Dynamic-Static Synergistic Selection Method for Candidate Code Solutions with Generated Test Cases

Ren-Biao Liu[1,2], **Jiang-Tian Xue**[1,2], **Chao-Zeng Ma**[1,2], **Hui Sun**[1,2], **Xin-Ye Li**[1,2], **Ming Li**[*1,2]

[1]National Key Laboratory for Novel Software Technology, Nanjing University, China
[2]School of Artificial Intelligence, Nanjing University, China
{liurb, xuejt, macz, sunh, lixy, lim}@lamda.nju.edu.cn

Abstract

Large language models (LLMs) show significant improvement in code generation. A common practice is sampling multiple candidate codes to increase the likelihood of producing an accurate solution. However, effectively identifying the best candidate from the pool is a significant challenge. Although existing code consensus methods attempt to solve this issue, they suffer from a critical problem: relying on test cases generated by LLMs, which can be flawed or provide incomplete coverage. This problem can result in erroneous validations, causing correct code to fail flawed tests and preventing the detection of functional differences in candidate code solutions. To address these issues, we present the Dynamic-Static Synergistic Selection Method, a novel framework that combines two complementary analytical approaches. First, it uses the abstract syntax tree (AST) to detect and filter candidate solutions and test cases. Second, the method statically analyzes the quality of the solutions and then dynamically validates functional consistency based on the execution results of the extracted inputs, thereby neutralizing the impact of faulty tests. Extensive experiments demonstrate that this synergistic approach significantly outperforms existing methods, substantially enhancing the correctness of the selected code.

Introduction

Code generation (Jiang et al. 2024; Roziere et al. 2023; Hui et al. 2024) aims to automatically produce functionally correct code based on natural language descriptions, representing a core technology for intelligent programming assistance tools. The field is characterized by rapid advancements and new models, with emerging approaches indicating an intense research focus on LLMs specifically for programming tasks. This shift moves beyond generic language understanding to domain-specific code generation.

Despite significant advancements in LLMs for code generation, practical applications still face notable quality issues. Directly generating a single code instance often results in outputs prone to semantic deviations, logical flaws, or security risks due to influences from temperature parameters, prompt biases, or errors in contextual understanding (Chen et al. 2021; Liu et al. 2023). While increasing the number of sampled candidate codes improves the probability of containing correct answers, manually validating large volumes of candidates is costly and time-consuming. This presents a fundamental trade-off: achieving higher accuracy often necessitates generating more candidates, which introduces a significant human cost for verification, thereby creating a practical barrier for LLM-based programming assistance.

Code ranking models are proposed to address the challenge of selecting correct or high-quality outputs from generated candidates. These approaches evolved from static preference modeling (Ni et al. 2023; Liu et al. 2024) to dynamic frameworks incorporating execution feedback (Li et al. 2024; Sun et al. 2024), and functional similarity analysis (To et al. 2023; Inala et al. 2022; Zhang et al. 2023a; Lyu et al. 2025). While effective, these approaches typically rely on training separate ranking models and designing task-specific strategies, often requiring substantial computational resources and depending on lexical-level code features. Relying on a ranking model has a generalization issue. This weakness has prompted a shift toward test-time scaling, which enhances selection accuracy by increasing computational effort and the number of candidates.

To address these issues, code consensus methods select and categorize functionally consistent code sets from multiple LLM-generated candidates through dynamic testing and validation, mitigating the impact of generation uncertainty (Chen et al. 2023, 2024; Zhang et al. 2023b; Huang et al. 2023; Wang et al. 2022; Lahiri et al. 2022; Zuo et al. 2025; Shi et al. 2022). However, existing code consensus methods mainly rely on LLM-generated test cases for validation, which suffer from two critical issues: First, erroneous candidate codes may pass flawed test cases due to coexisting errors in both, leading to misjudgments; Second, test cases exhibit coverage insufficiency and redundancy, resulting in limited validation on covered code portions while ignoring potential discrepancies in uncovered regions. For code consensus methods, relying solely on LLM-generated test cases can lead to false positives due to inherent flaws in the content generated by LLMs and fail to account for incomplete test coverage, resulting in unreliable assessments of functional consistency. This concern underscores the need to move beyond a fixed set of generated tests. Instead, validation should be treated as a scalable computation performed during testing. A more robust approach would use test-time techniques

[*]Ming Li is the corresponding author.

to expand the test suite.

To tackle these challenges, this paper proposes **Dynamic-Static Synergistic Selection Method ($\mathcal{DS}^3$)** for Python-based function-level code generation tasks. This method addresses erroneous test case evaluations and incomplete coverage by integrating dynamic validation with static analysis to transcend the single-dimensional validation paradigm of existing approaches. The synergistic mechanism is central to this proposition. It implies that the combined approach yields a greater benefit than the sum of its parts, effectively overcoming the complex, intertwined challenges of LLM-generated code quality. In the *static dimension*, the method combines **abstract syntax tree (AST) filtering** and **static analysis scoring** to capture hidden discrepancies that are unidentifiable through dynamic validation alone. In the *dynamic dimension*, it employs **execution equivalence** and **functional consistency** to eliminate faulty or redundant test interference, precisely identifying functionally consistent candidates. The $\mathcal{DS}^3$framework establishes a foundational validation system through dynamic execution and static analysis, addressing confusing ranks caused by faulty test cases and incomplete coverage. Experimental results demonstrate that the proposed methods outperform existing code consensus approaches on multiple benchmark datasets, significantly improving code generation quality.

The main contributions are highlighted as follows:

- **A Novel Synergistic Framework for Code Selection** The paper introduces $\mathcal{DS}^3$, a novel framework to improve code selection from LLM-generated candidates. It is designed to overcome the critical limitations of existing consensus methods by robustly handling flawed test cases and incomplete coverage.
- **A Hybrid Methodology Integrating Dynamic Validation and Static Analysis** The method combines dynamic validation and static analysis. It ranks functionally equivalent code using dynamic execution results, while statically analyzing to identify subtle discrepancies missed due to incomplete test coverage.
- **Demonstrated State-of-the-Art Performance through Empirical Validation** Extensive experiments on multiple benchmark datasets empirically validate the approach. The results confirm that $\mathcal{DS}^3$significantly outperforms existing code consensus methods and substantially enhances the final quality and correctness of the selected code.

