

# 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.

Recent frameworks such as AgentCoder (Huang et al., 2023), MetaGPT (Hong et al., 2023), and

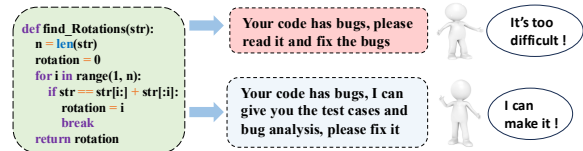


Figure 1: When code contains bugs, simply reporting failure is insufficient. Providing execution results, test cases, and bug analysis significantly facilitates effective debugging.

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 rarely model debugging as a *cognitive process of understanding* a program’s internal behavior, 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 precisely localize root causes.

We take a different stance: debugging is not merely an optimization step, but an epistemic process of hypothesis testing. 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, HLDA treats execution traces as first-class reasoning signals. By leveraging intermediate variable states and failure contexts, the model develops a causal understanding of its own program behavior.

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 simulation:** Rather than post-hoc reflection, the model simulates the human debugging process through hypothesis validation and iterative correction.
- **From benchmark success to reasoning transfer:** The same architecture generalizes from simple tasks to contest-level programming, indicating that structured feedback reshapes LLM reasoning.

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 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.

## 2 Related Works

### 2.1 Employing LLM as an Agent

As LLMs have evolved, efforts have been made to utilize them as agents in everyday scenarios. In this context, an ‘agent’ refers to an entity where we input information such as objectives, environment, and feedback, allowing the LLM to assist in decision-making. Starting with the concept of Chain-of-Thought ((Wei et al., 2022)) and Tree-of-Thought ((Yao et al., 2023)), researchers have been teaching LLMs to approach goals incrementally, similar to human processes. In ReAct ((Yao

et al., 2022)), there is a shift towards guiding LLMs to generate reasons in addition to actions. Additionally, recent studies ((Chang et al., 2023),(Hu et al., 2023)) are exploring the use of reinforcement learning to provide feedback to LLMs, enhancing their performance. The application of LLMs as agents spans numerous fields. For instance, they have been used in gaming ((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)). The expansion of LLMs continues to open up new possibilities for their application in various domains.

### 2.2 Code Generation

The task of generating code from high-level specifications has a long history. Early approaches were predominantly rule-based, as exemplified by Woods (Woods, 1973), while the advent of deep learning shifted the paradigm toward data-driven sequence modeling (Xiao et al., 2016). With the emergence of large language models (LLMs), code generation has seen substantial performance gains across a wide range of benchmarks. Recent research has focused on improving the reliability of LLM-generated code by incorporating execution feedback and iterative refinement. For instance, LEVER (Ni et al., 2023) and LATS (Zhou et al., 2023) integrate program execution into the generation loop, using test outcomes to guide search or selection among multiple candidate programs. ReAct (Yao et al., 2022) further unifies reasoning and acting by interleaving code generation with environment feedback. Beyond output-level validation, several recent studies emphasize leveraging intermediate execution signals. Teaching Large Language Models to Self-Debug (Chen et al., 2023) enables LLMs to inspect runtime errors and revise code accordingly, while Debug Like a Human (Zhong et al., 2024) proposes step-by-step execution verification using intermediate variable states. Similarly, Leveraging Print Debugging (Hu et al., 2024) demonstrates that exposing execution traces via print statements can substantially improve debugging effectiveness. Despite these advances, most existing methods still rely primarily on final execution outcomes or coarse-grained feedback, and lack an explicit mechanism for causal reasoning over execution traces. As a result, error correction often remains heuristic or trial-and-error driven. In contrast, our work explicitly models debugging as an execution-grounded

