

# Precise Debugging Benchmark: Is Your Model Debugging or Regenerating?

Anonymous ACL submission

## Abstract

Unlike code completion, debugging requires localizing faults and applying targeted edits. We observe that frontier LLMs often *regenerate* correct but over-edited solutions during debugging. To evaluate how far LLMs are from precise debugging, we introduce the PRECISE DEBUGGING BENCHMARKING (PDB) framework, an automatic pipeline that converts any coding dataset into a debugging benchmark with precision-aware evaluation. PDB generates buggy programs by synthesizing verified atomic bugs and composing them into multi-bug programs. We further define two novel metrics, edit-level *precision* and bug-level *recall*, which provide fine-grained measurements of how many necessary edits are made and how many bugs are resolved. We release PDB-HARD, a 5,734-example evaluation set for measuring debugging precision and recall. Experiments show that frontier models, such as GPT-5.1-Codex and DeepSeek-V3.2-Thinking, achieve unit-test pass rates above 78% but exhibit precision below 45%, even when explicitly instructed to perform minimal debugging. Finally, we show that iterative and agentic debugging strategies do not substantially improve precision or recall, highlighting the need to rethink post-training pipelines for coding models.

## 1 Introduction

Large Language Models (LLMs) have reshaped the programming landscape through their remarkable capabilities in code generation (Chen et al., 2021; Li et al., 2022a). From synthesizing complex algorithms from natural language prompts to translating entire codebases, modern LLMs excel at producing code from scratch. However, real-world software development is dominated not by generation but by debugging and maintenance (Glass, 2002). When applied to debugging tasks, we observe that frontier LLMs often default to regeneration, *i.e.*, rewriting large portions, or even the entirety, of a program

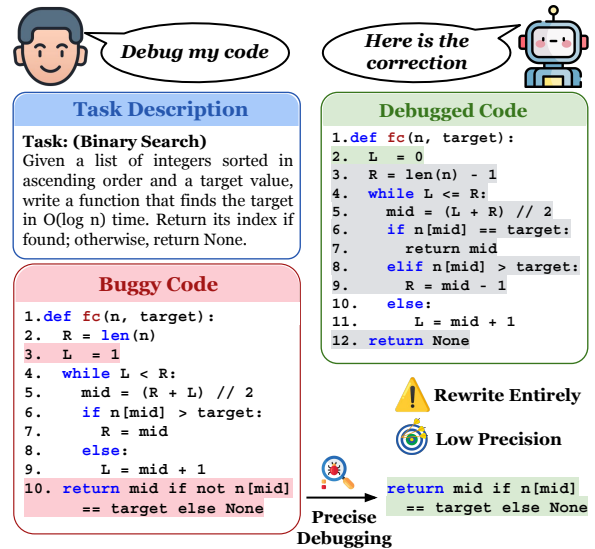


Figure 1: Real example from GPT-5.2 debugging a binary search program, where the model rewrites the entire solution. Green lines mark precise edits; gray lines highlight over-edits.

when presented with buggy code (Figure 1). While often effective at passing tests, this brute-force strategy is poorly suited for realistic codebases, where large-scale rewrites are costly, risky, and difficult to review (Sobania et al., 2023). In contrast, targeted debugging requires precise fault localization and minimal, intent-preserving edits. This raises a fundamental question: *How far are LLMs from precise debugging, rather than merely reverting to their strength in code regeneration?*

Existing debugging benchmarks focus on unit-test only evaluation and fail to evaluate these capabilities. Under such evaluation, models are rewarded equally for regenerating a full solution, hard-coding outputs, or performing a minimal targeted fix. Moreover, unit-test evaluation obscures incremental progress: a model that correctly repairs only one defect in a multi-bug program receives the same score as a model that fixes none. This misalignment with real-world debugging practice

limits our ability to understand how LLMs reason about bugs and code edits.

To address this gap, we introduce the PRECISE DEBUGGING BENCHMARKING (PDB) framework, an automatic pipeline that rigorously evaluates LLM debugging behavior independently of code generation. PDB provides a plug-and-play framework that converts existing coding datasets into debugging benchmarks through two steps: (1) synthesizing verified atomic bugs to produce ground-truth edit scripts, and (2) composing these bugs into multi-bug programs while preserving bug independence (*i.e.*, avoiding compounding interactions). Beyond binary test outcomes, PDB evaluates model patches using novel edit-level *precision* and bug-level *recall*, explicitly rewarding targeted fixes and penalizing unnecessary modifications.

We constructed a 5,734-example PDB-HARD evaluation set with the PDB framework, from BigCodeBench (Zhuo et al., 2024) and LiveCodeBench (Zhu et al., 2024). Experiments on PDB-HARD reveal behaviors that unit tests fail to capture. First, frontier models exhibit strikingly different rankings under edit-level evaluation. Models such as GPT-5.1-Codex (OpenAI, 2025a) and DeepSeek-V3.2-Thinking (Liu et al., 2025) achieve high unit-test pass rates (>78%) but low edit precision (<45%), while Qwen3-Coder-480B (Qwen, 2025) attains comparatively lower unit-test pass rates (71%) yet substantially higher precision (66%). Additionally, we show that though iterative and agentic debugging strategies can improve unit-test performance, they do not meaningfully improve precision or recall. Our findings demonstrate the necessity of PDB for revealing true debugging capabilities and distinguishing model strategies beyond surface-level correctness, and highlight a fundamental limitation in current post-training pipelines for coding-focused LLMs.

## 2 Precise Debugging Setup

We begin by formally defining the components of the automated debugging task. An automated debugging system  $\mathcal{M}$ , takes the initial buggy program  $C_b$  and a natural language task description  $x$  as input, and returns the predicted program revision  $\hat{C} = \mathcal{M}(C_b, x)$ . The conventional debugging pipeline evaluates the system’s final output purely on its functional correctness, using a binary evaluation function  $F_{\mathcal{U}}(C) \rightarrow \{0, 1\}$ , where  $\mathcal{U} = \{u_1, u_2, \dots, u_n\}$  is a suite of designed unit

tests. The evaluation function  $F_{\mathcal{U}}$  returns 1 if the program  $C$  passes all tests in  $\mathcal{U}$ , and 0 otherwise.

While straightforward, this method cannot penalize unnecessary edits or wholesale rewrites when the final program merely passes the test, nor can it distinguish between partially correct solutions and entirely incorrect ones. The precise debugging setup shifts the evaluation from program-level to the specific set of edits proposed by the model.

**Minimal corrections.** We denote a line-edit on line  $l$  as  $e_l$ , and a set of line-edits as  $E$ . For a buggy program  $C_b$ , we denote the set of *minimal corrections* by

$$\mathcal{E}_{C_b} = \arg \min_E |E| \text{ s.t. } F_{\mathcal{U}}(\text{apply}(E, C_b)) = 1,$$

where the apply function applies line-edits on  $C_b$ . Similarly, we can apply reverse edits  $\bar{E}$  on ground-truth program  $C_{\text{gt}}$  to derive buggy program  $C_b$ .

**Atomicity.** We define the bug in a buggy program  $C_b$  as *atomic* when a minimal correction consists of edits on a contiguous sequence of lines. Formally,  $\exists E \in \mathcal{E}_{C_b}$  such that  $E = \{e_i, e_{i+1}, \dots, e_{i+n}\}$ .

**Independence.** Intuitively, independence means that fixing one bug neither introduces nor removes edits required to fix the other. For two edit sets  $E_1 \in \mathcal{E}_{C_{b1}}$  and  $E_2 \in \mathcal{E}_{C_{b2}}$  corresponding to the same ground-truth program  $C_{\text{gt}}$ , we can construct a composed buggy program  $C_{b3} = \text{apply}(\bar{E}_1 \cup \bar{E}_2, C_{\text{gt}})$ . If the set of minimal corrections is the pairwise union of corrections from  $\mathcal{E}_{C_{b1}}$  and  $\mathcal{E}_{C_{b2}}$ , we consider bugs in  $C_{b1}$  and  $C_{b2}$  to be *independent*.

**Semantic correctness.** Consider a buggy program  $C_b$  containing  $k$  atomic and independent bugs, and a revision  $\hat{C} = \text{apply}(\hat{E}, C_b)$ , where  $\hat{E}$  is the predicted edits. We define *bug-level semantic correctness* as follows.

Let  $E_{\text{gt}} \in \mathcal{E}_{C_b}$  be the set of ground-truth edits, which can be decomposed as  $E_{\text{gt}} = E_1 \cup E_2 \cup \dots \cup E_k$ , where  $E_1, \dots, E_k$  are contiguous and non-overlapping. We employ a function, denoted as  $\text{map}$ , which pairs each  $E_i$  with the closest edits in  $\hat{E}$ . For each bug  $i$ , we construct a pseudo-revision  $\hat{C}_i = \text{apply}((E_{\text{gt}} \setminus E_i) \cup \text{map}(E_i), C_b)$ , which replaces the ground-truth edits  $E_i$  with the predicted edits  $\text{map}(E_i)$ . We define a candidate  $\hat{C}_i$  as *semantically correct* for bug  $i$  if  $F_{\mathcal{U}}(\hat{C}_i) = 1$ . Examples of this evaluation are shown in Figure 2.

