

ConFreeze-Code: Validation-Consensus Freezing for Cost-Effective Multi-Model Code Generation

Tianqi Hao¹, Xiaoting Du¹, Chenglong Li², and Yi Wang¹

¹ Beijing University of Posts and Telecommunications, Beijing, China
2024110779@bupt.cn, duxiaoting@bupt.edu.cn, oliver.wangyi@gmail.com

² Beijing University of Technology, Beijing, China
lichenglong@bjut.edu.cn

³ * Corresponding author: Xiaoting Du (duxiaoting@bupt.edu.cn)

Abstract. Large language models are increasingly used for code generation, but improving reliability through multi-model refinement is not free. A second peer-aware round — where models revise programs after viewing peer code and test outcomes — can repair some incorrect programs, but it also adds substantial token cost and may revise initially correct programs into failing ones. This paper studies when such refinement should be invoked in execution-based code generation. We propose *ConFreeze-Code*, a training-free validation-consensus freezing mechanism. Three code LLMs first generate candidate programs independently and execute them on available base tests. If all candidates pass, *ConFreeze-Code* freezes the task and skips the second round; otherwise, the task enters peer-aware refinement. Each policy submits one selected program per task, and final correctness is evaluated using stronger base-plus tests. On HumanEval+ and MBPP+, matched-trajectory simulation shows that *ConFreeze-Code* achieves comparable or higher selected pass@1 than initial-only decoding and full refinement, while reducing token usage by 45.22% and 46.40% relative to full refinement. Further analysis shows that validation consensus identifies tasks with limited repair upside but non-negligible harmful-revision risk. These results suggest that multi-model code refinement should be invoked selectively rather than applied by default.

Keywords: large language models; code generation; multi-model refinement; dependable intelligent systems; token efficiency

1 Introduction

Large language models (LLMs) are increasingly used for software engineering tasks such as code generation, debugging, and program repair. However, generated code remains difficult to trust: a syntactically plausible program may still fail hidden tests, edge cases, or stronger validation suites. Execution-based benchmarks therefore evaluate generated programs by running unit tests rather

than comparing surface forms [1, 3]. To improve reliability, recent systems often use multiple models, multiple samples, or iterative feedback, where candidate programs are compared, executed, or revised in a later round [2, 12, 14].

Despite their potential, indiscriminate multi-model refinement faces two practical challenges. The first is cost: a second refinement round requires additional model calls and substantially increases token consumption, especially when several models participate. The second is non-monotonic correctness: refinement is not guaranteed to improve code. A later round may revise a first-round program that already satisfies the available validation tests into one that fails stronger tests. We refer to this failure mode as a *harmful revision*. This risk is especially relevant because execution feedback is partial. Passing base tests does not guarantee full functional correctness, while failing tests may provide only incomplete evidence about the underlying defect. Prior work on code self-repair has also found that repair gains can be modest, inconsistent, or even negative once cost is considered [13]. These challenges motivate the central question of this paper: *when should multi-model refinement be invoked, and when should it be skipped?*

Our key observation is that code generation provides a useful validation signal before refinement. When all first-round candidates independently pass the base tests, the task exhibits *validation consensus*. This does not prove correctness under stronger tests, but it suggests that another peer-aware round may have limited repair upside while still exposing the selected program to harmful revision. In contrast, when at least one candidate fails the base tests, the initial round reveals implementation uncertainty or incomplete reliability, and further refinement may still be worth the additional cost.

Based on this observation, we propose *ConFreeze-Code*, a lightweight validation-consensus freezing mechanism for multi-model code generation. *ConFreeze-Code* adapts consensus freezing to an execution-based setting by replacing label unanimity with executable validation. By default, it uses a strict validation-consensus gate: an item is frozen only when all three first-round candidates pass the base tests. Otherwise, the models enter a peer-aware refinement round in which they observe peer code and execution status. For each task, the system submits one selected program, and final evaluation uses stronger base-plus tests. The mechanism is training-free, does not require an additional judge model, and can be wrapped around existing multi-model refinement pipelines.

This design directly addresses the two challenges above. By freezing validation-consensus items, *ConFreeze-Code* skips unnecessary refinement calls and reduces token cost. At the same time, it avoids revising programs that already satisfy the available validation evidence, thereby reducing exposure to harmful revisions. Non-consensus items are still refined, preserving the opportunity to repair programs whose first-round candidates reveal failures or disagreement. In this way, *ConFreeze-Code* treats multi-model code refinement as a selective, validation-guided decision rather than an always-on pipeline.

We organize our empirical study around three research questions. **RQ1** asks whether *ConFreeze-Code* can improve selected-code correctness while reducing refinement cost. **RQ2** examines whether validation consensus identifies where

refinement is useful. **RQ3** studies the sensitivity of *ConFreeze-Code* to gate threshold and model-team composition. Together, these questions evaluate not only final correctness, but also whether refinement budget is spent on tasks that can plausibly benefit from it.