177	causal reasoning process, integrating variable-level	227
178	traces, specification-consistent pseudo tests, and	228
179	structured feedback to localize root causes of fail-	229
180	ure. This human-like debugging perspective en-	230
181	ables more precise and interpretable code correc-	231
182	tion, leading to consistent improvements over prior	232
183	methodologies.	233
184	<b>3 Architecture</b>	234
185	<b>Our Human-Like Debugging Architecture</b>	235
186	<b>(HLDA)</b> is designed to replicate the reasoning pro-	236
187	cess through which human programmers concep-	237
188	tualize, execute, and refine their code. Rather than	238
189	performing code generation as a one-shot predic-	239
190	tion, HLDA structures it as a multi-stage reasoning	240
191	loop that integrates explicit planning, controlled	241
192	execution, semantic diagnosis, and iterative feed-	242
193	back (Fig. 2). This architecture enables the model	243
194	to reason not only about the correctness of the code	244
195	but also about the causal chain that led to success	245
196	or failure, thereby providing a principled bridge	246
197	between neural code generation and debugging.	247
198	<b>3.1 Planning Module</b>	248
199	Human developers rarely begin coding without	249
200	forming a mental map of the solution. They first	250
201	identify the target function, select suitable algo-	251
202	rithms or data structures, and consider corner cases.	252
203	Our Planning Module mirrors this cognitive phase	253
204	by decomposing the natural-language description	254
205	of a problem into explicit reasoning steps and	255
206	control logic. The output is a structured plan	256
207	that includes the goal, reasoning outline, poten-	257
208	tial variables, and possible error sources. For in-	258
209	stance, given a sorting problem, the module spec-	259
210	ifies substeps such as “parse input,” “apply sort-	260
211	ing algorithm,” and “return ordered output.” This	261
212	plan functions as the LLM’s internal reasoning	262
213	scaffold, guiding subsequent modules while main-	263
214	taining a transparent mapping between intent and	264
215	implementation. By decoupling reasoning from	265
216	code generation, HLDA enforces consistency in	266
217	the output and improves interpretability and mod-	267
218	ularity—enabling systematic failure analysis in later	268
219	stages. Empirically, this step prevents cascading	269
220	logic errors, as each subsequent regeneration refers	270
221	back to the same structured reasoning anchor.	271
222	<b>3.2 Code Generator</b>	272
223	The Code Generator translates the structured plan	273
224	into executable Python code. Each reasoning ele-	274
225	ment from the planning stage is realized as a cor-	275
226	responding code block or function. This modu-	276
	lar translation preserves the causal dependencies	
	among variables and functions. The generator em-	
	ploys an instruction-driven prompting format that	
	encourages the model to “follow its own reasoning”	
	instead of retrieving memorized templates. For ex-	
	ample, if the plan specifies a boundary check or a	
	special case, the generator explicitly implements	
	it, ensuring alignment between reasoning and syn-	
	tax. Such interpretability is critical because the	
	debugging process can later link an observed fail-	
	ure directly to its corresponding planning node or	
	generated segment. This linkage allows the sys-	
	tem to perform targeted correction, similar to how	
	a human developer would adjust one step of the	
	logic without rewriting the entire function. More-	
	over, HLDA maintains detailed internal associa-	
	tions between lines of code and their originating	
	plan elements, enabling fine-grained causal tracing.	
	<b>3.3 Code Executor</b>	
	In human programming, running and observing	
	the code is as crucial as writing it. Our Code Ex-	
	ecutor 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 con-	
	trol 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 struc-	
	tured 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 out-	
	puts 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 de-	
	bugging behaviors—where developers insert print	
	statements or use breakpoints to inspect intermedi-	
	ate results. The execution report thus provides the	
	foundation for reasoning in the subsequent mod-	
	ule, offering <i>trace-level causal visibility</i> rather than	
	simple binary pass/fail feedback.	
	<b>3.4 Pseudo Test Case Generator</b>	
	Benchmark-provided test cases are reliable but	
	often limited in coverage, leaving many edge	
	conditions and latent failure modes unexposed.	