Related Work

LLM for Code Generation

LLMs (Brown et al. 2020; Achiam et al. 2023; Zhao et al. 2023) have emerged as powerful tools for generating language. Codex (Chen et al. 2021) first demonstrates that large-scale pre-training on code enables models to capture the mappings between natural and programming languages. Subsequently, large multilingual code models, such as CodeLLaMA (Roziere et al. 2023) and CodeGeeX (Zheng et al. 2023), further demonstrated the cross-lingual transferability of syntactic structures in code. At the model architecture level, CodeGen (Nijkamp et al. 2023) employs a decoding mechanism that separates control and data flow modeling. Regarding the training strategy, AlphaCode (Li et al. 2022) uses reinforcement learning to refine its generation policy using reward signals. Consequently, CodeLLMs (Bai et al. 2023; Guo et al. 2024; Liu et al. 2025) are trained for programming tasks through specific optimizations such as syntax-aware learning (Gong, Elhoushi, and Cheung 2024), specialized tokenization (Bavarian et al. 2022), and program translation (Du et al. 2025; Li et al. 2025).

Code Solution Ranking

Post-processing of generated code has evolved from solution ranking to preference modeling frameworks. Early approaches, such as that described in (Liu et al. 2024), used synthetic data to model human preferences. Sun et al. (2024) used a multi-task framework for classification and feedback generation, while Li et al. (2024) embedded a lightweight environment for dynamic verification. To et al. (2023) developed a ranking system based on functional similarity analysis of code graphs. The decoupled verifier co-design in LEVER by Ni et al. (2023) marks a recent paradigm shift towards modularity. This approach establishes a comprehensive evaluation framework that enables the co-optimization of verifiers and generation models. CODERANKER (Inala et al. 2022) models the code selection as a binary classification and sorts the candidates according to predicted scores. The CODER-REVIEWER (Zhang et al. 2023a) proposes a novel code ranking method according to the Bayes method. Lyu et al. (2025) introduced a learning framework to optimize the Pass@k metric.

Code Consensus Method

The quality assurance of generated code has evolved from dynamic verification to formal modeling using the code consensus method. Early explorations, such as the dual-path generation framework in CODET (Chen et al. 2023), embed dynamic execution to filter candidates based on test outcomes. The approach was mathematically refined in B4 (Chen et al. 2024), which used a Bayesian joint model to establish a quantitative relationship between test credibility and code quality, thereby mitigating noise. A significant advancement came with the integration of formal verification methods. ALGO (Zhang et al. 2023b) utilizes mathematical proofs in the generation workflow to ensure logical completeness, a gap left by traditional testing methods. The method was further extended by MPSC (Huang et al. 2023), which introduced a multi-perspective framework for cross-validating solutions, specifications, and test cases through graph-structured consensus modeling. The code consensus method shifts from verifying functional correctness to trustworthiness and has emerged as a key technique for ensuring the consistency and correctness of candidate solutions.

Preliminaries

Problem Definition

Code generation aims to automatically generate a code solution x from a given prompt. In detail, the prompt contains

a natural language problem description and a code snippet that includes statements such as imports and the function header. A code solution is the completed function that solves the programming problem described in the prompt. Generally, we sample a set of N code solutions, denoted as $\mathcal{X} = \{x_1, x_2, \cdots, x_N\}$, based on the prompt using a pre-trained language model. In addition to generating code solutions, we must use test cases to evaluate their accuracy. Each test case includes an input and an expected output for a function specified in the prompt. We then feed the original prompt and the specially designed instruction to the language model, and sample a set of M test cases, denoted as $\mathcal{Y} = \{y_1, y_2, \cdots, y_M\}$, from the model output.

The correctness of the generated code solutions and test cases is initially unknown. The observable outcomes are captured in a matrix $\mathbf{E} \in \{0,1\}^{N \times M}$, where $e_{i,j} = 1$ if the i-th code solution passes the j-th test case, and $e_{i,j} = 0$ otherwise. We refer to $\mathbf{E}$ as the execution matrix and each element $e_{i,j}$ as an execution result. We assume the set of generated solutions contains at least one correct code solution because otherwise, a selection strategy would be meaningless. All correct code solutions are functionally equivalent, and test outcomes are deterministic.

Consensus Solution

Previous research aimed to identify the most extensive possible set of code-test pairs, $\mathcal{S}_{(\hat{\mathcal{X}}, \hat{\mathcal{Y}})}$, in which all constituent code solutions exhibit identical behavior across all constituent test cases. It is usually referred to as a *consensus set*.

Definition 1 (Consensus Set) *A consensus set $\mathcal{S}$ is defined as a pair of subsets $(\hat{\mathcal{X}}, \hat{\mathcal{Y}})$, where $\hat{\mathcal{X}} \subseteq \mathcal{X}$ and $\hat{\mathcal{Y}} \subseteq \mathcal{Y}$, such that all code solutions in $\hat{\mathcal{X}}$ exhibit functionally identical behavior on all test cases in $\hat{\mathcal{Y}}$. This behavioral consistency can be expressed in the following two equivalent ways. From the testing perspective, every code solution in $\hat{\mathcal{X}}$ passes every test case in $\hat{\mathcal{Y}}$:*

$$\forall x_i \in \hat{\mathcal{X}},\ \forall y_k \in \hat{\mathcal{Y}}, \quad e_{i,k} = 1.$$

From the function perspective, for any two solutions in $\hat{\mathcal{X}}$, their execution results are identical on all test cases in $\hat{\mathcal{Y}}$:

$$\forall x_i, x_j \in \hat{\mathcal{X}},\ x_i \equiv x_j \iff \forall y_k \in \hat{\mathcal{Y}}, \quad e_{i,k} = e_{jk}.$$

In other words, all code solutions in the consensus set not only pass the same set of test cases but also exhibit indistinguishable behavior on them. It assumes that a consensus set containing more code solutions and test cases indicates a higher level of consensus, and thus, the more likely they are to be correct. Recent studies aim to use $\mathbf{E}$ to assess the correctness of code solutions and select the optimal one.