This paper makes three contributions. First, we formulate multi-model code refinement as a budget-aware dependability problem and provide a stability-oriented evaluation protocol for beneficial revision, harmful revision, and false freeze. Second, we introduce *ConFreeze-Code*, a training-free strict validation-consensus gate that skips peer refinement only when all first-round candidates pass the available base tests. Third, through empirical evaluation on HumanEval+ and MBPP+, we show that *ConFreeze-Code* achieves comparable or higher selected pass@1 than initial-only decoding and full refinement while substantially reducing token consumption and harmful revisions. Our analysis further shows that validation consensus identifies a region with little repair upside, and that the preferred freezing threshold can vary with model-team reliability.

2 Related Work

2.1 Code Generation and Execution-Based Evaluation

Execution-based code-generation benchmarks evaluate whether generated programs satisfy functional tests rather than only matching reference text. HumanEval introduced hand-written Python programming problems and popularized pass@k as a functional-correctness metric for code-generating LLMs [3]. MBPP provides a complementary set of short Python tasks derived from natural-language problem descriptions and accompanied by automated tests [1]. These benchmarks are useful for evaluating program synthesis, but their original test suites can be incomplete. EvalPlus addresses this issue by augmenting HumanEval and MBPP with stronger automatically generated test inputs, producing HumanEval+ and MBPP+ and showing that weak tests can overestimate model correctness [11]. Our study relies on this distinction: base tests are treated as deployable validation signals for freezing, while base-plus tests are used for final evaluation.

Several lines of work use execution or feedback signals to improve generated code. CodeT generates tests and uses execution agreement between generated code and generated tests to select candidate programs [2]. Reflexion uses verbal feedback from task outcomes to improve later attempts by language agents, including programming tasks [14]. Self-Refine similarly studies iterative feedback and refinement without model training [12]. However, refinement is not guaranteed to be monotonic. Olausson et al. study self-repair for code generation and find that gains are often modest, vary substantially across subsets, and are sometimes absent when the cost of repair is considered [13]. This finding motivates our stability metrics: a refinement policy should be judged not only by whether it sometimes repairs code, but also by whether it introduces harmful revisions and whether its additional token cost is justified.

2.2 Multi-Agent Debate and Multi-Model Refinement

Multi-agent debate uses multiple LLM instances to propose, critique, and revise answers over multiple rounds. Du et al. show that multiagent debate can improve factuality and reasoning by allowing models to exchange arguments before converging on a final response [4]. Liang et al. propose a judge-managed debate framework to encourage divergent thinking and address limitations of self-reflection [10]. More generally, multi-model sampling and voting can improve performance by increasing candidate diversity [8]. However, later studies caution that multi-agent debate does not uniformly outperform simpler prompting strategies and can incur nontrivial cost and instability [15]. These observations align with our findings in code generation: full peer refinement is unevenly useful and can introduce harmful revisions.

There are also multi-agent systems designed specifically for code generation. MapCoder uses multiple specialized agents to emulate a human programming cycle of retrieval, planning, code generation, and debugging [6]. CODESIM extends this line by using a multi-agent framework for simulation-driven planning, coding, and debugging, reporting strong pass@1 results across several code-generation benchmarks [7]. These systems show that multi-agent collaboration is a promising design pattern for code generation. Our goal is complementary: rather than building a richer coding agent or a new debugging pipeline, we study when a lightweight multi-model refinement round should be skipped under executable validation feedback. To our knowledge, validation-consensus freezing has not been systematically studied as a gate for multi-model code refinement.

2.3 Efficient and Selective Test-Time Computation

A broader line of work studies adaptive test-time computation, including self-consistency, early-stopping self-consistency, difficulty-adaptive self-consistency, and selective multi-agent deliberation [5, 9, 16–19]. These methods share the idea that extra inference should be allocated selectively, but they usually rely on answer distributions, learned selectors, confidence signals, or communication sparsification. *ConFreeze-Code* differs in both signal and objective: it uses external execution results from multiple first-round code candidates and asks when peer-aware refinement may be unnecessary or harmful. This makes validation-consensus freezing a code-specific gate rather than a general confidence- or convergence-based early-stopping rule.

Recent work also studies selective or efficient deliberation in multi-agent settings. S²-MAD reduces token usage inside multi-agent debate by sparsifying communication and filtering redundant exchanges among agents [19]. iMAD learns a debate-decision classifier from structured self-critique features and triggers debate only when it is predicted to be beneficial [5]. SELENE combines selective debate initiation with evidence-weighted self-consistency, skipping well-aligned queries and replacing a single judge with evidence-weighted aggregation [16]. These systems demonstrate that adaptive test-time compute is an important

direction, but they typically introduce learned selectors, confidence or semantic-disagreement modules, or redesigned aggregation schemes.

