DeepDebateCoder: Meta-Policy Modeling and Few-Shot Anchored Asymmetric Debate for Reliable Code Generation

1st Yunpeng Wang
Taiyuan University of Technology
Taiyuan, China
wyp_nanxun@163.com

2nd Yangzhi Han
Taiyuan University of Technology
Taiyuan, China
2023005003@link.tyut.edu.cn

3rd Jiaying Gao
Taiyuan University of Technology
Taiyuan, China
2023002247@link.tyut.edu.cn

4th Wenrui Zhang
Taiyuan University of Technology
Taiyuan, China
2024006341.tyut@vip.163.com

Jianfeng Wang*
Taiyuan University of Technology
Taiyuan, China
wangjianfeng@tyut.edu.cn

Wei Zhang*
Taiyuan University of Technology
Taiyuan, China
wei.zhang.2@ki.se

***Abstract*—Large language models (LLMs) are increasingly effective at generating code, but they still struggle on specification-intensive tasks where reliability is critical. Such tasks typically come with long problem statements, layered constraints, and many corner cases, and they are often judged by hidden tests. To address this gap, we introduce DeepDebateCoder, a closed-loop framework designed to improve robustness and structural consistency in LLM-generated code. DeepDebateCoder first builds an explicit Policy Graph that captures module boundaries, branching logic, state transitions, and I/O constraints. This graph then serves as a structured guide for code synthesis and refinement. To go beyond example-level correctness, it generates policy-aligned tests that target boundary conditions and exception paths, and validates them before using them for refinement. Finally, DeepDebateCoder uses execution feedback to drive an asymmetric debate between a strategy expert and a code expert. Instead of repeatedly patching surface-level errors, the debate focuses on revising the underlying decision path. Across HumanEval and APPS, DeepDebateCoder consistently improves pass@1 and robustness over competitive baselines. The gains are especially pronounced on APPS, with the largest improvements on long-specification and high-difficulty tasks, underscoring the value of meta-policy modeling and few-shot-anchored asymmetric debate for achieving highly reliable code generation. More broadly, our framework points to a new paradigm for reliability-oriented program synthesis.**

***Index Terms*—LLMs, program synthesis, trustworthy code generation, Policy Graph modeling, execution-guided feedback**

I. Introduction

Large language models (LLMs) have improved substantially in program synthesis, with solid performance on code completion, program generation, and automated debugging. *Reliable* code generation remains challenging for specification-intensive programming tasks where problem statements are long, constraints are multi-faceted, control flow is highly branching, and numerous corner cases must be handled to pass hidden tests. In benchmarks such as HumanEval and APPS, common failure modes include inconsistent branching priorities, unstable state updates, and missing boundary checks, which often surface only under hidden tests. These characteristics reflect real-world information-system engineering workloads where robustness, maintainability, and auditability are as critical as functional correctness.

However, current LLM-based generation pipelines often break down once tasks require consistent multi-branch control flow, careful state handling, and robust boundary checks. We argue that this stems from three core limitations. First, generated code often exhibits flattened structure, inconsistent branching logic, and unstable state management [1], [2]; yet most approaches lack an analyzable *policy-level* representation to make control structures, state transitions, and I/O constraints explicit, leaving little intermediate structure to inspect or revise during generation. Second, standard benchmark tests and naive auto-test generation do not reliably push code toward robustness: boundary and exception paths are under-covered, and model-generated tests can be noisy or incorrect, thereby weakening the execution signal for iterative refinement. Third, although multi-agent and self-healing systems can improve local correctness, they typically operate at the candidate-code level and apply patch-style fixes, lacking mechanisms to reconstruct *global* decision paths and branching structures in a verifiable way.

In this work, we present **DeepDebateCoder** (abbreviated as **DDCoder**), a closed-loop framework for robust code generation under long specifications and hidden-test evaluation. Our key contributions are threefold: (i) we introduce a Policy Graph that makes global decision structure, branching priorities, and I/O constraints explicit and auditable for specification-intensive tasks; (ii) we propose policy-aligned test augmentation with secondary validation to filter ill-formed or self-contradictory cases, providing a cleaner execution sig-

* Co-corresponding authors.

nal for robustness-oriented refinement; and (iii) we develop an execution-guided asymmetric debate loop anchored by a small set of high-quality few-shot exemplars to reduce topic drift and stabilize repair reasoning, which updates the underlying meta-policy (decision path) rather than applying patch-centric code edits, improving reliability under long-spec hidden-test evaluation.

To support reproducibility, we provide detailed hyperparameter settings and implementation details in the Appendix. A complete replication package (source code, prompt templates, agent orchestration logic, and one-click run scripts) is publicly available on Zenodo (see Appendix A for details).

II. Related Work

We organize related work into three threads: (i) single-agent code generation with planning and refinement, (ii) multi-agent code generation with execution feedback, and (iii) debate-based multi-agent reasoning.

A. Single-Agent Code Generation and Task Planning

Pretrained code models have improved code synthesis capabilities. Codex [3] showed that large-scale pretraining improves syntactic and semantic correctness; WizardCoder [5] improved instruction-following; and AlphaCode [6] achieved competitive performance on challenging problems. LLMs have also been used for auxiliary tasks such as unit test generation [7]. To improve reliability, a common direction augments a single agent with lightweight planning or iterative refinement, including chain-of-thought prompting [9], Reflexion-style self-feedback [10], self-refinement/self-debugging [11], analogical prompting (APE) [12], and test-driven generation [4].

While effective for reducing superficial mistakes, these methods often struggle on long-spec, corner-case-heavy problems: they may produce brittle defensive logic and exhibit weak handling of malformed inputs or boundary conditions [8]. More importantly, without an explicit policy-level representation, refinement tends to be *patch-centric*—fixing observed symptoms in code rather than correcting the underlying global decision structure—which contributes to flattened structure and inconsistent branching logic [1], [2].

