Enhancing Code Generation in Large Language Models: A Human-Like Debugging Approach for Competitive Problem Solving

Anonymous ACL submission

Abstract

Recent advances in Large Language Models (LLMs) have shown impressive code generation capabilities, yet their reasoning often collapses when faced with algorithmically complex or error-sensitive programming tasks. This gap stems not from limited knowledge, but from a lack of self-diagnostic awareness—the ability to understand why a piece of code fails and how to repair it. We argue that effective code generation requires integrating cognitive processes analogous to human debugging: perceiving failures, tracing intermediate states, hypothesizing causes, and revising accordingly. Building on this perspective, we propose a Human-Like Debugging Architecture (HLDA) that operationalizes these processes within LLMs. HLDA systematically executes generated code, records fine-grained variable traces, interprets discrepancies, and iteratively regenerates improved solutions. Unlike prior self-editing or executor-based approaches that rely on outcome-level feedback, our method introduces trace-level causal feedback, enabling models to reason over "how the code went wrong" rather than merely "what went wrong." This leads to substantial performance gains, setting new state-of-the-art results on HumanEval, MBPP, and contest-level benchmarks such as APPS and CodeContest.

1 Introduction

Large Language Models (LLMs) have rapidly evolved from natural language generators into powerful semantic reasoners capable of producing executable code. Yet despite remarkable fluency, most LLMs still lack the self-awareness and reasoning granularity characteristic of human debugging. When their code fails, the difficulty lies not in missing algorithmic knowledge, but in the inability to observe, interpret, and reason about their own execution process. In particular, existing models often know that a program fails, but struggle to identify how the failure emerges from intermediate execution states.

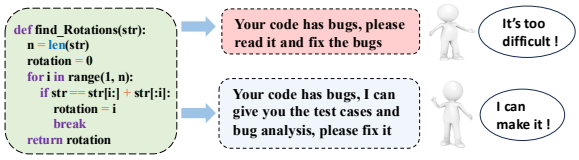

Figure 1: When code contains bugs, simply reporting failure is insufficient. Providing execution results, test cases, and bug analysis significantly facilitates effective debugging.

Recent frameworks such as AgentCoder (Huang et al., 2023), MetaGPT (Hong et al., 2023), and MapCoder (Islam et al., 2024) improve LLM-based code generation through agentic coordination, modular planning, or execution-guided refinement. Execution-aware debugging approaches, including Teaching Large Language Models to Self-Debug (Chen et al., 2023), Debug Like a Human (Zhong et al., 2024), and Leveraging Print Debugging (Hu et al., 2024), further incorporate runtime signals to guide error correction. Despite these advances, most methods still frame debugging as an iterative *search over candidate outputs* or high-level fixes driven by execution outcomes. They primarily use binary pass/fail signals or coarse textual feedback, and often lack explicit mechanisms to reason over fine-grained execution dynamics—such as intermediate variable trajectories, loop invariants, or control-flow decisions—required to causally localize root failures.

We take a different stance: debugging is not merely an optimization step, but an epistemic process of hypothesis testing over program behavior. Human programmers form causal links between observed failures and internal execution states (e.g., a premature counter termination indicating an incorrect loop condition). To emulate this introspective process, we propose a **Human-Like Debugging Architecture (HLDA)** that transforms code generation from one-shot synthesis into an iterative

causal reasoning loop.

Our architecture consists of six interconnected modules: Planning, Code Generation, Execution with variable tracing, Debugging with semantic validation, Feedback-driven regeneration, and a pseudo-test mechanism. Crucially, the novelty of HLDA lies not simply in orchestrating these modules, but in redesigning the feedback signal for code repair. HLDA treats execution traces as first-class reasoning evidence and explicitly connects variable trajectories, control-flow behaviors, and failure contexts to the code logic that causes them. By leveraging these trace-level signals, the model develops a causal understanding of its own program behavior rather than relying only on outcome-level trial-and-error refinement.

This perspective leads to three key insights:

- **From correctness to cognition:** Reasoning over intermediate states enables logic-level understanding beyond final outcomes.
- **From reflection to causal introspection:** Rather than post-hoc reflection, the model analyzes execution trajectories to localize root causes and validate repair hypotheses.
- **From agent orchestration to feedback design:** HLDA shifts code repair from outcome supervision toward execution-grounded causal feedback.

Through extensive experiments on HumanEval, MBPP, and contest-level datasets, we show that HLDA achieves superior Pass@1 accuracy with competitive computational cost. More importantly, it provides interpretability through structured, execution-grounded feedback, revealing how trace-level causal introspection can advance self-corrective code generation systems.

As illustrated in Fig. 1, vague feedback such as *"your code has bugs"* rarely helps humans identify errors, whereas access to execution traces, test cases, and intermediate states enables effective causal reasoning. Existing LLM-based systems resemble the former scenario, relying on binary feedback and often producing superficial fixes. HLDA is designed to emulate the latter, explicitly exposing execution-level diagnostics and semantic reasoning cues to enable targeted, interpretable corrections instead of trial-and-error refinement. In this sense, HLDA is not merely an integration of existing agentic components, but a feedback signal design paradigm that moves from outcome-level supervision to execution-grounded causal supervision.

2 Related Works

Employing LLM as an Agent Recent work has increasingly explored using LLMs as agents that interact with objectives, environments, and feedback to support decision-making. Chain-of-Thought (Wei et al., 2022) and Tree-of-Thought (Yao et al., 2023) encourage models to decompose problems into intermediate reasoning steps, while ReAct (Yao et al., 2022) further combines reasoning with action generation. Other studies investigate reinforcement-learning-based feedback mechanisms to improve agent behavior (Chang et al., 2023; Hu et al., 2023). LLM agents have been applied to diverse domains, including games (Qiao et al., 2023; Akoury et al., 2023), customer service (Cai et al., 2023; Soni, 2023), and code generation (Ni et al., 2023; Zhou et al., 2023). These works demonstrate the potential of agentic LLMs, but often rely on high-level environmental feedback rather than fine-grained execution-state reasoning.

Code Generation Code generation has evolved from rule-based systems (Woods, 1973) and neural sequence models (Xiao et al., 2016) to large language models capable of producing executable programs. Recent methods improve reliability by incorporating execution feedback and iterative refinement. For example, LEVER (Ni et al., 2023) and LATS (Zhou et al., 2023) use program execution to guide candidate selection or search, while ReAct (Yao et al., 2022) interleaves reasoning, action, and environment feedback. Beyond final-output validation, several works leverage runtime information for debugging. Self-Debugging (Chen et al., 2023) enables LLMs to inspect errors and revise code, Debug Like a Human (Zhong et al., 2024) verifies intermediate execution states, and Print Debugging (Hu et al., 2024) exposes execution traces to improve correction. However, most methods still depend mainly on final outcomes or coarse feedback, leaving error repair largely heuristic. In contrast, our work models debugging as execution-grounded causal reasoning, using variable-level traces, specification-consistent pseudo tests, and structured feedback to localize root causes and enable more interpretable code correction.