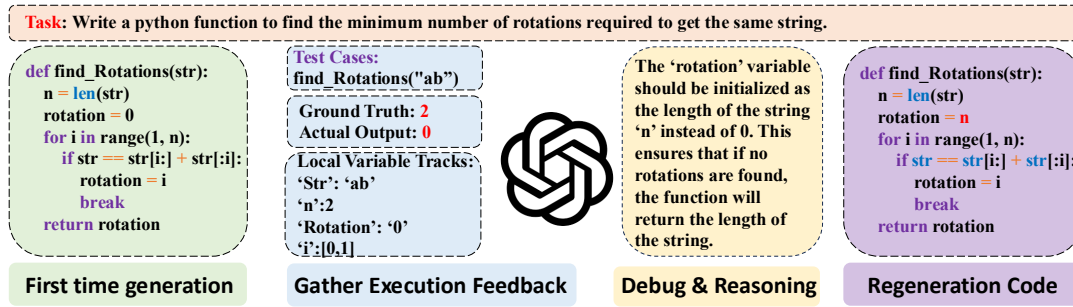


Figure 2: The pipeline of our model. We start with a task and a planning module to create a step-by-step plan for code generation. Then, we use LLM to generate an initial code and our code executor to evaluate the generated code using the test cases. If there are bugs in the generated code, we use our 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.

To broaden behavioral coverage without relying on reference solutions or ground-truth supervision, HLDA introduces an *oracle-free* Pseudo Test Case Generator that constructs additional execution probes as *inputs paired with specification-consistent properties*, rather than exact input-output pairs.

Given a natural-language problem description and the structured plan from the planning module, the generator prompts the LLM to propose diverse candidate inputs by perturbing boundary values, input sizes, distributions, and corner cases (e.g., empty inputs, repeated elements, extreme values). The LLM has access only to the textual specification, type signatures, and explicit constraints, and is neither asked to infer correct outputs nor to access any reference implementation.

Instead of predicting expected outputs, the LLM generates a set of *verifiable properties* implied by the specification, such as invariants, metamorphic relations, or consistency constraints. Examples include permutation preservation or monotonicity for sorting tasks, idempotence or length constraints for string manipulation, and boundary or symmetry conditions for numerical functions. These properties encode semantic expectations while remaining agnostic to exact output values.

Each pseudo test is represented as a tuple  $(x, \mathcal{P})$ , where  $x$  denotes the input and  $\mathcal{P}$  denotes executable property checks. During execution, the Code Executor runs the candidate program on  $x$  and evaluates whether  $\mathcal{P}$  holds using both final outputs and intermediate variable states. If any property is violated, the pseudo test yields a valid failure signal; otherwise, it is discarded. At no point is the program output compared against the pseudo-test output, ensuring that pseudo tests do not function

as correctness oracles.

Because LLM-generated properties may be noisy or underspecified, HLDA applies a lightweight semantic validation step before debugging, filtering out ambiguous or specification-inconsistent cases. Validated pseudo tests are then used exclusively as execution probes. When a property violation occurs, the resulting execution traces—including variable updates, control-flow transitions, and intermediate states—are passed to the Debug Module for causal reasoning and feedback generation. Thus, pseudo tests contribute only execution contexts and failure signals, while all debugging decisions are grounded in runtime behavior rather than pseudo-test outputs.

In summary, the Pseudo Test Case Generator serves as a coverage-enhancing component rather than a source of supervision. By decoupling pseudo-test generation from correctness evaluation, HLDA avoids dependence on canonical solutions or hidden oracles while still exposing latent bugs through execution-grounded property violations. ing latent bugs through execution-grounded property violations.

### 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

350 pseudo test cases, it first verifies the test’s valid-  
 351 ity to prevent spurious corrections caused by mis-  
 352 specified examples. Once validated, the module  
 353 conducts a detailed causal attribution by mapping  
 354 variable deviations to their respective code frag-  
 355 ments and planning nodes. This process generates  
 356 a natural-language explanation that includes: (1)  
 357 the probable root cause, (2) the affected code lines  
 358 or logical condition, and (3) a proposed fix based on  
 359 reasoning consistency. For instance, in a rotation-  
 360 counting example, the debug module might detect  
 361 that the initialization of the “rotation” variable to  
 362 0 rather than  $n$  caused early termination. Such ex-  
 363 planations are not heuristic guesses but structured  
 364 causal hypotheses derived from execution evidence.  
 365 By explicitly separating test case validation from  
 366 bug localization, the module enhances both sta-  
 367 bility and interpretability. The result is a robust  
 368 debugging pipeline that faithfully mirrors human  
 369 reasoning, balancing exploration and precision.