B. Multi-Agent Code Generation and Execution Feedback

Multi-agent collaboration has been explored to overcome limitations of single-agent systems by decomposing roles and enabling critique, review, and parallel reasoning. Early self-collaboration and role-based systems [13], [14] introduced multi-role interactions to improve solution completeness. Later frameworks such as MetaGPT [15] and ChatDev [17] modeled broader software engineering workflows, while CAMEL [16] studied role-playing and conversation control to improve coordination. More recent approaches scale multi-agent coding via structured reasoning chains and coordination mechanisms, including CodeCoT [18], CodeCoR [19], and self-organized agent collectives [20].

TABLE I
Comparison of existing code generation methods and their core capabilities.

Method	Planning	Examples	Test Generation	Diagnosis	Evolution
Reflexion [10]			✓	✓	
Analogical Prompting [12]	✓	✓			
MetaGPT [15]	✓			✓	
ChatDev [17]	✓		✓		
MapCoder [21]	✓	✓		✓	
CodeSIM [29]	✓	✓		✓	
DDCoder (Ours)	✓	✓	✓	✓	✓

Execution feedback is a particularly important signal for reliable code generation, and several frameworks incorporate runtime outcomes to guide repairs, e.g., MapCoder [21]. However, in many existing systems, execution feedback is primarily used to trigger *local* edits on candidate code. Strategy evolution is often implicit, and updates rarely reconstruct global branching structures or correct inconsistent decision paths. As a result, deep structural issues such as redundant logic, unclosed branches, or fragile boundary handling may persist even after multiple repair iterations [1], [2]. DDCoder differs by making the policy explicit and using execution to drive *policy-level* revision.

C. Debate-Based Multi-Agent Reasoning

Debate-based multi-agent reasoning improves rigor by assigning differentiated roles and perspectives and enforcing cross-examination protocols. ReConcile [22] proposed structured debate procedures, and subsequent work explored heterogeneous agent pools and collaboration efficiency, including collaborating-agent QA systems [23].

Applying debate to code generation introduces additional challenges: code synthesis requires precise structural constraints, alignment between high-level intent and executable artifacts, and grounding through verifiable signals (e.g., execution). Many coding-agent frameworks rely on serialized collaboration without adversarial argumentation (e.g., MetaGPT [15], ChatDev [17]), while SWE-Debate [24] introduces competitive debate but mainly targets fault localization rather than revising strategy-level design decisions. Moreover, generic debate protocols are often weakly coupled to execution feedback, limiting their ability to converge on code-specific, verifiable consensus.

Existing debate-based frameworks for code generation lack (i) *asymmetric* role specialization, (ii) an explicit *policy-level* representation of global decision structure, and (iii) tight coupling with execution feedback to support iterative global reconstruction. These limitations motivate our design choices.

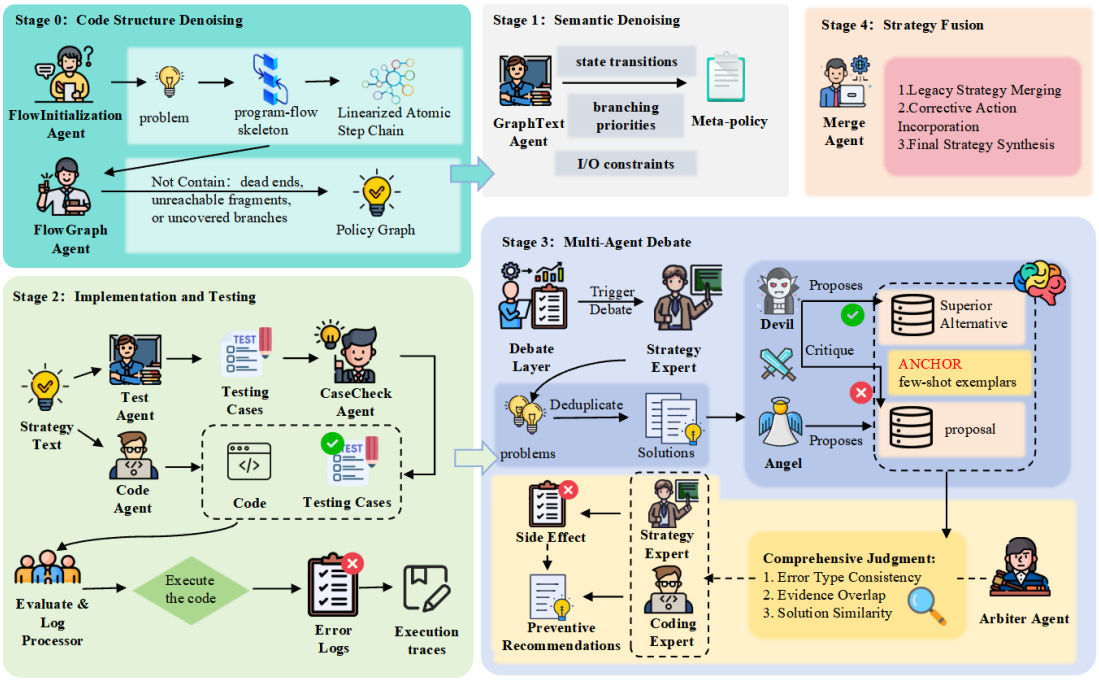


Fig. 1. The overall architecture diagram of the DDCoder framework.