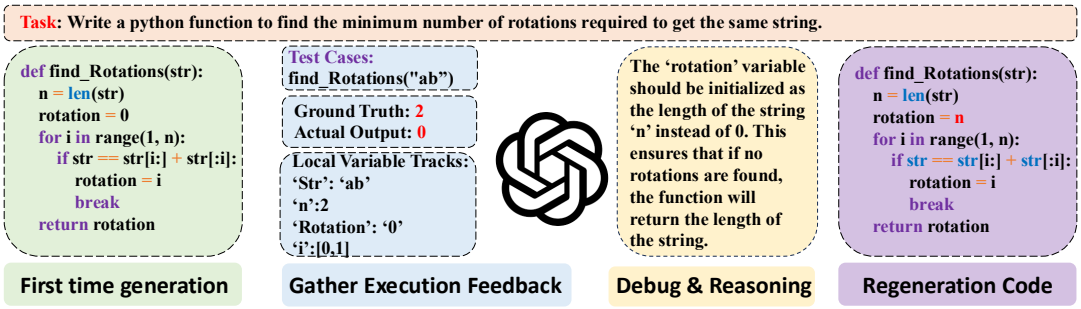

Figure 2: The pipeline of our model. We start with a task and a planning module to create a step-by-step plan for code generation. Then, we use LLM to generate an initial code and our code executor to evaluate the generated code using the test cases. If there are bugs in the generated code, we use our debug module to track the intermediate variables and other information. Based on the collected information, our architecture will try to provide a reason and solution to correct the bugs. Then the LLM can start the next iteration.

3 Architecture

Our **Human-Like Debugging Architecture (HLDA)** models code generation as an iterative debugging process rather than a one-shot prediction. It integrates planning, controlled execution, semantic diagnosis, and feedback-driven refinement (Fig. 2), enabling the model to reason about both code correctness and the causal execution path that leads to success or failure.

3.1 Planning Module

Human developers rarely begin coding without forming a mental map of the solution. They first identify the target function, select suitable algorithms or data structures, and consider corner cases. Our Planning Module mirrors this cognitive phase by decomposing the natural-language description of a problem into explicit reasoning steps and control logic. The output is a structured plan that includes the goal, reasoning outline, potential variables, and possible error sources. For instance, given a sorting problem, the module specifies substeps such as "parse input," "apply sorting algorithm," and "return ordered output." This plan functions as the LLM's internal reasoning scaffold, guiding subsequent modules while maintaining a transparent mapping between intent and implementation. By decoupling reasoning from code generation, HLDA enforces consistency in the output and improves interpretability and modularity—enabling systematic failure analysis in later stages. Empirically, this step prevents cascading logic errors, as each subsequent regeneration refers back to the same structured reasoning anchor.

3.2 Code Generator

The Code Generator translates the structured plan into executable Python code. Each reasoning element from the planning stage is realized as a corresponding code block or function. This modular translation preserves the causal dependencies among variables and functions. The generator employs an instruction-driven prompting format that encourages the model to "follow its own reasoning" instead of retrieving memorized templates. For example, if the plan specifies a boundary check or a special case, the generator explicitly implements it, ensuring alignment between reasoning and syntax. Such interpretability is critical because the debugging process can later link an observed failure directly to its corresponding planning node or generated segment. This linkage allows the system to perform targeted correction, similar to how a human developer would adjust one step of the logic without rewriting the entire function. Moreover, HLDA maintains detailed internal associations between lines of code and their originating plan elements, enabling fine-grained causal tracing.

3.3 Code Executor

In human programming, running and observing the code is as crucial as writing it. Our Code Executor reproduces this introspection by executing the generated code within a controlled sandbox. Before execution, the source code is transformed to insert lightweight trace hooks at critical control points—capturing loop counters, conditional branches, variable updates, and return values. This mechanism allows the executor to record both the final outputs and the full evolution of the program's internal state. All traces are serialized into structured logs that detail, for each execution step, the

corresponding line of code and associated variable states. When execution completes, the module checks whether the results match the expected outputs from both ground-truth and pseudo test cases. If all test cases pass, the result is marked correct. Otherwise, the executor returns a comprehensive diagnostic report including failing inputs, expected versus actual outputs, and variable traces at failure points. This design directly parallels human debugging behaviors—where developers insert print statements or use breakpoints to inspect intermediate results. The execution report thus provides the foundation for reasoning in the subsequent module, offering *trace-level causal visibility* rather than simple binary pass/fail feedback.

3.4 Pseudo Test Case Generator

Benchmark-provided test cases are reliable but often limited in coverage, leaving edge cases and latent failures unexposed. To broaden behavioral coverage without using reference solutions or additional ground-truth supervision, HLDA introduces an *oracle-free* Pseudo Test Case Generator. Unlike conventional test generation methods that construct input–output pairs, our pseudo tests are execution probes consisting of inputs paired with specification-consistent properties.

Given a natural-language problem description and the structured plan from the Planning Module, the generator proposes diverse candidate inputs by perturbing boundary values, input sizes, value distributions, and corner cases, such as empty inputs, repeated elements, extreme values, or degenerate cases. The LLM has access only to the textual specification, type signatures, and explicit constraints. It is not asked to infer expected outputs and does not access any reference implementation.

Instead of generating pseudo labels, the LLM produces *verifiable properties* implied by the specification, including invariants, metamorphic relations, and consistency constraints. For example, sorting tasks may require permutation preservation or monotonicity; string-manipulation tasks may require length preservation, idempotence, or format consistency; numerical tasks may involve boundary, symmetry, or monotonicity constraints. These properties capture semantic expectations without specifying exact outputs.

Formally, each pseudo test is represented as a tuple $(x, \mathcal{P})$, where x is the generated input and $\mathcal{P}$ is a set of executable property checks. During execution, the Code Executor runs the candidate program on x and evaluates whether $\mathcal{P}$ holds using final outputs and intermediate variable states. If a property is violated, the pseudo test yields a failure signal and the corresponding execution trace is collected; otherwise, it is discarded. Crucially, the program output is never compared with an LLM-generated expected output, so pseudo tests do not serve as correctness oracles or direct supervision for repair.

Because LLM-generated properties may be noisy or underspecified, HLDA applies a lightweight semantic validation step before execution. This step only filters ambiguous or specification-inconsistent properties; it does not certify program correctness or generate debugging labels. Validated pseudo tests are used solely as execution probes. When a violation occurs, runtime information, including variable updates, control-flow transitions, return values, and intermediate states, is passed to the Debug Module for causal reasoning and feedback generation.

This design avoids the circular dependency that would arise if an LLM generated and validated pseudo input–output pairs. Even if an imperfect property passes validation, it can only trigger additional traces; it cannot inject an incorrect target output, because no pseudo-test label is used as ground-truth supervision. All debugging decisions remain grounded in runtime behavior and trace-level causal deviation analysis.

In summary, the Pseudo Test Case Generator acts as a coverage-enhancing component rather than a source of supervision. By decoupling pseudo-test generation from correctness evaluation, HLDA exposes latent failures through execution-grounded property violations while avoiding dependence on canonical solutions, hidden oracles, or LLM-generated pseudo labels.