### 370 3.6 Feedback Module

371 The Feedback Module closes the human-like rea-  
 372 soning loop. In practical debugging, developers it-  
 373 eratively revise their code by reflecting on observed  
 374 errors and progressively refining logic. HLDA  
 375 implements this behavior by feeding the diagnos-  
 376 tic explanation from the debug module back into  
 377 the code generator as augmented contextual input.  
 378 The regenerated prompt retains the original plan  
 379 and problem statement but appends the specific  
 380 failure description and suggested correction strat-  
 381 egy. For example, the feedback may instruct: “Ini-  
 382 tialize rotation with  $n$  instead of 0 to ensure cor-  
 383 rect behavior when no match is found.” This new  
 384 prompt drives the next iteration of code synthesis.  
 385 The architecture tracks performance across itera-  
 386 tions—monitoring whether new versions resolve  
 387 prior issues or introduce regressions. To avoid  
 388 infinite loops or diminishing returns, the process  
 389 is capped at eight iterations, returning the best-  
 390 performing version if full success is not achieved.  
 391 Over successive iterations, HLDA accumulates  
 392 an internal causal map linking observed errors to  
 393 reasoning flaws, thus learning from its own mis-  
 394 takes. This feedback mechanism transforms code  
 395 generation into a reflective learning cycle, where  
 396 execution-grounded evidence continuously informs  
 397 improved reasoning. The result is an adaptive, in-  
 398 terpretable, and scalable debugging framework that  
 399 mirrors the human process of testing, understand-  
 400 ing, and refinement.

## 4 Experiments 401

### 4.1 Comparison with State-of-the-arts 402

**Simple Programming problem** This section 403  
 compares our method with several other methods 404  
 using different LLMs. In table 1, we show our 405  
 result compared with other methods in the MBPP 406  
 dataset and the HumanEval dataset, and our method 407  
 achieves State-of-the-art in both datasets. 408

**Contest level programming problem** Contrast- 409  
 level programming problems are much more com- 410  
 plicated than simple programming problems. It 411  
 contains complex problem descriptions, and its so- 412  
 lutions are more challenging to get. In table 2, we 413  
 compare our results with several typical methods 414  
 and the current state-of-the-art. 415

### 4.2 Ablation Studies 416

In this section, all experiments are done using GPT- 417  
 3.5-turbo and tested on the HumanEval dataset. 418

#### 4.2.1 Influence of different level feedback 419

In this part of our study, we evaluate how varying 420  
 degrees of feedback our feedback module provides 421  
 affect its performance. This module can deliver in- 422  
 put at multiple levels, such as assessing the code’s 423  
 correctness, analyzing specific test cases, and exam- 424  
 ining the intermediate variables produced during 425  
 code execution. As illustrated in table 3 of our 426  
 report, it becomes evident that the LLM’s ability 427  
 to identify bugs and enhance the overall quality of 428  
 the final output is significantly improved with more 429  
 comprehensive feedback. 430

This trend mirrors the human approach to debug- 431  
 ging. Just as a programmer equipped with more 432  
 information can more easily locate and rectify bugs 433  
 in the code, the LLM’s performance in identifying 434  
 and correcting errors is similarly enhanced with 435  
 richer feedback. In contrast, limited information 436  
 can make debugging more challenging and less ef- 437  
 ficient. Our findings reinforce the notion that the 438  
 depth and detail of feedback are crucial in effec- 439  
 tively guiding both human and machine learning 440  
 processes in code debugging. 441

#### 4.2.2 Influence of the temperature 442

We study how the temperature parameter affects the 443  
 code generation performance of LLMs. Temper- 444  
 ature controls the randomness in text generation: 445  
 low values yield more deterministic and conser- 446  
 vative outputs, while high values produce more 447  
 diverse but potentially less coherent results. 448

Table 1: Quantitative results of our proposed architecture, the best results are highlighted in **bold**.