III. Method

A. Overview: Three-Layer Iterative Denoising

[1] We treat robust code generation as a closed loop that makes the decision structure explicit, turns it into executable code with trustworthy tests, and then uses execution feedback to refine both. Fig. 1 summarizes the workflow, which we write as the composition

$$\text{Final Code} = f_{\text{Merge}}\Big(f_{\text{Behavior}}\big(f_{\text{Semantic}}(f_{\text{Structural}}(P))\big)\Big), \quad (1)$$

where P is the problem statement with I/O specification. The structural layer $f_{\text{Structural}}$ builds an analyzable control model; the semantic layer f_{Semantic} converts it into code and validated tests; the behavioral layer f_{Behavior} performs execution-grounded repair via asymmetric debate; and f_{Merge} finalizes the revised program.

a) Remark on "denoising": We use "denoising" to describe filtering and correcting artifacts that are common in LLM-generated programs. *Structural denoising* rejects ill-formed control models (e.g., unreachable branches or incomplete guards); *semantic denoising* filters noisy/incorrect augmented tests and misaligned Policy Graph-to-code translations; *behavioral denoising* reduces execution failures measured by failed tests (and ultimately improves pass@k). The three layers are illustrated in Fig. 2.

b) Notations: Let P be the problem statement (including constraints and I/O format). Let G denote the Policy Graph produced by the structural layer. Let C be a candidate program produced by the semantic layer. Let T be a test suite; we distinguish T_{base} (provided tests), T_{enh} (agent-generated enhanced tests), and T_{val} (validated enhanced tests). Executing C on T yields a set of failed tests $T_{\text{fail}} \subseteq T$ and per-test traces $\text{trace}(t, C)$ (stdout/stderr/exception). Let t_i denote the i-th failed test case in T_{fail}.

[1]Technical expressions in this section were refined with assistance from ChatGPT-5.2 [30].

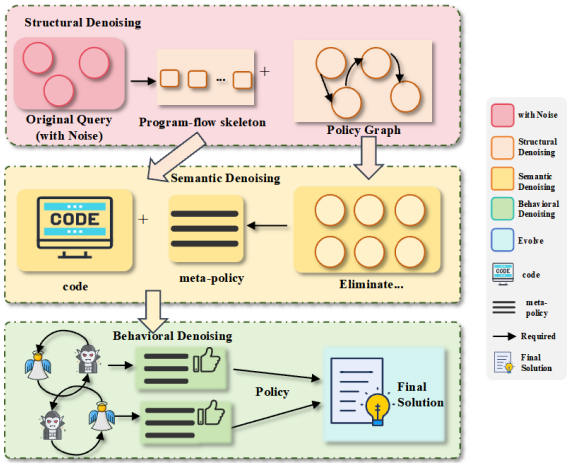


Fig. 2. Three-layer iterative denoising process (structural, semantic, and behavioral) in the DDCoder framework.

B. Structural Denoising Layer: Policy-Graph Based Control Modeling

a) Flow skeleton.: We start with a coarse program-flow skeleton that captures the macro execution order (e.g., "parse input → compute → output"). We represent it as a lightweight directed graph $F = (V_F, E_F)$, where nodes are

high-level stages and edges specify the order. This skeleton is intentionally simple: it is not a statement-level CFG, but a stable backbone that keeps later refinements from drifting.

b) Policy Graph as a typed attributed directed graph.: We model the decision structure with a *typed, attributed directed graph*

$$G := (V, E, \tau, \phi), \tag{2}$$

where V is the set of nodes, $E \subseteq V \times V$ is the set of directed edges, $\tau : V \to \mathcal{T}$ assigns each node a type from a finite set $\mathcal{T}$, and ϕ stores structured attributes. In our implementation, $\mathcal{T}$ includes GUARD, COMPUTE, TRANSFORM, IO, FORMAT, and TERMINATE. For each node $v \in V$, $\phi(v)$ records its intent, typed input/output schema, and preconditions/postconditions (constraints/invariants). For each edge $(u, v) \in E$, $\phi(u, v)$ optionally stores a guard expression (for GUARD branching) and a priority (integer tie-breaker) when deterministic successor selection is needed.

c) Schema-constrained JSON representation.: In practice, we serialize G as schema-constrained JSON (metadata, nodes, edges), which makes validity checks straightforward and keeps the interface stable across agents.

d) Structural denoising via constraint-and-regeneration.: A useful policy model must at least be well-formed: it should not contain dead ends, unreachable fragments, or uncovered branches. We therefore enforce a small set of structural constraints $\{\mathcal{C}_k\}$ and regenerate the Policy Graph when violations occur. Violations are detected by (i) JSON schema validation, (ii) reachability analysis, and (iii) sanity checks on branching guards and successor determinism.

We enforce the following constraints on the Policy Graph: $\mathcal{C}_1$ (Type validity): every node has a valid type $\tau(v) \in \mathcal{T}$ and required attributes in $\phi(v)$ (intent, typed I/O, pre/post). $\mathcal{C}_2$ (Top-level acyclicity): the top-level graph is acyclic to keep the global execution order unambiguous; iteration (if needed) must be encapsulated inside a TRANSFORM/COMPUTE node with explicit exit conditions in postconditions. $\mathcal{C}_3$ (Entry/exit discipline): after IO, the next operational node is a GUARD validating inputs; invalid cases must route to an explicit termination behavior (error output or exception), and the last node must be FORMAT or TERMINATE. $\mathcal{C}_4$ (Guard completeness): GUARD nodes are restricted to binary branching and their outgoing edges must include explicit guard expressions intended to be complementary (no uncovered cases). $\mathcal{C}_5$ (Deterministic successors): non-GUARD nodes have exactly one outgoing edge; if multiple successors are allowed, the successor is chosen deterministically by highest priority in $\phi(u, v)$. $\mathcal{C}_6$ (Connectivity and termination): there are no unreachable nodes or non-terminal nodes without successors, and every path ends in FORMAT/TERMINATE. $\mathcal{C}_7$ (Data-flow sanity): every produced value is consumed downstream, appears in the final output, or is declared auxiliary in intent/postconditions.