3.5 Debug Module

The Debug Module forms the analytical core of HLDA. It consumes execution logs and test outcomes to perform semantic diagnosis of failures. Upon encountering a mismatch between expected and actual results, the module examines the corresponding variable trajectories to identify the smallest causal chain that explains the error. For example, if a variable remains zero when it should have been incremented, the debug module inspects the preceding loop iterations and control conditions to determine the faulty logic. When operating on pseudo test cases, it first verifies the test's valid-

ity to prevent spurious corrections caused by misspecified examples. Once validated, the module conducts a detailed causal attribution by mapping variable deviations to their respective code fragments and planning nodes. This process generates a natural-language explanation that includes: (1) the probable root cause, (2) the affected code lines or logical condition, and (3) a proposed fix based on reasoning consistency. For instance, in a rotation-counting example, the debug module might detect that the initialization of the "rotation" variable to 0 rather than n caused early termination. Such explanations are not heuristic guesses but structured causal hypotheses derived from execution evidence. By explicitly separating test case validation from bug localization, the module enhances both stability and interpretability. The result is a robust debugging pipeline that faithfully mirrors human reasoning, balancing exploration and precision.

3.6 Feedback Module

The Feedback Module closes the human-like reasoning loop. In practical debugging, developers iteratively revise their code by reflecting on observed errors and progressively refining logic. HLDA implements this behavior by feeding the diagnostic explanation from the debug module back into the code generator as augmented contextual input. The regenerated prompt retains the original plan and problem statement but appends the specific failure description and suggested correction strategy. For example, the feedback may instruct: "Initialize rotation with n instead of 0 to ensure correct behavior when no match is found." This new prompt drives the next iteration of code synthesis. The architecture tracks performance across iterations—monitoring whether new versions resolve prior issues or introduce regressions. To avoid infinite loops or diminishing returns, the process is capped at eight iterations, returning the best-performing version if full success is not achieved. Over successive iterations, HLDA accumulates an internal causal map linking observed errors to reasoning flaws, thus learning from its own mistakes. This feedback mechanism transforms code generation into a reflective learning cycle, where execution-grounded evidence continuously informs improved reasoning. The result is an adaptive, interpretable, and scalable debugging framework that mirrors the human process of testing, understanding, and refinement.

4 Experiments

4.1 Comparison with State-of-the-arts

Simple Programming problem In table 1, we show our result compared with other methods in the MBPP dataset, the HumanEval dataset, and LiveCodeBench dataset, and our method achieves State-of-the-art in both datasets.

Contest level programming problem Contrast-level programming problems are much more complicated than simple programming problems. It contains complex problem descriptions, and its solutions are more challenging to get. In table 2, we compare our results with several typical methods and the current state-of-the-art.

4.2 Ablation Studies

In this section, all experiments are done using GPT-3.5-turbo and tested on the HumanEval dataset.

4.2.1 Influence of different level feedback

In this part of our study, we evaluate how varying degrees of feedback our feedback module provides affect its performance. This module can deliver input at multiple levels, such as assessing the code's correctness, analyzing specific test cases, and examining the intermediate variables produced during code execution. As illustrated in table 3 of our report, it becomes evident that the LLM's ability to identify bugs and enhance the overall quality of the final output is significantly improved with more comprehensive feedback.

This trend mirrors the human approach to debugging. Just as a programmer equipped with more information can more easily locate and rectify bugs in the code, the LLM's performance in identifying and correcting errors is similarly enhanced with richer feedback. In contrast, limited information can make debugging more challenging and less efficient. Our findings reinforce the notion that the depth and detail of feedback are crucial in effectively guiding both human and machine learning processes in code debugging.

4.2.2 Influence of the temperature

We study how the temperature parameter affects the code generation performance of LLMs. Temperature controls the randomness in text generation: low values yield more deterministic and conservative outputs, while high values produce more diverse but potentially less coherent results.

Table 1: Quantitative results on standard code generation benchmarks. The best results within each comparable block are highlighted in **bold**. "–" indicates unavailable or not evaluated under the same protocol.

Method		HumanEval	MBPP	LiveCodeBench
LLMs (zero-shot prompting)	AlphaCode (Li et al., 2022)	17.1	–	–
	Incoder (Fried et al., 2022)	15.2	17.6	15.3
	CodeX (Brown et al., 2020)	47.0	58.1	17.2
	PalmCoder (Chowdhery et al., 2023)	43.9	32.3	16.7
	StarCoder (Li et al., 2023)	33.6	52.7	13.2
	Llama-70B (Touvron et al., 2023)	30.5	45.4	19.4
	Code Llama-7B (Touvron et al., 2023)	33.5	41.4	20.1
	GPT-3.5-turbo (Achiam et al., 2023)	56.4	52.6	22.6
	Claude-instance-1	31.1	26.9	22.1
	GPT-4-turbo (Achiam et al., 2023)	58.6	64.8	26.7
	GPT-4 (Achiam et al., 2023)	66.1	69.3	33.9
	DeepSeek-V3 (Liu et al., 2024)	65.2	75.4	19.4
	DeepSeek-R1-0528 (Guo et al., 2025)	–	–	73.1
	Qwen3-235B-A22B (Yang et al., 2025)	–	–	65.9
	Qwen3-Coder-480B-A35B (Cao et al., 2026)	–	–	72.5
	Gemini-2.5-Pro-05-06 (Comanici et al., 2025)	–	–	71.8
	Claude-Opus-4 (Thinking) (Anthropic, 2025)	–	–	56.6
	GPT-5 (OpenAI, 2025)	–	–	70.2
With GPT-3.5-turbo	CoT (Wei et al., 2022)	46.9	54.8	29.4
	ReAct (Yao et al., 2022)	56.9	67.0	29.8
	Reflexion (Shinn et al., 2023)	68.1	70.0	30.1
	ToT (Yao et al., 2023)	54.4	65.8	29.4
	RAP (Hao et al., 2023)	63.1	71.4	28.6
	Self-Edit (Zhang et al., 2023)	62.2	56.4	30.1
	Self-Planning (Jiang et al., 2023)	65.2	58.6	31.5
	Self-debugging (Chen et al., 2023)	61.6	60.1	35.8
	INTERVENOR (Wang et al., 2023)	75.6	69.8	37.3
	LATS (Zhou et al., 2023)	83.8	81.1	31.6
	AgentCoder (Huang et al., 2023)	79.9	89.9	40.1
	Ours	**89.6**	**91.3**	**42.1**
With GPT-4	Reflexion (Shinn et al., 2023)	91.0	77.1	35.9
	Self-debugging (Chen et al., 2023)	–	80.6	37.2
	MetaGPT (Hong et al., 2023)	85.9	87.7	36.8
	LATS (Zhou et al., 2023)	94.4	–	40.1
	AgentCoder (Huang et al., 2023)	96.3	91.8	50.3
	Ours	**97.6**	**93.8**	**55.9**
With StarCoder	Ours	67.0	72.1	47.6
With Claude-instance-1	Ours	68.2	81.3	44.5
With PalmCoder	Ours	68.9	79.2	41.9
With Code Llama-7B	Ours	73.7	83.5	42.7
With GPT-4-turbo	Ours	91.4	93.7	48.9
With GPT-4o	Ours	98.8	95.2	62.1
With GPT-5	Ours	**99.4**	**97.5**	**73.2**

As shown in Fig. 3, we observe that moderate temperature levels leads to better performance. At very low temperatures, the model lacks creativity, limiting its ability to explore effective fixes. At very high temperatures, increased randomness introduces incoherent or buggy code.

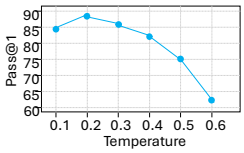

Figure 3: Effect of temperature on debugging. Moderate temperatures yield the best trade-off between exploration and stability.