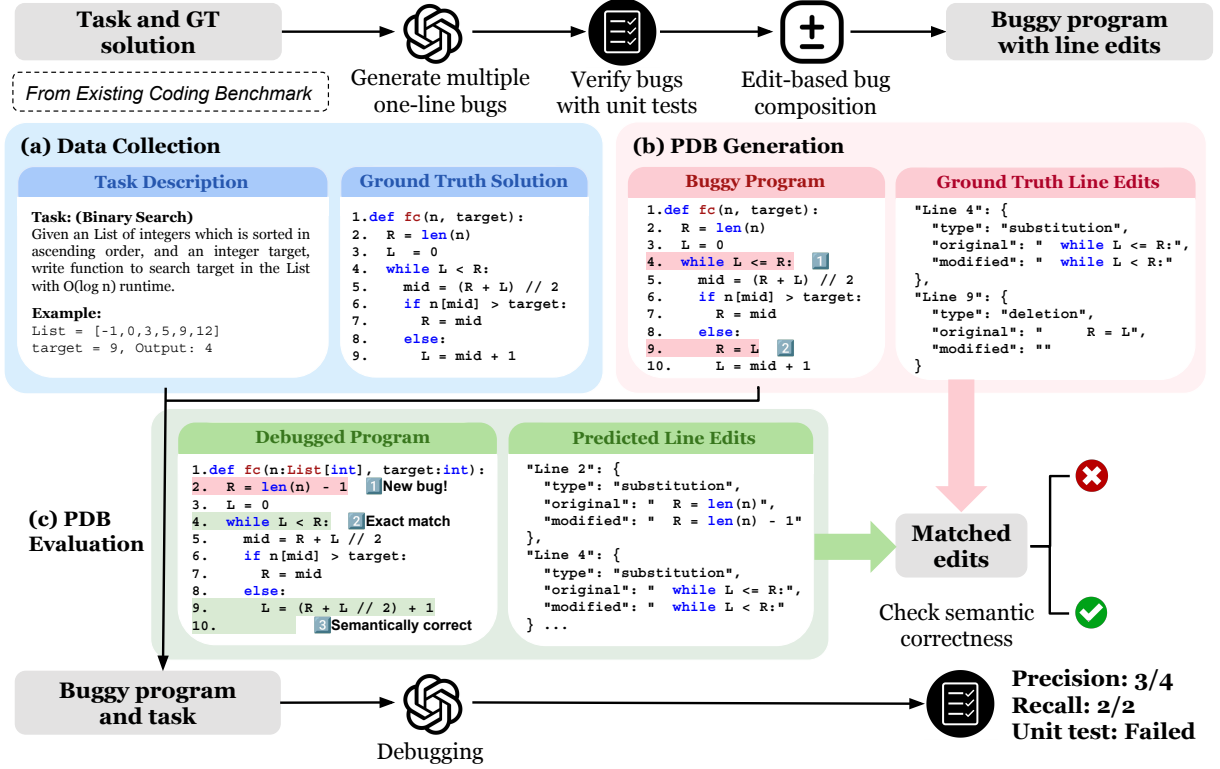


Figure 2: PDB pipeline. Generation: LLMs first synthesize and verify single-line bugs from existing coding datasets, which are then composed into multi-bug programs. Evaluation: Automated debugging systems are evaluated on these programs using both unit-test accuracy and edit-level precision and bug-level recall.

Based on this, we define precision and recall as:

$$\text{precision} = \frac{1}{|\hat{E}|} \sum_{i=1}^k F_{\mathcal{U}}(\hat{C}_i) \cdot |E_i|, \quad (1)$$

$$\text{recall} = \frac{1}{k} \sum_{i=1}^k F_{\mathcal{U}}(\hat{C}_i). \quad (2)$$

We note that precision functions as an edit-level metric by averaging over the edits  $|\hat{E}|$ , while recall is a bug-level metric averaged over the  $k$  bugs.

Since our goal is to discourage full-solution re-generation rather than to enforce strictly minimal edits, we relax the precision metric in Eq. (1) by introducing a tolerance parameter  $\epsilon$ , allowing up to  $\epsilon$  additional lines of edits for each bug  $i$ . Formally, we denote the *effective edit size* for each bug  $i$  as  $(|\hat{E}_i|)_{\epsilon} = \min(|\text{map}_{\epsilon}(E_i)|, |E_i| + \epsilon)$ . We detail the matching functions  $\text{map}$  and  $\text{map}_{\epsilon}$  in Appendix C. Accordingly, the precision with a tolerance  $\epsilon$  is

$$\text{precision}_{\epsilon} = \frac{1}{|\hat{E}|} \sum_{i=1}^k F_{\mathcal{U}}(\hat{C}_i) \cdot (|\hat{E}_i|)_{\epsilon}. \quad (3)$$

### 3 Generation and Evaluation Pipeline

As illustrated in Figure 2, PDB consists of two stages: *generation* and *evaluation*. During the

PDB generation stage, we first use LLMs to synthesize atomic bugs from existing coding datasets. After verifying buggy programs with unit tests, we record the corresponding edit sets and compose them to construct multi-bug programs. During the PDB evaluation stage, we prompt an automated debugging system  $\mathcal{M}$  to revise the buggy programs, and evaluate its performance using both traditional unit-test accuracy and our proposed edit-level precision and bug-level recall metrics. All prompt templates are provided in Appendix E.

#### 3.1 PDB generation

Starting from an existing coding benchmark, we preprocess the data by executing unit tests on the annotated “ground-truth” programs and retaining only *verified* programs that pass all tests without errors (e.g., timeouts). For each task description  $x$  and ground-truth program  $C_{\text{gt}}$ , we generate buggy programs across five Orthogonal Defect Classification (ODC; Chillarege et al., 1992) categories: *Assignment*, *Checking*, *Algorithm*, *Build/Package/Merge*, and *Timing/Serialization*. Each category further contains several subcategories, listed in Table 5.

**Atomic bug generation.** To ensure atomicity, we restrict bug generation to single-line edits, so that the minimal correction satisfies  $|E| = 1$ . We consider three types of line-level operations: *insertion*, *deletion*, and *substitution*. We first apply a rule-based filter to identify lines that are not safely deletable (e.g., causing indentation errors) or not editable (e.g., function headers). To promote diversity, we then randomly select (i) one operation type, (ii) one bug category, and (iii) a subset of editable lines compatible with the chosen operation. An LLM from a generator pool is prompted to modify one of the selected lines to produce a single-line buggy program. We repeat this process  $m_1$  times per  $(x, C_{gt})$  pair and retain only programs that fail unit tests, ensuring the validity of injected bugs.

**Bug composition.** To create more challenging debugging scenarios, we compose multiple atomic bugs into a single program. For each  $(x, C_{gt})$  pair and a target bug count  $k$ , we randomly sample  $k$  distinct single-line edits from the generated atomic bugs. To encourage independence between bugs, we enforce a *stride* constraint, requiring any two selected edits to be at least  $s$  lines apart. For each bug count  $k \in \{2, \dots, k_{max}\}$ , we repeat this process  $m_2$  times per  $(x, C_{gt})$  pair and record all composed multi-bug programs that satisfy the constraint.

**Subsampling.** To avoid over-representation of  $(x, C_{gt})$  pairs with many successful generations, we subsample the data by randomly selecting at most  $m_3$  buggy programs per bug count per  $(x, C_{gt})$  pair.

### 3.2 PDB evaluation

During evaluation, debugging systems, either single-pass LLMs or LLM-based agents, are instructed to debug a buggy program  $C_b$ , given the task description  $x$  and, optionally, access to unit tests  $\mathcal{U}$  and unit-test error feedback.

We use the precision, recall equations as in Eq. (2, 3), and report the unit-test score at Pass@1 (Kulal et al., 2019). Finally, although subsampling reduces imbalance, the dataset may still be skewed toward certain bug counts. We therefore report *micro-averaged* on all metrics by first averaging over examples with the same bug count and then across different bug counts.

## 4 PDB-HARD Evaluation Set

We construct PDB-HARD, an evaluation set of 5,734 examples, using the PDB generation and

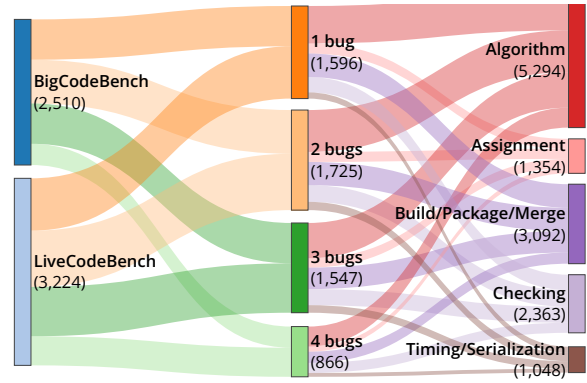


Figure 3: Data distribution of PDB-HARD.

evaluation pipeline. Dataset statistics are summarized in Figure 3.

We source tasks from two existing coding benchmarks, BigCodeBench, which focuses on API usage, and LiveCodeBench, which emphasizes algorithmic reasoning. Our bug-generation pool consists of three frontier LLMs: GPT-5.1-Codex, Claude-4.5-Sonnet (Anthropic, 2025), and Gemini-2.5-Pro (Comanici et al., 2025). For each task, we generate  $m_1 = 20$  single-line bugs, compose up to  $m_2 = 100$  multi-bug variants with at most  $k_{max} = 4$  independent bugs per program, and subsample  $m_3 = 5$  buggy programs per bug count per task. We enforce a stride of at least  $s = 3$  lines apart during composition.