	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
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-Planing (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	<b>89.6</b>	<b>91.3</b>	<b>42.1</b>
	With GPT-4	Reflexion (Shinn et al., 2023)	91.0	77.1
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		<b>97.6</b>	<b>93.8</b>	<b>55.9</b>
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

Table 2: Quantitative results of our proposed architecture in contest-level programming dataset, the best results are highlighted in **bold**.

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	<b>15.9</b>	<b>33.7</b>	<b>17.1</b>
GPT4	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	<b>27.8</b>	<b>52.2</b>	<b>34.8</b>

As shown in Fig. 3, we observe that moderate temperature levels lead 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.

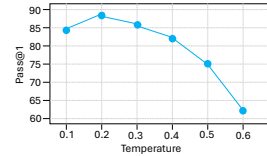


Figure 3: Effect of temperature on debugging performance. Moderate temperatures yield the best trade-off between exploration and stability.

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

Table 3: Ablation study for different level feedback. The result will improve with more feedback.

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

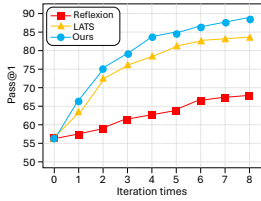


Figure 4: Ablation Study for iteration times. With the increase in iteration times, we also compare the influence of iteration in Reflexion and LATS. Notice that in the first several iterations, our method has a significant improvement compared to Reflexion.

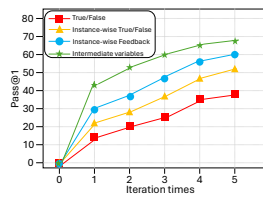


Figure 5: Result of the Accuracy after debugging by our architecture. With more feedback, the accuracy will be higher after debugging. This is intuitive, just like humans, provided more information will help humans better locate and correct the bugs.

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 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), 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 Multi-function Test

While most existing benchmarks evaluate models on single-function tasks, real-world programming often involves multi-function interactions, where the correctness of one module depends on another. To assess our model's ability to reason across func-

Table 4: Accuracy of different methods on multi-function tasks.

Method	GPT-4	Reflexion	AgentCoder	GPT-4 + Our Architecture
Accuracy	41.5%	52.0%	63.5%	<b>81.0%</b>

Table 5: Overall token usage and per-problem overhead (HumanEval / 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	<b>97.6</b>	<b>93.8</b>	63.4K / 75.9K	279.8 / 425.4

tion boundaries, we designed a controlled benchmark of 200 multi-function tasks by extending single-function problems from HumanEval and MBPP into multi-component programs.

Each problem consists of 2–4 interdependent functions that together implement a higher-level objective. For instance, one task extends a simple sorting function into a pipeline that combines data parsing, filtering, and sorting, where an error in one sub-function can propagate to others. This design evaluates whether the model can maintain logical consistency and causal reasoning across dependent modules.

During evaluation, our architecture reuses the same planning, execution, and feedback modules but applies debugging iteratively across all functions. If an error in one function affects downstream outputs, the system tracks cross-function variable dependencies to isolate the root cause. This differs from prior methods such as Reflexion and AgentCoder, which operate on each function independently.

As shown in Table 4, our method (GPT-4 + HLDA) achieves an accuracy of 81%, outperforming Reflexion (52%) and AgentCoder (63.5%). This demonstrates that detailed trace-level feedback enables consistent reasoning even under compositional task settings, where function-level coordination is essential. We also observe that intermediate-variable feedback plays a pivotal role in maintaining coherent behavior across modules, highlighting the potential of execution-grounded introspection for large-scale program synthesis.

#### 4.5 Inference Cost

While our approach introduces an additional execution and debugging phase, the overall computational overhead remains moderate. Table 5 summarizes the average token usage and per-problem inference cost on HumanEval and MBPP. Com-

pared 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.