This suggests the existence of an optimal temperature range where the model maintains a balance between creativity and control—producing code that is both innovative and reliable. Tuning temperature is therefore critical for achieving robust and effective code generation.

4.2.3 Influence of the max iteration number

After receiving feedback, our code generator will generate a new code. However, we cannot guarantee that our method will always provide a correct code. Hence, we must set a max iteration number to avoid our method being stuck in a loop. As shown in Fig. 4, we can see that with the number of max iterations increasing, the result will first increase and then remain the same, which indicates that for some challenging problems, it's difficult for LLM to generate a correct code. However, for some median difficulty code tasks, giving LLM several chances will help it develop the correct code. We also compare the performance of LATS and Reflexion when the iteration numbers increase.

As shown in Fig. 4, LATS and our model will have a noticeable performance improvement in the

Table 2: Quantitative results on contest-level programming benchmarks. The best results within each comparable block are highlighted in **bold**. "–" indicates unavailable or not evaluated under the same protocol.

LLM	Method	APPS	XcodeEval	CodeContest
GPT-3.5-turbo	Direct	8.0	17.9	5.5
	CoT (Wei et al., 2022)	7.3	23.6	6.1
	Self-Planning (Jiang et al., 2023)	9.3	18.9	6.1
	Analogical (Yasunaga et al., 2023)	6.7	15.1	7.3
	MapCoder (Islam et al., 2024)	11.3	27.4	12.7
	Ours	**15.9**	**33.7**	**17.1**
GPT-4	Direct	12.7	32.1	12.1
	CoT (Wei et al., 2022)	11.3	36.8	5.5
	Self-Planning (Jiang et al., 2023)	14.7	34.0	10.9
	Analogical (Yasunaga et al., 2023)	12.0	26.4	10.9
	MapCoder (Islam et al., 2024)	22.0	45.3	28.5
	Ours	**27.8**	**52.2**	**34.8**
DeepSeek-V3 (Liu et al., 2024)	–	6.7	13.9	3.6
DeepSeek-R1-0528 (Guo et al., 2025)	–	29.0	58.4	35.0
Qwen3-235B-A22B (Yang et al., 2025)	–	26.0	53.7	30.8
Qwen3-Coder-480B-A35B (Cao et al., 2026)	–	28.8	59.6	34.6
Gemini-2.5-Pro-05-06 (Comanici et al., 2025)	–	28.5	58.9	34.2
Claude-Opus-4 (Thinking) (Anthropic, 2025)	–	22.1	47.8	25.4
GPT-5 (OpenAI, 2025)	–	27.8	57.2	33.3
GPT-5	Ours	**31.2**	**63.6**	**38.9**

first several iterations. In contrast, the Reflection model does not have a pronounced improvement in the first several iterations. Our analysis of this phenomenon is that LATS and our method will provide high-level explainable feedback rather than low-level implicit feedback. The high-level feedback will help the code generator better understand why the previously code-generated code is wrong and help it better correct its bugs.

4.3 Using our method to debug code

In this part, we explore an experimental approach distinct from previous methods. While earlier methods concentrated on creating accurate code in response to a coding task, and our methodology has demonstrated superior accuracy, our architecture's capabilities extend beyond mere code generation. Inspired by human debugging processes, we were intrigued to discover if our architecture could also effectively debug existing erroneous code.

We made specific alterations to adapt our architecture for debugging rather than code generation. Our architecture comprises five components, but we omitted the code generator in the initial iteration for the debugging task. Since we already had code containing bugs, generating new code was unnecessary. The other components, however, remained unchanged from their roles in the code generation process.

The method for acquiring flawed code involved using the HumanEval and MBPP datasets. We employed various Large Language Models (LLMs),

Table 3: Ablation study for different levels of feedback. Performance improves as richer feedback is provided.

True/False	Instance-wise True/False	Instance-wise Feedback	Intermediate Variables	Pass@1
✗	✗	✗	✗	59.1
✓	✗	✗	✗	67.0
✓	✓	✗	✗	77.4
✓	✓	✓	✗	83.5
✓	✓	✓	✓	89.6

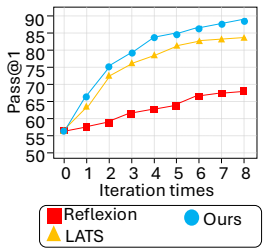

Figure 4: Ablation Study for iteration times. With the increase in iteration times, we also compare the influence of iteration in Relexion and LATS. Notice that in the first several iterations, our method has a significant improvement compared to Reflexion.

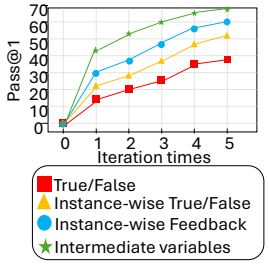

Figure 5: Result of the Accuracy after debugging by our architecture. With more feedback, the accuracy will be higher after debugging. This is intuitive, just like humans, provided more information will help humans better locate and correct the bugs.

Table 4: Overall token usage and per-problem overhead on HumanEval and MBPP.

Method	HumanEval	MBPP	Tokens (HumanEval / MBPP)	Overhead (HumanEval / MBPP)
MetaGPT	85.9	87.7	138.2K / 206.5K	1248.5 / 1583.6
AgentCoder	96.3	91.8	56.9K / 66.3K	228.7 / 365.9
Ours	**97.6**	**93.8**	63.4K / 75.9K	279.8 / 425.4

including GPT-3.5, GPT-4, and LLama, to create code straightforwardly without employing special techniques. The generated code was then tested using the provided test cases, and we collected all code samples that failed these tests. Due to the absence of specialized strategies in code creation, the errors in these samples varied widely, including incorrect function names, input values, and land logical errors.

In the debugging phase, our code executor gathered more comprehensive information than in its previous use to account for the diverse nature of the bugs. This information included specifics like TypeError and AttributeError. The debug module, therefore, had a broader scope of errors to address and correct. Once these initial errors were rectified, the module shifted to identifying and fixing any logical flaws in the code, using the test cases as a guide, akin to the process in our code generation experiments.

Post-debugging, the original flawed code and the feedback from our debugging process were fed back to the code generator. This step was crucial in generating a corrected version of the code based on the erroneous version and the feedback provided. We then evaluated this newly generated code against all test cases to determine its accuracy. Through this comprehensive process, we aimed not only to identify but also to correct a wide range of coding errors, thereby evaluating the effectiveness of our architecture in a debugging context, an area previously unexplored in our research.

To provide a complete understanding of the effectiveness of our different levels of feedback, we experimented with varying levels of feedback, which have similar settings in our first ablation study. The result can be seen in Fig 5

4.4 Inference Cost

While our approach introduces an additional execution and debugging phase, the overall computational overhead remains moderate. Table 4 summarizes the average token usage and per-problem inference cost on HumanEval and MBPP. Compared to AgentCoder, HLDA incurs slightly higher overhead due to the inclusion of variable tracing and structured feedback generation (approximately +18% tokens), yet it remains significantly more efficient than MetaGPT, which requires extensive multi-agent communication. We view this as a reasonable trade-off: the architecture achieves higher reliability and interpretability at manageable cost.