This procedure yields an initial evaluation set, PDB-FULL, containing 7,591 examples (see Appendix B.2 for details). We evaluate PDB-FULL on 9 models, including thinking models: GPT-5.1-Codex, Claude-4.5-Sonnet, Gemini-2.5-Pro, Grok-Code-Fast (xAI, 2025), DeepSeek-V3.2-Thinking, and Kimi-K2-Thinking (Kimi et al., 2025); and non-thinking models: Qwen3-Coder-480B, DeepSeek-V3.2, and Kimi-K2-Instruct (Kimi et al., 2025). All models are prompted to produce minimal code edits. We use a maximum output length of 32,000 tokens for thinking models and 8,000 tokens for non-thinking models, with a temperature of 1.0 throughout.

We then apply model-based filtering to identify easy examples. We use a tolerance  $\epsilon = 2$  for precision evaluation with Eq. (3). An example is labeled easy if it achieves perfect precision, recall, and unit-test score for at least 7 out of the 9 evaluated models ( $> 77\%$ ). Applying this criterion removes 1,857 examples, resulting in the final PDB-HARD benchmark of 5,734 challenging examples.

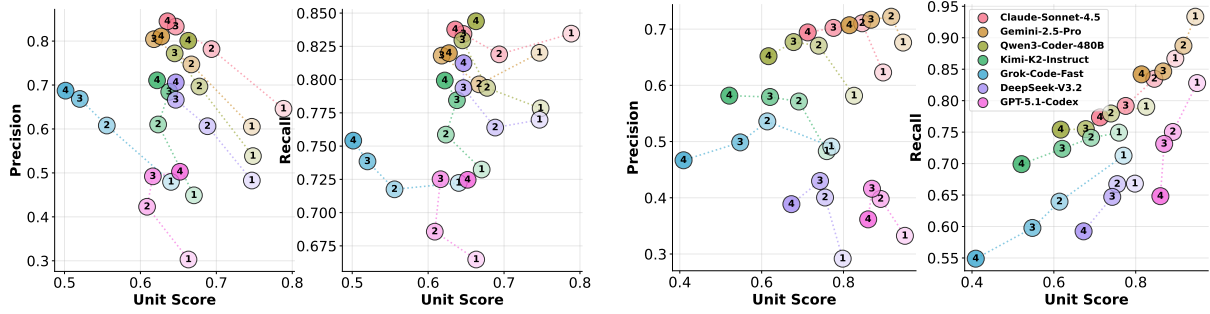


Figure 4: Correlation between precision, recall, and unit-test score across bug counts. Results are shown on subsets of PDB-HARD from BigCodeBench (left) and LiveCodeBench (right), with bug counts indicated by numbers. As the number of bugs increases, precision generally exhibits a negative correlation with unit-test score, while recall displays dataset-dependent behavior.

Model	Precision	Recall	Unit (%)
Claude-Sonnet-4.5	71.9±0.9	82.6±0.8	76.7±1.1
Gemini-2.5-Pro	71.7±0.9	85.2±0.7	79.7±1.0
Qwen3-Coder-480B	66.4±0.9	78.7±0.9	71.0±1.2
Kimi-K2-Instruct	57.2±1.0	74.6±0.9	65.4±1.2
Grok-Code-Fast	54.6±1.0	67.6±1.0	58.8±1.3
Kimi-K2-Thinking	51.4±0.9	77.3±0.9	75.7±1.1
DeepSeek-V3.2	47.5±1.0	70.8±1.0	72.3±1.2
DeepSeek-V3.2-Thinking	44.5±0.9	71.9±1.0	81.2±1.0
GPT-5.1-Codex	39.7±0.8	72.7±0.9	78.1±1.1

Table 1: Precision, recall, and unit score on the PDB-HARD set. Blue indicates better performance, while red indicates worse.

## 5 Experiment Results

By evaluating and analyzing both LLMs and LLM-based agents on PDB-HARD, we show that current systems remain far from achieving precise, edit-aware debugging.

### 5.1 PDB-HARD overview

**Divergence in model debugging behaviors.** Even when explicitly instructed to perform minimal debugging, 6 of 9 frontier models fail to reach 60% precision. Table 1 shows that while unit-test pass rates suggest comparable performance among top models, edit-level precision reveals substantial differences in debugging strategies. For example, DeepSeek-V3.2-Thinking achieves a higher unit-test score than Claude-Sonnet-4.5 (81.2% vs. 76.7%), yet exhibits much lower precision (44.5% vs. 71.9%), indicating regenerator-style behavior that relies on broad rewrites. In contrast, models such as Claude-Sonnet-4.5 and Gemini-2.5-Pro achieve both high unit-test scores ( $> 76\%$ ) and high precision ( $> 71\%$ ), reflecting precise fault localization and minimal edits. These results un-

derscore the necessity of edit-level evaluation to distinguish true debugging from regeneration.

### Negative correlation between unit score and precision.

We further analyze model performance as the number of injected bugs increases ( $k \in \{1, 2, 3, 4\}$ ), corresponding to increasing problem complexity. As shown in Figure 4, unit-test scores consistently decrease across all models as the number of bugs increases. At the same time, we observe an inverse trend for edit-level precision. Because models tend to over-edit, increasing the number of bugs raises the likelihood that a model modifies at least one necessary line, but also increases the amount of unnecessary edits, leading to lower precision overall. This trend is further supported by our analysis in Appendix B, which shows that precision degrades as buggy code length increases.

In contrast, recall primarily reflects debugging difficulty per bug, as it measures the fraction of bugs successfully addressed. Consistent with this interpretation, recall exhibits dataset-dependent behavior. On the API-focused BigCodeBench benchmark, where the difficulty of fixing individual bugs remains relatively stable, recall varies by less than 5% across bug counts from 1 to 4. On the algorithm-focused LiveCodeBench benchmark, where debugging difficulty increases with the number of injected bugs, recall shows a clear positive correlation with unit-test scores.

### 5.2 Iterative and agentic debugging

We randomly sample 500 instances from PDB-HARD to evaluate model behavior under *iterative* and *agentic* debugging settings. In *iterative debugging*, models produce an initial single-shot solution and are then allowed up to three revision attempts per problem. The process terminates early if a re-

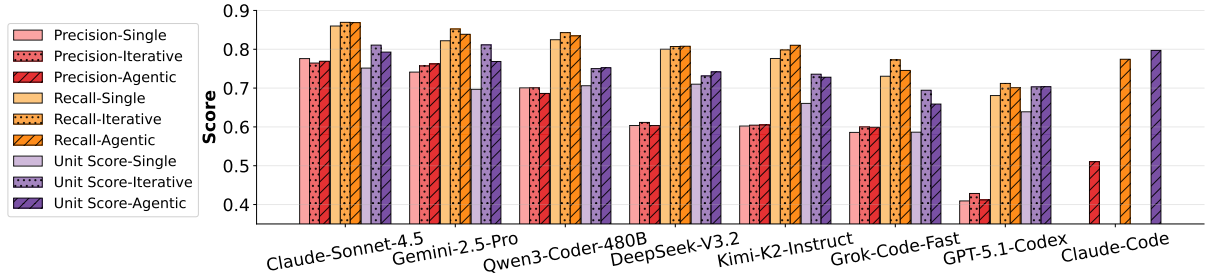


Figure 5: Both iterative and agentic setups on PDB-HARD improve unit-test pass rates and recall over single-shot debugging, indicating higher functional success. However, edit-level precision does not improve and sometimes degrades. Notably, even Claude-Code with access to unit-test and execution feedback exhibits only 50% precision.

vision passes all unit tests. Models have access to their previous failed outputs, approximating an interactive debugging workflow commonly used by human programmers. In the *agentic* setting, models are likewise permitted up to three attempts, but additionally receive unit tests and execution error feedback at each step, resembling a simple agentic debugging pipeline with explicit external feedback.

**Functional gains without precision improvements.** As shown in Figure 5, both iterative and agentic settings consistently improve unit-test scores and recall, indicating a higher likelihood of eventually producing functionally correct programs and resolving a larger fraction of ground-truth bugs. However, these gains do not translate into improved edit-level precision. In most cases, precision remains unchanged or degrades relative to single-shot debugging. This pattern suggests that iterative interaction primarily improves correctness by expanding the scope of code modifications, rather than by refining or localizing edits toward minimal repairs.

**Ineffective use of feedback in agentic debugging.** Despite direct access to unit tests and execution feedback, most models fail to leverage this information to improve edit-level behavior in the agentic setting (Figure 5). In particular, agentic debugging often underperforms iterative debugging in precision, suggesting that additional feedback may exacerbate regeneration-oriented strategies. Rather than supporting fault localization, test outcomes and error messages are frequently treated as coarse success signals that trigger further broad rewrites. These results indicate that access to feedback alone is insufficient to induce edit-aware debugging.