## 649 Limitations

650 Despite the strong empirical results, several limita-  
651 tions remain. **(i) Oracle and specification ambigu-**  
652 **ity.** Our framework relies on benchmark-provided  
653 unit tests (e.g., HumanEval/MBPP) as the primary  
654 correctness signal. For pseudo tests beyond the  
655 benchmark suite, fully automatic construction of  
656 *unambiguous* and *spec-consistent* checks remains  
657 challenging when problem statements are under-  
658 specified; noisy or ambiguous pseudo tests may  
659 still mislead debugging, even with filtering. **(ii)**  
660 **Program scale.** We validate HLDA on single-  
661 function tasks and a controlled multi-function set-  
662 ting (2–4 interdependent functions). However, the  
663 current framework does not yet address large-scale,  
664 multi-file software projects that involve complex  
665 dependencies, build systems, persistent state, and  
666 long-horizon codebase reasoning. **(iii) Language**  
667 **scope.** Our executor and variable-level tracing  
668 are currently implemented for Python. Extending  
669 HLDA to other languages (e.g., C++/Java) requires  
670 language-specific sandboxing and tracing hooks  
671 (e.g., bytecode-, VM-, or compiler-level instrumen-  
672 tation), which we leave for future work. **(iv) Re-**  
673 **liability of LLM-based diagnosis.** The debug  
674 module performs causal attribution and patch pro-  
675 posals using LLM reasoning over execution traces.  
676 While traces improve grounding, diagnosis can still  
677 be brittle for deeply nested control flow, complex  
678 data structures, or tasks requiring extensive domain  
679 knowledge, and the model may occasionally pro-  
680 duce plausible but incorrect rationales.

## 681 AI Disclosure

682 We used ChatGPT solely for grammar correction  
683 and language polishing. All research content, lit-  
684 erature analysis, and writing were conducted inde-  
685 pendently by the authors.

## 686 References

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

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

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten 697  
Bosma, Henryk Michalewski, David Dohan, Ellen 698  
Jiang, Carrie Cai, Michael Terry, Quoc Le, and 1 699  
others. 2021. Program synthesis with large language 700  
models. *arXiv preprint arXiv:2108.07732*. 701

Tom Brown, Benjamin Mann, Nick Ryder, Melanie 702  
Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind 703  
Neelakantan, Pranav Shyam, Girish Sastry, Amanda 704  
Askell, and 1 others. 2020. Language models are 705  
few-shot learners. *Advances in neural information 706*  
*processing systems*, 33:1877–1901. 707

Yuzhe Cai, Shaoguang Mao, Wenshan Wu, Zehua Wang, 708  
Yaobo Liang, Tao Ge, Chenfei Wu, Wang You, Ting 709  
Song, Yan Xia, and 1 others. 2023. Low-code 710  
llm: Visual programming over llms. *arXiv preprint 711*  
*arXiv:2304.08103*. 712

Jonathan D Chang, Kianté Brantley, Rajkumar Rama- 713  
murthy, Dipendra Misra, and Wen Sun. 2023. Learn- 714  
ing to generate better than your llm. *arXiv preprint 715*  
*arXiv:2306.11816*. 716

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, 717  
Henrique Ponde De Oliveira Pinto, Jared Kaplan, 718  
Harri Edwards, Yuri Burda, Nicholas Joseph, Greg 719  
Brockman, and 1 others. 2021. Evaluating large 720  
language models trained on code. *arXiv preprint 721*  
*arXiv:2107.03374*. 722

Xinyun Chen, Maxwell Lin, Nathanael Schärli, and 723  
Denny Zhou. 2023. Teaching large language models 724  
to self-debug. *arXiv preprint arXiv:2304.05128*. 725

Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, 726  
Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul 727  
Barham, Hyung Won Chung, Charles Sutton, Sebas- 728  
tian Gehrmann, and 1 others. 2023. Palm: Scaling 729  
language modeling with pathways. *Journal of Ma- 730*  
*chine Learning Research*, 24(240):1–113. 731

Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, 732  
Eric Wallace, Freda Shi, Ruiqi Zhong, Wen-tau Yih, 733  
Luke Zettlemoyer, and Mike Lewis. 2022. Incoder: 734  
A generative model for code infilling and synthesis. 735  
*arXiv preprint arXiv:2204.05999*. 736