ConFreeze-Code is complementary to this literature but differs in both signal and objective. Unlike general selective test-time computation, which usually asks when additional inference is likely to improve an answer, *ConFreeze-Code* asks when an additional peer-aware refinement round may be unnecessary or even harmful for code. Unlike confidence-based gates, it does not rely on self-reported model confidence or a learned difficulty predictor; the gate is defined by an external execution signal that is naturally available in code-generation workflows. Unlike conventional early stopping, which often monitors convergence along a sampling or reasoning trajectory, *ConFreeze-Code* uses cross-model validation consensus: all first-round candidates must independently pass the available base tests before refinement is skipped. This design makes the mechanism deliberately simple, but its contribution is to expose and evaluate a code-specific trade-off among executable validation, hidden-style correctness, harmful revision, and token cost.

3 Methodology

Figure 1 summarizes the workflow of *ConFreeze-Code*. Given a programming task, three code LLMs independently generate candidate programs in the first round. The system executes these candidates on the available base tests and records both the selected Round-1 program and the validation status of all candidates. If all three candidates pass the base tests, the strict validation-consensus gate freezes the item and keeps the Round-1 selection, thereby skipping the second refinement round. Otherwise, the item remains eligible for peer-aware refinement, where models revise their programs using peer code and execution status. The system finally selects one program and evaluates it using the stronger base-plus tests.

3.1 Problem Setting

For each programming task x , a team of $M = 3$ models produces first-round candidate programs $c_1^{(1)}, c_2^{(1)}, c_3^{(1)}$. Each candidate is evaluated against a set of base tests available during inference. Since the system may internally generate multiple candidates, we use a deterministic selection rule whenever one program must be submitted: candidates with better base-test execution status are preferred, ties are broken by shorter code, and remaining ties are broken by a fixed model order. We use the status order $\text{PASS} > \text{FAIL} > \text{RUNTIME} > \text{SYNTAX}$. This same rule selects the Round-1 program, the Round-2 program, and the final submitted program under each policy.

The system must finally submit exactly one candidate program c^* , which is evaluated using stronger base-plus tests. We use selected pass@1 as the primary correctness metric, computed on the selected program. A method receives credit on a task only if its selected output passes the final base-plus evaluation. We

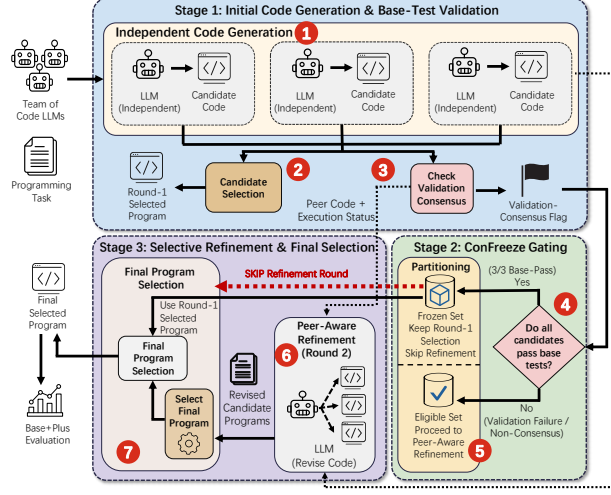


Fig. 1. Overview of *ConFreeze-Code* for validation-guided multi-model code refinement.

additionally record any-pass statistics as candidate-pool diagnostics, but they are not used as the main reported performance.

3.2 Validation-Consensus Freezing

The default *ConFreeze-Code* gate is a *strict validation-consensus gate*. Let $b_i \in \{0, 1\}$ indicate whether candidate $c_i^{(1)}$ passes the base tests. *ConFreeze-Code* freezes a task only when all first-round candidates pass the base tests:

$$\text{freeze}(x) = \mathbb{I} \left[\sum_{i=1}^3 b_i = 3 \right]. \quad (1)$$

If the item is frozen, no second-round API calls are made and the system submits the first-round selected candidate. If the item is not frozen, the task enters a peer-aware refinement round. This strict gate is deliberately training-free and requires no additional judge model or learned selector.

We also evaluate two more aggressive gate variants as ablations: freezing when at least one model passes the base tests, and freezing when at least two models pass the base tests. These variants can save more tokens, but they may also increase false-freeze risk because they freeze on weaker validation evidence.

3.3 Peer-Aware Refinement

For non-frozen items, each model receives the original task, its own previous candidate, peer candidates, and execution-status feedback from the base tests. The status feedback indicates whether each candidate is classified as PASS,

FAIL, RUNTIME, or SYNTAX under the base tests. It does not include concrete failing inputs, expected outputs, stack traces, natural-language rationales, or structured bug diagnoses. The model then produces a revised program. We keep this refinement protocol lightweight to avoid confounding the gate with a complex repair agent. The final candidate is selected using the deterministic rule defined in the problem setting. This rule ensures reproducibility and prevents random tie-breaking.