**Regeneration persists even in Claude-Code.** We observe similar trends in Claude-Code, which

Data	Precision	Recall	Unit (%)
Raw	73.0	83.1	76.2
Rewrite-Same-Gen	<b>76.5</b>	86.6	<b>76.4</b>
Rewrite-Different-Gen	75.8	<b>86.8</b>	74.8

Table 2: Rewriting ground truth data always makes it easier for models to debug more precisely on the buggy data, but only with different generators the model is harder to debug successfully on the buggy data.

achieves the highest precision among agentic methods but still attains only approximately 50% precision. As shown in Figure 5, this result indicates that even more sophisticated, end-to-end agentic systems largely rely on regeneration rather than precise editing, reinforcing the conclusion that current debugging agents lack robust mechanisms for localized, minimal code repair.

### 5.3 Analysis of prompting and data generation

We randomly sample 500 examples from PDB-FULL to analyze how prompting and data generation strategies affect model debugging behavior.

**Freeform vs. minimal debugging.** In our main experiments, models are explicitly instructed to perform debugging with minimal edits. To assess the impact of this constraint, we conduct an ablation in which models are instead prompted to debug freely, without any restriction on edit scope (see Appendix E for prompt details). Figure 6 compares model performance under freeform and minimal-debug prompts.

Across all evaluated models, freeform prompting results in a substantial drop in edit-level precision and bug-level recall. Even the strongest models, including Claude-Sonnet-4.5 and Qwen3-Coder-480B, achieve less than 60% precision under freeform prompting. Gemini-2.5-Pro exhibits a



Figure 6: Comparison of model performance under minimal-debug and freeform prompting on a subset of PDB-FULL. Freeform prompting leads to substantial drops in precision and recall across all models, indicating prompt-level constraints are necessary to increase debugging precision.

405 40% absolute drop in precision, indicating that its  
 406 apparent debugging precision largely stems from in-  
 407 struction following rather than intrinsic edit aware-  
 408 ness. GPT-5.1-Codex performs particularly poorly  
 409 under freeform prompts, failing to reach 20% pre-  
 410 cision. These results reinforce the regeneration  
 411 behavior discussed in the introduction and demon-  
 412 strate that *prompt-level constraints are necessary*  
 413 *but insufficient*: while minimal-debug prompts  
 414 reduce over-editing, they do not fundamentally  
 415 change underlying model behavior.

416 **Regeneration vs. contamination.** Although re-  
 417 generation dominates model behavior on PDB, an  
 418 open question is whether this tendency is driven  
 419 by data contamination, *i.e.*, overlap between bench-  
 420 mark solutions and model pretraining data. To dis-  
 421 entangle these effects, we conduct two controlled  
 422 analyses. First, we rewrite ground-truth solutions  
 423 using rewriter models (Claude-Sonnet-4.5 or GPT-  
 424 5.1-Codex), producing semantically equivalent but  
 425 surface-diverse references. Second, we generate  
 426 buggy programs using either the same model as the  
 427 rewriter or a different generator model.

428 As shown in Table 2, rewriting ground-truth solu-  
 429 tions consistently makes debugging slightly easier,  
 430 improving edit-level precision by 2.8-3.5% on aver-  
 431 age. This suggests that increased surface diversity  
 432 reduces incidental overlap and modestly improves  
 433 precise debugging. In contrast, when buggy pro-  
 434 grams are generated by a different model from the  
 435 rewriter, performance degrades, with unit-test pass  
 436 rates dropping by up to 1.4%. This indicates that  
 437 cross-model generation introduces additional vari-  
 438 ability that is more difficult for models to resolve.  
 439 Taken together, these results suggest that while  
 440 *data contamination* may marginally influence de-  
 441 bugging performance, it does not account for the  
 442 pervasive regeneration behavior observed on PDB.

#### 443 5.4 Metric verification and error analysis

444 We conduct a qualitative error analysis by manu-  
 445 ally inspecting two categories of failures: (1) cases  
 446 passing unit tests with imperfect precision or recall,  
 447 and (2) cases failing unit tests despite containing  
 448 partially correct edits. This analysis assesses the  
 449 robustness of our precision and recall metrics. We  
 450 randomly selected 240 examples from these cate-  
 451 gories and derived the taxonomy described below;  
 452 detailed examples are provided in Appendix D.

#### 453 **Passing unit tests with imperfect precision or recall.**

454 In this category, models successfully re-  
 455 solve the intended bug but introduce extraneous  
 456 modifications. In 83.5% of cases where unit tests  
 457 pass, the recall score is also 1.

458 *Precision Analysis:* In scenarios where unit tests  
 459 pass but precision < 1, we observe that 9.8% of edits  
 460 add redundant guard checks, 66.8% modify correct  
 461 code blocks, 13.7% apply correct but non-minimal  
 462 edits, and 7.8% fully regenerate the solution. No-  
 463 tably, the remaining 1.9% of patches have low pre-  
 464 cision because they fix bugs that were missing from  
 465 the ground-truth solutions. Thus, our *edit-level pre-*  
 466 *cision* accurately captures unnecessary edits.

467 *Recall Analysis:* Conversely, 16.5% of passing ex-  
 468 amples exhibit imperfect recall. In this scenario,  
 469 we find that 70% of examples are functionally cor-  
 470 rect but evade recall detection due to over-editing  
 471 or structural rewrites. Furthermore, 10% involve  
 472 compounding bugs, where an injected bug alters  
 473 program logic such that other bugs change context,  
 474 and another 20% arise because a single bug allows  
 475 for multiple minimal correct fixes. This indicates  
 476 that, provided bug independence is maintained dur-  
 477 ing dataset creation, our *bug-level recall* is accurate  
 478 for over 97.5% of all the data.

#### 479 **Failing unit tests with partially correct edits.**

480 In this category, models apply some correct edits

BugGen Model	Count	Precision	Recall	Unit (%)
GPT-5.1-Codex	1809	61.5	83.1	78.8
Gemini-2.5-Pro	1937	58.1	75.7	71.0
Claude-Sonnet-4.5	1988	49.9	67.6	67.8

Table 3: Precision, recall, and unit score comparison across source bug generation models for PDB-HARD, averaged over debug models.

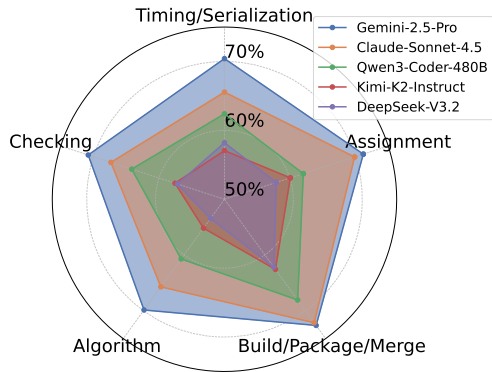


Figure 7: Recall distribution over bug categories.

but fail to fully resolve existing bugs or introduce new ones. We classify these failures into three types: (1) *Under-repair* (Recall  $< 1$ , Precision = 1): The model fixes some bugs without unnecessary edits but fails to apply all fixes (31.4%). (2) *Imprecise repair* (Recall  $< 1$ , Precision  $< 1$ ): The model both misses fixes and introduces unnecessary or harmful edits (29.4%). (3) *Regressive repair* (Recall = 1): The model fixes all original bugs but introduces new errors that cause unit tests to fail; this accounts for the majority (39.2%). This gap highlights a silent reasoning challenge unique to debugging: models must understand program structure and preserve original intent while restoring functional correctness, rather than merely generating working code.

### 5.5 Categorical analysis

We first analyze model debugging behavior by examining how performance varies across bug generation sources and defect categories. Table 3 reveals a clear ranking in task difficulty across the three bug generators in our pool: bugs generated by GPT-5.1-Codex are consistently the easiest to debug, while those generated by Claude-Sonnet-4.5 are the hardest. This ordering is consistent across all evaluated metrics.

We further examine bug-level recall across defect categories (Figure 7). With the exception of Gemini-2.5-Pro, which exhibits relatively uniform

recall of  $\sim 70\%$  across all categories, other models show markedly higher recall on the *Build/Package/Merge* category. We hypothesize that this advantage arises from the higher prevalence of such defects in model pretraining data, making them easier for models to recognize and repair.

## 6 Related Works

Debugging is a critical yet time-consuming stage of the software development lifecycle (Glass, 2002), making it a natural target for automation with LLMs. Recent systems increasingly emulate real-world debugging workflows, achieving improved performance through hierarchical multi-agent architectures (Han et al., 2024; Bouzenia et al., 2024) and agent-based data synthesis with explicit communication (Yang et al., 2024b). To evaluate such approaches, several general-purpose debugging benchmarks have been introduced, including those mined from historical bug-fixing commits (Tian et al., 2024; Siddiq et al., 2024) and those expanding coverage across programming languages (Ma et al., 2023). Other benchmarks focus on specific dimensions of debugging, such as code editing (Guo et al., 2024), systematic analyses of automated bug fixing (Sobania et al., 2023), or broader coverage of debugging scenarios (Yuan et al., 2025; Huang et al., 2025; Chai et al., 2024).

However, existing benchmarks predominantly rely on unit-test-based evaluation, which rewards models equally for rewriting large portions of code and for making minimal, targeted fixes. In contrast, PDB introduces edit-level precision and bug-level recall, exposing fundamental shortcomings in current debugging systems and aligning evaluation more closely with real-world practices.

