

SolidCoder: Bridging the Mental-Reality Gap in LLM Code Generation through Concrete Execution

Woojin Lee Jin-Xia Huang

Electronics and Telecommunications Research Institute, Republic of Korea
{writerwoody, hgh}@etri.re.kr

Abstract

State-of-the-art code generation frameworks rely on *mental simulation*, where LLMs internally trace execution to verify correctness. We expose a fundamental limitation: the **Mental-Reality Gap**—where models hallucinate execution traces and confidently validate buggy code. This gap manifests along two orthogonal dimensions: the **Specification Gap** (overlooking edge cases during planning) and the **Verification Gap** (hallucinating correct behavior for flawed code). We propose **SolidCoder** with a simple principle: *don't imagine—execute*. The **S.O.L.I.D.** architecture addresses both dimensions by forcing edge-case awareness before algorithm design and replacing imagined traces with sandboxed execution using property-based oracles. With GPT-4o, SolidCoder achieves state-of-the-art pass@1 performance: **95.7%** on HumanEval (+0.6%p), **77.0%** on CodeContests (+4.3%p), and **26.7%** on APPS (+3.4%p). Ablation reveals that edge-case awareness provides the largest individual gain, while execution grounding catches categorically different errors that specification improvements cannot address. These gains generalize to RL post-trained models, validating that bridging both gap dimensions is essential for robust code synthesis. We release our code and framework to facilitate future research.¹

1 Introduction

Recent large language models have demonstrated remarkable programming capabilities (Ge et al., 2025). Foundation model families such as GPT (Achiam et al., 2023), Gemini (Team et al., 2023), Claude (Anthropic, 2024), DeepSeek (Liu et al., 2025), Kimi (Team et al., 2025), and Qwen (Yang et al., 2025) have shown impressive proficiency in generating code from natural language descriptions. However, synthesizing correct programs remains

inherently challenging, requiring deep understanding of natural language processing, algorithmic design, data structure selection, and problem-solving strategies. These challenges are compounded for competitive programming and advanced software engineering tasks, where adherence to strict constraints and passing comprehensive test suites are paramount (Chen et al., 2021; Hendrycks et al., 2021; Li et al., 2022; Jimenez et al., 2023; Jain et al., 2024).

To overcome these challenges, a rich line of research has emerged. Early methods employed direct prompting (Chen et al., 2021), Chain-of-Thought reasoning (Wei et al., 2022), retrieval-augmented generation (Parvez et al., 2021), and various in-context exemplar-based strategies (Zhang et al., 2022; Shum et al., 2023). Recent paradigms have shifted toward plan-based decomposition (Jiang et al., 2024), tree-searching approaches (Zhou et al., 2023), analogical reasoning (Yasunaga et al., 2023), and diverse agent-based architectures (Shinn et al., 2023; Qian et al., 2024). Among these, MapCoder (Islam et al., 2024) marked a significant advance by introducing a multi-agent pipeline that separates concerns: one agent retrieves analogous solved problems, another constructs execution plans, a third generates code, and a fourth handles debugging through iterative refinement. Building on this foundation, CodeSIM (Islam et al., 2025) achieved state-of-the-art results by leveraging *Mental Simulation* at two critical phases: (1) in the planning agent, the LLM simulates execution to verify the plan before code generation, and (2) in the debugging agent, when code fails tests, simulation is employed again to trace through the logic and identify bugs.

However, this approach has a fundamental flaw: **LLMs hallucinate**. In complex algorithmic scenarios, models frequently “imagine” execution traces that do not reflect actual program behavior—effectively engaging in wishful thinking where they

¹<https://github.com/10kH/SolidCoder>

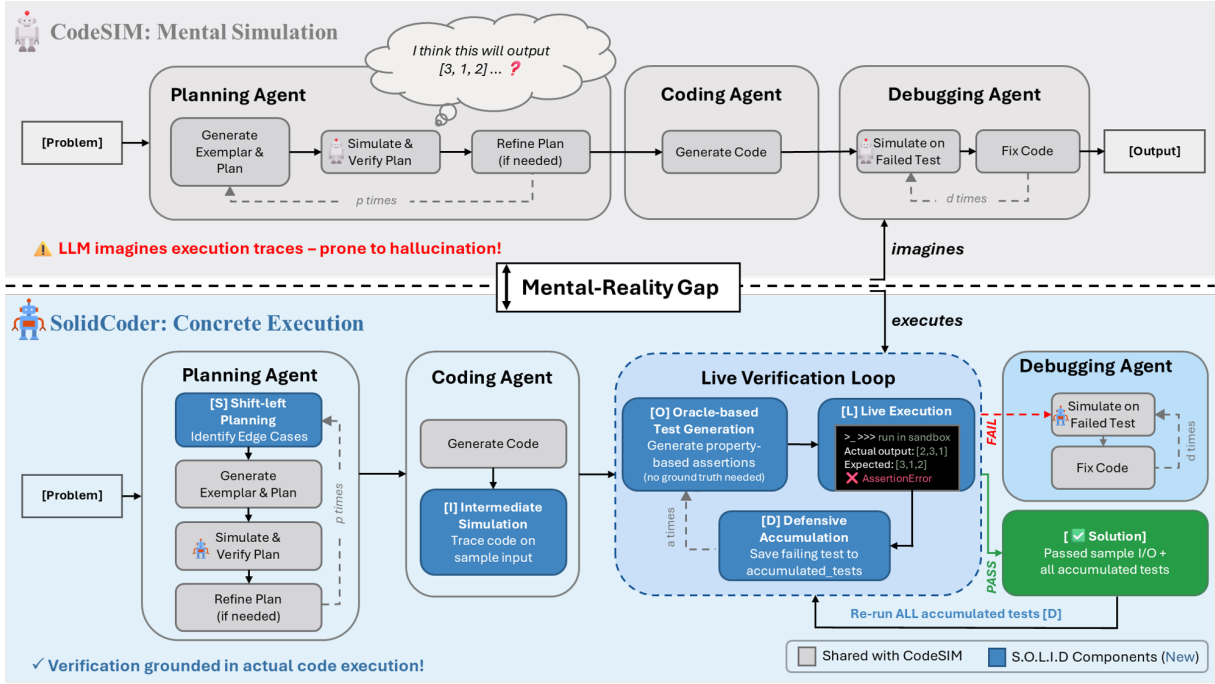


Figure 1: **Comparative Architecture Overview: CodeSIM vs. SolidCoder.** The top pipeline illustrates the CodeSIM baseline, which relies on mental simulation ("imagines") for verification during planning and debugging, resulting in a "Mental-Reality Gap" prone to LLM hallucination. The bottom pipeline demonstrates SolidCoder, which bridges this gap by grounding verification in concrete execution ("executes"). SolidCoder incorporates S.O.L.I.D. enhancements highlighted in blue: (1) The **Planning Agent** integrates Shift-left Planning to identify edge cases before plan generation; (2) The **Coding Agent** applies Intermediate Simulation to verify code traces; (3) The central **Live Verification Loop** replaces mental checks with sandbox Live Execution utilizing Oracle-based assertions. Failing tests are stored via Defensive Accumulation, and fixed code must pass *all* accumulated tests ([D]) to prevent regression before being accepted as a solution.

convince themselves that flawed code works correctly (Huang et al., 2025). This is akin to playing blindfold chess and declaring victory without ever moving the pieces on the board. We term this phenomenon the **"Mental-Reality Gap"**—the disconnect between how an agent *believes* a program behaves and how it *actually* behaves at runtime. This gap causes agents to confidently validate buggy code, submitting incorrect solutions as final answers. Worse, such a self-affirming verification loop can reinforce errors rather than correcting them.

The most direct way to bridge the Mental-Reality Gap is to *actually execute* the code rather than imagining its behavior. Two barriers have historically prevented this approach. The first is **technical feasibility**: when LLMs were pure text generators, interaction with external environments felt unnatural, and incorporating execution results into reasoning was awkward. This barrier has largely fallen—with tool use and function calling emerging as core capabilities (Yao et al., 2022; Schick et al., 2023),

modern agents can execute code, interpret error messages, and refine their approach based on empirical evidence. The second barrier is more fundamental: the **Missing Oracle problem**. Executing code requires test cases with expected outputs, but for novel algorithmic problems, the correct output is precisely what we are trying to discover.