Notably, most additional computation occurs during the early iterations (typically within the first 3–4 loops), after which the system quickly converges to a stable solution. This demonstrates that structured introspection provides rapid learning signals, enabling efficient correction with minimal repeated inference. In practice, our approach increases per-problem latency by roughly 5–10 seconds while yielding substantial accuracy gains, improving Pass@1 by over 3–5 points across datasets. We also observe that roughly two-thirds of the additional tokens are spent on execution and trace recording, which directly contribute to causal reasoning rather than redundant text generation.

5 Conclusion

In this work, we proposed **HLDA**, a Human-Like Debugging Architecture that redefines code generation as an iterative reasoning and self-correction process rather than a single text-to-code translation. By integrating explicit planning, execution tracing, semantic-level debugging, and structured feedback, HLDA enables large language models to reason over their own execution behavior, identify causal sources of failure, and iteratively refine logic with grounded evidence. Extensive experiments across HumanEval, MBPP, and contest-level programming benchmarks show that HLDA achieves superior accuracy, robustness, and interpretability compared to existing reflection-based systems while maintaining competitive efficiency. More importantly, our results demonstrate that incorporating trace-level introspection fundamentally changes how LLMs approach problem solving—transforming them from reactive generators into reflective reasoners. We believe this paradigm opens new avenues for building generalizable and cognitively aligned code agents capable of continual improvement, scalable debugging, and adaptive reasoning across diverse domains. In future work, we plan to extend HLDA toward multi-function and multi-agent settings, integrating richer forms of execution feedback and tool interaction to further enhance reasoning transparency and autonomy.

Limitations

Despite the strong empirical results, several limitations remain. **(i) Oracle and specification ambiguity.** Our framework relies on benchmark-provided unit tests (e.g., HumanEval/MBPP) as the primary correctness signal. For pseudo tests beyond the benchmark suite, fully automatic construction of *unambiguous* and *spec-consistent* checks remains challenging when problem statements are underspecified; noisy or ambiguous pseudo tests may still mislead debugging, even with filtering. **(ii) Program scale.** We validate HLDA on single-function tasks and a controlled multi-function setting (2–4 interdependent functions). However, the current framework does not yet address large-scale, multi-file software projects that involve complex dependencies, build systems, persistent state, and long-horizon codebase reasoning. **(iii) Language scope.** Our executor and variable-level tracing are currently implemented for Python. Extending HLDA to other languages (e.g., C++/Java) requires language-specific sandboxing and tracing hooks (e.g., bytecode-, VM-, or compiler-level instrumentation), which we leave for future work. **(iv) Reliability of LLM-based diagnosis.** The debug module performs causal attribution and patch proposals using LLM reasoning over execution traces. While traces improve grounding, diagnosis can still be brittle for deeply nested control flow, complex data structures, or tasks requiring extensive domain knowledge, and the model may occasionally produce plausible but incorrect rationales.

AI Disclosure

We used ChatGPT solely for grammar correction and language polishing. All research content, literature analysis, and writing were conducted independently by the authors.

References

Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, and 1 others. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*.

Nader Akoury, Qian Yang, and Mohit Iyyer. 2023. A framework for exploring player perceptions of llm-generated dialogue in commercial video games. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pages 2295–2311.

Anthropic. 2025. Introducing claude 4. https://www.anthropic.com/news/claude-4.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and 1 others. 2021. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, and 1 others. 2020. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901.

Yuzhe Cai, Shaoguang Mao, Wenshan Wu, Zehua Wang, Yaobo Liang, Tao Ge, Chenfei Wu, Wang You, Ting Song, Yan Xia, and 1 others. 2023. Low-code llm: Visual programming over llms. *arXiv preprint arXiv:2304.08103*.

Ruisheng Cao, Mouxiang Chen, Jiawei Chen, Zeyu Cui, Yunlong Feng, Binyuan Hui, Yuheng Jing, Kaixin Li, Mingze Li, Junyang Lin, and 1 others. 2026. Qwen3-coder-next technical report. *arXiv preprint arXiv:2603.00729*.

Jonathan D Chang, Kiante Brantley, Rajkumar Ramamurthy, Dipendra Misra, and Wen Sun. 2023. Learning to generate better than your llm. *arXiv preprint arXiv:2306.11816*.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, and 1 others. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. 2023. Teaching large language models to self-debug. *arXiv preprint arXiv:2304.05128*.

Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, and 1 others. 2023. Palm: Scaling language modeling with pathways. *Journal of Machine Learning Research*, 24(240):1–113.

Gheorghe Comanici, Eric Bieber, Mike Schaekermann, Ice Pasupat, Noveen Sachdeva, Inderjit Dhillon, Marcel Blistein, Ori Ram, Dan Zhang, Evan Rosen, and 1 others. 2025. Gemini 2.5: Pushing the frontier with advanced reasoning, multimodality, long context, and next generation agentic capabilities. *arXiv preprint arXiv:2507.06261*.

Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen-tau Yih, Luke Zettlemoyer, and Mike Lewis. 2022. Incoder: A generative model for code infilling and synthesis. *arXiv preprint arXiv:2204.05999*.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Peiyi Wang, Qihao Zhu, Runxin Xu, Ruoyu Zhang, Shirong Ma, Xiao Bi, and 1 others. 2025. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*.

Shibo Hao, Yi Gu, Haodi Ma, Joshua Jiahua Hong, Zhen Wang, Daisy Zhe Wang, and Zhiting Hu. 2023. Reasoning with language model is planning with world model. *arXiv preprint arXiv:2305.14992*.

Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and 1 others. 2021. Measuring coding challenge competence with apps. *arXiv preprint arXiv:2105.09938*.

Sirui Hong, Xiawu Zheng, Jonathan Chen, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, and 1 others. 2023. Metagpt: Meta programming for multi-agent collaborative framework. *arXiv preprint arXiv:2308.00352*.

Bin Hu, Chenyang Zhao, Pu Zhang, Zihao Zhou, Yuanhang Yang, Zenglin Xu, and Bin Liu. 2023. Enabling efficient interaction between an algorithm agent and an llm: A reinforcement learning approach. *arXiv preprint arXiv:2306.03604*.

Xueyu Hu, Kun Kuang, Jiankai Sun, Hongxia Yang, and Fei Wu. 2024. Leveraging print debugging to improve code generation in large language models. *arXiv preprint arXiv:2401.05319*.

Dong Huang, Qingwen Bu, Jie M Zhang, Michael Luck, and Heming Cui. 2023. Agentcoder: Multi-agent-based code generation with iterative testing and optimisation. *arXiv preprint arXiv:2312.13010*.

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

Xue Jiang, Yihong Dong, Lecheng Wang, Qiwei Shang, and Ge Li. 2023. Self-planning code generation with large language model. *arXiv preprint arXiv:2303.06689*.

Mohammad Abdullah Matin Khan, M Saiful Bari, Xuan Long Do, Weishi Wang, Md Rizwan Parvez, and Shafiq Joty. 2023. xcodeeval: A large scale multilingual multitask benchmark for code understanding, generation, translation and retrieval. *arXiv preprint arXiv:2303.03004*.

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, and 1 others. 2023. Starcoder: may the source be with you! *arXiv preprint arXiv:2305.06161*.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, and 1 others. 2022. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097.

Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, and 1 others. 2024. Deepseek-v3 technical report. *arXiv preprint arXiv:2412.19437*.