Previous Modeling

Prior approaches have predominantly focused on information extraction or feature processing concerning the matrix $\mathbf{E}$. This subsection provides a brief review of some representative modeling methods for solving this problem.

The first family of methods, collectively referred to as MAX-PASS (Lahiri et al. 2022; Li et al. 2022; Le et al. 2022; Rozière et al. 2022), selects the candidate solution that passes the maximum number of test cases. This approach can be considered a specialized form of broader techniques, such as Self-CONSISTENCY (Wang et al. 2022) and MAJORITY VOTE (Zuo et al. 2025), adapted for test-time scaling in code generation. Formally, the optimal code solution is selected by maximizing the number of passed test cases:

$$x_i, \text{ where } i = \arg\max_{i \in [N]} \sum_{j=1}^{M} e_{i,j}.$$

Another alternative family of methods clusters code solutions based on functional consensus, which is determined by their execution outcomes on a shared set of test cases (Li et al. 2022; Shi et al. 2022; Chen et al. 2023). A representative method from this family is CODET (Chen et al. 2023). It seeks to identify the most significant consensus set from K candidates. The significance of a set $\mathcal{S}$, termed its capacity, is defined as the product of the number of its constituent solutions $|\hat{\mathcal{X}}|$ and test cases $|\hat{\mathcal{Y}}|$: $|\mathcal{S}| = |\hat{\mathcal{X}}| \times |\hat{\mathcal{Y}}|$. Consequently, CODET scores each consensus set by its capacity and selects a solution from the set with the maximum score:

$$x_i \in \mathcal{S}_k, \text{ where } k = \arg\max_{k \in [K]} |\hat{\mathcal{X}}_k| \times |\hat{\mathcal{Y}}_k|.$$

Similarly, other ranking methods, such as MBR-EXEC (Shi et al. 2022) and AlphaCode-C (Li et al. 2022), score individual solutions based on their empirical error on a given set of test cases. These methods formulate the selection of an optimal code solution as a Minimum Bayes Risk (MBR) problem, where risk is measured by execution consistency among a set of candidate solutions. To estimate functional equivalence, each candidate solution is executed on a set of test cases, and its outputs are compared against those of other candidates to compute an empirical agreement score. Formally, the solution with the lowest risk is selected:

$$x_i, \text{ where } i = \arg\max_{i \in [N]} \sum_{j=1}^{N} \mathbb{I}[e_{i,\cdot} = e_{j,\cdot}],$$

where $\mathbb{I}[\cdot]$ is the indicator function. This objective corresponds to selecting the candidate with the highest consensus in functional behavior under model-based sampling. From a Bayesian perspective, this approach reflects the core idea of the optimization objective: it approximates the posterior distribution over candidate programs through sampling. It selects the solution with maximum expected utility.

Methods

Revisiting Consensus in Code Generation Models such as GPT-4 (Achiam et al. 2023) and CodeLlama (Roziere et al. 2023) can simultaneously generate candidate code solutions and their corresponding test cases. However, the correctness of code and test cases generated by LLMs is not guaranteed, primarily due to model hallucinations. Traditional static analysis tools can detect violations of coding

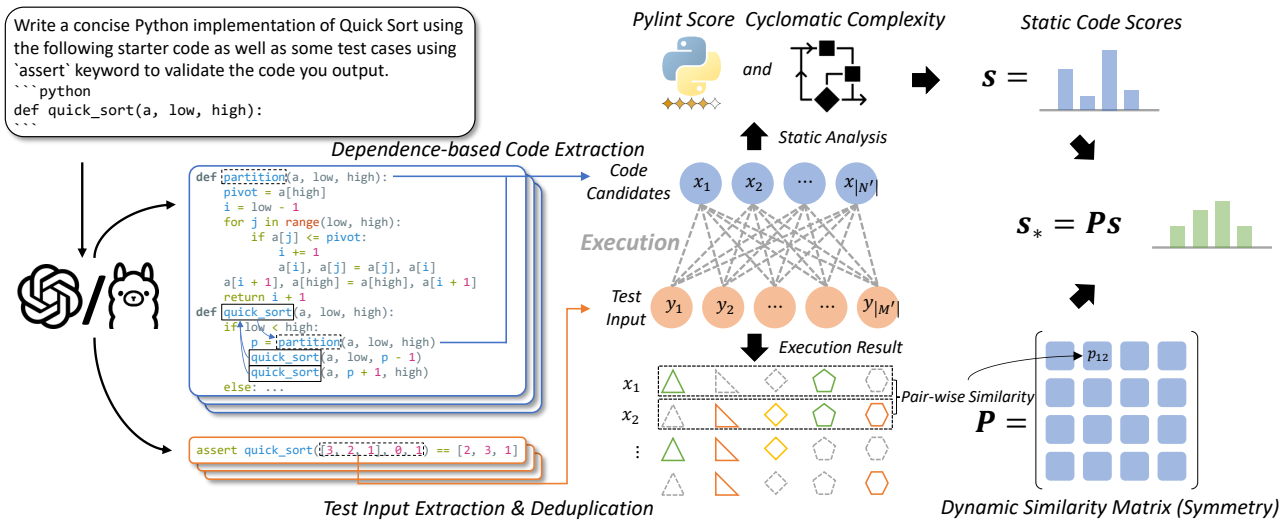

Figure 1: Overview of the $\mathcal{DS}^3$ framework. Candidate code snippets and test cases generated by the LLM are first sanitized and filtered to produce complete code and valid test inputs. Static analysis then computes divergence scores for code candidates. Each code candidate is also executed on all test inputs to collect outputs, which are compared to build a dynamic similarity matrix. Finally, static and dynamic signals are combined to compute an overall ranking score for each candidate.

standards, but cannot perform dynamic validation. They rely solely on syntactic pattern matching and rule-based inference, which fail to capture the code's actual runtime behavior. Conversely, dynamic validation methods capture runtime behavior through test execution. Its simplistic counting strategy has two primary limitations. First, evaluating only the number of passed tests disregards structural differences in code implementations. For example, two sorting algorithms, such as QuickSort and BubbleSort, might pass the same set of tests. Under this evaluation criterion, they are considered equivalent, despite potentially having a difference in complexity. Second, the absence of a quality assessment mechanism for test cases means faulty tests can incorrectly reject valid code. These limitations underscore the shortcomings of current methods for assessing code quality and ensuring robustness against flawed test cases.