## 7 Discussion

We show that while frontier LLMs often succeed at passing unit tests, they remain far from precise debugging, frequently relying on solution regeneration rather than targeted edits. By introducing PDB and edit-level precision and bug-level recall, we expose behaviors that unit-test-only evaluation fails to capture, revealing substantial gaps between functional correctness and genuine debugging. Results on PDB-HARD demonstrate that improving debugging performance requires rethinking both evaluation and post-training objectives, with an explicit focus on fault localization and edit minimality.

## 559 Limitation

560 First, PDB assumes bug independence when com-  
561 posing multi-bug programs, which can be difficult  
562 to guarantee in realistic software systems where  
563 bugs may interact. While such interactions are pos-  
564 sible, we empirically observe that violations of this  
565 assumption constitute only a small fraction of cases  
566 when data are generated using the PDB framework.  
567 Nonetheless, handling interacting or compounding  
568 bugs remains an open challenge.

569 Second, to ensure atomicity and reliable ground-  
570 truth edit scripts, PDB-HARD is constructed exclu-  
571 sively from single-line bugs. This design choice  
572 limits the coverage of more complex, multi-line de-  
573 fects. However, we release the full PDB pipeline,  
574 enabling future work to extend it to multi-line bug  
575 generation. Importantly, our results show that even  
576 under this simplified setting, frontier models strug-  
577 gle to perform precise debugging, indicating that  
578 the observed limitations are fundamental rather  
579 than an artifact of bug complexity.

580 Finally, although our edit-level precision and  
581 bug-level recall metrics provide a more accurate  
582 characterization of debugging behavior than unit-  
583 test-only evaluation, they may still fail to capture  
584 certain correct but semantically equivalent fixes.  
585 Incorporating more flexible semantic evaluation  
586 mechanisms, such as LLM-as-a-judge, may help  
587 address these edge cases. More broadly, reliably  
588 evaluating semantic correctness of code edits re-  
589 mains an open problem.

## 590 References

591 Loubna Ben Allal, Raymond Li, Denis Kocetkov,  
592 Chenghao Mou, Efrat Bitton, Yacine Nako, Shang-  
593 Wen Lo, Thomas Wolf, Colin Raffel, Róger Gontijo-  
594 Lopes, and 1 others. 2023. SantaCoder: don't reach  
595 for the stars! *arXiv preprint arXiv:2301.03988*.

596 Anthropic. 2025. Claude sonnet 4.5.  
597 <https://www.anthropic.com/news/claude-sonnet-4-5>.  
598

599 Islem Bouzenia, Premkumar Devanbu, and Michael  
600 Pradel. 2024. Repairagent: An autonomous, llm-  
601 based agent for program repair. *arXiv preprint*  
602 *arXiv:2403.17134*.

603 Linzheng Chai, Shukai Liu, Jian Yang, Yuwei Yin,  
604 Ke Jin, Jiaheng Liu, Tao Sun, Ge Zhang, Changyu  
605 Ren, Hongcheng Guo, and 1 others. 2024. Mce-  
606 val: Massively multilingual code evaluation. *arXiv*  
607 *preprint arXiv:2406.07436*.

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

614 Zhaoyang Chen, Yiling Liu, Haoyu Wang, Zhaoxue Liu,  
615 and Yuming Sun. 2023. Large language models for  
616 test-free fault localization. In *Proceedings of the 38th*  
617 *IEEE/ACM International Conference on Automated*  
618 *Software Engineering*, pages 680–692.

619 Ram Chillarege, Inderpal S. Bhandari, Jarir K. Chaar,  
620 Michael J. Halliday, Diane S. Moebus, Bonnie K.  
621 Ray, and Man-Yuen Wong. 1992. Orthogonal defect  
622 classification—a concept for in-process measurements.  
623 *IEEE Trans. Softw. Eng.*, 18(11):943–956.

624 Gheorghe Comanici, Eric Bieber, Mike Schaekermann,  
625 Ice Pasupat, Noveen Sachdeva, Inderjit Dhillon, Mar-  
626 cel Blistein, Ori Ram, Dan Zhang, Evan Rosen, and  
627 1 others. 2025. Gemini 2.5: Pushing the frontier with  
628 advanced reasoning, multimodality, long context, and  
629 next generation agentic capabilities. *arXiv preprint*  
630 *arXiv:2507.06261*.

631 Yiling Fan and Xin Xia. 2024. Copiloting the copilots:  
632 Fusing large language models with completion en-  
633 gines for automated program repair. In *Proceedings*  
634 *of the 46th IEEE/ACM International Conference on*  
635 *Software Engineering*, pages 63–75.

636 Jialun Fu, Jingyi Chen, Binglei Li, and Siyuan Liu.  
637 2023. A study on robustness and reliability of large  
638 language model code generation. *arXiv preprint*  
639 *arXiv:2308.13888*.

640 Robert L Glass. 2002. *Facts and fallacies of software*  
641 *engineering*. Addison-Wesley Professional.

642 Jiawei Guo, Ziming Li, Xueling Liu, Kaijing Ma,  
643 Tianyu Zheng, Zhouliang Yu, Ding Pan, Yizhi Li,  
644 Ruibo Liu, Yue Wang, and 1 others. 2024. Codeedi-  
645 torbench: Evaluating code editing capability of large  
646 language models. *arXiv preprint arXiv:2404.03543*.

647 Don-Yeong Han, Minki Kang, Seong-Hyeon Kim, and  
648 Geon-Woo Lee. 2024. Fixagent: Hierarchical multi-  
649 agent framework for unified software debugging. In  
650 *Proceedings of the 46th IEEE/ACM International*  
651 *Conference on Software Engineering*, pages 49–62.

652 Dan Hendrycks, Steven Basart, Saurav Kadavath, Man-  
653 tas Mazeika, Akul Arora, Ethan Guo, Collin Burns,  
654 Samir Puranik, Horace He, Dawn Song, and 1 others.  
655 2021. Measuring coding challenge competence with  
656 apps. *arXiv preprint arXiv:2105.09938*.

657 Jinyang Huang, Xiachong Feng, Qiguang Chen, Hanjie  
658 Zhao, Zihui Cheng, Jiesong Bai, Jingxuan Zhou, Min  
659 Li, and Libo Qin. 2025. Mldebugging: Towards  
660 benchmarking code debugging across multi-library  
661 scenarios. *arXiv preprint arXiv:2506.13824*.