3.4 Metrics

We use task-level binary indicators over N matched tasks, where “matched” means that all compared policies are simulated on the same task set and the same Full Refinement trajectories. For task i , let $p_i^{(1)} \in \{0, 1\}$ denote whether the first-round selected program passes the final base-plus tests, and let $p_i^{(f)} \in \{0, 1\}$ denote whether the final selected program passes the same tests. Let $z_i \in \{0, 1\}$ indicate whether the policy freezes after the first round, and let $q_i^{(1)} \in \{0, 1\}$ indicate whether the first-round selected program passes the base tests. The primary reliability metric is system-level selected pass@1:

$$\text{Pass@1} = \frac{1}{N} \sum_{i=1}^N p_i^{(f)}. \quad (2)$$

Stability is measured by beneficial and harmful revisions relative to the first-round selected program:

$$B_i = \mathbb{I}[p_i^{(1)} = 0 \wedge p_i^{(f)} = 1], \quad H_i = \mathbb{I}[p_i^{(1)} = 1 \wedge p_i^{(f)} = 0]. \quad (3)$$

We report $\frac{1}{N} \sum_i B_i$ and $\frac{1}{N} \sum_i H_i$ unless a table explicitly states a conditional denominator, such as among debated or would-freeze items.

Gate reliability is measured by false-freeze rate. A false freeze occurs when an item is frozen after passing the available base validation but its first-round selected program fails the stronger base-plus evaluation:

$$\text{FalseFreeze} = \frac{\sum_{i=1}^N z_i \mathbb{I}[q_i^{(1)} = 1 \wedge p_i^{(1)} = 0]}{\sum_{i=1}^N z_i}. \quad (4)$$

For the default strict validation-consensus gate, $q_i^{(1)} = 1$ holds for frozen items by construction; the metric therefore captures the gap between base-test validation and stronger final testing.

Cost is measured by average tokens per task and second-round coverage $\frac{1}{N} \sum_i (1 - z_i)$. For a policy π , token saving is computed against Full Refinement on the same matched tasks:

$$\text{TokenSaving}(\pi) = 1 - \frac{\sum_{i=1}^N T_i^\pi}{\sum_{i=1}^N T_i^{\text{Full}}}, \quad (5)$$

where T_i^π is the total prompt and completion tokens used by policy π on task i .

4 Experimental Setup

4.1 Benchmarks

We evaluate on two execution-based code-generation benchmarks. HumanEval contains 164 hand-written Python programming tasks and is widely used to evaluate functional correctness of generated code [3]. MBPP contains short Python programming problems derived from natural-language descriptions and accompanied by tests [1]. We use the EvalPlus versions of both benchmarks, HumanEval+ and MBPP+, which augment the original tests with stronger test inputs and expose failures missed by the base suites [11]. In our protocol, base tests are available during inference and serve as the validation signal for freezing, while the stronger base-plus tests are used only for final evaluation. This separation prevents the final evaluation suite from becoming a freezing oracle.

4.2 Model Team and Decoding

Our main experiments use a heterogeneous three-model team from different providers: OpenAI’s gpt-4.1-mini (April 2025), Anthropic’s claude-haiku-4-5-20251001 (October 2025), and DeepSeek’s deepseek-v4-flash (April 2026). We choose lightweight or efficient models because the paper studies cost-aware multi-model refinement rather than maximum single-model coding performance. All model calls use temperature 0.7, the same output parser, the same retry logic, and the same deterministic selection rule across methods.

4.3 Compared Policies

We compare three policy families. *Initial Round* stops after the first round and submits the selected first-round program; it is the low-cost baseline. *Full Refinement* invokes the second round for every task and is the high-cost baseline. *ConFreeze-Code* is our default selective policy: it uses the strict validation-consensus gate, freezing only when all three first-round candidates pass the base tests and refining the remaining items. The at-least-one and at-least-two base-pass variants are not separate baselines; we analyze them in RQ2 as more aggressive gate settings.

Refinement prompt. We use a lightweight peer-aware refinement prompt in the second round. Each model receives its own first-round program, peer programs, and the base-test execution status of these programs. We intentionally do not provide detailed failure traces, natural-language rationales, or additional repair-specific agents. This design keeps the protocol close to a deployable multi-model refinement setting and helps isolate the effect of selective invocation rather than combining it with stronger debugging modules.

4.4 Evaluation Protocol

To avoid confounding policy effects with sampling variance, our main comparison uses matched-trajectory offline simulation from the Full Refinement logs. For each task, the same first-round and second-round candidates are reused to evaluate *Initial Round*, *Full Refinement*, and *ConFreeze-Code*. This isolates the policy choice from sampling variation across separate API runs. Token saving is computed against *Full Refinement* on the same matched task set, and each method is evaluated by the selected program’s base-plus correctness.