Overall Framework This study proposes the Dynamic-Static Synergistic Selection Method ($\mathcal{DS}^3$) to address the dual limitations of methods that rely on static analysis or dynamic execution alone. Its core innovation is the breakthrough from the traditional, single-dimensional validation paradigm. As illustrated in Figure 1, the proposed $\mathcal{DS}^3$ method comprises two main components: a dynamic validation component for code consensus scoring and a static analysis component for code divergence scoring. The static analysis component assigns each candidate solution a score based on static metrics. Departing from static analysis that relies solely on semantic rules, our approach constructs a multidimensional divergence metric by incorporating lexical-level code quality and syntactic-level code complexity. The dynamic validation component assesses the degree of consensus between code solutions by evaluating the consistency of their execution outputs with those of other candidates, thereby quantifying functional agreement with the candidate set. Furthermore, in contrast to traditional dynamic validation, our method achieves fine-grained assessment by analyzing the execution results of candidate code.

Filtering and Scoring Based on Static Analysis

The first challenge arises from the potential for syntax errors in the test cases generated by the model. Due to noise in the pre-trained data and the model's ability, these test cases often contain incorrect assertions or invalid inputs. More critically, if the model is not powerful enough, the generated test cases and candidate code may be biased. For instance, when tackling complex problems, the model may produce a test case with the correct code but the wrong test, or vice versa. This type of failure renders traditional validation mechanisms ineffective as erroneous implementations and faulty tests create a logical closed loop, increasing the risk of selecting an incorrect code candidate.

Sanitize Code Solutions We introduce a sanitizing pipeline to enhance the reliability of code produced by LLMs, starting with the identification of the longest contiguous valid code block using AST. It then discards the malformed or incomplete trailing tokens in the generated outputs. After extraction, we perform a static analysis recognizing all top-level nodes as imports, class definitions, function definitions, or global variable assignments. A key component of this method is its dependency-driven filtering mechanism. Given a specified entry point, the mechanism constructs a dependency graph of all definitions. Traversing this graph identifies the minimum set of reachable definitions, eliminating dead code and pruning any functions or variables that are not essential for executing the entry point. Finally, the sanitized code is reconstructed with the neces-

sary import statements and transitively reachable definitions. It results in a self-contained program that is ready for execution or further analysis and is syntactically correct.

Filter Test Input We use a multi-stage analysis methodology to ensure the semantic validity and robustness of the generated test cases. First, we perform the syntax analysis to parse each candidate test case. We traverse the AST to locate the target function invocation and extract all arguments. This initial step confirms that the test case is syntactically correct and structurally relevant to the target function. Next, we preliminarily validate the extracted arguments against the function's signature. At this stage, the function's body is not executed. Instead, we perform prior validation on the arguments to confirm their conformance to the types and constraints specified in the function's signature. A set of arguments is considered valid only if it successfully passes both the static extraction and preliminary validation stages. Arguments that fail either stage are rejected to ensure that only syntactically and semantically valid inputs are retained.

Static Quality Score To assess and differentiate code quality before dynamic analysis, we perform a static assessment of each candidate solution to quantify its adherence to coding standards and style conventions and detect potential logical errors. Let $\widetilde{\mathcal{X}} = \{x_1, x_2, \ldots, x_{N'}\}$ and $\widetilde{\mathcal{Y}} = \{y_1, y_2, \ldots, y_{M'}\}$ denote the filtered sets of code functions and test cases. We employ the `Pylint` static analysis tool for this purpose. For each solution x_i, we compute a Pylint score, $\boldsymbol{s}$, which is normalized to a $[0, 1]$ scale. This score reflects aspects of code quality independent of its execution. Similarly, another score is computed based on cyclomatic complexity. This score is then normalized to $[0, 1]$, whereby lower complexity corresponds to a higher score. The final static score is calculated using a weighted sum of these two metrics. These scores form a static quality vector $\boldsymbol{s}$, where each element $s_i \in [0, 1], \forall i \in [N']$.

Metric	CG	CX	SC	CL	DS
Syntactic Correctness	62.2	76.5	62.7	62.4	61.5
Execute Correctness	61.3	75.0	60.9	61.5	60.2
Semantic Correctness	40.4	48.9	33.3	40.3	45.1
Line Coverage	86.3	86.1	86.0	86.4	90.0
Branch Coverage	62.2	62.9	62.9	62.6	60.0

Table 1: Accuracy (%) comparison of test cases generated by different models across multiple dimensions on HumanEval, where **CG** = CodeGen, **CX** = CodeX, **SC** = StarCoder, **CL** = CodeLlama, and **DS** = DeepSeek-Coder.

Consistency Based on Dynamic Execution

The second challenge lies in the structural bias of model-generated test case coverage. For example, test data for HumanEval (Chen et al. 2021) in Table 1 show that, while test cases generated by LLMs achieve an average line coverage of approximately 86% for candidate code, their average branch coverage is only around 62%, revealing an incomplete validation process. This significant discrepancy suggests that LLMs tend to generate test cases for explicit execution paths. However, they cannot cover critical branches, such as conditional logic and exception handling. Consequently, this coverage bias can allow divergences in critical branch paths to go undetected among different implementations. Different implementations may exhibit functional divergences within uncovered code segments, yet still be grouped into the same consensus set. Consequently, code with boundary condition defects may be overlooked due to a lack of targeted testing.