CodeSIM’s authors confronted this oracle challenge directly. They explored augmenting test cases through self-consistency—generating tests twice and retaining only those appearing in both responses. The result was a **9.3%p performance decline**, leading them to conclude that “generating additional I/Os is a non-trivial task that may negatively impact final outcomes” (Islam et al., 2025). However, we argue that the failure stemmed not from test generation itself, but from attempting to predict *exact outputs* without ground truth. Our key insight is that verification need not require exact answers: by checking *properties* rather than values (e.g., “output length equals input length,” “result is a permutation of input”), we can judge correctness

without an oracle. This property-based approach closes the verification loop that CodeSIM left open.

Two Dimensions of the Mental-Reality Gap. Our analysis reveals that this gap manifests along two orthogonal dimensions: the **Specification Gap** (overlooking edge cases during planning) and the **Verification Gap** (hallucinating execution traces during validation). These are independent—addressing one leaves the other open. Our ablation (Table 3) confirms: removing Shift-left Planning ([S]) causes a -23.7% drop, removing Live Execution ([L]) causes -7.9% , and the full system achieves performance that neither alone can reach.

Building on this insight, we propose **SolidCoder**, a multi-agent framework addressing *both* dimensions. In the **S.O.L.I.D.** architecture: **Shift-left Planning** addresses the Specification Gap by forcing edge-case awareness before algorithm design; **Live Execution** addresses the Verification Gap by replacing imagined traces with concrete runtime feedback; **Oracle-based Assertions** enable verification without ground truth; **Intermediate Simulation** provides a cost-effective pre-filter; and **Defensive Accumulation** prevents regression. Our contributions are:

- We identify and characterize the **Mental-Reality Gap** as a *two-dimensional* phenomenon: the *Specification Gap* (edge-case blindness) and the *Verification Gap* (execution hallucination)—a fundamental limitation in existing multi-agent code generation frameworks.
- We propose **SolidCoder**, a framework that jointly addresses *both* dimensions through the **S.O.L.I.D.** architecture: Shift-left Planning for the Specification Gap, Live Execution for the Verification Gap, with Oracle assertions, Intermediate simulation, and Defensive accumulation as supporting mechanisms.
- We achieve new state-of-the-art pass@1 results with GPT-4o: **95.7%** on HumanEval (+0.6%p), **77.0%** on CodeContests (+4.3%p), and **26.7%** on APPS (+3.4%p). Our ablation demonstrates that both dimensions contribute substantially, with edge-case awareness providing the largest individual gain and execution grounding providing essential complementary value—together outperforming either component alone.

2 Related Work

2.1 Program Synthesis and LLMs for Code

Program synthesis has a long-standing history in AI, evolving from early search-based methods (Manna and Waldinger, 1971; Gulwani, 2011) to neural approaches (Feng et al., 2020; Ahmad et al., 2021). The advent of Large Language Models (LLMs) has fundamentally transformed this landscape.

Codex (Chen et al., 2021) pioneered LLMs for code. GPT-4 (Achiam et al., 2023) and GPT-4o (Hurst et al., 2024) marked a turning point, with other frontier models (Team et al., 2023; Anthropic, 2024) demonstrating comparable capabilities. Open-source alternatives (Roziere et al., 2023; Guo et al., 2024; Hui et al., 2024) have rapidly closed the gap. More recently, reasoning models (Jaech et al., 2024; Guo et al., 2025) have advanced code generation through techniques like RLVR (Wang et al., 2025), using execution feedback as training signal. As these models continue to improve, they increasingly serve as the “brain” for autonomous agents.

2.2 Approaches to LLM Code Generation

With capable LLMs as the foundation, the question becomes *how* to best orchestrate them. Various prompting strategies have been developed to enhance code generation. Chain-of-Thought (CoT) (Wei et al., 2022) decomposes complex problems into intermediate reasoning steps. Retrieval-augmented generation (Parvez et al., 2021) leverages similar examples to guide generation. Self-planning (Jiang et al., 2024) generates algorithmic plans before coding. Analogical reasoning (Yasunaga et al., 2023) recalls and adapts solutions from related problems. ReAct (Yao et al., 2022) interleaves reasoning with action execution. While these techniques improve single-pass generation, they remain limited by the LLM’s inability to verify its own outputs through actual execution.

To address this limitation, recent research has shifted towards multi-agent frameworks that simulate the software development lifecycle. MapCoder (Islam et al., 2024) introduced a pipeline of retrieval, planning, coding, and debugging agents, significantly outperforming single-agent baselines. CodeSIM (Islam et al., 2025) further enhanced this by incorporating simulation-driven planning and debugging, allowing agents to verify their logic through step-by-step execution traces. Other ap-

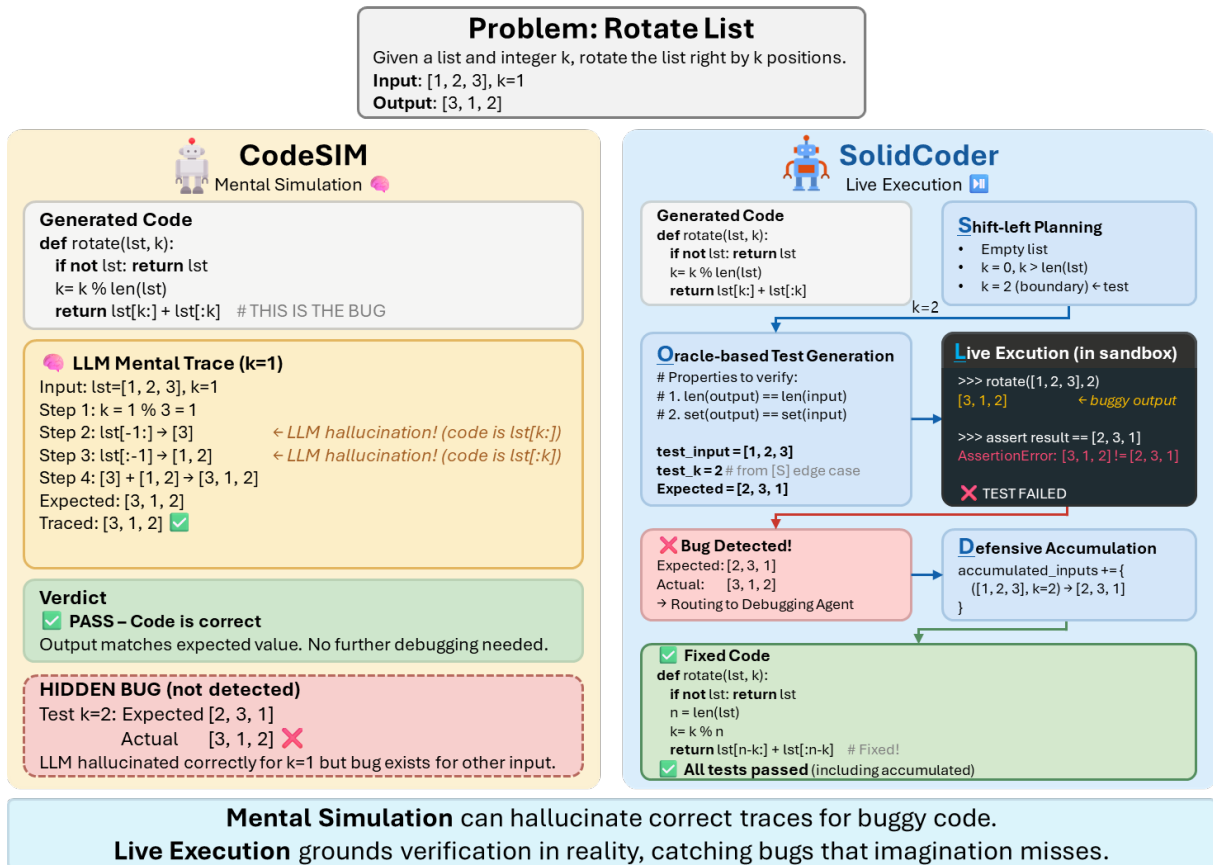


Figure 2: **The Mental-Reality Gap in Action.** A comparative example on a list rotation problem. **Left (CodeSIM):** Mental Simulation traces through the code and incorrectly concludes “Output matches expected. PASS”—the LLM hallucinates correct behavior despite a bug. **Right (SolidCoder):** Live Execution runs the actual code, revealing the bug through concrete failure: `AssertionError: [3, 1, 2] != [2, 3, 1]`. This demonstrates how grounding verification in reality catches bugs that mental simulation misses. (This example focuses on the Live Verification Loop; [I] Intermediate Simulation is depicted in Figure 1.)

proaches like Reflexion (Shinn et al., 2023) and LATS (Zhou et al., 2023) leverage iterative self-correction and tree search, respectively. However, these frameworks still rely on the LLM’s internal reasoning for verification, making them susceptible to hallucinated execution traces.

