| = 6 (greater than 2, no edge)
- |3 - 8| = 5 (greater than 2, no edge)
- |9 - 8| = 1 (less than 2, edge between indices 3 and 4)

From this, we can identify connected components:
- Component 1: Indices [0, 2, 1] (values [1, 3, 5])
- Component 2: Indices [3, 4] (values [9, 8])

Sorting each component:
- Component 1 sorted: [1, 3, 5]
- Component 2 sorted: [8, 9]

Reconstructing the array with sorted components:
- [1, 3, 5, 8, 9]

Thus, the lexicographically smallest array that can be obtained is [1, 3, 5, 8, 9].
```
[1, 3, 5, 8, 9]
```
````

Figure 15: LLM Execution (Correct).

```python
def lexicographicallySmallestArray(nums: List[int], limit: int) -> List[int]:
    from collections import defaultdict, deque

    n = len(nums)
    graph = defaultdict(list)

    # Build the graph
    for i in range(n):
        for j in range(i + 1, n):
            if abs(nums[i] - nums[j]) <= limit:
                graph[i].append(j)
                graph[j].append(i)

    visited = [False] * n
    components = []

    # Function to perform BFS and find connected components
    def bfs(start):
        queue = deque([start])
        component = []
        visited[start] = True
        while queue:
            node = queue.popleft()
            component.append(node)
            for neighbor in graph[node]:
                if not visited[neighbor]:
                    visited[neighbor] = True
                    queue.append(neighbor)
        return component

    # Find all connected components
    for i in range(n):
        if not visited[i]:
            component = bfs(i)
            components.append(component)

    # Sort each component and place back into nums
    for component in components:
        indices = component
        values = [nums[i] for i in indices]
        values.sort()
        for idx, val in zip(indices, values):
            nums[idx] = val

    return nums
```

Figure 16: Generated Code (Implementation Error).

````
# Inputs
## Problem
```plaintext
{{ problem }}
```

## Starter code
```python
{{ starter_code }}
```

# Instruction
I provided you a coding problem.
You need to write a pseudocode for the problem.
Here are some conditions.
- The output should be written in a separate code block using three backticks (```) at the beginning and end.
- The part surrounded by double curly braces in the output format below is a placeholder. You need to replace it with an appropriate value to generate
↪  the output.
- Print only one code block. Do not print any other code block.

# Output Format{% raw %}
```plaintext
{{ pseudocode }}
```{% endraw %}
````

Figure 17: Prompt template for pseudocode generation.

````
# Inputs
## Problem
```plaintext
{{ problem }}
```

## Starter code
```python
{{ starter_code }}
```

## Pseudocode
```plaintext
{{ pseudocode }}
```

# Instruction
You need to write a python solution for the problem.
Here are some conditions.
- The code should implement the same algorithm as the given pseudocode.
- The output should be written in a separate code block using three backticks (```) at the beginning and end.
- Do not include `self` in the function input arguments.
- The part surrounded by double curly braces in the output format is a placeholder. You need to replace it with an appropriate value to generate the
↪ output.
- Print only one code block. Do not print any other code block.

# Output Format{% raw %}
```python
{{ code }}
```{% endraw %}
````

Figure 18: Prompt template for code generation.

````
# Inputs
## Problem
```plaintext
{{ problem }}
```

## Testcase Input
```
{{ tc_input }}
```

# Instruction
You need to predict the output of the testcase input based on the provided coding problem.
Here are some conditions.
- Write the reasoning steps before providing the final answer.
- Do not write the correct code or code to execute the test case; instead, write the expected output.
- The final output should not be in natural language format but should consist of the output itself only.
- The part surrounded by double curly braces in the output format below is a placeholder. You need to replace it with an appropriate value to generate
↪ the output.
- The final output should be written in a code block using three backticks (```) at the beginning and end.

# Output Format{% raw %}
{{ reasoning }}
```json
{{ output }}
```{% endraw %}
````

Figure 19: Prompt template for LLM-based execution (no grounding).

`````
# Inputs
## Problem
```plaintext
{{ problem }}
```

## Starter code
```python
{{ starter_code }}
```

## Pseudocode
```plaintext
{{ pseudocode }}
```

## Testcase Input
```plaintext
{{ tc_input }}
```

# Instruction
Predict the testcase output that would result from executing the given testcase input on the provided problem.
Here are some conditions.
- Before arriving at the final answer, understand the algorithm described in the pseudocode and write reasoning that explains the process of deriving the
↪  final answer based on the given testcase input.
- The final output should not be in natural language format but should consist of the output itself only.
- The part surrounded by double curly braces in the output format below is a placeholder. You need to replace it with an appropriate value to generate
↪  the output.
- Your final expected output should always be placed at the end of your response, enclosed by a pair of ```.

# Output Format{% raw %}
{{ reasoning }}
```
{{ expected_output }}
```{% endraw %}
`````

Figure 20: Prompt template for LLM-based execution (pseudocode grounding).

`````
# Inputs
## Problem
```plaintext
{{ problem }}
```

## Starter code
```python
{{ starter_code }}
```

## Code
```python
{{ pseudocode }}
```

## Testcase Input
```plaintext
{{ tc_input }}
```

# Instruction
Predict the testcase output that would result from executing the given testcase input on the provided problem.
Here are some conditions.
- Before arriving at the final answer, understand the algorithm described in the code and write reasoning that explains the process of deriving the final
↪  answer based on the given testcase input.
- The final output should not be in natural language format but should consist of the output itself only.
- The part surrounded by double curly braces in the output format below is a placeholder. You need to replace it with an appropriate value to generate
↪  the output.
- Your final expected output should always be placed at the end of your response, enclosed by a pair of ```.

# Output Format{% raw %}
{{ reasoning }}
```
{{ expected_output }}
```{% endraw %}
`````

Figure 21: Prompt template for LLM-based execution (code grounding).

````
# Inputs
## Problem
```plaintext
{{ problem }}
```

## Starter Code
```python
{{ starter_code }}
```

# Instruction
You need to generate testcase inputs for the problem.
Here are some conditions.
- Provide reasoning before generating the main output.
- Write exactly 3 test case inputs.
- If the test case input format supports executing multiple test cases at once, ensure each test case tests only a single scenario.
- Do not generate test case outputs.
- If starter code is provided, format the input as a JSON string representing a dictionary where the keys correspond to the input argument names of the
↪  given function.
- If starter code is not provided, format the input as expected by the program via standard input (stdin) so it can be directly used as input.
- Replace placeholders surrounded by double curly braces in the output format with appropriate values.
- Continue writing test case inputs up to a total of 3, following the provided format.
- Enclose all outputs in separate code blocks using three backticks (```).

# Output Format{% raw %}
## Reasoning
{{ reasoning }}
## Test Case Inputs
### Test Case Input 1
```
{{ testcase input 1 }}
```
### Test Case Input 2
```
{{ testcase input 2 }}
```
### Test Case Input 3
```
{{ testcase input 3 }}
```
{% endraw %}
````

Figure 22: Prompt template for test input suite generation.