The cornerstone of our method is the concept of dynamic execution consistency. This concept posits that, despite different implementations, high-quality solutions will exhibit similar behavior when presented with a given set of inputs. We quantify this behavioral similarity using a pairwise similarity matrix $\mathbf{P} \in [0, 1]^{N' \times N'}$. The similarity between two distinct solutions, x_i and x_j, is calculated as the average consistency score across test inputs. For a given test case, y_k, with input arguments, a_k, the execution-level similarity is based on the condition of functional equivalence; both solutions, x_i and x_j, must execute successfully on a_k and produce identical outputs. This condition ensures the comparison is restricted to functionally equivalent solutions for a given input. The test-wise similarity can be calculated:

$$\sigma(x_i, x_j, y_k) = \begin{cases} 1 & \text{if } \operatorname{exec}(x_i, y_k) == \operatorname{exec}(x_j, y_k) \\ 0 & \text{otherwise} \end{cases}$$

The final similarity score $p_{i,j}$ in the matrix is the mean of these values over all M test cases, providing a robust measure of overall behavioral similarity:

$$p_{ij} = \frac{1}{|M'|} \sum_{k=1}^{|M'|} \sigma\left(x_i, x_j, y_k\right)$$

For the diagonal elements, we set $p_{i,i} = 1$, as a solution is perfectly similar to itself. Note that the matrix $\mathbf{P}$ is symmetric, *i.e.*, $p_{i,j} = p_{j,i}$. Summing the similarity scores (*i.e.*, the row sums of $\mathbf{P}$) would treat all solutions as equally valid. However, our static analysis has already provided an estimate of the intrinsic quality of each solution. We use this information to compute a more refined final score. The final score for candidate solution x_i is a weighted sum of its similarity scores with all other solutions. Specifically, the score for x_i is the dot product of the i-th row of the similarity matrix $\mathbf{P}$ and the static score vector $\boldsymbol{s}$:

$$\boldsymbol{s}_* = \mathbf{P}\boldsymbol{s}$$

This scoring mechanism represents a consensus-based evaluation. A solution receives a high score if it behaves similarly to others with high static quality. In other words, the high value of the final score means that the code is a reliable solution, *i.e.*s_i is high, and it is consistent with many other solutions, *i.e.*the sum of $p_{i,\cdot}$ is high as well. This approach effectively marginalizes solutions that pass all tests but do so inconsistently compared to other well-structured solutions. It favors candidates who belong to a cluster of high-quality, behaviorally consistent programs. After computing the aggregate score for each candidate solution, the solution with the highest score is selected as the optimal one.

Experiment

Benchmark We conducte experiments on widely used code generation benchmarks: HumanEval(+) and MBPP. HumanEval (Chen et al. 2021) and MBPP (Austin et al. 2021) are datasets comprising hand-written Python programming problems. HumanEval+ (Liu et al. 2023) extends HumanEval by incorporating additional unit tests.

Metric In the context of Code Ranking, the metric Pass@k (Chen et al. 2021) is pivotal for evaluating model performance. It assesses the probability of a model generating at least one correct code sample out of k attempts for a given problem. Mathematically, Pass@k is expressed as Pass@$k = 1 - (1 - p)^k$, where p is the probability of generating a correct sample. To address the computational challenges of directly calculating Pass@k, an unbiased estimation method is employed:

$$\text{Pass@}k = 1 - \frac{\binom{n-c}{k}}{\binom{n}{k}}$$

where $\binom{n}{k}$ is the binomial coefficient. This method provides a reliable measure of the model's performance with significantly reduced computational cost. In the context of code ranking, it denotes the probability that at least one of the top-k candidate solutions passes all test cases.

Baselines We compare our method against two categories of baselines: several general or specialized LLMs, such as GPT-3.5-Turbo, GPT-4 (Achiam et al. 2023), Code Llama (Roziere et al. 2023), WizardCoder (Luo et al. 2024), and Deepseek Coder (Guo et al. 2024); and various post-hoc approaches that enhance LLMs during test-time, including MAX-PASS (Wang et al. 2022), MBR-EXEC (Shi et al. 2022), CODET (Chen et al. 2023). We also compare with agent-based methods, Self-COLLABORATE (Dong et al. 2024) and MPSC (Huang et al. 2023). For the post-hoc approaches, we use GPT-3.5-Turbo as the base model to generate 200 candidate solutions for each problem. To align with the experimental setup of CODET (Chen et al. 2023), we use 500 test cases for the required baselines. For method MPSC, an additional 50 specifications are required. All baselines, where applicable, are evaluated with the same set of generated solutions and test cases to ensure a fair comparison.

Main Results

The experimental results on the HumanEval and MBPP benchmarks are summarized in Table 2. Our proposed method, $\mathcal{DS}^3$, consistently achieves strong performance across all metrics. Notably, it surpasses GPT-4 in Pass@1 on both benchmarks, establishing a new SOTA in this setting. The result highlights the effectiveness of our approach, even when built upon a weaker backbone model, demonstrating its ability to enhance code generation quality through consensus-based post-processing significantly. Compared to other augmentation methods such as MAX-PASS, MBR-EXEC, and CODET, our method exhibits consistent improvements across different k values. It is worth noting that, unlike approaches that benefit more from increasing k, our method already achieves strong results at $k = 1$, suggesting that our ranking strategy captures essential correctness signals early on. This property is particularly desirable in latency-sensitive or compute-constrained scenarios.

Result of Hard Test

Table 3 presents the evaluation results on the HumanEval+ benchmark, where our proposed method, $\mathcal{DS}^3$, significantly outperforms all baselines built upon the same foundation model. Notably, $\mathcal{DS}^3$ achieves the highest Pass@1 score, surpassing even the more powerful GPT-4 model. When compared with other post-hoc methods, such as MAX-PASS and MBR-EXEC, the improvement of $\mathcal{DS}^3$ over the GPT-3.5-Turbo baseline is substantially larger. The comparatively modest gains on the more challenging tasks highlight the importance of our consensus-driven scoring mechanism, particularly in the face of inherent distribution shifts. These results verify that $\mathcal{DS}^3$ maintains superior performance and exhibits remarkable resilience and scalability on more difficult and perturbed datasets.