A separate line of work has explored leveraging actual code execution for debugging. LDB (Zhong et al., 2024) segments programs into basic blocks and tracks intermediate variable values during runtime, identifying discrepancies step-by-step. MGDebugger (Shi et al., 2024) takes a hierarchical approach, decomposing code into a tree of sub-functions and debugging bottom-up. While these execution-based debuggers ground verification in reality, they operate as a *second pass* after initial code generation and require ground-truth test cases to detect errors. SolidCoder addresses this limitation by performing verification *in-loop* with property-based oracles rather than ground-truth

outputs, enabling autonomous verification under missing-oracle constraints.

3 SolidCoder Methodology

SolidCoder unifies the strengths of both paradigms while addressing their limitations. Like multi-agent frameworks, we operate within an integrated generation pipeline; like external debuggers, we leverage actual execution. Building upon the three-agent architecture of CodeSIM—comprising Planning, Coding, and Debugging agents—we introduce the **S.O.L.I.D.** framework to systematically address the Mental-Reality Gap. Our guiding principle is straightforward: *don’t imagine—execute*. Rather than relying on mental simulation for verification, we integrate concrete execution feedback directly into the generation loop.

Our key innovations are threefold: (1) we integrate live execution *into* the generation loop rather

Approach	Generation				Verification		
	Exemplar Retrieval	Plan	Edge Aware	Mental Simulation	Live Execution	Debug	Defensive Accumulation
Reflection	✗	✗	✗	✗	✗	✓	✗
Self-planning	✗	✓	✗	✗	✗	✗	✗
Analogical	✓	✗	✗	✗	✗	✗	✗
LATS	✗	✓	✗	✗	✗	✓	✗
MapCoder	✓	✓	✗	✗	✗	✓	✗
CodeSIM	✓	✓	✗	✓	✗	✓	✗
SolidCoder	✓	✓	✓	✓	✓	✓	✓

Table 1: Comparison of code generation approaches. SolidCoder integrates these capabilities within a single inference-time framework.

than applying it post-hoc, (2) oracle-based assertions enable property verification without ground-truth answers, and (3) Defensive Accumulation ensures discovered bugs never regress. Table 1 compares SolidCoder with existing code generation approaches.

3.1 S.O.L.I.D. Components

Shift-left Planning (S). Traditional approaches address edge cases reactively during debugging. SolidCoder extends the Planning Agent to identify potential failure cases *before* formulating the algorithmic plan. We prompt the LLM: “*What worst-case inputs could break a naive solution?*” These edge cases (empty inputs, boundary values, corner cases) are then incorporated into the planning prompt, forcing robust algorithm design from the outset.

Oracle-based Assertions (O). A fundamental challenge in test generation is the “oracle problem”—determining correct outputs for novel inputs. Mental Simulation sidesteps this by having the LLM predict outputs, reintroducing hallucination risk. SolidCoder addresses this through *property-based verification*: rather than checking exact outputs, we verify that results satisfy domain-invariant properties (e.g., a sorting function preserves length, maintains ordering, and produces a permutation). This transforms verification from “Is this output correct?” to “Does this output satisfy required properties?”—answerable through execution.

Live Execution (L). Rather than asking the LLM to imagine code behavior, we run it. Mental Simulation requires maintaining accurate state across potentially hundreds of execution steps—increasingly error-prone as complexity grows. Live Execution eliminates this by delegating state tracking to an actual interpreter. We execute generated code in a sandboxed environment (5-second timeout, re-

stricted filesystem access), routing results to debugging on assertion failures or runtime errors.

Intermediate Simulation (I). A verified plan does not guarantee correct code—translation errors can occur. Immediately after code generation, we prompt the LLM to trace the code on a sample input. Why retain mental simulation if unreliable? The key difference is *role*: in CodeSIM, it provides the terminal verdict; in SolidCoder, [I] is a preliminary filter followed by [L] Live Execution. If [I] misses a bug, [L] catches it through concrete execution. The critical guarantee: [L]—not [I]—provides the authoritative verdict.

Defensive Accumulation (D). Iterative debugging introduces regression risk: fixing bug A may reintroduce bug B from a previous iteration. Defensive Accumulation maintains a persistent test suite that grows throughout debugging. Whenever Live Execution discovers a failing test case, that input-assertion pair joins an `accumulated_tests` collection. Crucially, *all* accumulated tests are re-executed after every code modification, providing a monotonicity guarantee: once code passes a test, regression triggers immediate detection.

3.2 Integrated Workflow

The five components integrate into the workflow detailed in Algorithm 1 in Appendix A. The Planning Agent applies **S** to identify edge cases before generating a plan. The Coding Agent translates the plan to code, followed by **I** to catch translation errors. The code enters the Live Verification loop: **O** generates property-based tests, **L** provides concrete execution feedback, and **D** accumulates failing tests. The algorithm terminates when code passes all sample I/Os and accumulated tests, or after exhausting $p = 5$ planning and $d = 5$ debugging iterations (following CodeSIM).

4 Experiments

4.1 Datasets

We adopt the evaluation protocol and problem subsets from CodeSIM (Islam et al., 2025), assessing generalization across three benchmarks that span a difficulty gradient:

- **HumanEval** (Chen et al., 2021): 164 function-level problems with short docstring specifications, often solvable in 1–2 lines (*easy*).
- **CodeContests** (Li et al., 2022): 165 Codeforces problems requiring algorithmic reasoning with

precise I/O specifications and ~ 200 hidden tests per problem (*medium*).

- **APPS** (Hendrycks et al., 2021): 150 problems featuring complex rule-based tasks with ambiguous natural language descriptions (*hard*).

We deliberately prioritize hypothesis fit over benchmark breadth. HumanEval, CodeContests, and APPS keep the task close to function-level or competitive-programming code generation, allowing us to observe verification failures while minimizing confounds from repository navigation, dependency resolution, and edit localization. Repository-level benchmarks such as SWE-bench (Jimenez et al., 2023) are important future validations, but they stress a partially different bottleneck—finding and coordinating edits across large codebases—than the missing-oracle verification problem studied here.

4.2 Baselines and Metric

We adopt the comparison set from CodeSIM (Islam et al., 2025), excluding Reflexion (Shinn et al., 2023) and LATS (Zhou et al., 2023) because they are not reproducible under the publicly released CodeSIM codebase (no matching code for identical conditions). Included baselines are prompting methods (Direct, CoT (Wei et al., 2022), Self-Planning (Jiang et al., 2024)), retrieval-augmented approaches (Analogical (Yasunaga et al., 2023)), and multi-agent frameworks: MapCoder (Islam et al., 2024) and CodeSIM (Islam et al., 2025). We report **pass@1**: a problem is solved if the first generated solution passes all hidden test cases.

4.3 Experimental Setup

We evaluate on three models: **GPT-4o** (gpt-4o-2024-08-06) as our primary baseline for direct comparison with prior work (Islam et al., 2024, 2025); **GPT-OSS-120B** (Agarwal et al., 2025), an open-source GPT model post-trained with RLVR (Wang et al., 2025); and **Grok-4.1-Fast** (xAI, 2025), a non-GPT frontier model also post-trained with RL, with strong math and coding performance. This selection tests generalization across three axes: proprietary vs. open-source, pre- vs. post-RLVR, and GPT vs. non-GPT architectures.

For GPT-4o, we use the OpenAI API² with temperature $\tau=0$ and top- $p=0.95$, adopting Hu-

manEval baselines from the CodeSIM paper and reproducing CodeContests/APPS with the official codebase.³ The additional models use default configurations from their OpenRouter⁴ API pages. All experiments follow CodeSIM hyperparameters ($p=5$ planning iterations, $d=5$ debugging iterations); SolidCoder uses $a=3$ for assumption breaking.

5 Results

5.1 Main Results

Table 2 presents results across three benchmarks. SolidCoder matches or outperforms CodeSIM in all nine model-benchmark combinations. On GPT-4o, SolidCoder achieves **95.7%** on HumanEval (+0.6%p), **77.0%** on CodeContests (+4.3%p), and **26.7%** on APPS (+3.4%p)—enabling direct comparison with the CodeSIM paper (Islam et al., 2025).