Shibo Hao, Yi Gu, Haodi Ma, Joshua Jiahua Hong, 737  
Zhen Wang, Daisy Zhe Wang, and Zhiting Hu. 2023. 738  
Reasoning with language model is planning with 739  
world model. *arXiv preprint arXiv:2305.14992*. 740

Dan Hendrycks, Steven Basart, Saurav Kadavath, Man- 741  
tas Mazeika, Akul Arora, Ethan Guo, Collin Burns, 742  
Samir Puranik, Horace He, Dawn Song, and 1 others. 743  
2021. Measuring coding challenge competence with 744  
apps. *arXiv preprint arXiv:2105.09938*. 745

Sirui Hong, Xiawu Zheng, Jonathan Chen, Yuheng 746  
Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven 747  
Ka Shing Yau, Zijuan Lin, Liyang Zhou, and 1 748  
others. 2023. Metagpt: Meta programming for 749  
multi-agent collaborative framework. *arXiv preprint 750*  
*arXiv:2308.00352*. 751



856	<b>A Implementation Details</b>		
857	<b>A.1 Model Usage</b>		
858	We primarily use GPT-3.5-turbo and GPT-4 as our	forces and Kattis. There are 10,000 coding prob-	905
859	base language models for both code generation	lems in total, with 131,836 test cases. Due to the	906
860	and all LLM-driven modules (pseudo-test gener-	computing restrain, we use 169 problems to test	907
861	ator, debug module). For GPT-3.5-turbo, we set	our methods.	908
862	the temperature to 0.2, the maximum generation	<b>XcodeEval dataset</b> (Khan et al., 2023) The	909
863	length to 2048 tokens. For GPT-4 experiments, sim-	XcodeEval dataset consists of over 7.5k problem	910
864	ilar hyperparameters are adopted unless otherwise	with multiple tasks. In our experiment, we only	911
865	specified. The batch size for each round of code	focus on the code generation tasks and we use 178	912
866	generation is set to 1 (single problem per inference	problems to test our methods.	913
867	call), and each stage of the pipeline is executed	<b>CodeContest dataset</b> (Li et al., 2022) The Code-	914
868	sequentially.	Contest dataset is a competitive programming	915
869	<b>A.2 Baseline and Evaluation metrics</b>	dataset for machine-learning that was used when	916
870	<b>Evaluation Metrics</b> We use Pass@k as our evalua-	training AlphaCode. In our experiment, we used	917
871	tion metrics which is the same as previous works	192 problems to test our methods.	918
872	((Zhou et al., 2023)(Wang et al., 2023)(Shinn et al.,		
873	2023))	<b>B More Ablation Studies</b>	919
874	<b>Baseline</b> We compare our methods with several	<b>Effect of temperature beyond code generation.</b>	920
875	different architecture including Chain-of-Thought	To further analyze the reviewer’s concern, we con-	921
876	((Wei et al., 2022)), ReAct ((Yao et al., 2022)),	ducted a controlled module-wise temperature study.	922
877	etc. To better understand the effectiveness of our	All modules are fixed at temperature = 0.2, while	923
878	methods, we use different Large Language Models,	one target module varies its temperature. Results	924
879	including GPT-3.5-turbo and GPT-4, etc., as the	in Table 6 show that non-generation modules (plan-	925
880	LLM to test our method and other methods.	ning, debugging, feedback, and pseudo-test gener-	926
881	<b>A.3 dataset</b>	ation) strongly favor low temperatures for stabil-	927
882	In our experiments, we use five different datasets:	ity, whereas moderate temperature benefits code	928
883	<b>MBPP dataset</b> (Austin et al., 2021): The MBPP	generation. This supports our design choice of	929
884	dataset contains basic Python programming prob-	decoupling temperatures across modules and con-	930
885	lems stated in natural language. The dataset con-	firming that excessive randomness consistently harms	931
886	tains 974 problems. For every problem, the dataset	causal reasoning and execution-grounded debug-	932
887	contains three different test cases. Following the	ging.	933
888	previous methods((Ni et al., 2023)), we use the	<b>C Experimental Protocol and Version</b>	934
889	first test cases as part of the prompt to generate the	<b>Details</b>	935
890	template containing the function signatures. We	This section details the experimental protocol,	936
891	use all three test cases during the test, and only	model versions, dataset subsets, and evaluation pa-	937
892	when the generated code passes all three test cases	rameters used throughout our experiments, with	938
893	do we think the generated code is correct. If the	the goal of ensuring reproducibility and addressing	939
894	generated code fails in any test case, we will think	potential confounding factors arising from rapidly	940
895	the generated code is wrong and have bugs.	evolving LLM backbones.	941
896	<b>HumanEval dataset</b> (Chen et al., 2021): The Hu-	<b>C.1 Model Snapshots and Decoding</b>	942
897	manEval dataset provides 164 comment descrip-	<b>Parameters</b>	943
898	tions of functions paired with a canonical imple-	All experiments were conducted using fixed model	944
899	mentation of each function and several input-output	snapshots accessed via official APIs at the time of	945
900	pairs that the function should pass. We follow the	evaluation. For proprietary models, we report the	946
901	same evaluation method as the MBPP dataset.	exact model identifiers and do not rely on automatic	947
902	<b>APPS dataset</b> (Hendrycks et al., 2021) The APPS	version upgrades during the experimental period.	948
903	dataset consists of problems collected from dif-	Unless otherwise specified, decoding parameters	949
904	ferent open-access coding websites such as Code-	are kept identical across baselines and our method	950
		to ensure fair comparison.	951