Ansong Ni, Srini Iyer, Dragomir Radev, Veselin Stoyanov, Wen-tau Yih, Sida Wang, and Xi Victoria Lin. 2023. Lever: Learning to verify language-to-code generation with execution. In *International Conference on Machine Learning*, pages 26106–26128. PMLR.

OpenAI. 2025. Introducing gpt-5. https://openai.com/index/introducing-gpt-5/.

Dan Qiao, Chenfei Wu, Yaobo Liang, Juntao Li, and Nan Duan. 2023. Gameeval: Evaluating llms on conversational games. *arXiv preprint arXiv:2308.10032*.

Noah Shinn, Beck Labash, and Ashwin Gopinath. 2023. Reflexion: an autonomous agent with dynamic memory and self-reflection. *arXiv preprint arXiv:2303.11366*.

Vishvesh Soni. 2023. Large language models for enhancing customer lifecycle management. *Journal of Empirical Social Science Studies*, 7(1):67–89.

Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, and 1 others. 2023. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*.

Hanbin Wang, Zhenghao Liu, Shuo Wang, Ganqu Cui, Ning Ding, Zhiyuan Liu, and Ge Yu. 2023. Intervenor: Prompt the coding ability of large language models with the interactive chain of repairing. *arXiv preprint arXiv:2311.09868*.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, and 1 others. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in Neural Information Processing Systems*, 35:24824–24837.

William A Woods. 1973. Progress in natural language understanding: an application to lunar geology. In *Proceedings of the June 4-8, 1973, national computer conference and exposition*, pages 441–450.

Chunyang Xiao, Marc Dymetman, and Claire Gardent. 2016. Sequence-based structured prediction for semantic parsing. In *Annual meeting of the Association for Computational Linguistics (ACL)*, pages 1341–1350.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, and 1 others. 2025. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*.

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Thomas L Griffiths, Yuan Cao, and Karthik Narasimhan. 2023. Tree of thoughts: Deliberate problem solving with large language models. *arXiv preprint arXiv:2305.10601*.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2022. React: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629*.

Michihiro Yasunaga, Xinyun Chen, Yujia Li, Panupong Pasupat, Jure Leskovec, Percy Liang, Ed H Chi, and Denny Zhou. 2023. Large language models as analogical reasoners. *arXiv preprint arXiv:2310.01714*.

Kechi Zhang, Zhuo Li, Jia Li, Ge Li, and Zhi Jin. 2023. Self-edit: Fault-aware code editor for code generation. *arXiv preprint arXiv:2305.04087*.

Li Zhong, Zilong Wang, and Jingbo Shang. 2024. Debug like a human: A large language model debugger via verifying runtime execution step-by-step. *arXiv preprint arXiv:2402.16906*.

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 Implementation Details

A.1 Model Usage

We primarily use GPT-3.5-turbo and GPT-4 as our base language models for both code generation and all LLM-driven modules (pseudo-test generator, debug module). For GPT-3.5-turbo, we set the temperature to 0.2, the maximum generation length to 2048 tokens. For GPT-4 experiments, similar hyperparameters are adopted unless otherwise specified. The batch size for each round of code generation is set to 1 (single problem per inference call), and each stage of the pipeline is executed sequentially.

A.2 Baseline and Evaluation metrics

Evaluation Metrics We use Pass@k as our evaluation metrics which is the same as previous works ((Zhou et al., 2023)(Wang et al., 2023)(Shinn et al., 2023))

Baseline We compare our methods with several different architecture including Chain-of-Thought ((Wei et al., 2022)), ReAct ((Yao et al., 2022)), etc. To better understand the effectiveness of our methods, we use different Large Language Models, including GPT-3.5-turbo and GPT-4, etc., as the LLM to test our method and other methods.

A.3 dataset

In our experiments, we use five different datasets:

MBPP dataset(Austin et al., 2021): The MBPP dataset contains basic Python programming problems stated in natural language. The dataset contains 974 problems. For every problem, the dataset contains three different test cases. Following the previous methods((Ni et al., 2023)), we use the first test cases as part of the prompt to generate the template containing the function signatures. We use all three test cases during the test, and only when the generated code passes all three test cases do we think the generated code is correct. If the generated code fails in any test case, we will think the generated code is wrong and have bugs.

HumanEval dataset(Chen et al., 2021): The HumanEval dataset provides 164 comment descriptions of functions paired with a canonical implementation of each function and several input-output pairs that the function should pass. We follow the same evaluation method as the MBPP dataset.

APPS dataset(Hendrycks et al., 2021) The APPS dataset consists of problems collected from different open-access coding websites such as Code-

forces and Kattis. There are 10,000 coding problems in total, with 131,836 test cases. Due to the computing restrain, we use 169 problems to test our methods.

XcodeEval dataset(Khan et al., 2023) The XcodeEval dataset consists of over 7.5k problem with multiple tasks. In our experiment, we only focus on the code generation tasks and we use 178 problems to test our methods.

CodeContest dataset(Li et al., 2022) The CodeContest dataset is a competitive programming dataset for machine-learning that was used when training AlphaCode. In our experiment, we used 192 problems to test our methods.

B Multi-function Test

B.1 Dataset Build

While most existing code generation benchmarks evaluate models on single-function tasks, real-world programming often involves multiple interacting functions. In such settings, the correctness of the final output depends not only on the correctness of each individual function, but also on whether different functions are properly coordinated. To evaluate this ability, we construct a controlled benchmark of 200 multi-function tasks by extending single-function problems from HumanEval and MBPP into multi-component programs.

Each transformed problem consists of 2–4 interdependent functions that jointly implement a higher-level objective. Starting from an original single-function problem, we first identify its core logical operations, such as property checking, input parsing, filtering, aggregation, comparison, or boundary handling. We then decompose these operations into independent helper functions, where each helper function is assigned a clear semantic role and an explicit input-output contract. Finally, the original target function is rewritten as a main function that composes these helper functions to produce the final answer.

The construction follows four steps. First, we perform **core functionality identification**, where the original task is analyzed to extract its atomic logical components. Second, we apply **deterministic functional decomposition**, where each atomic operation is converted into a separate helper function. This ensures that the decomposition follows the natural computational structure of the original task rather than introducing arbitrary sub-functions. Third, we enforce **cross-function dependencies** by requiring the main function to call helper functions in sequence or combine their intermediate outputs. As a result, an error in one helper function may propagate to downstream computation and affect the final result. Fourth, we conduct **joint test generation**, where test cases are designed to evaluate the complete function pipeline rather than validating each helper function independently.

For example, a single-function string-processing task can be transformed into a multi-function problem requiring the model to implement `count_special_words(text: str) -> int`. The function counts words that satisfy two conditions: the word must be a palindrome and must contain at least one vowel. The transformed task explicitly requires two helper functions, `is_palindrome(word: str) -> bool` and `has_vowel(word: str) -> bool`. The main function then splits the input text into words, applies both helper functions to each word, and counts only the words satisfying both conditions. For the input `"madam racecar sky dry"`, the expected output is `2`, since `madam` and `racecar` satisfy both conditions.

This construction makes the benchmark more challenging than standard single-function evaluation. A model must preserve the intended semantics of each helper function, correctly pass intermediate values across functions, and maintain global consistency in the main function. Therefore, the benchmark directly tests cross-function reasoning, modular program synthesis, and error propagation in multi-component programs.