5 Results

We structure the results around three questions. RQ1 compares the three main policies and asks whether *ConFreeze-Code* improves the balance between final correctness, harmful revision, and token cost. RQ2 analyzes the validation-consensus signal itself, including both its ability to separate useful from low-value refinement regions and its false-freeze boundary. RQ3 examines whether this policy behavior is stable across gate thresholds and model-team compositions, and whether the preferred freezing threshold changes when the peer team becomes less reliable.

5.1 RQ1: Correctness and Cost under Selective Refinement

Table 1 reports the main comparison under matched-trajectory simulation. All policies reuse the same *Full Refinement* trajectories, so differences come from the policy choice rather than separate API sampling. *Full Refinement* is costly but does not reliably improve selected pass@1: it matches *Initial Round* on HumanEval+ and slightly underperforms it on MBPP+. This reflects the non-monotonicity of peer-aware refinement, where beneficial fixes and harmful revisions can offset each other.

ConFreeze-Code provides a better operating point. It obtains the highest point estimate of selected pass@1 among the compared policies on both benchmarks: 0.9390 on HumanEval+ and 0.8386 on MBPP+. These correctness gains should be interpreted together with paired uncertainty estimates, because the absolute pass@1 differences are small on HumanEval+. The main advantage of *ConFreeze-Code* is therefore not only the point-estimate improvement, but the fact that it maintains comparable or higher selected pass@1 while reducing second-round calls to 20.73% on HumanEval+ and 14.02% on MBPP+, saving 45.22% and 46.40% of tokens relative to *Full Refinement*. It also keeps most of the useful second-round fixes while reducing harmful revisions: from 0.0122 to 0.0061 on HumanEval+ and from 0.0159 to 0.0000 on MBPP+. Table 2 shows that *ConFreeze-Code* maintains comparable selected pass@1 on HumanEval+ and yields a statistically significant improvement over full refinement on MBPP+, while consistently reducing token consumption and harmful revisions.

Table 1. Main results on HumanEval+ and MBPP+ under matched-trajectory offline simulation.

Dataset	Method	N	Pass@1	Beneficial	Harmful	R2 call rate	Avg tokens	Token saving
HumanEval+	Initial	164	0.9329	0.0000	0.0000	0.0000	1333.0	63.66%
	Full Refinement	164	0.9329	0.0122	0.0122	1.0000	3668.5	0.00%
	<i>ConFreeze-Code</i>	164	0.9390	0.0122	0.0061	0.2073	2009.4	45.22%
MBPP+	Initial	378	0.8254	0.0000	0.0000	0.0000	1213.4	62.49%
	Full Refinement	378	0.8228	0.0132	0.0159	1.0000	3235.3	0.00%
	<i>ConFreeze-Code</i>	378	0.8386	0.0132	0.0000	0.1402	1734.0	46.40%

Table 2. Paired statistical tests on matched replay trajectories.

Dataset	Comparison	95% paired bootstrap CI	McNemar p
HumanEval+	Full – Initial	[-0.0305, 0.0305]	1.0000
HumanEval+	<i>ConFreeze-Code</i> – Initial	[-0.0183, 0.0183]	1.0000
HumanEval+	<i>ConFreeze-Code</i> – Full	[-0.0244, 0.0244]	1.0000
MBPP+	Full – Initial	[-0.0212, 0.0132]	1.0000
MBPP+	<i>ConFreeze-Code</i> – Initial	[0.0026, 0.0265]	0.0625
MBPP+	<i>ConFreeze-Code</i> – Full	[0.0053, 0.0291]	0.0313

5.2 RQ2: Validation Consensus as a Refinement Signal

We next examine whether the gate signal itself is meaningful. Table 3 varies the freeze threshold from aggressive to conservative. The at-least-one and at-least-two gates freeze on weaker evidence and therefore save more second-round calls, while the default strict gate freezes only when all three candidates pass. R2 coverage measures the fraction of tasks still refined, and false-freeze rate is conditional on the would-freeze subset.

The threshold defines a cost–risk frontier. Aggressive gates skip more second-round calls, but they freeze on weaker evidence and therefore depend more heavily on the completeness of base tests. The strict consensus gate spends more tokens than the aggressive variants, but it uses a stronger validation signal and yields the lowest false-freeze rate on HumanEval+. MBPP+ shows a larger gap between base-test success and final base-plus correctness than HumanEval+. This does not invalidate the use of validation consensus as a trigger for refinement decisions; rather, it highlights that the safety of freezing depends on the strength of the available tests. We use the strict validation-consensus gate as the default because it is the most conservative rule in this family: it requires agreement from all three candidates before skipping refinement, while still reducing a large fraction of second-round calls.

To test whether validation consensus actually separates low-value refinement cases from higher-value ones, we perform a counterfactual analysis on the full-refinement trajectories. For each gate, Table 4 separates items into two counterfactual groups under each gate: items that would be frozen after Round 1 (“Frozen” in the table), and items that would still enter the second refinement round (“Debated” in the table). For both groups, we use the *Full Refinement* trajectory to measure whether continuing to Round 2 would produce a benefi-

Table 3. Gate aggressiveness ablation.