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

- **GPT-3.5-turbo**: gpt-3.5-turbo, API snapshot (June 2024).
- **GPT-4**: gpt-4, API snapshot (June 2024).
- **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  $\text{top}_p = 1.0$ . 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.

## C.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.

## C.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.

## C.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.

## C.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.

## D Prompts for different level feedback

As shown in Table 7, we provide the difference between different levels of feedback. Moreover,

Table 7: Examples of Feedback at Varying Levels of Detail

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

in this section, we provide specific examples for different levels of feedback in Table 8

## E 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

Table 8: 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.	<b>Feedback:</b> "Your code is <b>wrong</b> ." <b>No additional explanation.</b>
Level 2: Instance-wise accurate/false feedback	Indicates the specific test case for which the code is incorrect.	<b>Feedback:</b> "Your code is <b>wrong</b> when the <b>test case</b> is 'ab'."
Level 3: Instance-wise feedback	Adds the incorrect output and the expected correct output for the failing test case.	<b>Feedback:</b> "Your code is <b>wrong</b> when the <b>test case</b> is 'ab', <b>your code output</b> is 0, <b>the right output</b> is 2."
Level 4: Intermediate Variable-Level Feedback	Provides complete trace-level feedback including intermediate values during execution.	<b>Feedback:</b> "Your code is <b>wrong</b> when the <b>test case</b> is 'ab', <b>your code output</b> is 0, <b>the right output</b> is 2. When your code is running, the <b>intermediate variables</b> are {'str': 'ab', 'n': 2, 'rotation': 0, 'i': 1}."

```
1125 -> Got False
1126
1127 Trace:
1128 Input: "Race car"
1129 s_reverse = "rac ecaR"
1130 s = "Race car"
1131 Output: False
```

Listing 5: Debug Module with Pseudo Test Case Prompt

```
1155
1156 Listing 6: Feedback Module Prompt
```

## F More examples

### Feedback Module

```
1133
1134 You are a feedback writer. Based on the error diagnosis and
1135 original code,
1136 generate a natural language explanation for the error, and a
1137 revised plan
1138 for code regeneration.
1139
1140 Code:
1141 def is_palindrome(s):
1142     s_reverse = [s[::-1]]
1143     s == s[::-1]
1144
1145 Bug:
1146 Failed on mixed-case and spacing input.
1147
1148 Feedback:
1149 The current implementation doesn't handle spaces and case
1150 normalization.
1151 Please update your function to preprocess the input by
1152 removing spaces
1153 and converting to lowercase before checking for a palindrome.
1154
```

**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  $i \times i$  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:])
```