B.2 Results Analysis

During evaluation, our architecture reuses the same planning, execution, and feedback modules as in the standard code generation setting, but applies debugging iteratively across all functions. Instead of treating each function as an isolated generation target, the executor traces intermediate variables and return values across the complete program pipeline. When an error in one function affects downstream outputs, the debug module analyzes cross-function variable dependencies to identify the likely root cause and produces targeted feedback for regeneration.

This setting is particularly challenging for prior reflection-based or agent-based methods. Reflexion mainly relies on outcome-level feedback and self-reflection, which can indicate that the final answer is wrong but often provides limited informa-

Table 5: Accuracy of different methods on multi-function tasks.

Method	GPT-4	Reflexion	AgentCoder	GPT-4 + Our Architecture
Accuracy	41.5%	52.0%	63.5%	**81.0%**

tion about which helper function causes the failure. AgentCoder introduces multi-agent collaboration and iterative testing, but its correction process still tends to focus on local code revisions rather than explicitly tracing how an error propagates across function boundaries. In contrast, our method uses trace-level feedback to connect final-output failures with intermediate variable states, making it easier to localize errors in dependent helper functions.

As shown in Table 5, GPT-4 direct generation achieves an accuracy of 41.5%, indicating that even strong LLMs struggle when a problem requires coordinated reasoning across multiple functions. Reflexion improves the result to 52.0%, suggesting that iterative self-reflection can help repair some errors. AgentCoder further improves the accuracy to 63.5%, showing the benefit of agentic coordination and iterative testing. However, our method achieves the best performance, reaching 81.0% accuracy with GPT-4 as the backbone.

The substantial improvement demonstrates that detailed trace-level feedback is especially useful in compositional programming tasks. By observing intermediate variables, function-level return values, and downstream failure points, our architecture can distinguish whether an error originates from an incorrect helper function, an incorrect composition strategy, or a mismatch between helper outputs and the main function logic. This enables more precise debugging than methods that rely only on final execution results. These results highlight the importance of execution-grounded introspection for multi-function code generation and suggest that our framework can better support scalable program synthesis beyond single-function benchmarks.

C More Ablation Studies

Effect of temperature beyond code generation. To further analyze the reviewer's concern, we conducted a controlled module-wise temperature study. All modules are fixed at temperature $= 0.2$, while one target module varies its temperature. Results in Table 6 show that non-generation modules (planning, debugging, feedback, and pseudo-test generation) strongly favor low temperatures for stability, whereas moderate temperature benefits code generation. This supports our design choice of decoupling temperatures across modules and confirms that excessive randomness consistently harms causal reasoning and execution-grounded debugging.

Influence of Pseudo Test Case Generator

We further conduct a controlled ablation to isolate the contribution of the Pseudo Test Case Generator under contest-level settings. In this experiment, we compare direct GPT-4 generation, HLDA without pseudo tests, and the full HLDA model. All HLDA variants keep the same planning, execution tracing, debugging, and feedback modules, and differ only in whether pseudo tests are used.

As shown in Table 7, removing pseudo tests consistently degrades performance across APPS, XcodeEval, and CodeContest. Compared with direct generation, HLDA without pseudo tests still brings substantial improvements, indicating that execution tracing and feedback-driven debugging are effective. However, the full model further improves performance from 24.9 to 27.8 on APPS, from 48.3 to 52.2 on XcodeEval, and from 31.6 to 34.8 on CodeContest. This confirms that pseudo tests provide useful additional execution probes, helping the Debug Module expose latent failures and generate more precise feedback.

D Experimental Protocol and Version Details

This section details the experimental protocol, model versions, dataset subsets, and evaluation parameters used throughout our experiments, with the goal of ensuring reproducibility and addressing potential confounding factors arising from rapidly evolving LLM backbones.

D.1 Model Snapshots and Decoding Parameters

All experiments were conducted using fixed model snapshots accessed via official APIs at the time of evaluation. For proprietary models, we report the exact model identifiers and do not rely on automatic version upgrades during the experimental period. Unless otherwise specified, decoding parameters are kept identical across baselines and our method to ensure fair comparison.

- **GPT-3.5-turbo**: `gpt-3.5-turbo`, API snapshot (June 2024).
- **GPT-4**: `gpt-4`, API snapshot (June 2024).

Table 6: Effect of temperature beyond code generation on HumanEval Pass@1. In each experiment, all modules are fixed at temperature $= 0.2$ except the target module, whose temperature is varied.

Module Varied	**Temp=0.1**	**Temp=0.2**	**Temp=0.4**	**Temp=0.6**	**Observation**
Planning Module	87.3	**89.6**	78.8	60.4	Higher temperature destabilizes multi-step planning; moderate temperature yields the most coherent and consistent plans.
Code Generation Module	84.9	**89.6**	82.5	63.1	Moderate temperature improves exploration and creativity, while high temperature introduces severe code corruption.
Pseudo-Test Generator	86.8	**89.6**	80.7	58.9	Increased randomness leads to noisy or specification-inconsistent pseudo tests, degrading debugging reliability.
Debug Module	86.1	**89.6**	79.4	57.2	Randomness impairs trace interpretation and causal attribution, resulting in faulty diagnoses at high temperatures.
Feedback Module	87.0	**89.6**	81.2	59.8	High temperature causes inconsistent or contradictory explanations, weakening feedback-driven regeneration.

Table 7: Ablation study of the Pseudo Test Case Generator on contest-level programming benchmarks.

Setting	APPS	XcodeEval	CodeContest
Direct Generation	12.7	32.1	12.1
HLDA w/o Pseudo Tests	24.9	48.3	31.6
Full HLDA	**27.8**	**52.2**	**34.8**

- **GPT-4-turbo**: `gpt-4-turbo`, API snapshot (June 2024).
- **Open-source models**: Code LLaMA-7B, StarCoder, and related variants are evaluated using their official released checkpoints.

For all LLMs, we use nucleus sampling with `top_p` $= 1.0$. Unless explicitly varied in ablation studies, the default temperature is set to 0.2 for planning, debugging, and feedback modules, and 0.2 for code generation. The maximum number of HLDA iterations is capped at 8.

D.2 Execution Environment and Timeouts

All generated programs are executed in a sandboxed Python environment. To avoid excessive runtime and ensure comparability across methods, we impose a fixed timeout of **10 seconds per execution**. If a program exceeds the timeout or raises runtime exceptions (e.g., `TypeError`, `IndexError`), the execution is treated as a failure and corresponding traces are recorded when applicable.

D.3 HumanEval and MBPP Evaluation

For HumanEval and MBPP, we follow the official evaluation protocols and test suites provided by the benchmarks. Pass@1 accuracy is reported throughout the paper. No additional test cases or hidden ground-truth information are used during evaluation. Pseudo tests generated by HLDA are employed solely as execution probes and do not contribute to correctness supervision or scoring.

D.4 LiveCodeBench Subset and Settings

LiveCodeBench is continuously updated, which can introduce ambiguity in reported results. To ensure clarity, we explicitly specify the subset used in our experiments.

We evaluate on the **code generation subset of LiveCodeBench**, covering problems released before March 2024. Only problems with executable Python reference solutions and deterministic evaluation scripts are included. Problems requiring external libraries, interactive I/O, or non-deterministic behavior are excluded. Each problem is evaluated using its official test cases, and Pass@1 accuracy is reported.