Dataset	Freeze criterion	Would-freeze rate	R2 Coverage	Policy Harmful	Policy Beneficial	False Freeze
HumanEval+	≥ 1 base-pass	0.9878	0.0122	0.0000	0.0061	0.0432
	≥ 2 base-pass	0.9207	0.0793	0.0061	0.0061	0.0331
	strict consensus (3/3)	0.7927	0.2073	0.0061	0.0122	0.0154
MBPP+	≥ 1 base-pass	0.9524	0.0476	0.0000	0.0106	0.1306
	≥ 2 base-pass	0.9180	0.0820	0.0000	0.0106	0.1326
	strict consensus (3/3)	0.8598	0.1402	0.0000	0.0132	0.1292

Table 4. Counterfactual validation-consensus analysis under Full Refinement trajectories.

Dataset	Gate	Frozen N	Frozen repair	Frozen damage	Debated N	Debated repair	Debated damage
HumanEval+	≥ 1 base-pass	162	1/162	2/162	2	1/2	0/2
	≥ 2 base-pass	151	1/151	1/151	13	1/13	1/13
	strict consensus (3/3)	130	0/130	1/130	34	2/34	1/34
MBPP+	≥ 1 base-pass	360	1/360	6/360	18	4/18	0/18
	≥ 2 base-pass	347	1/347	6/347	31	4/31	0/31
	strict consensus (3/3)	325	0/325	6/325	53	5/53	0/53

cial or harmful revision. Here, *repair* means that the selected program changes from failing to passing the final base-plus tests ($p_i^{(1)} = 0, p_i^{(2)} = 1$), while *damage* means that it changes from passing to failing ($p_i^{(1)} = 1, p_i^{(2)} = 0$). This counterfactual repair/damage analysis is distinct from false-freeze. A false-freeze occurs when an item is frozen and its first-round selected program fails the final tests; this rate is reported separately in Table 3.

The counterfactual results support the gate interpretation. Under the default strict validation-consensus gate, frozen items almost never benefit from continuing to Full Refinement: 0/130 on HumanEval+ and 0/325 on MBPP+. Yet they still contain harmful revisions if forced to continue. By contrast, the would-debate subset contains the remaining repair opportunity: 2/34 beneficial revisions on HumanEval+ and 5/53 on MBPP+. This suggests that the gate is not only saving tokens. It skips a group of tasks where another round rarely helps, while keeping the tasks with more observed repair potential eligible for refinement.

False freezes describe a different failure mode from harmful revision. Under the strict gate, false freezes occur in 2 of 130 frozen HumanEval+ tasks and 42 of 325 frozen MBPP+ tasks. These cases are the code-generation analogue of wrong consensus: all first-round candidates pass the available base tests, yet the selected program still fails the stronger base-plus evaluation. This distinction is important. A harmful revision is caused by the refinement process changing an initially correct selected program into an incorrect one; a false freeze is caused by incomplete validation evidence before refinement is invoked. Therefore, false freezes should not be interpreted as failures introduced by *ConFreeze-Code*, but as a boundary of any policy that relies on base tests as the deployable validation signal.

The difference between HumanEval+ and MBPP+ further shows that the reliability of validation consensus depends on test-suite strength. On HumanEval+, strict validation consensus is rarely false, suggesting that base-test agreement is

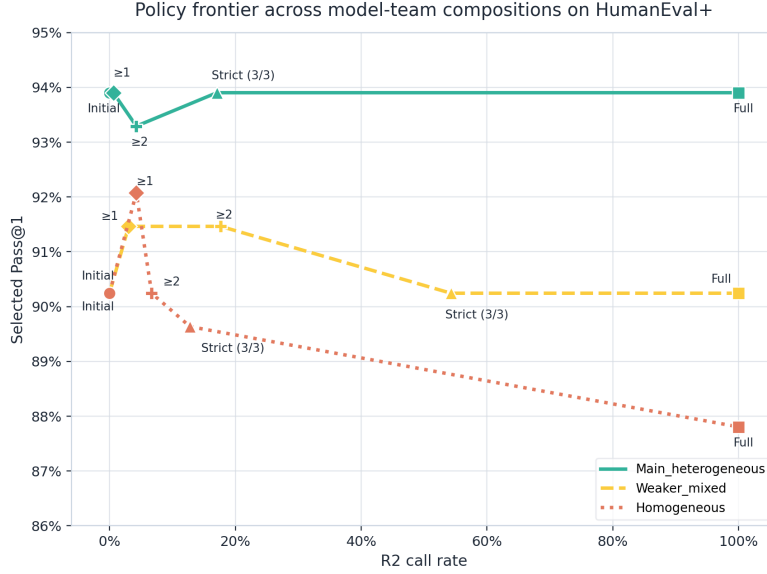


Fig. 2. Policy frontier on HumanEval+ under different model-team compositions.

a strong enough signal for many tasks. On MBPP+, the higher false-freeze rate indicates a larger gap between base tests and base-plus tests. In our matched Full Refinement trajectories, however, always applying the same lightweight refinement round does not reliably repair these false-freeze cases. This suggests that reducing false freezes is more likely to require stronger validation signals, such as generated tests, higher-coverage test suites, execution traces, or structured failure localization, rather than simply forcing peer-aware refinement on every task.