If a generated graph violates any $\mathcal{C}_k$, we reject it and request regeneration under the same schema and constraints until the constraints are satisfied or a preset regeneration budget is reached. In effect, this step keeps the global decision structure coherent before we start tuning code-level details.

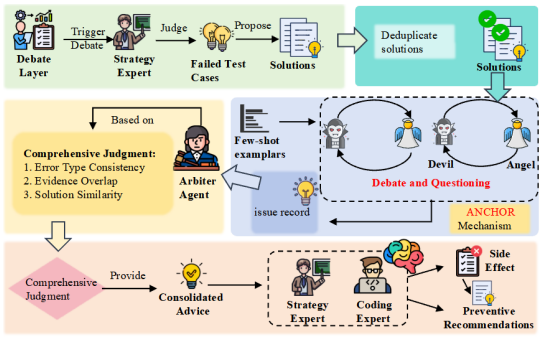


Fig. 3. Overview of the execution-guided asymmetric debate workflow in DDCoder.

C. Semantic Denoising Layer: Policy Graph-to-Code Translation and Test Validation

a) Policy Graph-to-code translation.: Given a structurally valid Policy Graph G, we first derive a high-level *meta-policy* S that specifies the execution plan implied by G (module boundaries, branching priorities, state transitions, and I/O constraints). We then translate S into a candidate program C while maintaining *policy–code alignment*. Concretely, each policy element in S (originating from a node type and its I/O contract in G) is realized as a corresponding code block (function/block/guard), and the resulting control structure preserves the branching semantics and priority order defined at the policy level. This reduces a common failure mode in long-spec tasks: code that appears locally plausible but violates global decision priorities.

b) Enhanced tests and semantic denoising.: Benchmarks often under-sample corner cases, so we generate an enhanced test set T_{enh} to stress boundary conditions, extreme values, and exceptional states. Since model-generated tests can be noisy or wrong [25], we validate them and keep only those that meet basic correctness and format requirements:

$$T_{\text{val}} = \text{ValidateTests}(T_{\text{enh}}, C). \tag{3}$$

Here, $\text{ValidateTests}(\cdot)$ returns the subset of T_{enh} whose I/O format matches P and whose expected outputs are well-defined.

c) Test validation protocol.: A dedicated CaseCheckAgent validates each generated test by (i) checking I/O format consistency with P, (ii) verifying that expected outputs are well-defined (e.g., no ill-typed or contradictory expectations), and (iii) optionally cross-checking expectations by executing C and ensuring internal consistency of the test specification. Tests that fail validation are discarded; the remaining T_{val} is used as a reliable feedback set for behavioral denoising.

D. Behavioral Denoising Layer: Execution-Guided Asymmetric Multi-Agent Debate

At this stage, we run C on T_{val} and obtain the failed-test set $T_{\text{fail}} \subseteq T_{\text{val}}$ together with per-test traces $\text{trace}(t, C)$

(stdout/stderr/exception). We then use debate to decide how to revise the underlying decision path (policy-level) and/or the implementation (code-level). Fig. 3 gives the high-level workflow. We iterate until all tests pass, a budget is reached, or improvements saturate.

a) Structured issue records from execution failures.: For each failed test case $t_i \in T_{\text{fail}}$, a strategy expert A_{strategy} writes a compact issue record:

$$R_i = A_{\text{strategy}}(P, G, C, t_i) = (\text{err}_i,\ \text{evid}_i,\ \text{sol}_i), \tag{4}$$

where err_i is a normalized error description, evid_i summarizes the trace and relevant code locations, and sol_i is an actionable plan that may include both policy edits and code edits.

b) Error extraction and normalization.: To avoid chasing superficial differences in error text, we normalize error strings before comparison:

$$\text{norm}(x) = \text{collapse\_spaces}\big(\text{remove\_punct}(\text{lower}(\text{strip}(x)))\big), \tag{5}$$

where $\text{remove\_punct}(\cdot)$ removes both English and Chinese punctuation, and $\text{collapse\_spaces}(\cdot)$ collapses repeated whitespace.

c) Deduplicating similar failures.: Multiple tests may fail due to the same underlying bug. We therefore deduplicate similar failure records before launching debates.We adopt a similarity-based failure deduplication strategy to avoid redundant computation, with empirical thresholds $\delta_1 = 0.85$ and $\delta_2 = 0.8$.

d) Asymmetric debate with few-shot anchors.: For each remaining issue record, we run an asymmetric debate between two specialized agents: an angel agent A_{angel} and a devil agent A_{devil}. The angel argues for a conservative fix consistent with current behavior, while the devil challenges it and proposes an alternative explanation and repair. We anchor both agents with a small set of high-quality exemplars to reduce unstable debate trajectories:

$$\text{Prompt}_{\text{debate}} = \text{Prompt}_0 \cup \{\text{FewShotExamples}\}. \tag{6}$$

e) Arbitration and risk auditing.: An arbiter A_{arbiter} selects the final action plan and attaches risk notes to prevent regressions:

$$D_i = A_{\text{arbiter}}(P, G, C, R_i) \to \{\text{steps}_i,\ \text{risk}_i\}, \tag{7}$$

where steps_i are concrete modification steps (policy updates and/or code edits) and risk_i highlights common pitfalls (e.g., I/O format drift, non-termination, or breaking previously passing cases). A risk auditor performs an additional check for typical regression patterns before applying edits.

f) Execution-grounded closed loop.: After applying the selected edits, we re-run the updated program on T_{val} to obtain a new failure set T_{fail}. This closes the loop and repeats until (i) $T_{\text{fail}} = \emptyset$, (ii) a maximum iteration budget is reached, or (iii) improvements saturate. In practice, this loop uses execution traces to update the meta-policy S (and thus the global decision path), rather than merely applying patch-style code tweaks.