662	Binyuan Hui, Jian Yang, Zeyu Cui, Jiayi Yang,	Yiqing Ma, Tri Le, Linyuan Dao, Buu Nguyen,	717
663	Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun	Hieu Nguyen, Van-Anh Nguyen, Vu Le-Hong, and	718
664	Zhang, Bowen Yu, Keming Lu, and 1 others. 2024.	Hieu Pham Nguyen. 2023. Mdeval: A massively mul-	719
665	Qwen2. 5-coder technical report. <i>arXiv preprint</i>	tilingual code debugging benchmark. <i>arXiv preprint</i>	720
666	<i>arXiv:2409.12186</i> .	<i>arXiv:2309.16885</i> .	721
667	Nafis Tanveer Islam, Joseph Khoury, Andrew Seong,	Ahmad Mohsin, Helge Janicke, Adrian Wood, Iqbal H	722
668	Mohammad Bahrami Karkevandi, Gonzalo De	Sarker, Leandros Maglaras, and Naeem Janjua. 2024.	723
669	La Torre Parra, Elias Bou-Harb, and Peyman Na-	Can we trust large language models generated code?	724
670	jafirad. 2024. Llm-powered code vulnerability repair	a framework for in-context learning, security pat-	725
671	with reinforcement learning and semantic reward.	terns, and code evaluations across diverse llms. <i>arXiv</i>	726
672	<i>arXiv preprint arXiv:2401.03374</i> .	<i>preprint arXiv:2406.12513</i> .	727
673	Wanjuan Jin, Itai Bar-Touv, Shaked Gersten, Stav Segal,	Christopher JC Ni, Jiacheng Wang, John Hewitt, Jackie	728
674	and Shai Ben-David. 2023. <b>Inferfix: End-to-end</b>	Cheung, and James L Priestley. 2022. <b>Lever: Learn-</b>	729
675	<b>program repair with LLMs</b> . In <i>2023 IEEE/ACM 45th</i>	<b>ing to verify language-to-code generation with exe-</b>	730
676	<i>International Conference on Software Engineering</i>	<b>cution</b> . In <i>Proceedings of the 44th International Con-</i>	731
677	( <i>ICSE</i> ), pages 1675–1687. IEEE.	<i>ference on Software Engineering</i> , pages 863–875.	732
678	Kimi, Yifan Bai, Yiping Bao, Guanduo Chen, Jiahao	Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan	733
679	Chen, Ningxin Chen, Ruijue Chen, Yanru Chen,	Wang, Yingbo Lin, Nazneen Rajani, Sergey Levine,	734
680	Yuankun Chen, Yutian Chen, and 1 others. 2025.	Yi Zhou, and Silvio Savarese. 2023. CodeGen: An	735
681	Kimi k2: Open agentic intelligence. <i>arXiv preprint</i>	open large language model for code with multi-turn	736
682	<i>arXiv:2507.20534</i> .	program synthesis. In <i>The Eleventh International</i>	737
683	Sumith Kulal, Panupong Pasupat, Kartik Chandra, Mina	<i>Conference on Learning Representations</i> .	738
684	Lee, Oded Padon, Alex Aiken, and Percy S Liang.	OpenAI. 2025a. <b>Gpt-5.1 codex</b> .	739
685	2019. Spoc: Search-based pseudocode to code. <i>Ad-</i>	<a href="https://openai.com/index/gpt-5-1-for-developers/">https://openai.com/index/gpt-5-1-for-developers/</a> .	740
686	<i>vances in Neural Information Processing Systems</i> ,	OpenAI. 2025b. <b>Gpt-5.2 codex</b> .	741
687	32.	<a href="https://openai.com/index/introducing-gpt-5-2-codex/">https://openai.com/index/introducing-gpt-5-2-</a>	742
688	Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas	<a href="https://openai.com/index/introducing-gpt-5-2-codex/">codex/</a> .	743
689	Muennighoff, Denis Kocetkov, Chenghao Mou, Marc	Long Phan, Hieu Tran, Daniel Le, Hieu Nguyen, James	744
690	Marone, Christopher Akiki, Jia Li, Jenny Chim, and	Anibal, Alec Peltekian, and Yanfang Ye. 2021. Co-	745
691	1 others. 2023. Starcoder: may the source be with	text: Multi-task learning with code-text transformer.	746
692	you! <i>arXiv preprint arXiv:2305.06161</i> .	<i>arXiv preprint arXiv:2105.08645</i> .	747
693	Yujia Li, David Choi, Junyoung Chung, Nate Kushman,	Qwen. 2025. <b>Qwen3 technical report</b> . <i>Preprint</i> ,	748
694	Julian Schrittwieser, Rémi Leblond, Tom Eccles,	<i>arXiv:2505.09388</i> .	749
695	James Keeling, Felix Gimeno, Agustin Dal Lago, and	Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten	750
696	1 others. 2022a. Competition-level code generation	Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi,	751
697	with alphacode. <i>Science</i> , 378(6624):1092–1097.	Jingyu Liu, Romain Sauvestre, Tal Remez, and 1	752
698	Yujia Li, David Choi, Junyoung Chung, Nate Kush-	others. 2023. Code llama: Open foundation models	753
699	man, Julian Schrittwieser, Rémi Leblond, Tom Ec-	for code. <i>arXiv preprint arXiv:2308.12950</i> .	754
700	cles, James Keeling, Felix Gimeno, Agustin Dal	Merlijn Sevenhuijsen, Khashayar Etemadi, and Mattias	755
701	Lago, Thomas Hubert, Peter Choy, Cyprien de Mas-	Nyberg. 2025. <b>Vecogen: Automating generation of</b>	756
702	son d’Autume, Igor Babuschkin, Xinyun Chen, Po-	<b>formally verified c code with large language models</b> .	757
703	Sen Huang, Johannes Welbl, Sven Gowal, Alexey	<i>Preprint</i> , <i>arXiv:2411.19275</i> .	758
704	Cherepanov, and 7 others. 2022b. <b>Competition-</b>	M. Ammar Siddiq, Islem Kaboré, Mojtaba Komeili,	759
705	<b>level code generation with alphacode</b> . <i>Science</i> ,	Homa Firooz, Abhinav Shrivastava, and Chitta Baral.	760
706	378(6624):1092–1097.	2024. Debugbench: Evaluating debugging capability	761
707	Aixin Liu, Aoxue Mei, Bangcai Lin, Bing Xue, Bingx-	of large language models. In <i>Proceedings of the</i>	762
708	uan Wang, Bingzheng Xu, Bochao Wu, Bowei	<i>IEEE/CVF Winter Conference on Applications of</i>	763
709	Zhang, Chaofan Lin, Chen Dong, and 1 others. 2025.	<i>Computer Vision</i> , pages 8647–8657.	764
710	Deepseek-v3. 2: Pushing the frontier of open large	Manav Singhal, Tushar Aggarwal, Abhijeet Awasthi,	765
711	language models. <i>arXiv preprint arXiv:2512.02556</i> .	Nagarajan Natarajan, and Aditya Kanade. 2024. No-	766
712	Fang Liu, Yang Liu, Lin Shi, Houkun Huang, Ruifeng	funeval: Funny how code lms falter on require-	767
713	Wang, Zhen Yang, Li Zhang, Zhongqi Li, and Yuchi	ments beyond functional correctness. <i>arXiv preprint</i>	768
714	Ma. 2024. Exploring and evaluating hallucinations	<i>arXiv:2401.15963</i> .	769
715	in llm-powered code generation. <i>arXiv preprint</i>		
716	<i>arXiv:2404.00971</i> .		

770	Dominik Sobania, Martin Briesch, Carol Hanna, and Justyna Petke. 2023. An analysis of the automatic bug fixing performance of chatgpt. In <i>2023 IEEE/ACM International Workshop on Automated Program Repair (APR)</i> , pages 23–30. IEEE.	825
771		826
772		827
773		828
774		
775	Florian Tambon, Arghavan Moradi-Dakhel, Amin Nikanjam, Foutse Khomh, Michel C Desmarais, and Giuliano Antoniol. 2025. Bugs in large language models generated code: An empirical study. <i>Empirical Software Engineering</i> , 30(3):65.	829
776		830
777		831
778		832
779		833
780	Runchu Tian, Yining Ye, Yujia Qin, Xin Cong, Yankai Lin, Yinxu Pan, Yesai Wu, Haotian Hui, Weichuan Liu, Zhiyuan Liu, and 1 others. 2024. Debugbench: Evaluating debugging capability of large language models. <i>arXiv preprint arXiv:2401.04621</i> .	834
781		835
782		836
783		837
784		838
785	Yue Wang, Zhe Wang, Dale Schuurmans, Hieu Le, Vincent Y Liu, Matt J Kusner, David Wang, Yiqin Li, Dusan Mandić, Yekun Shi, and 1 others. 2023. CodeT5+: Open code large language models for code understanding and generation. <i>arXiv preprint arXiv:2305.07922</i> .	839
786		840
787		841
788		842
789		843
790		844
791	xAI. 2025. Grok code fast 1. <a href="https://x.ai/news/grok-code-fast-1">https://x.ai/news/grok-code-fast-1</a> .	845
792		
793	Xin Xia and Yixuan Zhang. 2023. Automated program repair via conversation: Fixing 162 out of 337 bugs for \$0.42 each using chatgpt. <i>arXiv preprint arXiv:2304.00385</i> .	
794		
795		
796		
797	John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik R Narasimhan, and Ofir Press. 2024a. SWE-agent: Agent-computer interfaces enable automated software engineering. In <i>The Thirty-eighth Annual Conference on Neural Information Processing Systems</i> .	
798		
799		
800		
801		
802		
803	Weiqing Yang, Hanbin Wang, Zhenghao Liu, Xinze Li, Yukun Yan, Shuo Wang, Yu Gu, Minghe Yu, Zhiyuan Liu, and Ge Yu. 2024b. Coast: Enhancing the code debugging ability of llms through communicative agent based data synthesis. <i>arXiv preprint arXiv:2408.05006</i> .	
804		
805		
806		
807		
808		
809	Michihiro Yasunaga and Percy Liang. 2021. Break-it-fix-it: Unsupervised learning for program repair. In <i>International conference on machine learning</i> , pages 11941–11952. PMLR.	
810		
811		
812		
813	Pengcheng Yin, Bowen Deng, Edgar Chen, Bogdan Vasilescu, and Graham Neubig. 2018. Learning to mine aligned code and natural language pairs from stack overflow. In <i>2018 IEEE/ACM 15th international conference on mining software repositories (MSR)</i> , pages 476–486. IEEE.	
814		
815		
816		
817		
818		
819	Xingdi Yuan, Morgane M Moss, Charbel El Feghali, Chinmay Singh, Darya Moldavskaya, Drew MacPhee, Lucas Caccia, Matheus Pereira, Minseon Kim, Alessandro Sordoni, and 1 others. 2025. debug-gym: A text-based environment for interactive debugging. <i>arXiv preprint arXiv:2503.21557</i> .	
820		
821		
822		
823		
824		
	Li Zhong, Zilong Wang, and Jingbo Shang. 2024. Debug like a human: A large language model debugger via verifying runtime execution step-by-step. <i>arXiv preprint arXiv:2402.16906</i> .	
	Ming Zhong, Xiang Zhou, Ting-Yun Chang, Qingze Wang, Nan Xu, Xiance Si, Dan Garrette, Shyam Upadhyay, Jeremiah Liu, Jiawei Han, and 1 others. 2025. Vibe checker: Aligning code evaluation with human preference. <i>arXiv preprint arXiv:2510.07315</i> .	
	Yixuan Zhu, Zhitong Zeng, Zhaoxue Liu, Yixing Feng, Yuming Sun, Zhaoyang Chen, Yiling Liu, and Haoyu Wang. 2024. Livecodebench: Holistic and contamination free evaluation of large language models for code. In <i>Proceedings of the 12th International Conference on Learning Representations (ICLR)</i> .	
	Terry Yue Zhuo, Minh Chien Vu, Jenny Chim, Han Hu, Wenhao Yu, Ratnadira Widayarsi, Imam Nur Bani Yusuf, Haolan Zhan, Junda He, Indraneil Paul, and 1 others. 2024. Bigcodebench: Benchmarking code generation with diverse function calls and complex instructions. <i>arXiv preprint arXiv:2406.15877</i> .	