5.3 RQ3: Sensitivity to Gate Threshold and Model-Team Composition

RQ2 shows that validation consensus is a useful signal, but it does not imply that a single threshold is optimal in every deployment setting. The value of a second refinement round depends on the reliability and diversity of the peer models. To examine this sensitivity, we compare policy frontiers on HumanEval+ under three model-team compositions: the main heterogeneous team, a weaker mixed team, and a homogeneous GPT-4.1-mini team. The weaker mixed team uses GPT-4o-mini, Llama-3.1-8B-Instant, and DeepSeek-Chat.

Figure 2 plots the fraction of tasks entering the second refinement round against system-level selected pass@1. Each line connects operating points from the same model team, including initial-only decoding, more aggressive validation gates, the strict validation-consensus gate, and full refinement. The lines should

be read as policy frontiers rather than temporal trends: moving right increases the use of the second round, while moving left freezes more tasks earlier.

The same gate threshold does not behave identically across model teams. For the main heterogeneous team, the strict validation-consensus gate is a strong operating point: it improves selected pass@1 relative to both *Initial Round* and *Full Refinement* while keeping the R2 call rate at 0.2073. For the weaker mixed team, *Full Refinement* does not improve over *Initial Round*, and the strict-gate version of *ConFreeze-Code* mainly acts as a cost-control policy, reducing the R2 call rate to 0.5427 and saving 23.68% tokens. In the same frontier, the more aggressive gates move closer to the low-cost region and can match or exceed the strict gate. This suggests that when the second round provides limited net benefit, a more aggressive freezing threshold may become competitive.

As a stress-test case, the homogeneous GPT-4.1-mini team shows a clearer failure mode of always-on refinement. This suggests that repeated refinement among highly similar peers may amplify harmful revisions rather than add useful diversity.

Overall, these results suggest three deployment implications. First, if the only objective is minimum cost, *Initial Round* remains the strongest baseline and should not be ignored. Second, if a system can spend extra budget on uncertain cases, *ConFreeze-Code* provides a simple middle ground: it skips validation-consensus items while preserving refinement for failures or disagreements. Third, the gate threshold should be treated as a policy knob. Aggressive gates save more tokens but rely more heavily on incomplete base tests; conservative gates spend more tokens but reduce false-freeze risk. Choosing among these operating points is central to dependable use of multi-model code refinement.

6 Limitations

Offline simulation. Our main comparison uses matched Full Refinement trajectories to isolate policy effects from sampling variance. This makes the comparison fair across policies, but it does not capture all variability that may arise from independently re-running each policy online.

Validation signal. *ConFreeze-Code* uses base tests as the deployable validation signal and evaluates final correctness with stronger base-plus tests. False freezes occur when base tests are insufficient to reveal errors, so the safety of freezing depends on the strength and coverage of the available tests. Our results show that this boundary is benchmark-dependent: strict validation consensus is much more reliable on HumanEval+ than on MBPP+. Future versions of the mechanism could reduce false freezes by combining validation consensus with stronger generated tests, execution traces, or failure-localization feedback, but such enhancements are orthogonal to the lightweight freezing policy studied here.

Refinement protocol. We evaluate a lightweight peer-aware refinement protocol that shares peer code and execution status. Richer protocols with detailed failure traces, structured bug diagnosis, or specialized repair agents may improve second-round repair capability. Our claims therefore concern when to

invoke this lightweight refinement protocol, not the upper bound of multi-model code repair.

Scope. We evaluate two Python code-generation benchmarks and a limited set of model teams. The results suggest that validation-consensus freezing is useful for controlling unnecessary refinement, but the best freezing threshold may vary with model reliability, task difficulty, and available tests.

7 Conclusion

We presented *ConFreeze-Code*, a validation-consensus freezing mechanism for cost-effective multi-model code generation. Instead of applying peer-aware refinement to every task, *ConFreeze-Code* freezes tasks whose first-round candidates all pass the available base tests and reserves the second round for non-consensus cases. Under matched-trajectory simulation on HumanEval+ and MBPP+, *ConFreeze-Code* achieves the best selected pass@1 among the compared policies while reducing token consumption by 45.22% and 46.40%, respectively, relative to full refinement. Our analysis shows that validation-consensus items contain few useful second-round repairs, whereas non-consensus items preserve most of the observed repair opportunities. False freezes highlight an important boundary of the method: passing the available base tests does not always imply passing stronger final tests, and always-on refinement does not reliably repair these cases in our experiments. The policy-frontier analysis further shows that the preferred freezing threshold can vary with model-team reliability. Overall, our results suggest that multi-model code refinement should be treated as a selective, validation-guided decision rather than an always-on pipeline.