Execution efficiency

Besides demonstrating superior performance in accuracy, $\mathcal{DS}^3$ exhibits significant advantages in execution efficiency compared to representative augmentation methods, such as CODET. By introducing preliminary static analysis to filter candidate solutions and test cases, we substantially reduce the computational overhead typically associated with exhaustive dynamic executions. Empirical evaluations conducted on both HumanEval and MBPP benchmarks, of which the statistics are used to estimate the time complexity, indicate that $\mathcal{DS}^3$ reduces the total execution count by approximately 41.8% to 61.4%. These results underscore the efficacy of $\mathcal{DS}^3$ in improving code generation accuracy and significantly enhancing test-time computational efficiency.

Generalization over Models

We comprehensively compare Pass@1 performance across various foundation models to evaluate the generalization capabilities of our proposed method, with a primary focus on tasks that have at least one passed solution. As detailed in Table 4, the results demonstrate the consistent and robust performance of $\mathcal{DS}^3$ against several established baselines. It is evident that simpler methods, such as RANDOM and MAX-PASS, yield relatively low performance, while more sophisticated techniques, like MBR-EXEC, CODET, and $\mathcal{B}^4$ (Chen et al. 2024), offer substantial improvements. Notably, our method, $\mathcal{DS}^3$, consistently achieves highly competitive results across all tested models. These findings underscore the broad applicability and strong generalization of $\mathcal{DS}^3$, confirming its effectiveness across a broad spectrum of code generation architectures. $\mathcal{DS}^3$ introduces only one hyperparameter to balance the contribution of the cyclomatic complexity score and `Pylint` score. We also conduct a sensitivity analysis of the hyperparameter to support our method, which further demonstrates its robustness.

Method	HumanEval			MBPP		
	Pass@1	Pass@2	Pass@5	Pass@1	Pass@2	Pass@5
GPT-4	<u>81.48</u>	**86.31**	**90.46**	71.26	<u>74.27</u>	<u>76.99</u>
GPT-3.5-Turbo	68.38	76.24	<u>83.15</u>	66.80	72.34	76.60
DeepSeekCoder	79.30	-	-	70.00	-	-
WizardCoder	73.20	-	-	61.20	-	-
CodeLlama	62.20	-	-	61.20	-	-
MAX-PASS	$73.86_{+5.48}$	$73.93_{-2.31}$	$74.10_{-9.05}$	$71.70_{+4.90}$	$71.73_{-0.61}$	$71.82_{-4.78}$
MBR-EXEC	$72.96_{+4.58}$	$76.47_{+0.23}$	$79.00_{-4.15}$	$70.79_{+3.99}$	$73.14_{+0.80}$	$74.85_{-1.75}$
Self-COLLABORATE	$74.40_{+6.02}$	-	-	$68.20_{+1.40}$	-	-
CODET	$78.05_{+9.67}$	$78.05_{+1.81}$	$78.30_{-4.85}$	<u>71.90</u>$_{+5.10}$	$71.95_{-0.39}$	$72.02_{-4.58}$
MPSC	$74.17_{+5.79}$	$77.02_{+0.78}$	$78.53_{-4.62}$	$69.34_{+2.54}$	$70.06_{-2.28}$	$71.85_{-4.75}$
$\mathcal{DS}^3$	**81.71**$_{+13.33}$	<u>82.32</u>$_{+6.08}$	$82.93_{-0.22}$	**76.11**$_{+9.31}$	**77.28**$_{+4.94}$	**78.22**$_{+1.62}$

Table 2: Results on the HumanEval and MBPP code generation benchmarks. The foundation model for MPSC, MAX-PASS, MBR-EXEC, CODET, and Self-COLLABORATE is GPT-3.5-Turbo. Improvements are shown relative to GPT-3.5-Turbo. **Bold** and <u>underlined</u> numbers indicate the best and second-best scores, respectively.

Method	Pass@1	Pass@2	Pass@5
GPT-4	70.52	**75.48**	**79.54**
GPT-3.5-Turbo	58.75	66.58	73.96
MAX-PASS	$63.50_{+4.75}$	$64.70_{-1.88}$	$65.67_{-8.29}$
MBR-EXEC	$62.12_{+3.37}$	$67.08_{+0.50}$	$71.38_{-2.58}$
CODET	$67.87_{+9.12}$	$68.75_{+2.17}$	$69.65_{-4.31}$
MPSC	$65.05_{+6.30}$	$69.76_{+3.18}$	$71.72_{-2.24}$
$\mathcal{DS}^3$	**73.17**$_{+14.42}$	<u>75.00</u>$_{+8.42}$	<u>76.22</u>$_{+2.26}$

Table 3: Results on the HumanEval+ benchmark. All methods except GPT-4 are built on GPT-3.5-Turbo. The improvements are relative to GPT-3.5-Turbo.

Method	CG	CX	SC	CL	DS
RANDOM	32.5	39.2	29.8	34.1	65.3
MAX-PASS	28.6	57.8	32.2	40.6	58.2
MBR-EXEC	44.8	55.0	47.9	52.6	80.4
CODET	51.5	71.7	55.0	61.7	79.2
$\mathcal{B}^4$	58.0	73.1	59.3	64.8	80.5
$\mathcal{DS}^3$	59.1	77.5	59.3	66.2	88.7

Table 4: Pass@1 (%) on the HumanEval benchmark comparison across different models, where **CG** = CodeGen, **CX** = CodeX, **SC** = StarCoder, **CL** = CodeLlama, and **DS** = DeepSeek-Coder.

Ablation Study

Table 5 shows the ablation results on the HumanEval benchmark with GPT-3.5-Turbo. The full $\mathcal{DS}^3$ framework achieves the best performance, confirming the effectiveness of combining dynamic validation and static analysis. Removing either component (*w/o Dynamic* or *w/o Statics*) leads to performance drops, indicating their complementary roles. *w/o Ranking* further degrades results, underscoring the value of score integration. For the statistics on the MBPP benchmark, an extended ablation study leads to a similar conclusion. These findings highlight the necessity of each component and the overall framework.