## Appendix

<b>A Additional Related Works</b>	<b>12</b>
<b>B Additional Experiments</b>	<b>12</b>
B.1 Additional results on PDB-HARD	12
B.2 Additional results on PDB-FULL	13
<b>C Algorithm on Precision and Recall</b>	<b>13</b>
<b>D Examples of Debugging Categories</b>	<b>13</b>
<b>E Prompt templates</b>	<b>13</b>
<b>F Checklist Information</b>	<b>13</b>

### A Additional Related Works

**Code Generation.** The capability of LLMs in code generation has been transforming both academia and industry. Beginning with seminal models like Codex (Chen et al., 2021), the field has rapidly advanced with the introduction of dozens of powerful code-centric models including Code Llama (Roziere et al., 2023), StarCoder (Li et al., 2023; Allal et al., 2023), CodeGen (Nijkamp et al., 2023), CodeT5+(Wang et al., 2023), Qwen Coder Series (Hui et al., 2024; Qwen, 2025), and more recent GPT-5.1 and 5.2 codex (OpenAI, 2025a,b). These models, trained on vast web-scale datasets of code, excel at synthesizing end-to-end programs from natural language prompts. To evaluate their capabilities, numerous benchmarks have been established, ranging from function-level synthesis tasks like HumanEval (Chen et al., 2021) and CoNaLa (Yin et al., 2018), to more complex challenges including APPS (Hendrycks et al., 2021), CodeContests (Li et al., 2022b), SPOC (Kulal et al., 2019), and BigCodeBench (Zhuo et al., 2024). Beyond functional correctness, works like Vibe Checker (Zhong et al., 2025) and NoFunEval (Singhal et al., 2024) target the evaluation of models’ non-functional instruction-following abilities. Recent agent-based systems like SWE-Agent (Yang et al., 2024a) and AutoGen (Sevenhuijsen et al., 2025) demonstrate the potential of LLMs in autonomous development workflows.

**Debugging Frameworks.** As a critical and often time-consuming task, debugging has naturally emerged as another target for automation using LLMs. This need is further magnified by the fact that code generation models themselves are

Model	Precision	Recall	Unit (%)
Claude-Sonnet-4.5	<b>78.5±0.7</b>	86.7±0.6	82.2±0.9
Gemini-2.5-Pro	78.3±0.7	<b>88.5±0.6</b>	84.2±0.8
Qwen3-Coder-480B	74.2±0.8	83.7±0.7	77.7±0.9
Kimi-K2-Instruct	66.6±0.8	80.5±0.7	73.3±1.0
Grok-Code-Fast	64.2±0.9	<b>74.5±0.8</b>	<b>67.3±1.1</b>
Kimi-K2-Thinking	61.6±0.8	82.6±0.7	81.3±0.9
DeepSeek-V3.2	58.9±0.9	77.5±0.8	78.8±0.9
DeepSeek-V3.2-Thinking	56.1±0.9	78.3±0.8	<b>85.4±0.8</b>
GPT-5.1-Codex	<b>50.6±0.8</b>	78.3±0.8	82.6±0.9

Table 4: Precision, recall, and unit score on the PDB-FULL set. Blue indicates better performance, while red indicates worse.

a significant source of buggy and potentially vulnerable code (Ni et al., 2022; Fu et al., 2023; Jin et al., 2023; Mohsin et al., 2024; Tambon et al., 2025; Liu et al., 2024). Consequently, a spectrum of approaches have been proposed to leverage these models for program repair. Early work like Break-It-Fix-It (Yasunaga and Liang, 2021) introduced unsupervised learning for program repair, while CoText (Phan et al., 2021) explored multi-task learning with code-text transformers. Recent systems emulate real-world debugging workflows through sophisticated agent architectures: FixAgent (Han et al., 2024) employs hierarchical multi-agent frameworks, RepairAgent (Bouzenia et al., 2024) demonstrates autonomous repair capabilities, and COAST (Yang et al., 2024b) enhances debugging through communicative agent-based data synthesis. These approaches utilize techniques ranging from zero-shot prompting to multi-turn conversational agents (Chen et al., 2023; Fan and Xia, 2024; Xia and Zhang, 2023; Zhong et al., 2024; Islam et al., 2024).

### B Additional Experiments

We report additional experimental results in this section.

#### B.1 Additional results on PDB-HARD

We list the rewriting figure in Figure 8, which shares a similar but breakdown finding as Table 2. Moreover, we show the model-averaged performance on PDB-HARD over the distribution of buggy code length. All metrics have a similar performance drop, suggesting that increasing code length improves code completion difficulty and makes it harder for models to hit the necessary edits at the same time.

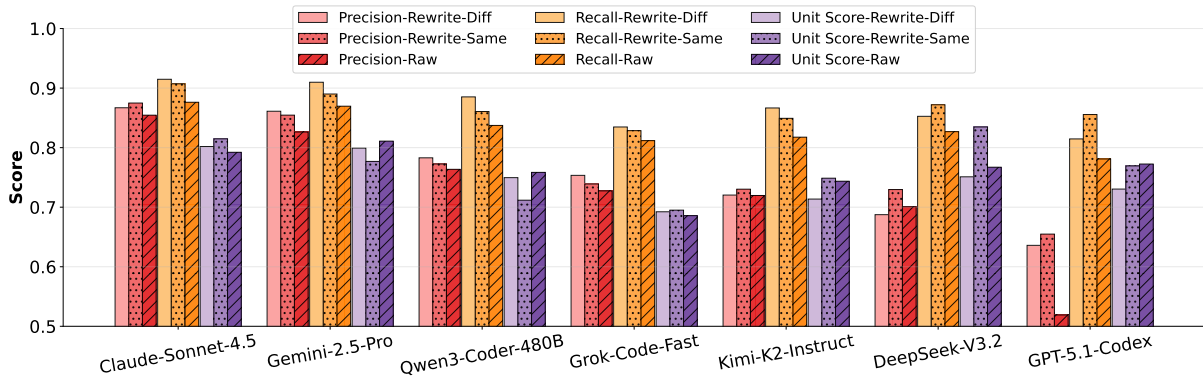


Figure 8: Model breakdown performance on PDB-HARD rewriting with the same generator, or a different generator.

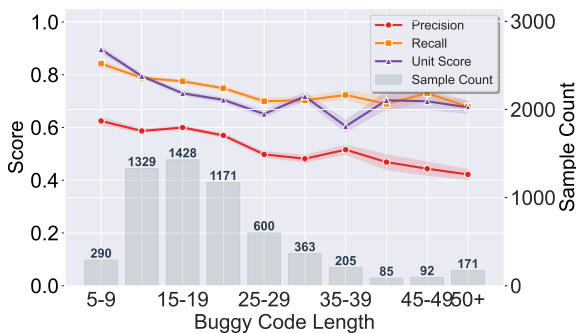


Figure 9: Model averaged performance on PDB-HARD over distribution of buggy code length. All metrics show a similar performance drop.

## B.2 Additional results on PDB-FULL

We list model performance on three metrics on PDB-FULL in Table 4, which are 4 – 8% higher than the results in Table 1 on PDB-HARD.

The negative correlation of precision and unit score over bug counts, and the positive correlation of recall and unit score over bug counts, are both clearer in the PDB-FULL set as described in Figure 10.

## C Algorithm on Precision and Recall

Following the definitions in §2, we formalize the block matching functions  $\text{map}$  and  $\text{map}_\epsilon$  in Algorithm 1 and Algorithm 2, respectively.

The function  $\text{map}$  performs edit alignment between ground-truth and predicted patches by first identifying exact line-level matches and then resolving block-level correspondences using structural containment, local contextual similarity, and content equality.

For cases where  $\epsilon$  relaxation is used, we have to verify redundant editing over  $\epsilon$ ,  $\text{map}_\epsilon$  extends this procedure by incorporating semantic verification

through unit-test evaluation, allowing a bounded tolerance of up to  $\epsilon$  additional edits. By explicitly validating semantic equivalence and minimizing effective edit scope within this tolerance,  $\text{map}_\epsilon$  yields a robust matching that supports relaxed precision evaluation while remaining faithful to targeted debugging behavior, which is also examined qualitatively in §5.4.

## D Examples of Debugging Categories

We show different categories of debugging with examples from Figure 11-19.

## E Prompt templates

We provide the prompt templates used in our experiments, ranging from bug injection and solution rewriting to minimal and free-form debugging with optional unit tests and execution feedback, shown in Figures 20–31.

## F Checklist Information

**Risks of malicious use of PDB pipeline.** PDB provides a systematic procedure for producing realistic buggy programs from existing code by prompting deliberate fault introduction. Hence, the same pipeline that supports controlled debugging evaluation could be repurposed for malicious bug-injection at scale, enabling automated generation of large quantities of plausible faulty code with minimal surface changes. Such capability may be misused to degrade software reliability in collaborative development settings, increase the review burden on maintainers, or seed low-quality code into shared repositories.