References

1. Austin, J., Odena, A., Nye, M., Bosma, M., Michalewski, H., Dohan, D., Jiang, E., Cai, C., Terry, M., Le, Q., Sutton, C.: Program synthesis with large language models. arXiv preprint arXiv:2108.07732 (2021)
2. Chen, B., Zhang, F., Nguyen, A., Zan, D., Lin, Z., Lou, J.G., Chen, W.: Codet: Code generation with generated tests. In: International Conference on Learning Representations (2023)
3. Chen, M., Tworek, J., Jun, H., Yuan, Q., Pinto, H.P.d.O., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., et al.: Evaluating large language models trained on code. arXiv preprint arXiv:2107.03374 (2021)
4. Du, Y., Li, S., Torralba, A., Tenenbaum, J.B., Mordatch, I.: Improving factuality and reasoning in language models through multiagent debate. In: International Conference on Machine Learning (2024)
5. Fan, W., et al.: imad: Intelligent multi-agent debate for efficient and accurate llm inference. In: Proceedings of the AAAI Conference on Artificial Intelligence (2026)
6. Islam, M.A., Ali, M.E., Parvez, M.R.: Mapcoder: Multi-agent code generation for competitive problem solving. arXiv preprint arXiv:2405.11403 (2024), <https://arxiv.org/abs/2405.11403>

7. Islam, M.A., Ali, M.E., Parvez, M.R.: CodeSim: Multi-agent code generation and problem solving through simulation-driven planning and debugging. In: Findings of the Association for Computational Linguistics: NAACL 2025. pp. 5128–5154. Association for Computational Linguistics, Albuquerque, New Mexico (Apr 2025). <https://doi.org/10.18653/v1/2025.findings-naacl.285>, <https://aclanthology.org/2025.findings-naacl.285/>
8. Li, J., Zhang, Q., Yu, Y., Fu, Q., Ye, D.: More agents is all you need. arXiv preprint arXiv:2402.05120 (2024)
9. Li, Y., Yuan, P., Feng, S., Pan, B., Wang, X., Sun, B., Wang, H., Li, K.: Escape sky-high cost: Early-stopping self-consistency for multi-step reasoning. In: International Conference on Learning Representations (2024), <https://arxiv.org/abs/2401.10480>
10. Liang, T., He, Z., Jiao, W., Wang, X., Wang, Y., Wang, R., Yang, Y., Shi, S., Tu, Z.: Encouraging divergent thinking in large language models through multi-agent debate. In: Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing (2024)
11. Liu, J., Xia, C.S., Wang, Y., Zhang, L.: Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. In: Advances in Neural Information Processing Systems (2023)
12. Madaan, A., Tandon, N., Gupta, P., Hallinan, S., Gao, L., Wiegrefe, S., Alon, U., Dziri, N., Prabhume, S., Yang, Y., et al.: Self-refine: Iterative refinement with self-feedback. In: Advances in Neural Information Processing Systems (2023)
13. Olausson, T.X., Inala, J.P., Wang, C., Gao, J., Solar-Lezama, A.: Is self-repair a silver bullet for code generation? In: The Twelfth International Conference on Learning Representations (ICLR) (2024), <https://openreview.net/forum?id=y0GJXRungR>
14. Shinn, N., Cassano, F., Gopinath, A., Narasimhan, K., Yao, S.: Reflexion: Language agents with verbal reinforcement learning. In: Advances in Neural Information Processing Systems (2023)
15. Smit, A., Raileanu, R., Hupkes, D., Bisk, Y.: Should we be going mad? a look at multi-agent debate strategies for llms. In: International Conference on Machine Learning (2024)
16. Verma, A., Gupta, S., Sircar, P., Pillai, S.: Selene: Selective and evidence-weighted llm debating for efficient and reliable reasoning. In: Proceedings of the 19th Conference of the European Chapter of the Association for Computational Linguistics: Industry Track (2026)
17. Wang, X., Feng, S., Li, Y., Yuan, P., Zhang, Y., Tan, C., Pan, B., Hu, Y., Li, K.: Make every penny count: Difficulty-adaptive self-consistency for cost-efficient reasoning. arXiv preprint arXiv:2408.13457 (2024), <https://arxiv.org/abs/2408.13457>
18. Wang, X., Wei, J., Schuurmans, D., Le, Q., Chi, E.H., Narang, S., Chowdhery, A., Zhou, D.: Self-consistency improves chain of thought reasoning in language models. In: International Conference on Learning Representations (2023), <https://arxiv.org/abs/2203.11171>
19. Zeng, Y., Huang, W., et al.: S²-mad: Breaking the token barrier to enhance multi-agent debate efficiency. In: Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (2025)