D.5 Fairness Considerations

Given that proprietary LLMs such as GPT-4 are subject to continuous updates, direct comparison across different time periods can be misleading. To mitigate this issue, we emphasize two evaluation principles: (i) comparisons are conducted under fixed backbone snapshots whenever possible, and (ii) our method is evaluated across multiple LLM backbones, including both proprietary and open-source models. This design allows us to isolate method-level improvements from gains due to model scaling or backend updates.

Overall, this protocol ensures that all reported results are obtained under controlled and reproducible settings, and that observed performance gains can be attributed to the proposed architecture rather than undocumented changes in model versions or evaluation data.

E Prompts for different level feedback

As shown in Table 8, we provide the difference between different levels of feedback. Moreover, in this section, we provide specific examples for different levels of feedback in Table 9

F Prompt Template

Planning Module

```
You are an expert problem solver. Given a natural language
      programming task,
decompose it into a plan of reasoning steps before writing
      code. Your plan should include:

1. Clarify the problem requirements.
2. Choose a suitable algorithm or data structure.
3. Identify edge cases to consider.
4. Describe the step-by-step logic to solve the task.

Task:
"Write a function that returns True if a string is a
      palindrome, ignoring spaces and case."
```

Listing 1: Planning Module Prompt

Code Generator

```
You are a code generation assistant. Based on the task and
      the reasoning plan,
write clean and correct Python code.

Task:
"Write a function that returns True if a string is a
      palindrome, ignoring spaces and case."

Function signature:
def is_palindrome(s: str) -> boolean

Plan:
1. Normalize the string by removing spaces and converting to
      lowercase.
2. Compare the normalized string to its reverse.
3. Return True if they are equal, False otherwise.
```

Listing 2: Code Generator Prompt

Pseudo-Test Case Generator

```
You are a test case generator. Without access to the
      implementation,
write diverse test cases that check the correctness of the
      following function description.

Function Description:
"Return True if a string is a palindrome, ignoring spaces and
      case."

Function signature:
def is_palindrome(s: str) -> boolean

Generate:
- 2 basic tests
- 2 edge tests (e.g., empty string, symbols)
- 1 stress test (long string)
```

Listing 3: Pseudo-Test Case Generator Prompt

Debug Module (Ground Truth)

```
You are a debugging assistant. Analyze the execution trace
      and test failures
to identify possible bugs in the code.

Task:
"Return True if a string is a palindrome, ignoring spaces and
      case."

Code:
def is_palindrome(s):
    s_reverse = [s::-1]
    s == s[::-1]

Test Failure:
assert is_palindrome("Race car") == True
-> Got False

Trace:
Input: "Race car"
s_reverse = "rac ecaR"
s = "Race car"
Output: False
```

Listing 4: Debug Module with Ground Truth Prompt

Debug Module (Pseudo Test Case)

```
You are a debugging expert. Given:
- A failed test case (pseudo-generated),
- A code snippet,
- And the expected + actual outputs,

you must determine:

Step 1: Whether the test case is valid.
Step 2: If valid, identify the bug and suggest a correction.
If invalid, return a correction for the test case.

Task:
"Return True if a string is a palindrome, ignoring spaces and
      case."

Code:
def is_palindrome(s):
    s_reverse = [s::-1]
    s == s[::-1]

Test Failure:
assert is_palindrome("Race car") == True
-> Got False

Trace:
Input: "Race car"
s_reverse = "rac ecaR"
s = "Race car"
Output: False
```

Listing 5: Debug Module with Pseudo Test Case Prompt

Feedback Module

```
You are a feedback writer. Based on the error diagnosis and
      original code,
generate a natural language explanation for the error, and a
      revised plan
for code regeneration.

Code:
def is_palindrome(s):
    s_reverse = [s::-1]
    s == s[::-1]

Bug:
Failed on mixed-case and spacing input.

Feedback:
The current implementation doesn't handle spaces and case
      normalization.
Please update your function to preprocess the input by
      removing spaces
and converting to lowercase before checking for a palindrome.
```

Listing 6: Feedback Module Prompt

Table 8: Examples of Feedback at Varying Levels of Detail

Feedback Type	**Example**
Basic Feedback	Your code is **wrong**.
Instance-wise accurate/false feedback	Your code is **wrong** when the **test case** is `[input]`.
Instance-wise Feedback	Your code is **wrong**, when the **test case** is `[input]`, **your code output** is `[output]`, **the right output** is `[expected]`.
Intermediate Variable Feedback	Your code is **wrong**, when the **test case** is `[input]`, **your code output** is `[output]`, **the right output** is `[expected]`. When your code is running, the **intermediate variables** are `[variable traces]`.

G More examples

Q: Write a python function to count the number of squares in a rectangle.
A: def count_Squares(m,n):
    return (min(m, n) * (min(m, n) + 1) * (2 * min(m, n) + 1)) // 6
Q: Your generated code is wrong when the test case is count_Squares(4,3): Your code output of the test case is 14, the correct output of the test case is 20, the local variables when running your code are {'m': 4, 'n': 3}. You should calculate the number of squares of size i x i that can fit in the rectangle using the formula (m - i + 1) * (n - i + 1). please fix your code.
A: def count_Squares(m, n):
    total_squares = 0
    for i in range(1, min(m, n) + 1):
        total_squares += (m - i + 1) * (n - i + 1)
    return total_squares

Q: Write a function to convert snake case string to camel case string..
A: def snake_to_camel(word):
    parts = word.split('_')
    return parts[0] + ''.join(x.title() for x in parts[1:])
Q: Your generated code is wrong when the test case is snake_to_camel('python_program'). Your code output of the test case is 'pythonProgram', the correct output of the test case is 'PythonProgram', the local variables when running your code are {'word': 'python_program', 'parts': ['python', 'program']}. You should consider the first word. please fix your code.
A: def snake_to_camel(word):
    parts = word.split('_')
    return parts[0].title() + ''.join(x.title() for x in parts[1:])

Table 9: Illustrative Examples Across Different Feedback Levels

Original Questions: Write a python function to find the minimum number of rotations required to get the same string.
Original Code:

```
def find_Rotations(str):
    n = len(str)
    rotation = 0
    for i in range(1, n):
        if str == str[i:] + str[:i]:
            rotation = i
            break
    return rotation
```

Feedback Level	**Description**	**Example**
Level 1: Basic Feedback	Only tells the user whether the code is right or wrong. No further details.	**Feedback:** "Your code is **wrong**." **No additional explanation.**
Level 2: Instance-wise accurate/-false feedback	Indicates the specific test case for which the code is incorrect.	**Feedback:** "Your code is **wrong** when the **test case** is 'ab'."
Level 3: Instance-wise feedback	Adds the incorrect output and the expected correct output for the failing test case.	**Feedback:** "Your code is **wrong** when the **test case** is 'ab', **your code output** is 0, **the right output** is 2."
Level 4: Intermediate Variable-Level Feedback	Provides complete trace-level feedback including intermediate values during execution.	**Feedback:** "Your code is **wrong** when the **test case** is 'ab', **your code output** is 0, **the right output** is 2. When your code is running, the **intermediate variables** are `{'str': 'ab', 'n': 2, 'rotation': 0, 'i': 1}`."