The pattern of improvements reveals a systematic relationship with benchmark difficulty. HumanEval, where current LLMs already exceed 97% average performance, shows minimal gains—the benchmark is approaching saturation, leaving little room for verification-based improvements. CodeContests, by contrast, exhibits the largest and most consistent gains across all models: GPT-4o +4.3%p, GPT-OSS-120B +4.2%p, and Grok-4.1-Fast +3.0%p (averaging +3.8%p). We attribute this to CodeContests occupying a “sweet spot” in difficulty—problems are complex enough that mental simulation becomes unreliable, yet remain tractable enough for execution-grounded verification to effectively catch and correct errors. Notably, Grok-4.1-Fast with SolidCoder reaches **98.2%**, approaching ceiling performance on this competitive programming benchmark.

On APPS, improvements vary by model: GPT-4o gains +3.4%p, while RL post-trained models show smaller improvements (GPT-OSS-120B +1.4%p, Grok-4.1-Fast +0.7%p). APPS poses such extreme difficulty that the bottleneck shifts from *verification* to *generation*—even perfect verification cannot salvage fundamentally flawed algorithms. However, the same RL models still show substantial gains on CodeContests (+4.2%p and +3.0%p), indicating that while RLVR improves code generation, models still rely on mental simulation when evaluating their outputs at inference time. SolidCoder’s live ex-

²<https://platform.openai.com/docs/models/gpt-4o>

³<https://github.com/kagnlp/CodeGenerator>

⁴<https://openrouter.ai>

Model	Direct	CoT	SelfPlanning	Analogical	MapCoder	CodeSIM	SolidCoder
HumanEval (Chen et al., 2021) (EASY)							
GPT-4o*	90.2	90.9	89.0	88.4	90.2	95.1	95.7
GPT-OSS-120B	69.5	96.3	90.8	89.0	61.0	98.2	98.2
Grok-4.1-Fast	88.4	96.9	96.9	95.7	95.7	97.6	97.6
<i>Average</i>	82.7	94.7	92.2	91.0	82.3	97.0	97.2
CodeContests (Li et al., 2022) (MEDIUM)							
GPT-4o*	42.4	44.2	49.1	30.3	69.1	72.7	77.0
GPT-OSS-120B	75.8	75.2	75.2	75.2	44.8	87.9	92.1
Grok-4.1-Fast	79.4	85.4	81.2	77.0	83.6	95.2	98.2
<i>Average</i>	65.9	68.3	68.5	60.8	65.8	85.3	89.1
APPS (Hendrycks et al., 2021) (HARD)							
GPT-4o*	10.7	17.3	14.7	14.0	20.7	23.3	26.7
GPT-OSS-120B	35.3	32.7	34.7	30.7	24.0	39.3	40.7
Grok-4.1-Fast	37.3	34.7	37.3	36.0	33.3	41.3	42.0
<i>Average</i>	27.8	28.2	28.9	26.9	26.0	34.6	36.5

Table 2: Pass@1 (%) across seven strategies and three models. Benchmarks are ordered by difficulty. *GPT-4o enables comparison with prior work; the remaining models are RL post-trained. SolidCoder consistently outperforms CodeSIM, with the largest gain on CodeContests (+3.8%p avg). **Bold**: best per row.

ecution provides value by replacing this unreliable self-assessment with concrete feedback. Overall, these results validate that the Mental-Reality Gap is a fundamental limitation of mental simulation itself, and execution-grounded verification provides consistent improvements across model families and training paradigms.

5.2 Ablation Study

To quantify the contribution of each S.O.L.I.D. component, we conduct a leave-one-out ablation study on CodeContests using GPT-4o, our primary model for comparison with prior work. We select this benchmark because it offers an appropriate difficulty level where component contributions are likely to be clearly distinguishable. On HumanEval, current LLMs approach saturation (95%+), which would make subtle variations difficult to observe reliably. Conversely, APPS poses such high difficulty that component differences may be obscured by uniformly low scores. CodeContests strikes a balance, providing sufficient headroom to reveal each component’s contribution.

Each ablation removes exactly one component from the full pipeline. In *w/o [L] Live Execution*, the concrete sandbox execution stage is replaced by mental assumption-breaking, where the model reasons about correctness through text-based tracing rather than actual execution; this isolates the value of execution grounding. In *w/o [O] Oracle-*

Configuration	Pass@1	Δ
GPT-4o Direct	42.4	−34.6
w/o Shift-left Planning	53.3	−23.7
w/o Intermediate Simulation	64.0	−13.0
w/o Oracle-based Assertions	65.4	−11.6
w/o Live Execution	69.1	−7.9
w/o Defensive Accumulation	70.3	−6.7
GPT-4o SolidCoder (Full)	77.0	–

Table 3: Ablation on CodeContests (GPT-4o). Δ indicates drop from Full. Components ranked by impact.

based Assertions, live execution is retained but tests rely on crash detection rather than model-generated property assertions; this isolates the contribution of oracle-style specifications.

Table 3 presents the results. The findings validate our two-dimensional analysis of the Mental-Reality Gap introduced in Section 1.

Specification Gap Dominates on CodeContests.

The largest drops come from specification-oriented components: removing [S] Shift-left Planning causes −23.7%p, and removing [I] Intermediate Simulation causes −13.0%p. Together, these account for the majority of SolidCoder’s improvement over the Direct baseline. This confirms that for algorithmic competition problems, *edge-case blindness* is the predominant failure mode—models generate solutions that work on typical inputs but fail on boundary conditions.

	GPT-4o*			GPT-OSS-120B			Grok-4.1-Fast		
	MapCoder	CodeSIM	SolidCoder	MapCoder	CodeSIM	SolidCoder	MapCoder	CodeSIM	SolidCoder
HumanEval (Chen et al., 2021) (EASY)									
Pass@1 (%)	90.2	95.1	95.7	61.0	98.2	98.2	95.7	97.0	97.6
API Calls	9	<u>4</u>	8	6	<u>3</u>	19	8	<u>4</u>	12
Tokens (K)	6.6	<u>3.8</u>	7.5	10.0	<u>5.6</u>	40.9	18.7	<u>14.3</u>	47.8
CodeContests (Li et al., 2022) (MEDIUM)									
Pass@1 (%)	69.1	72.7	77.0	44.8	87.9	92.1	83.6	95.2	98.2
API Calls	11	<u>11</u>	16	<u>4</u>	6	15	8	<u>4</u>	10
Tokens (K)	20.9	<u>20.9</u>	31.9	<u>11.5</u>	28.3	58.3	44.2	<u>35.0</u>	96.7
APPS (Hendrycks et al., 2021) (HARD)									
Pass@1 (%)	20.7	23.3	26.7	24.0	39.3	40.7	33.3	41.3	42.0
API Calls	<u>19</u>	26	35	<u>9</u>	21	48	15	<u>20</u>	47
Tokens (K)	<u>36.4</u>	49.3	60.4	<u>31.3</u>	73.3	177.4	97.5	<u>266.5</u>	520.9

Table 4: Efficiency comparison across models. Underline: most efficient; **bold**: highest Pass@1 per model-benchmark. GPT-OSS-120B shows degraded MapCoder performance due to XML parsing incompatibility. *Primary model for comparison with prior work.

Verification Gap Remains Critical. Despite smaller absolute contribution, [L] Live Execution’s -7.9% p drop represents cases where mental simulation would have *incorrectly validated buggy code*. These are precisely the “Mental-Reality Gap” failures that motivated our work: the model traces through incorrect logic and hallucinates a correct result. Without [L], such errors escape detection entirely. The 7.9% p improvement may appear modest relative to [S], but it reflects a categorically different error type that specification improvements cannot address.

Component Synergy. The full system (77.0%) substantially outperforms any single-component removal, indicating synergistic interaction. Edge-case identification ([S]) generates more comprehensive test scenarios, which live execution ([L]) then validates with concrete feedback. Neither component alone achieves comparable results: [S] without [L] still relies on hallucination-prone mental verification; [L] without [S] only tests the happy path that the model already handles well. The near-additivity of component contributions supports this interpretation—both components are essential for addressing both gaps.

Supporting Components. [O] Oracle-based Assertions (-11.6% p) enables property verification without ground truth—a prerequisite for [L] to function on novel inputs. [D] Defensive Accumulation (-6.7% p) prevents regression during iterative debugging, ensuring that discovered bugs stay fixed.

5.3 Efficiency Analysis

SolidCoder’s execution-grounded verification introduces additional cost. Table 4 compares efficiency across models.