Algorithm 1 DDCoder (Ultra-Minimal)

Require: $P, s, N_{\text{iter}}, r, E$
Ensure: $C^\star$

$G \leftarrow f_{\text{Structural}}(P)$; $S \leftarrow \text{DeriveMetaPolicy}(G)$
$(S, C) \leftarrow \text{EnforceSVT}(S, \text{Translate}(S))$ {Policy-code alignment}
$T_{\text{val}} \leftarrow \text{ValidateTests}(\text{GenPolicyAlignedTests}(P, G), C)$
for $k = 1$ to N_{iter} **do**
  $T_{\text{pass}}, T_{\text{fail}}, \{\text{trace}(t, C)\}_{t \in T_{\text{fail}}} \leftarrow \text{RunAndSplit}(C, T_{\text{val}})$
  acc $\leftarrow |T_{\text{pass}}|/|T_{\text{val}}|$
  if acc $\geq s$ **then**
    break
  end if{Only exit condition: accuracy $\geq s$}
  $\mathcal{R} \leftarrow \text{BuildAndDedupIssues}(P, G, S, C, T_{\text{fail}}, \{\text{trace}(t, C)\})$
  for all $R_i \in \mathcal{R}$ **do**
    $\Delta S_i \leftarrow \text{AsymDebate}(P, G, S, C, R_i, E, r)$ {Debate per deduplicated issue}
    $S \leftarrow \text{UpdateMetaPolicy}(S, \Delta S_i)$; $G \leftarrow \text{UpdatePolicyGraph}(G, \Delta S_i)$
    $(S, C) \leftarrow \text{EnforceSVT}(S, \text{Translate}(S))$ {Policy-code alignment, s not involved}
  end for
end for
$C^\star \leftarrow C$;
return $C^\star$

IV. Experimental Results

A. Experimental Setup

We evaluate *DDCoder* on three widely used benchmarks with increasing task complexity: *HumanEval* [3], *HumanEval-ET* [3], and *APPS* [26]. These benchmarks cover basic algorithmic reasoning, robustness under distribution shifts, and specification-intensive multi-step programming tasks with hidden tests.[2]

Performance is reported using the standard *pass@k* metric:

$$\text{pass@k} = \frac{1}{N} \sum_{i=1}^{N} \left(1 - \frac{\binom{n-c_i}{k}}{\binom{n}{k}} \right), \tag{8}$$

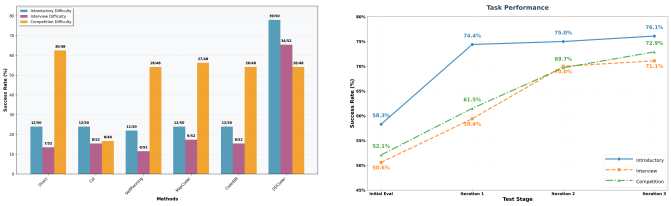
where n is the number of generated samples per task, c_i is the number of correct samples for task i, and N is the total number of tasks.

Our goal is *reliability-oriented* LLM-based code generation. In addition to functional correctness (pass@k), we analyze robustness-related behaviors that commonly appear in long-specification tasks, such as boundary handling, exception paths, and globally consistent control logic. We further provide a qualitative case study (Sec. IV-C) to illustrate representative failure modes and how DDCoder resolves them through explicit decision-structure modeling and execution-grounded refinement.

[2]The presentation of results and discussion in this section was improved with assistance from ChatGPT-5.2 [30].

TABLE II
COMPARISON OF PASS@1 SCORES (%). BEST RESULTS PER COLUMN ARE HIGHLIGHTED IN BOLD.

Method	HumanEval	HumanEval-ET	APPS
DeepSeek-Chat			
Direct	73.17	75.00	32.67
CoT [9]	91.46	79.90	18.07
Self-Planning [27]	91.46	84.10	28.67
Analogical [28]	92.07	80.49	28.00
MapCoder [21]	95.73	84.10	32.00
CodeSIM [29]	**97.60**	87.20	30.67
DDCoder (Ours)	92.68	**93.29**	**66.00**
Qwen-Plus			
Direct	92.68	86.00	21.33
CoT [9]	91.50	82.90	20.67
Self-Planning [27]	93.29	84.10	23.33
Analogical [28]	92.68	81.70	17.33
MapCoder [21]	93.29	75.60	25.33
CodeSIM [29]	**97.00**	87.20	28.67
DDCoder (Ours)	93.90	**92.68**	**45.34**



(a) Performance on APPS across difficulty levels.

(b) Algorithm convergence across different difficulty tiers.

Fig. 4. Comparative performance of DDCoder across APPS difficulty tiers and iterative refinement rounds.

TABLE III
ABLATION RESULTS ON APPS (PASS@1, %).

Variant	Pass@1	Drop (pp)
w/o Policy Graph	64.00	2.00
w/o Enhanced Tests	64.67	1.33
w/o Debate	53.33	**12.67**
w/o Few-shot Exemplars	63.33	2.67
Full DDCoder	66.00	–

All methods are evaluated under identical decoding settings on two commercial LLM backends (*DeepSeek-Chat* and *Qwen-Plus*). These include *Direct*, *CoT* [9], *Self-Planning* [27], *Analogical Reasoning* [28], *MapCoder* [21], and *CodeSIM* [29]. Hyperparameter settings for DDCoder are given in the Appendix.