Method	Pass@1	Pass@2	Pass@5
$\mathcal{DS}^3$	81.71	82.32	82.93
w/o Dynamic	68.29	74.39	81.10
w/o Statics	79.88	80.49	81.10
w/o Ranking	70.99	76.83	82.32
w/o ALL	68.38	76.24	83.15

Table 5: Ablation results on the HumanEval benchmark using GPT-3.5-Turbo as the generation model. "w/o Dynamic" corresponds to fixing $p_{ij} = 1$ for all i, j, "w/o Statics" corresponds to fixing $s_i = 1$, and "w/o Ranking" is equivalent to removing both the dynamic and static components

Conclusion

This work presents $\mathcal{DS}^3$, a novel Dynamic-Static Synergistic Selection framework that ranks generated code based on quality. This approach overcomes the limitations of existing consensus methods. Unlike previous approaches, which rely solely on test cases sampled by LLMs, $\mathcal{DS}^3$ integrates dynamic validation and static analysis to create a more reliable, multidimensional code selection strategy. It mitigates the risk of flawed test case evaluations and reduces the impact of coverage gaps by dynamically verifying functional consistency and statically identifying discrepancies in candidate solutions. Extensive empirical studies across multiple benchmarks demonstrate that our framework achieves state-of-the-art performance, substantiating its generalization and effectiveness in selecting correct, high-quality code. It provides a comprehensive approach to achieving more reliable and robust test-time scaling in code generation with LLMs.

Acknowledgments

This paper is supported by the Major Program (JD) of Hubei Province (2023BAA024) and NSFC (62076121). The authors thank Dr. Wang-Zhou Dai for his insightful discussions regarding this work.

References

Achiam; Josh; Adler; Steven; Agarwal; Sandhini; Ahmad; Lama; Akkaya; Ilge; Aleman; Leoni, F.; Almeida; Diogo; Altenschmidt; Janko; Altman; Sam; Anadkat; Shyamal; et al. 2023. Gpt-4 Technical Report. arXiv:2303.08774.

Austin; Jacob; Odena; Augustus; Nye; Maxwell; Bosma; Maarten; Michalewski; Henryk; Dohan; David; Jiang; Ellen; Cai; Carrie; Terry; Michael; Le; Quoc; et al. 2021. Program Synthesis with Large Language Models. arXiv:2108.07732.

Bai; Jinze; Bai; Shuai; Chu; Yunfei; Cui; Zeyu; Dang; Kai; Deng; Xiaodong; Fan; Yang; Ge; Wenbin; Han; Yu; Huang; Fei; et al. 2023. Qwen technical report.

Bavarian; Mohammad; Jun; Heewoo; Tezak; Nikolas; Schulman; John; McLeavey; Christine; Tworek; Jerry; Chen; and Mark. 2022. Efficient training of language models to fill in the middle. arXiv:2207.14255.

Brown, T. B.; Mann, B.; Ryder, N.; Subbiah, M.; Kaplan, J.; Dhariwal, P.; Neelakantan, A.; Shyam, P.; Sastry, G.; Askell, A.; Agarwal, S.; Herbert-Voss, A.; Krueger, G.; Henighan, T.; Child, R.; Ramesh, A.; Ziegler, D. M.; Wu, J.; Winter, C.; Hesse, C.; Chen, M.; Sigler, E.; Litwin, M.; Gray, S.; Chess, B.; Clark, J.; Berner, C.; McCandlish, S.; Radford, A.; Sutskever, I.; and Amodei, D. 2020. Language Models are Few-Shot Learners. In *Proc. of NeurIPS*.

Chen; Mark; Tworek; Jerry; Jun; Heewoo; Yuan; Qiming; Pinto; Oliveira, H. P. D.; Kaplan; Jared; Edwards; Harri; Burda; Yuri; Joseph; Nicholas; Brockman; Greg; et al. 2021. Evaluating Large Language Models Trained on Code. arXiv:2107.03374.

Chen; Mouxiang; Liu; Zhongxin; Tao; He; Hong; Yusu; Lo; David; Xia; Xin; Sun; and Jianling. 2024. B4: Towards Optimal Assessment of Plausible Code Solutions with Plausible Tests. In *Proc. of ASE*, 1693–1705.

Chen, B.; Zhang, F.; Nguyen, A.; Zan, D.; Lin, Z.; Lou, J.; and Chen, W. 2023. CodeT: Code Generation with Generated Tests. In *Proc. of ICLR*.

Dong; Yihong; Jiang; Xue; Jin; Zhi; Li; and Ge. 2024. Self-collaboration code generation via chatgpt. *ACM Transactions on Software Engineering and Methodology*, 1–38.

Du; Yali; Sun; Hui; Li; and Ming. 2025. Post-Incorporating Code Structural Knowledge into LLMs via In-Context Learning for Code Translation. arXiv:2503.22776.

Gong, L.; Elhoushi, M.; and Cheung, A. 2024. AST-T5: Structure-Aware Pretraining for Code Generation and Understanding. In *Proc. of ICML*.

Guo; Daya; Zhu; Qihao; Yang; Dejian; Xie; Zhenda; Dong; Kai; Zhang; Wentao; Chen; Guanting; Bi; Xiao; Wu; Yu; Li; YK; et al. 2024. DeepSeek-Coder: When the Large Language Model Meets Programming–The Rise of Code Intelligence. arXiv:2401.14196.

Huang; Baizhou; Lu; Shuai; Chen; Weizhu; Wan; Xiaojun; Duan; and Nan. 2023. Enhancing Large Language Models in Coding through Multi-Perspective Self-Consistency. arXiv:2309.17272.

Hui; Binyuan; Yang; Jian; Cui; Zeyu; Yang; Jiaxi; Liu; Dayiheng; Zhang; Lei; Liu; Tianyu; Zhang; Jiajun; Yu; Bowen; Lu; Keming; et al. 2024. Qwen2. 5-Coder Technical Report. arXiv:2409.12186.

Inala; Priya, J.; Wang; Chenglong; Yang; Mei; Codas; Andres; Encarnación; Mark; Lahiri; Shuvendu; Musuvathi; Madanlal; Gao; and Jianfeng. 2022. Fault-aware neural code rankers. In *Proc. of NeurIPS*, 13419–13432.

Jiang; Juyong; Wang; Fan; Shen; Jiasi; Kim; Sungju; Kim; and Sunghun. 2024. A Survey on Large Language Models for Code Generation. arXiv:2406.00515.