On easy problems like HumanEval, SolidCoder’s overhead is inefficient: GPT-4o uses $+97\%$ more tokens for only $+0.6\%$ p gain. However, on problems where base LLMs struggle—CodeContests and APPS—the additional cost yields proportional improvements: $+50\%$ tokens for $+4.3\%$ p on CodeContests, $+23\%$ for $+3.4\%$ p on APPS. RL post-trained models show similar patterns, achieving consistent gains on CodeContests ($+4.2\%$ p for GPT-OSS, $+3.0\%$ p for Grok) despite higher token overhead.

This suggests that difficulty-aware routing could maximize SolidCoder’s value: applying lightweight paths to simple problems while reserving the full verification loop for complex tasks where mental simulation is most likely to fail. In this sense, SolidCoder can be viewed not only as a fixed pipeline but also as an inference-time framework whose verification intensity can be adapted to task difficulty.

5.4 Qualitative Example

Figure 2 concretizes the Mental-Reality Gap on a list rotation problem: mental simulation hallucinates correct behavior for buggy code, whereas SolidCoder’s Live Execution immediately exposes the fault through a concrete `AssertionError`. A complete end-to-end trace of the full S.O.L.I.D. pipeline, including all LLM inputs and outputs, is provided on a separate problem in Appendix D.

6 Conclusion and Future Work

We proposed **SolidCoder** with the **S.O.L.I.D.** architecture to bridge the Mental-Reality Gap through concrete execution, guided by a simple principle: *don't imagine—execute*. Our analysis showed that this gap manifests in two orthogonal dimensions—the Specification Gap and the Verification Gap—and our ablation indicates that [S] reduces the dominant failure mode in algorithmic tasks, [O] enables verification without ground-truth outputs, and [L] catches errors that specification improvement alone cannot remove. With GPT-4o, SolidCoder achieves **95.7%** on HumanEval, **77.0%** on CodeContests, and **26.7%** on APPS; these gains generalize across RL post-trained models and are most pronounced on medium-difficulty tasks where mental simulation is unreliable yet problems remain tractable. More broadly, our findings suggest that the next bottleneck in code generation lies not only in stronger generation but in stronger self-evaluation. Even though our experiments focus on function-level benchmarks, the underlying principle generalizes: as models become more capable generators, it matters increasingly how they falsify and revise their own hypotheses. Future work will explore difficulty-aware routing and more structural ways to reduce self-verification errors, and we release our code and framework to support that line of research.

Limitations

Our work has several limitations that point to future research directions.

First, evaluation scope. Live Execution currently supports Python only; extending to other languages requires language-specific sandboxing—a tractable engineering effort. Additionally, our evaluation focuses on function-level benchmarks that cleanly isolate the Mental-Reality Gap. Generalization to repository-level tasks (e.g., SWE-bench (Jimenez et al., 2023)) remains to be validated, though S.O.L.I.D. components could complement existing repository-level frameworks.

Second, LLM dependency. While Oracle-based Assertions reduce reliance on exact output prediction, hallucinated properties remain possible, and [O] is most effective when correctness can be partially expressed through structural constraints or invariants rather than near-exact outputs. Moreover, when the same LLM generates code, properties, and validates tests, systematic biases may

propagate. [L] partially mitigates this via model-independent feedback, but heterogeneous model combinations are an important future direction.

Third, experimental scope. We evaluate three models (GPT-4o, GPT-OSS-120B, Grok-4.1-Fast) but did not test Gemini or Claude due to cost constraints, and benchmark contamination cannot be ruled out—though identical evaluation conditions preserve relative comparisons. SolidCoder also consumes more tokens than CodeSIM (+50% on CodeContests, +97% on HumanEval), and our ablation covers only CodeContests with GPT-4o; compute-matched comparisons and cross-benchmark ablations remain for future work.

Ethical Considerations

Our work aims to improve the reliability of autonomous code generation, which inherently carries dual-use implications. While **SolidCoder** enhances developer productivity and debugging capabilities, we acknowledge that the same mechanisms could potentially be misused to generate malicious software with higher success rates.

To mitigate operational risks associated with **Live Execution** of untrusted model-generated code, we implemented sandboxing measures in our experiments, including network blocking and execution timeouts. We strongly emphasize that any real-world deployment of execution-based agents must incorporate robust, OS-level containment (e.g., containers or microVMs) to prevent sandbox escapes or resource exhaustion.

Furthermore, our framework relies on large language models, and thus generated code or test cases may reflect biases present in the training data. Users should treat the system's outputs as suggestions requiring human review, particularly in safety-critical applications. Finally, we utilized standard public benchmarks (HumanEval, CodeContests, APPS) that do not contain personally identifiable information.

Acknowledgements

This work was supported by Institute of Information and Communications Technology Planning and Evaluation(IITP) grant funded by the Korea government(MSIT) (RS-2019-II190004, Development of semi-supervised learning language intelligence technology and Korean tutoring service for foreigners).

References

- Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altschmidt, Sam Altman, Shyamal Anadkat, et al. 2023. GPT-4 technical report. *arXiv preprint arXiv:2303.08774*.
- Sandhini Agarwal, Lama Ahmad, Jason Ai, Sam Altman, Andy Applebaum, Edwin Arbus, Rahul K Arora, Yu Bai, Bowen Baker, Haiming Bao, et al. 2025. GPT-OSS-120B & GPT-OSS-20B model card. *arXiv preprint arXiv:2508.10925*.
- Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Unified pre-training for program understanding and generation. In *Proceedings of the 2021 conference of the North American chapter of the association for computational linguistics: human language technologies*, pages 2655–2668.
- Anthropic. 2024. **The Claude 3 model family: Opus, Sonnet, Haiku**. Model card, Anthropic. Accessed: 2025-12-15.
- Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.
- Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. CodeBERT: A pre-trained model for programming and natural languages. In *Findings of the association for computational linguistics: EMNLP 2020*, pages 1536–1547.
- Yuyao Ge, Lingrui Mei, Zenghao Duan, Tianhao Li, Yujia Zheng, Yiwei Wang, Lexin Wang, Jiayu Yao, Tianyu Liu, Yujun Cai, et al. 2025. A survey of vibe coding with large language models. *arXiv preprint arXiv:2510.12399*.
- Sumit Gulwani. 2011. Automating string processing in spreadsheets using input-output examples. *ACM Sigplan Notices*, 46(1):317–330.
- Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Peiyi Wang, Qihao Zhu, Runxin Xu, Ruoyu Zhang, Shirong Ma, Xiao Bi, et al. 2025. DeepSeek-R1: Incentivizing reasoning capability in LLMs via reinforcement learning. *arXiv preprint arXiv:2501.12948*.
- Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Yifan Wu, YK Li, et al. 2024. DeepSeek-Coder: When the large language model meets programming—the rise of code intelligence. *arXiv preprint arXiv:2401.14196*.
- Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, et al. 2021. Measuring coding challenge competence with APPS. *arXiv preprint arXiv:2105.09938*.
- Lei Huang, Weijiang Yu, Weitao Ma, Weihong Zhong, Zhangyin Feng, Haotian Wang, Qianglong Chen, Weihua Peng, Xiaocheng Feng, Bing Qin, et al. 2025. A survey on hallucination in large language models: Principles, taxonomy, challenges, and open questions. *ACM Transactions on Information Systems*, 43(2):1–55.
- Binyuan Hui, Jian Yang, Zeyu Cui, Jiayi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, et al. 2024. Qwen2.5-Coder technical report. *arXiv preprint arXiv:2409.12186*.
- Aaron Hurst, Adam Lerer, Adam P Goucher, Adam Perelman, Aditya Ramesh, Aidan Clark, AJ Ostrom, Akila Welihinda, Alan Hayes, Alec Radford, et al. 2024. GPT-4o system card. *arXiv preprint arXiv:2410.21276*.
- Md Ashraful Islam, Mohammed Eunus Ali, and Md Rizwan Parvez. 2024. MapCoder: Multi-agent code generation for competitive problem solving. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 4912–4944.
- Md Ashraful Islam, Mohammed Eunus Ali, and Md Rizwan Parvez. 2025. CodeSIM: Multi-agent code generation and problem solving through simulation-driven planning and debugging. In *Findings of the Association for Computational Linguistics: NAACL 2025*, pages 5113–5139.
- Aaron Jaech, Adam Kalai, Adam Lerer, Adam Richardson, Ahmed El-Kishky, Aiden Low, Alec Helyar, Aleksander Madry, Alex Beutel, Alex Carney, et al. 2024. OpenAI o1 system card. *arXiv preprint arXiv:2412.16720*.
- Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. 2024. LiveCodeBench: Holistic and contamination free evaluation of large language models for code. *arXiv preprint arXiv:2403.07974*.
- Xue Jiang, Yihong Dong, Lecheng Wang, Zheng Fang, Qiwei Shang, Ge Li, Zhi Jin, and Wenpin Jiao. 2024. Self-planning code generation with large language models. *ACM Transactions on Software Engineering and Methodology*, 33(7):1–30.
- Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2023. SWE-bench: Can language models resolve real-world GitHub issues? *arXiv preprint arXiv:2310.06770*.
- Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. 2022. Competition-level code generation with AlphaCode. *Science*, 378(6624):1092–1097.
- Aixin Liu, Aoxue Mei, Bangcai Lin, Bing Xue, Bingxuan Wang, Bingzheng Xu, Bochao Wu, Bowei Zhang,