B. Comparison Experiments

Table II reports pass@1 across all datasets. DDCoder achieves the best performance among evaluated methods on *HumanEval-ET* and *APPS* for both LLM backends. On APPS—a benchmark with long specifications, multi-constraint reasoning, and hidden robustness checks—DDCoder improves pass@1 by 34 points on DeepSeek-Chat (32.00% to 66.00%) and by 17 points on Qwen-Plus (28.67% to 45.34%).

On *HumanEval*, where tasks are short and less constraint-heavy, DDCoder remains competitive but does not consistently surpass the strongest prompting baselines. This suggests that the primary benefits of our closed-loop refinement emerge when global decision structure, boundary handling, and execution robustness dominate the difficulty.

Fig. 4 breaks down APPS performance by difficulty tiers (*Introductory*, *Interview*, and *Competition*). DDCoder matches or exceeds the best baseline across all tiers, with gains larger on higher-difficulty tiers. Panel (b) illustrates convergence across difficulty levels, where iterative refinement steadily improves pass rates over multiple rounds.

C. Qualitative Analysis: Case Study

To illustrate how DDCoder mitigates structural and execution-level failure modes in specification-intensive tasks, we present a case study on the "Zonk Dice Game Score Calculator" problem from APPS. This task involves multiple scoring rules, complex branching logic, and numerous corner cases (e.g., straight, three pairs, various n-of-a-kind combinations, and single-die scoring), making it representative of long-specification programs where reliability requires consistent global decision paths and robust boundary handling.

Fig. 5 compares representative implementations generated by different methods under identical decoding settings. We observe recurring failure patterns in baseline outputs, including mis-specified rule priority, incomplete handling of special combinations, and dice-consumption conflicts that lead to double counting or missed scores. Some methods also exhibit unstable state updates (e.g., premature resets) that break subsequent scoring logic. In contrast, DDCoder produces a solution that explicitly validates inputs, enforces a coherent rule priority, tracks dice usage to avoid conflicts, and correctly handles all special combinations. This example demonstrates how Policy Graph modeling and execution-guided asymmetric debate enable global decision-path correction beyond local patching.

D. Ablation Study

Table III reports ablation results on APPS. Removing the debate layer causes the largest performance drop (-12.67 pp), indicating that execution-grounded asymmetric debate is a major driver of gains on complex, long-specification tasks. Removing the Policy Graph module yields a consistent degradation (-2.00 pp), suggesting that explicit decision-structure modeling contributes beyond prompting alone. Removing validated enhanced tests (-1.33 pp) and few-shot exemplars (-2.67 pp) further reduces performance, highlighting the importance of reliable feedback signals and anchored debate dynamics. Overall, the improvements are driven by the synergy of Policy Graph modeling, test augmentation, and debate-based refinement rather than any single component.

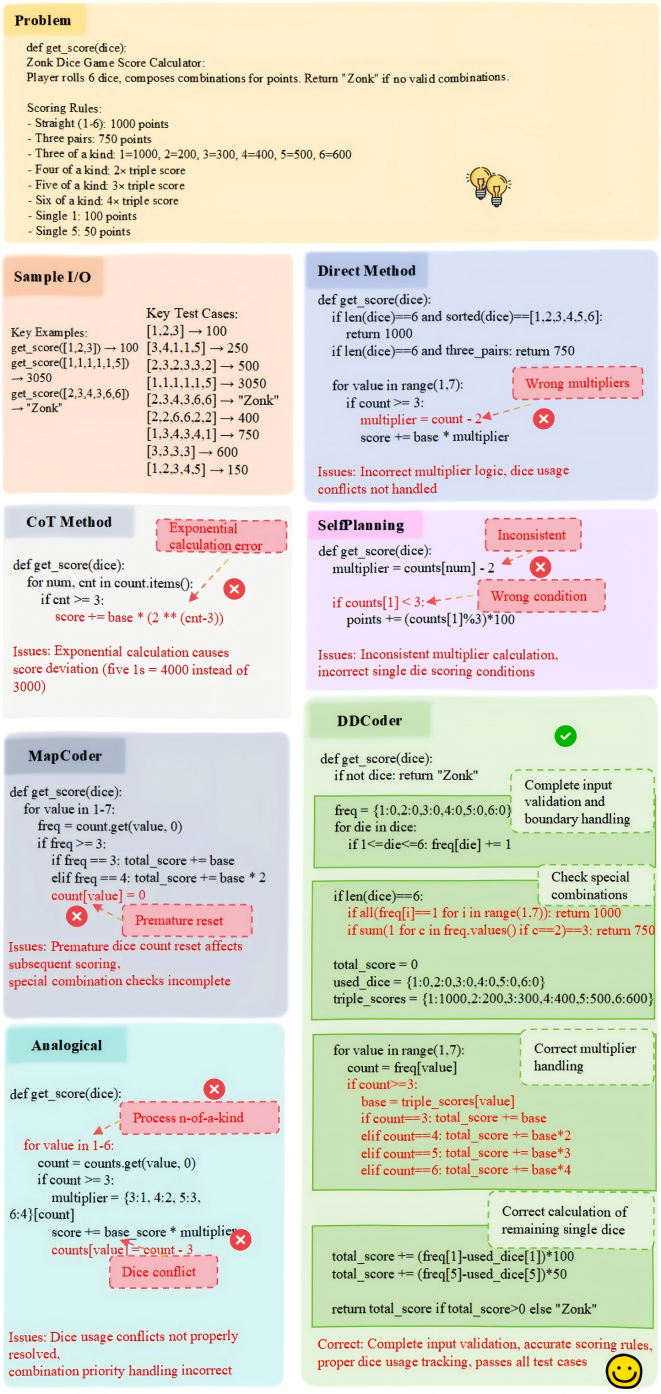


Fig. 5. Qualitative comparison of code implementations for the "Zonk Dice Game Score Calculator" problem. The figure highlights representative failure patterns in baseline outputs (e.g., rule-priority conflicts, unstable state updates, and dice-consumption inconsistencies) versus DDCoder's robust implementation with complete validation and consistent scoring logic.