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

Le, H.; Wang, Y.; Gotmare, A. D.; Savarese, S.; and Hoi, S. C. 2022. CodeRL: Mastering Code Generation through Pretrained Models and Deep Reinforcement Learning. In *Proc. of NeurIPS*.

Li; Haau-Sing; Fernandes; Patrick; Gurevych; Iryna; Martins; and T., A. F. 2024. DOCE: Finding the Sweet Spot for Execution-Based Code Generation. arXiv:2408.13745.

Li; Xin-Ye; Du; Ya-Li; Li; and Ming. 2025. Enhancing LLMs in Long Code Translation through Instrumentation and Program State Alignment. arXiv:2504.02017.

Li; Yujia; Choi; David; Chung; Junyoung; Kushman; Nate; Schrittwieser; Julian; Leblond; Rémi; Eccles; Tom; Keeling; James; Gimeno; Felix; Lago, D.; Agustin; et al. 2022. Competition-Level Code Generation with Alphacode. *Science*, 1092–1097.

Liu; Jiawei; Nguyen; Thanh; Shang; Mingyue; Ding; Hantian; Li; Xiaopeng; Yu; Yu; Kumar; Varun; Wang; and Zijian. 2024. Learning Code Preference Via Synthetic Evolution. arXiv:2410.03837.

Liu, J.; Xia, C. S.; Wang, Y.; and Zhang, L. 2023. Is Your Code Generated by ChatGPT Really Correct? Rigorous Evaluation of Large Language Models for Code Generation. In *Proc. of NeurIPS*.

Liu, R.-B.; Li, A.; Yang, C.; Sun, H.; and Li, M. 2025. Revisiting Chain-of-Thought in Code Generation: Do Language Models Need to Learn Reasoning before Coding? In *Proc. of ICML*, volume 267 of *Proceedings of Machine Learning Research*, 38809–38826. PMLR.

Luo, Z.; Xu, C.; Zhao, P.; Sun, Q.; Geng, X.; Hu, W.; Tao, C.; Ma, J.; Lin, Q.; and Jiang, D. 2024. WizardCoder: Empowering Code Large Language Models with Evol-Instruct. In *Proc. of ICLR*.

Lyu; Zhicun; Li; Xinye; Xie; Zheng; Li; and Ming. 2025. Top Pass: Improve Code Generation by Pass@ K-Maximized Code Ranking. *Frontiers of Computer Science*, 198341.

Ni, A.; Iyer, S.; Radev, D.; Stoyanov, V.; Yih, W.; Wang, S. I.; and Lin, X. V. 2023. LEVER: Learning to Verify Language-to-Code Generation with Execution. In *Proc. of ICML*, 26106–26128.

Nijkamp, E.; Pang, B.; Hayashi, H.; Tu, L.; Wang, H.; Zhou, Y.; Savarese, S.; and Xiong, C. 2023. CodeGen: An Open Large Language Model for Code with Multi-Turn Program Synthesis. In *Proc. of ICLR*.

Roziere; Baptiste; Gehring; Jonas; Gloeckle; Fabian; Sootla; Sten; Gat; Itai; Tan; Ellen, X.; Adi; Yossi; Liu; Jingyu; Sauvestre; Romain; Remez; Tal; et al. 2023. Code Llama: Open Foundation Models for Code. arXiv:2308.12950.

Rozière, B.; Zhang, J.; Charton, F.; Harman, M.; Synnaeve, G.; and Lample, G. 2022. Leveraging Automated Unit Tests for Unsupervised Code Translation. In *Proc. of ICLR*.

Shi; Freda; Fried; Daniel; Ghazvininejad; Marjan; Zettlemoyer; Luke; Wang; and I., S. 2022. Natural Language to Code Translation with Execution. In *Proc. of EMNLP*, 3533–3546.

Sun; Zhihong; Wan; Yao; Li; Jia; Zhang; Hongyu; Jin; Zhi; Li; Ge; Lyu; and Chen. 2024. Sifting through the Chaff: On Utilizing Execution Feedback for Ranking the Generated Code Candidates. In *Proc. of ASE*, 229–241.

To; Quoc, H.; Nguyen; Huynh, M.; Bui; and Q., N. D. 2023. Functional Overlap Reranking for Neural Code Generation. arXiv:2311.03366.

Wang; Xuezhi; Wei; Jason; Schuurmans; Dale; Le; Quoc; Chi; Ed; Narang; Sharan; Chowdhery; Aakanksha; Zhou; and Denny. 2022. Self-consistency improves chain of thought reasoning in language models. arXiv:2203.11171.

Zhang; Tianyi; Yu; Tao; Hashimoto; Tatsunori; Lewis; Mike; Yih; Wen-tau; Fried; Daniel; Wang; and Sida. 2023a. Coder reviewer reranking for code generation. In *Proc. of ICML*, 41832–41846.

Zhang, K.; Wang, D.; Xia, J.; Wang, W. Y.; and Li, L. 2023b. ALGO: Synthesizing Algorithmic Programs with Generated Oracle Verifiers. In *Proc. of NeurIPS*.

Zhao; Xin, W.; Zhou; Kun; Li; Junyi; Tang; Tianyi; Wang; Xiaolei; Hou; Yupeng; Min; Yingqian; Zhang; Beichen; Zhang; Junjie; Dong; Zican; et al. 2023. A survey of large language models. arXiv:2302.06318.

Zheng, Q.; Xia, X.; Zou, X.; Dong, Y.; Wang, S.; Xue, Y.; Shen, L.; Wang, Z.; Wang, A.; Li, Y.; Su, T.; Yang, Z.; and Tang, J. 2023. CodeGeeX: A Pre-Trained Model for Code Generation with Multilingual Benchmarking on HumanEval-X. In *Proc. of KDD*, 5673–5684.

Zuo; Yuxin; Zhang; Kaiyan; Sheng; Li; Qu; Shang; Cui; Ganqu; Zhu; Xuekai; Li; Haozhan; Zhang; Yuchen; Long; Xinwei; Hua; Ermo; et al. 2025. Ttrl: Test-time reinforcement learning. arXiv:2504.16084.