- Chaofan Lin, Chen Dong, et al. 2025. DeepSeek-v3.2: Pushing the frontier of open large language models. *arXiv preprint arXiv:2512.02556*.
- Zohar Manna and Richard J Waldinger. 1971. Toward automatic program synthesis. *Communications of the ACM*, 14(3):151–165.
- Md Rizwan Parvez, Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Retrieval augmented code generation and summarization. In *Findings of the Association for Computational Linguistics: EMNLP 2021*, pages 2719–2734.
- Chen Qian, Wei Liu, Hongzhang Liu, Nuo Chen, Yufan Dang, Jiahao Li, Cheng Yang, Weize Chen, Yusheng Su, Xin Cong, et al. 2024. ChatDev: Communicative agents for software development. In *Proceedings of the 62nd annual meeting of the association for computational linguistics (volume 1: Long papers)*, pages 15174–15186.
- Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, et al. 2023. Code Llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*.
- Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. 2023. Toolformer: Language models can teach themselves to use tools. *Advances in neural information processing systems*, 36:68539–68551.
- Yuling Shi, Songsong Wang, Chengcheng Wan, Min Wang, and Xiaodong Gu. 2024. From code to correctness: Closing the last mile of code generation with hierarchical debugging. *arXiv preprint arXiv:2410.01215*.
- Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. Reflexion: Language agents with verbal reinforcement learning. *Advances in neural information processing systems*, 36:8634–8652.
- KaShun Shum, Shizhe Diao, and Tong Zhang. 2023. Automatic prompt augmentation and selection with chain-of-thought from labeled data. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pages 12113–12139.
- Gemini Team, Rohan Anil, Sebastian Borgeaud, Jean-Baptiste Alayrac, Jiahui Yu, Radu Soricut, Johan Schalkwyk, Andrew M Dai, Anja Hauth, Katie Millican, et al. 2023. Gemini: a family of highly capable multimodal models. *arXiv preprint arXiv:2312.11805*.
- Kimi Team, Yifan Bai, Yiping Bao, Y Charles, Cheng Chen, Guanduo Chen, Haiting Chen, Huarong Chen, Jiahao Chen, Ningxin Chen, et al. 2025. Kimi K2: Open agentic intelligence. *arXiv preprint arXiv:2507.20534*.
- Yiping Wang, Qing Yang, Zhiyuan Zeng, Liliang Ren, Liyuan Liu, Baolin Peng, Hao Cheng, Xuehai He, Kuan Wang, Jianfeng Gao, et al. 2025. Reinforcement learning for reasoning in large language models with one training example. *arXiv preprint arXiv:2504.20571*.
- Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837.
- xAI. 2025. **Grok-4 model card**. Model card, xAI. File-name: 2025-08-20-grok-4-model-card.pdf. Accessed: 2025-12-15.
- An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. 2025. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*.
- Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R Narasimhan, and Yuan Cao. 2022. ReAct: Synergizing reasoning and acting in language models. In *The eleventh international conference on learning representations*.
- Michihiro Yasunaga, Xinyun Chen, Yujia Li, Panupong Paspapat, Jure Leskovec, Percy Liang, Ed H Chi, and Denny Zhou. 2023. Large language models as analogical reasoners. *arXiv preprint arXiv:2310.01714*.
- Zhuosheng Zhang, Aston Zhang, Mu Li, and Alex Smola. 2022. Automatic chain of thought prompting in large language models. *arXiv preprint arXiv:2210.03493*.
- Li Zhong, Zilong Wang, and Jingbo Shang. 2024. Debug like a human: A large language model debugger via verifying runtime execution step by step. In *Findings of the Association for Computational Linguistics: ACL 2024*, pages 851–870.
- Andy Zhou, Kai Yan, Michal Shlapentokh-Rothman, Haohan Wang, and Yu-Xiong Wang. 2023. Language agent tree search unifies reasoning acting and planning in language models. *arXiv preprint arXiv:2310.04406*.

A Algorithm Details

Algorithm 1 presents the complete SolidCoder workflow. The algorithm consists of three main stages: (1) edge case identification and plan generation via Shift-left Planning, (2) verification through Oracle tests and Live Execution in the Assumption Breaking Phase, and (3) iterative code refinement in the Traditional Debugging Phase. Each S.O.L.I.D. component is marked in **blue**.

Algorithm 1 The SolidCoder Workflow

Require: Problem P , Sample I/O T , max_plan p , max_debug d , max_assume a
Ensure: Solution Code S

```
1: accumulated  $\leftarrow \emptyset$ 
2: for  $i = 1$  to  $p$  do
3:   [S] edges  $\leftarrow$  IDENTIFYEDGECASES( $P$ )
4:   plan  $\leftarrow$  GENERATEPLAN( $P$ , edges)
5:   valid  $\leftarrow$  SIMULATEPLAN( $P$ , plan)  $\triangleright$  Mental Verification
6:   if  $\neg$ valid then
7:     plan  $\leftarrow$  REFINEPLAN( $P$ , plan)
8:   end if
9:    $S \leftarrow$  GENERATECODE( $P$ , plan)
10:  [I]  $S \leftarrow$  INTERMEDIATESIMULATION( $P$ ,  $S$ )  $\triangleright$  Complementary filter
11:  for  $k = 1$  to  $a$  do  $\triangleright$  Assumption Breaking Phase
12:    [O] test  $\leftarrow$  GENERATEORACLETEST( $P$ ,  $S$ )
13:    [L] result  $\leftarrow$  LIVEEXECUTE( $S$ , test)
14:    if result = FAIL then
15:      [D] accumulated  $\leftarrow$  accumulated  $\cup$  {test}
16:       $S \leftarrow$  FIXCODE( $P$ ,  $S$ , test)
17:    else
18:      break  $\triangleright$  Verification Passed
19:    end if
20:  end for
21:  if PASSALL( $S$ ,  $T \cup$  accumulated) then
22:    return  $S$ 
23:  end if
24:  for  $j = 1$  to  $d$  do  $\triangleright$  Traditional Debugging Phase
25:     $S \leftarrow$  DEBUGCODE( $P$ , plan,  $S$ )
26:    if PASSALL( $S$ ,  $T \cup$  accumulated) then
27:      return  $S$ 
28:    end if
29:  end for
30: end for
31: return  $S$ 
```

B S.O.L.I.D. Component Prompts

This appendix presents the complete prompts used in SolidCoder’s S.O.L.I.D. components. Variables in curly braces (e.g., {problem}) are replaced with actual values at runtime.

B.1 [S] Shift-left Planning Prompt

The Shift-left prompt extends standard planning by requiring edge case identification *before* algorithm design:

[S] Shift-left Planning Prompt

```
You are an expert software engineer.
You are given a problem and you need to
generate a detailed plan to solve it.

## Problem
{problem}

## Killer Edge Cases
Identify 3 potential edge cases that could
break a naive solution:
1. Empty/minimal input
2. Maximum constraint input
3. Special pattern (all same, alternating,
etc.) or Boundary values

## Plan
Write a detailed plan that handles these
edge cases.
The plan should be step-by-step and easy to
implement.
```

B.2 [O] Oracle-based Assertion Prompt

The Oracle prompt instructs the LLM to act as a “Red Team” tester, generating property-based assertions rather than exact output predictions:

[O] Oracle-based Assertion Prompt

```
You are an expert software tester (Red
Team).
Your goal is to break the following code by
finding hidden assumptions.

## Problem
{problem}

## Code
“{language}
{code}
“

## Your Task
1. Identify a weak assumption (Type, Value,
Structure, or Relationship).
2. Produce a **Python test script** that
calls the target function with a breaking
input.
3. Include an assert that would fail if the
assumption is violated.

Format:
Assumption: <short text>
Test Script:
“python
# call the function defined above
result = <call>
assert <oracle about result>
“
```

B.3 [L] Live Execution

Live Execution does not use a separate prompt—it directly executes the test scripts generated by the Oracle component in a sandboxed Python environment. The execution wrapper is shown below:

[L] Live Execution Wrapper

```
def _concrete_verify_script(self, code,
test_script, timeout=5):
    full_code = f"""
    import sys
    import math
    from typing import List, Dict, Any,
Optional, Union, Tuple
    {code}
    {test_script}
    """
    try:
        safe_builtins = builtins.__dict__.copy()
        safe_builtins["input"] = lambda *_ , **__:
\
        ( _ for _ in
        ()).throw(RuntimeError("input() disabled"))
        exec_globals = {"__builtins__":
safe_builtins}
        function_with_timeout(exec, (full_code,
exec_globals), timeout)
        return "PASS"
    except AssertionError:
        return "FAIL_ASSERT"
    except Exception:
        return "FAIL_CRASH"
```

See Appendix C for detailed safety mechanisms.