E. Model Performance Analysis

Fig. 6 illustrates sensitivity to SVT (code validity, denoted as s), which determines the acceptance standard, and to iteration limits. Both backends exhibit smooth improvements as SVT increases and more refinement rounds are permitted,

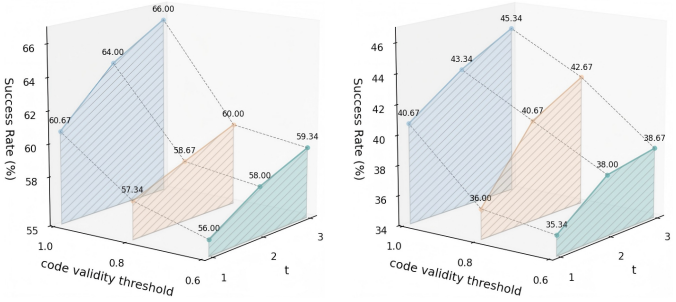


Fig. 6. Sensitivity to threshold and iteration count.

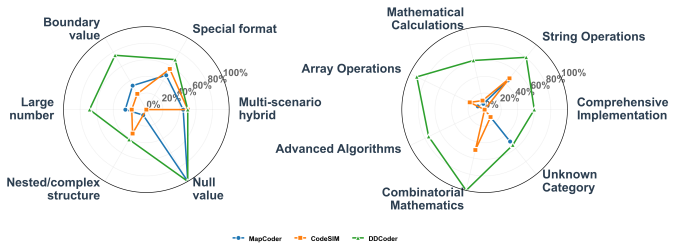


Fig. 7. Scenario-level (left) and task-type-level (right) performance on APPS.

with diminishing returns after 2–3 iterations, suggesting that DDCoder can be budgeted with a small number of refinement steps for practical usage.

DDCoder introduces additional inference-time computation due to its closed-loop design (Policy Graph construction, enhanced test generation/validation, and multi-agent debate refinement). However, this cost is explicitly controllable through the refinement budget (e.g., SVT and the number of refinement iterations N_{iter}). As shown in Fig. 6, even lightweight configurations yield competitive pass@1, while increasing the budget improves pass@1 smoothly and saturates after 2–3 iterations. This enables a practical budget-aware deployment strategy: low-budget refinement can be used for rapid iteration or easy instances, and higher-budget refinement can be selectively allocated to difficult problems where robustness is critical.

To assess robustness in diverse execution conditions, we categorize APPS tasks along two orthogonal dimensions: (1) **input robustness scenarios** (boundary, null, irregular formats, nested structure, large-number conditions), and (2) **task types** (arrays, combinatorics, string manipulation, advanced algorithms, etc.). Fig. 7 shows that DDCoder outperforms baselines across these categories, with larger gains on boundary/irregular inputs and large-number scenarios, which are common sources of failures in real-world execution.

V. Conclusion

This paper presents DeepDebateCoder, a closed-loop framework for generating code from long, complex specifications. The framework improves generation reliability by employing an explicit and modifiable high-level decision structure. DeepDebateCoder first constructs a typed Policy Graph to encode branching priorities, state transitions, and I/O constraints, and enforces structural constraints to reduce logical ambiguity. It

then translates the policy representation into executable code, and generates enhanced tests to probe boundary cases and exceptional behaviors. A test validation protocol filters out ill-formed or self-contradictory tests to suppress noise in execution feedback. The execution-guided asymmetric debate loop generates concrete repair plans that revise both policy-level decision paths and code-level implementations. Experiments on the HumanEval-ET and APPS datasets with two LLM backends demonstrate consistent performance improvements for DeepDebateCoder, with the most significant gains achieved on harder, longer-specification tasks. Experimental results also indicate that future work can focus on more robust validation for a broader range of task types and more budget-efficient refinement strategies.

Appendix A: Hyperparameters and Implementation Details

In our DDCoder experiments, the core hyperparameters include the number of enhanced test cases (n_{test}), SVT (code validity, denoted as s), which determines the acceptance standard, the number of refinement iterations (N_{iter}), and the number of debate rounds (r). Specifically, n_{test} is set to 10, controlling the number of additional test cases generated for each problem; s is set to 1; N_{iter} is set to 3, representing the maximum number of refinement iterations; and r is set to 2, specifying the number of debate rounds for each failed test case. These hyperparameters jointly control the depth of refinement and the compute–accuracy trade-off.

Our experiments use the OpenAI API [30] (via `langchain-openai`) for LLM calls, among other libraries. The full environment specification (`requirements.txt`), complete prompt templates, and agent orchestration logic are released with the code package on Zenodo: https://doi.org/10.5281/zenodo.19212523.

Acknowledgments

We acknowledge the use of ChatGPT-5.2 [30] for editing and polishing portions of this manuscript, including refinement of technical expressions and improvements to the presentation of experimental results and discussion.

References

[1] Pasquale, L., Sabetta, A., d'Amorim, M., Hegedűs, P., Tarrit Mirakhorli, M., Okhravi, H.: Challenges to using large language models in code generation and repair. IEEE Security & Privacy 23(2), 81–88 (2025).

[2] Clark, A., Igbokwe, D., Ross, S., Zibran, M.F.: A quantitative analysis of quality and consistency in AI-generated code. In: 7th International Conference on Software and System Engineering (ICoSSE 2024), pp. 1–8. IEEE, Paris (2024).