B.4 [I] Intermediate Simulation Prompt

The Intermediate Simulation prompt asks the LLM to mentally trace code execution before live verification:

[I] Intermediate Simulation Prompt

```
## Problem
{problem}

## Code
“{language}
{code}
“

## Your Task: Mental Execution
1. Select a sample input.
2. Trace the code execution step-by-step.
3. Track variable values.
4. Predict the final output.

If you find a logic error or
mismatch with the plan, output:
**CODE_SIMULATION_FAILED**
If the code seems correct, output:
**CODE_SIMULATION_PASSED**
```

B.5 [D] Defensive Accumulation

Defensive Accumulation is implemented as a runtime mechanism rather than a prompt. When Live Execution detects a failing test case, the test is automatically added to an `accumulated_tests` collection. All subsequent code modifications must pass *every* accumulated test before being accepted:

[D] Defensive Accumulation Logic

```
# In the assumption breaking loop:
verify_status =
self._concrete_verify_script(code,
test_script)
if verify_status in {"FAIL_CRASH",
"FAIL_ASSERT"}:
    # [D] Defensive Accumulation: store failing
test
    if self.enable_defensive_test:
        self.accumulated_inputs.append(test_script)
        additional_io.append(test_script)
    # Fix the code based on the failing test
code = self._fix_with_test_script(problem,
code, test_script)
    continue # Re-verify with accumulated tests
```

B.6 Test Case Validation (Judge Prompt)

To prevent false positives from invalid test cases, SolidCoder includes a Judge component that validates generated tests:

Judge Prompt

```
You are an impartial Judge.

## Problem
{problem}

## Proposed Test Case
“python
{test_script}
“

## Task
Determine if this test case is VALID and
CORRECT for the given problem.
1. Does the input satisfy all constraints
(e.g. range, type, format)?
2. Is the asserted output (if any) logically
correct?
3. Is it a fair test?

If the test case is valid, output: **VALID**
If it violates constraints or expects wrong
output, output: **INVALID**
```

B.7 Code Fix Prompt

When a vulnerability is found, SolidCoder uses this prompt to fix the code:

Code Fix Prompt

```
## Problem
{problem}

## Code
“{language}
{code}
“

## Vulnerability Found
The following test script failed (Crash or
Assertion Error):
“python
{test_script}
“

## Task
Fix the code to handle this edge case.
Wrap the code in “{language} ... “ block.
```

C Live Execution Safety Mechanisms

SolidCoder implements several safety mechanisms for Live Execution to prevent malicious or buggy code from affecting the host system. Table 5 summarizes these mechanisms. Full implementation details are provided in the [L] [Live Execution](#) code listing in Appendix B.

Mechanism	Implementation	Purpose & Rationale
Timeout	5-second limit per test via <code>function_with_timeout</code>	Prevents infinite loops from blocking the pipeline. Correct solutions complete in milliseconds; 5s provides margin while catching runaway executions.
Network Blocking	<code>block_network=True</code> (default)	Prevents generated code from making outbound network calls, eliminating data exfiltration and external dependency risks during sandboxed execution.
Input Blocking	<code>input()</code> → <code>RuntimeError</code> stub	Prevents scripts from hanging indefinitely waiting for user input that will never arrive in automated execution.
Isolated Namespace	Fresh <code>exec_globals</code> per execution	Prevents test scripts from interfering with each other or accumulating state. Each test starts with a clean environment.
Exception Classification	<code>PASS</code> , <code>FAIL_ASSERT</code> , <code>FAIL_CRASH</code>	Distinguishes logical failures (<code>AssertionError</code>) from runtime bugs. Both trigger Defensive Accumulation with different diagnostic signals.

Table 5: Safety mechanisms implemented in SolidCoder’s Live Execution component to ensure reliable and secure code evaluation.

D Complete Example

This section presents a complete execution trace of SolidCoder solving HumanEval problem #0 (has_close_elements). The trace demonstrates the full S.O.L.I.D. pipeline: **[S]** Shift-left Planning identifies edge cases before generating the plan; **[I]** Intermediate Simulation traces the generated code mentally; **[O]** Oracle-based Assumption Breaking generates adversarial test cases; and **[L]** Live Execution validates correctness through actual runtime. In this example, the code passes on the first attempt, so **[D]** Defensive Accumulation is not triggered. All prompts and LLM responses are shown verbatim from the experiment log.

Complete Execution Trace: HumanEval/0 (has_close_elements) with GPT-4o

Problem

```
from typing import List

def has_close_elements(numbers: List[float], threshold: float) -> bool:
    """ Check if in given list of numbers, are any two numbers closer to each
        other than given threshold.
    """
    >>> has_close_elements([1.0, 2.0, 3.0], 0.5)
    False
    >>> has_close_elements([1.0, 2.8, 3.0, 4.0, 5.0, 2.0], 0.3)
    True
    """
```

[S] Shift-left Planning

Prompt:

You are an expert software engineer.
You are given a problem and you need to generate a detailed plan to solve it.

```
## Problem
[problem code shown above]

## Killer Edge Cases
Identify 3 potential edge cases that could break a naive solution:
1. Empty/minimal input
2. Maximum constraint input
3. Special pattern (all same, alternating, etc.) or Boundary values
```

```
## Plan
Write a detailed plan that handles these edge cases.
The plan should be step-by-step and easy to implement.
```

LLM Response:

To solve the problem of determining if any two numbers in a list are closer to each other than a given threshold, we need to consider potential edge cases and develop a robust plan. Here's a detailed plan to address the problem and handle edge cases:

Edge Cases

1. Empty/Minimal Input:

- An empty list or a list with a single element should return False because there are no pairs of

numbers to compare.

2. Maximum Constraint Input:

- Consider the scenario where the list contains the maximum number of elements allowed by the system's memory. The solution should be efficient in terms of time and space complexity to handle large inputs.

3. Special Pattern or Boundary Values:

- Lists where all elements are the same, which should return True if the threshold is greater than zero.
- Lists with alternating values or values that are very close to each other but not within the threshold.
- Lists with very large or very small floating-point numbers to test precision and handling of floating-point arithmetic.

Plan

1. Initial Checks:

- If the list `numbers` is empty or contains only one element, return `False` immediately since no pairs exist.

2. Sorting:

- Sort the list of numbers. Sorting helps in efficiently finding close elements because if two numbers are close, they will be adjacent in a sorted list.

3. Iterate and Compare:

- Iterate through the sorted list and compare each pair of adjacent elements.
- For each pair (`numbers[i]`, `numbers[i+1]`), calculate the absolute difference.
- If the difference is less than the threshold, return `True`.

4. Return Result:

- If no such pair is found after checking all adjacent pairs, return `False`.

5. Complexity Consideration:

- Sorting the list takes $O(n \log n)$ time, and iterating through the list takes $O(n)$ time. Thus, the overall time complexity is $O(n \log n)$, which is efficient for large inputs.

6. Precision Handling:

- Ensure that floating-point arithmetic is handled correctly by using Python's built-in functions which are designed to manage floating-point precision.

Plan Simulation

Prompt:

You are a programmer tasked with verifying a plan to solve a given problem using the `Python3` programming language.

[Problem and Plan shown above]

`**Expected Output:**`

Your response must be structured as follows:

`### Simulation`

- Take a sample input and apply plan step by step to get the output.
- Compare the generated output with the sample output to verify if your plan works as expected.

`### Plan Evaluation`

- If the simulation is successful write `**No Need to Modify Plan**`.
- Otherwise write `**Plan Modification Needed**`.

LLM Response:

`### Simulation`

Let's simulate the plan with the provided test cases to verify its correctness.

Test Case 1: numbers = [1.0, 2.0, 3.0], threshold = 0.5

1. **Initial Checks:** The list has more than one element, so we proceed.
2. **Sorting:** The list is already sorted: [1.0, 2.0, 3.0].
3. **Iterate and Compare:**
 - Compare 1.0 and 2.0: $|1.0 - 2.0| = 1.0$, which is not less than 0.5.
 - Compare 2.0 and 3.0: $|2.0 - 3.0| = 1.0$, which is not less than 0.5.
4. **Return Result:** No pairs found with a difference less than 0.5, so return False.

Output: False (Matches expected output)

Test Case 2: numbers = [1.0, 2.8, 3.0, 4.0, 5.0, 2.0], threshold = 0.3

1. **Initial Checks:** The list has more than one element, so we proceed.
2. **Sorting:** Sort the list: [1.0, 2.0, 2.8, 3.0, 4.0, 5.0].
3. **Iterate and Compare:**
 - Compare 1.0 and 2.0: $|1.0 - 2.0| = 1.0$, which is not less than 0.3.
 - Compare 2.0 and 2.8: $|2.0 - 2.8| = 0.8$, which is not less than 0.3.
 - Compare 2.8 and 3.0: $|2.8 - 3.0| = 0.2$, which is less than 0.3.
4. **Return Result:** A pair (2.8, 3.0) is found with a difference less than 0.3, so return True.

Output: True (Matches expected output)

`### Plan Evaluation`

The simulation of the plan with the provided test cases shows that the implementation works as expected. The plan correctly identifies whether there are any two numbers in the list that are closer to each other than the given threshold.

No Need to Modify Plan

Code Generation

Prompt:

You are a programmer tasked with solving a given problem using the `**Python3**` programming language. See the plan to solve the plan and implement code.

[Problem and Plan shown above]

`**Important Instructions:**`

- Do not add any explanation.
- The generated `**Python3**` code must be inside a triple backtick (`````) block.

LLM Response:

```
from typing import List

def has_close_elements(numbers: List[float], threshold: float) -> bool:
    # Step 1: Initial check for minimal input
    if len(numbers) < 2:
        return False

    # Step 2: Sort the list
    numbers.sort()

    # Step 3: Iterate and compare adjacent elements
    for i in range(len(numbers) - 1):
        if abs(numbers[i] - numbers[i + 1]) < threshold:
            return True

    # Step 4: Return result if no close elements found
    return False
```

[I] Intermediate Simulation

Prompt:

`## Problem`

[problem code]

`## Code`

[generated code]

`## Your Task: Mental Execution`

1. Select a sample input.
2. Trace the code execution step-by-step.
3. Track variable values.
4. Predict the final output.

If you find a logic error or mismatch with the plan,

output: `**CODE_SIMULATION_FAILED**`

If the code seems correct,

output: `**CODE_SIMULATION_PASSED**`

LLM Response:

Let's select a sample input and trace the code execution step-by-step.

Sample Input:

```
numbers = [1.0, 2.8, 3.0, 4.0, 5.0, 2.0]
threshold = 0.3
```

Code Execution:**1. Initial Check:**

- The code checks if the length of numbers is less than 2.
- $\text{len}(\text{numbers}) = 6$, so the check fails, and we proceed to the next step.

2. Sorting the List:

- The list numbers is sorted.
- Sorted numbers = [1.0, 2.0, 2.8, 3.0, 4.0, 5.0].

3. Iterate and Compare Adjacent Elements:

We iterate over the sorted list and compare adjacent elements.

- **Iteration 1** ($i = 0$):
 - Compare numbers[0] and numbers[1]: $|1.0 - 2.0| = 1.0$
 - $1.0 \geq 0.3$, so continue to the next iteration.
- **Iteration 2** ($i = 1$):
 - Compare numbers[1] and numbers[2]: $|2.0 - 2.8| = 0.8$
 - $0.8 \geq 0.3$, so continue to the next iteration.
- **Iteration 3** ($i = 2$):
 - Compare numbers[2] and numbers[3]: $|2.8 - 3.0| = 0.2$
 - $0.2 < 0.3$, so the condition is satisfied, and the function returns True.

4. Return Result:

- Since we found two numbers (2.8 and 3.0) that are closer than the threshold (0.3), the function returns True.

Conclusion: The code correctly identifies that there are two numbers in the list that are closer to each other than the given threshold. The logic and execution match the expected behavior.

Output: CODE_SIMULATION_PASSED

[O] Oracle-based Assumption Breaking (Round 1/3)**Prompt:**

You are an expert software tester (Red Team).
Your goal is to break the following code by finding hidden assumptions.

```
## Problem
[problem code]
```

```
## Code
```

[generated code]

Your Task

1. Identify a weak assumption (Type, Value, Structure, or Relationship).
2. Produce a **Python test script** that calls the target function with a breaking input.
3. Include an assert that would fail if the assumption is violated.

Format:

Assumption: <short text>

Test Script:

```
```python
result = <call>
assert <oracle about result>
```
```

LLM Response:

Assumption: The function assumes that the input list numbers contains only valid floating-point numbers and does not handle special floating-point values like NaN (Not a Number) or inf (infinity), which can lead to unexpected behavior.

Test Script:

```
# call the function defined above
result = has_close_elements([1.0, float('nan'), 3.0], 0.5)
assert result == False, "The function should return False when the list
contains NaN, as NaN comparisons are not valid."
```

```
# call the function defined above
result = has_close_elements([1.0, float('inf'), 3.0], 0.5)
assert result == False, "The function should return False when the list
contains infinity, as infinity cannot be close to any finite number."
```

In this test script, we are testing the function with lists that include NaN and inf. The function should ideally handle these cases gracefully, but due to the nature of these special floating-point values, the current implementation may not behave as expected. The assertions are designed to fail if the function does not correctly handle these cases.

[L] Live Execution + Judge

Judge Prompt:

You are an impartial Judge.

Problem

[problem code]

Proposed Test Case

[test script from Red Team]

Task

Determine if this test case is VALID and CORRECT for the given problem.

1. Does the input satisfy all constraints (e.g. range, type, format)?
2. Is the asserted output (if any) logically correct?

3. Is it a fair test?

If the test case is valid, output: ****VALID****

If it violates constraints or expects wrong output, output: ****INVALID****

Judge Response:

To determine if the proposed test cases are valid and correct for the given problem, let's analyze each aspect:

1. Input Constraints: The function `has_close_elements` accepts a list of floats and a float threshold. The proposed test cases use lists containing `float('nan')` and `float('inf')`, which are valid float values in Python. Therefore, the input satisfies the type constraints.

2. Logical Correctness of the Asserted Output:

- For the test case with `float('nan')`: In Python, any comparison with NaN (Not a Number) is always false, including equality and inequality checks. Therefore, the presence of NaN should not affect the determination of whether any two numbers are closer than the threshold. The function should return `False` as no valid comparisons can be made with NaN.
- For the test case with `float('inf')`: Infinity is not a finite number, and it cannot be close to any finite number within a finite threshold. Therefore, the function should return `False` as no finite number can be within a finite threshold of infinity.

3. Fairness of the Test: The test cases are fair as they check the function's behavior with special float values (NaN and Infinity), which are edge cases that can occur in floating-point computations. Given these points, the proposed test cases are logically correct and do not violate any constraints. They are valid tests for the function's behavior with special float values.

Verdict: VALID

Live Execution Result:

Executing test script with generated code...

```
def _concrete_verify_script(code, test_script, timeout=5):
    full_code = f"""
    {code}
    {test_script}
    """
    try:
        exec(full_code, exec_globals)
        return "PASS"
    except AssertionError:
        return "FAIL_ASSERT"
    except Exception:
        return "FAIL_CRASH"
```

Result: PASS (no assertion errors, no crashes)

Final Result

PASSED after self-check

Total API Calls: 6
Total Time: 20.8 seconds
Result: Solved (1/1 = 100%)

The generated code correctly handles all edge cases including NaN and infinity without requiring any fixes. This demonstrates how SolidCoder's Live Execution component grounds verification in runtime reality rather than relying on mental simulation alone. The Judge correctly validated the Red Team's test cases as fair, and the Live Execution confirmed the code's robustness.