[3] Chen, M., Tworek, J., Jun, H., et al.: Evaluating large language models trained on code. arXiv preprint arXiv:2107.03374 (2021). (Unpublished; later published.)

[4] Chen, B., Zhang, F., Nguyen, A., Zan, D., Lin, Z., Lou, J.-G., Chen, W.: CodeT: code generation with generated tests. In: ICLR (2023).

[5] Luo, Z., Xu, C., Zhao, P., Sun, Q., Geng, X., Hu, W., Tao, C., Ma, J., Lin, Q., Jiang, D.: WizardCoder: empowering code large language models with Evol-Instruct. In: ICLR (2024).

[6] Li, Y., Choi, D., Chung, J., Kushman, N., Schrittwieser, J., Leblond, R., et al.: Competition-level code generation with AlphaCode. Science 378(6624), 1092–1097 (2022).

[7] Fakhoury, S., Naik, A., Sakkas, G., Chakraborty, S., Lahiri, S.K.: LLM-based test-driven interactive code generation: user study and empirical evaluation. IEEE Trans. Softw. Eng. 50(9), 2254–2268 (2024).

[8] Zhang, C., Wang, Z., Zhao, R., Mangal, R., Fredrikson, M., Jia, L., Pasareanu, C.: Attacks and defenses for large language models on coding tasks. In: ASE 2024, pp. 2268–2272. ACM (2024).

[9] Wei, J., Wang, X., Schuurmans, D., Bosma, M., Ichter, B., Xia, F., Chi, E., Le, Q., Zhou, D.: Chain-of-thought prompting elicits reasoning in large language models. NeurIPS 35, 24824–24837 (2022).

[10] Shinn, N., Cassano, F., Berman, E., Gopinath, A., Narasimhan, K., Yao, S.: Reflexion: language agents with verbal reinforcement learning. NeurIPS 36, 8634–8652 (2023).

[11] Madaan, A., Tandon, N., Gupta, P., et al.: Self-Refine: iterative refinement with self-feedback. arXiv preprint arXiv:2303.17651 (2023). (Unpublished.)

[12] 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. NeurIPS 36, 21558–21572 (2023).

[13] Dong, Y., Jiang, X., Jin, Z., Li, G.: Self-collaboration code generation via ChatGPT. ACM Trans. Softw. Eng. Methodol. 33(7), Article 189 (2024).

[14] Talebirad, Y., Nadiri, A.: Multi-agent collaboration: harnessing collective intelligence with LLMs. arXiv preprint arXiv:2306.03314 (2023). (Unpublished.)

[15] Hong, S., Zhuge, M., Chen, J., et al.: MetaGPT: meta programming for multi-agent collaborative framework. arXiv preprint arXiv:2308.00352 (2023). (Unpublished.)

[16] Li, G., Hammoud, H.A.A.K., Itani, H., Khizbullin, D., Ghanem, B.: CAMEL: communicative agents for "mind" exploration of large language model society. NeurIPS 36 (2023).

[17] Qian, C., Liu, W., Liu, H., et al.: ChatDev: communicative agents for software development. arXiv preprint arXiv:2307.07924 (2023). (Unpublished.)

[18] Huang, D., Bu, Q., Qing, Y., Cui, H.: CodeCoT: tackling code syntax errors in CoT reasoning for code generation. arXiv preprint arXiv:2308.08784 (2024). (Unpublished.)

[19] Pan, R., Zhang, H., Liu, C.: CodeCoR: an LLM-based self-reflective multi-agent framework for code generation. arXiv preprint arXiv:2501.07811 (2025). (Unpublished.)

[20] Ishibashi, Y., Nishimura, Y.: Self-organized agents: a large language model multi-agent framework for ultra-large-scale code generation and optimization. arXiv preprint arXiv:2404.02183 (2024). (Unpublished.)

[21] Islam, M.A., Ali, M.E., Parvez, M.R.: MapCoder: multi-agent code generation for competition-level problem solving. arXiv preprint arXiv:2405.11403 (2024). (Unpublished.)

[22] Chen, J.C.-Y., Saha, S., Bansal, M.: ReConcile: round-table conference improves reasoning via consensus among diverse LLMs. In: ACL 2024, pp. 381–394 (2024).

[23] Huynh, T., Nguyen, D.: Collaborating agents with large language models for question-answering tasks. In: SEAI 2025. IEEE (2025).

[24] Li, H., Shi, Y., Lin, S., Gu, X., Lian, H., Wang, X., Jia, Y., Huang, T., Wang, Q.: SWE-Debate: competitive multi-agent debate for software issue resolution. arXiv preprint arXiv:2507.23348 (2025). (Unpublished.)

[25] Baqar, M., Khanda, R.: The future of software testing: AI-powered test case generation and validation. In: CompCom 2025. Springer (2025).

[26] Hendrycks, D., Basart, S., Kadavath, S., et al.: Measuring coding challenge competence with APPS. NeurIPS 34 (2021).

[27] Jiang, X., Dong, Y., Wang, L., et al.: Self-planning code generation with large language models. ACM Trans. Softw. Eng. Methodol. 33, 1–30 (2024).

[28] Yasunaga, M., Chen, X., Li, Y., Pasupat, P., Leskovec, J., Liang, P., Chi, E.H., Zhou, D.: Large language models as analogical reasoners. In: ICLR (2024).

[29] Islam, M.A., Ali, M.E., Parvez, M.R.: CodeSIM: multi-agent code generation and problem solving through simulation-driven planning and debugging. In: NAACL Findings (2025).

[30] OpenAI. ChatGPT-5.2. https://openai.com/. Accessed: 2026-01.