Another risk concern is potential data poisoning and model capability shaping. Because PDB converts coding data into structured buggy program

---

**Algorithm 1:** map: Match predicted edits to ground-truth edit blocks and construct equivalence-test candidates
 

---

**Input:** Buggy program  $C_b$ ; ground-truth diff  $\Delta_{gt}$ ; predicted diff  $\Delta_{pred}$ 
**Output:** Matched blocks  $\mathcal{M}$ ; unmatched predicted edits  $\Delta_{pred}^-$ ; unmatched ground-truth edits  $\Delta_{gt}^-$ ; equivalence-test set  $\mathcal{T}$ 
 $\mathcal{M} \leftarrow \emptyset, \mathcal{T} \leftarrow \emptyset, \Delta_{pred}^- \leftarrow \emptyset$ 
 $\Delta_{gt}^{rem} \leftarrow \Delta_{gt}, \Delta_{pred}^{rem} \leftarrow \Delta_{pred}$ 
 $\mathcal{B}_{gt}^{rem} \leftarrow \text{PARSETOBLOCKS}(\Delta_{gt}^{rem})$ 
 $\mathcal{B}_{pred}^{rem} \leftarrow \text{PARSETOBLOCKS}(\Delta_{pred}^{rem})$ 
 $L \leftarrow \text{SPLITLINES}(C_b)$ 
**Pass 1: Exact line-level matches (EM).**
**foreach** predicted edit  $(\ell, v) \in \Delta_{pred}^{rem}$  **in descending**  $\ell$  **do**
 $\ell' \leftarrow \ell$  // line index in GT (same coordinate system)

**if**  $\ell' \in \Delta_{gt}^{rem}$  **and**  $\text{EQUAL}(v, \Delta_{gt}^{rem}[\ell'])$  **then**
 $\mathcal{M} \leftarrow \mathcal{M} \cup \{\text{MAKEEMBLOCK}(\ell, v)\}$ 

 remove  $\ell'$  from  $\Delta_{gt}^{rem}$  and remove  $\ell$  from  $\Delta_{pred}^{rem}$ 

 remove the GT block starting at  $\ell$  from  $\mathcal{B}_{gt}^{rem}$ 
 $\mathcal{B}_{pred}^{rem} \leftarrow \text{PARSETOBLOCKS}(\Delta_{pred}^{rem})$ 
**Pass 2: Block-level matching.**
**for**  $j \leftarrow 1$  **to**  $|\mathcal{B}_{pred}^{rem}|$  **do**
 $B^{pred} \leftarrow \mathcal{B}_{pred}^{rem}[j]$ 
 $\mathcal{G} \leftarrow \emptyset$ 

// matched GT blocks for this predicted block

**(2.1) Wrap match: predicted block covers GT block start.**
**foreach**  $B^{gt} \in \mathcal{B}_{gt}^{rem}$  **do**
**if**  $B^{pred}.start \leq B^{gt}.start \leq B^{pred}.end$  **then**
 $\mathcal{G} \leftarrow \mathcal{G} \cup \{B^{gt}\}$ 
**(2.2) Near match: context-line overlap before and after.**
**if**  $\mathcal{G} = \emptyset$  **then**
 $S_{pred}^- \leftarrow \text{CONTEXTBEFORE}(L, B^{pred}, \mathcal{B}_{pred}^{rem})$ 
 $S_{pred}^+ \leftarrow \text{CONTEXTAFTER}(L, B^{pred}, \mathcal{B}_{pred}^{rem})$ 
**foreach**  $B^{gt} \in \mathcal{B}_{gt}^{rem}$  **do**
 $S_{gt}^- \leftarrow \text{STRIDECONTEXTBEFORE}(L, B^{gt})$ 
 $S_{gt}^+ \leftarrow \text{STRIDECONTEXTAFTER}(L, B^{gt})$ 
**if**  $\text{LINESETMATCH}(S_{pred}^-, S_{gt}^-)$  **and**  $\text{LINESETMATCH}(S_{pred}^+, S_{gt}^+)$  **then**
 $\mathcal{G} \leftarrow \{B^{gt}\}$ 
**break**
**(2.3) Distant-but-identical: single-line equality.**
**if**  $\mathcal{G} = \emptyset$  **and**  $|B^{pred}.\Delta| = 1$  **then**
**foreach**  $B^{gt} \in \mathcal{B}_{gt}^{rem}$  **do**
**if**  $\text{EQUAL}(\text{ONLYVALUE}(B^{pred}.\Delta), \text{ONLYVALUE}(B^{gt}.\Delta))$  **then**
 $\mathcal{G} \leftarrow \{B^{gt}\}$ 
**break**
**Update matches and construct equivalence tests.**
**if**  $\mathcal{G} \neq \emptyset$  **then**
 $I \leftarrow \{B.id \mid B \in \mathcal{G}\}$ 
 $B^{test} \leftarrow (\text{ALLGTBLOCKS} \setminus \{B : B.id \in I\}) \cup \{B^{pred}\}$ 
 $\Delta^{test} \leftarrow \text{EXPANDBLOCKS}(B^{test})$ 
 $C^{test} \leftarrow \text{APPLY}(C_b, \Delta^{test})$ 
 $\mathcal{T} \leftarrow \mathcal{T} \cup \{(j, C^{test})\}$ 
 $\tau \leftarrow \max(\min(|\mathcal{G}| \cdot (\epsilon - 1), |B^{pred}.\Delta| - |\mathcal{G}|), 0)$ 
 $\mathcal{M} \leftarrow \mathcal{M} \cup \{\text{MAKEMATCH}(B^{pred}, \mathcal{G}, I, \tau)\}$ 

 remove all blocks in  $\mathcal{G}$  from  $\mathcal{B}_{gt}^{rem}$ 
**else**
 $\Delta_{pred}^- \leftarrow \Delta_{pred}^- \cup B^{pred}.\Delta$ 
 $\Delta_{gt}^- \leftarrow \text{EXPANDBLOCKS}(\mathcal{B}_{gt}^{rem})$ 
**return**  $\mathcal{M}, \Delta_{pred}^-, \Delta_{gt}^-, \mathcal{T}$ 


---

---

**Algorithm 2:**  $\text{map}_\epsilon$ :  $\epsilon$ -tolerant edit matching with semantic verification

---

**Input:** Buggy program  $C_b$ ; GT blocks  $\mathcal{B}_{\text{gt}}$ ; diffs  $\Delta_{\text{gt}}, \Delta_{\text{pred}}$ ; tolerance budget  $\epsilon$ ; unit-test oracle  $F_U(\cdot)$

**Output:** Final matching  $\mathcal{M}_\epsilon$  with fields {success, tolerance, effective\_starter}

**Step 1: Candidate matching via map.**

$(\mathcal{M}, \Delta_{\text{pred}}^-, \Delta_{\text{gt}}^-, \mathcal{T}) \leftarrow \text{map}(C_b, \Delta_{\text{gt}}, \Delta_{\text{pred}})$

//  $\mathcal{T}$  contains equivalence-test candidates  $(\text{id}, C^{\text{test}})$  built by replacing matched GT blocks with predicted blocks.

**Step 2: Semantic equivalence verification using  $F_U$ .**

**foreach**  $(\text{id}, C^{\text{test}}) \in \mathcal{T}$  **do**

**if**  $F_U(C^{\text{test}}) = 1$  **then**

$\mathcal{M}[\text{id}].\text{success} \leftarrow \text{True}$

**else**

$\mathcal{M}[\text{id}].\text{success} \leftarrow \text{False}$

$\mathcal{M}[\text{id}].\text{tolerance} \leftarrow 0$ ;

$\mathcal{M}[\text{id}].\text{gt\_match\_count} \leftarrow 0$ ;

**Step 3: Deep redundancy check to realize effective  $\epsilon$ -tolerance.**

**foreach** *match record*  $r \in \mathcal{M}$  **with**  $r.\text{success} = \text{True}$  **and**  $r.\text{tolerance} > 0$  **do**

$B^{\text{pred}} \leftarrow r.\text{pred\_block}$ ;  $I \leftarrow r.\text{gt\_match\_ids}$

$\mathcal{S} \leftarrow \emptyset$

    // candidate sub-blocks

    // Enumerate smaller contiguous sub-diffs within the predicted block.

    let  $\text{items}(B^{\text{pred}}.\Delta) = [(\ell_1, v_1), \dots, (\ell_m, v_m)]$  ordered by  $\ell$

**for**  $\tau \leftarrow 0$  **to**  $r.\text{tolerance} - 1$  **do**

**for**  $j \leftarrow 1$  **to**  $m - \tau$  **do**

$B^{\text{sub}} \leftarrow \text{SUBBLOCK}(B^{\text{pred}}, j, \tau + 1)$

$B^{\text{test}} \leftarrow (\mathcal{B}_{\text{gt}} \setminus \{B : B.\text{id} \in I\}) \cup \{B^{\text{sub}}\}$

$C^{\text{sub}} \leftarrow \text{APPLY}(C_b, \text{EXPANDBLOCKS}(B^{\text{test}}))$

$\mathcal{S} \leftarrow \mathcal{S} \cup \{(\tau, j, C^{\text{sub}})\}$

    // Find the smallest  $\tau$  (and a starter index  $j$ ) that still passes  $F_U$ .

$(\tau^*, j^*) \leftarrow (+\infty, \perp)$

**foreach**  $(\tau, j, C^{\text{sub}}) \in \mathcal{S}$  **do**

**if**  $F_U(C^{\text{sub}}) = 1$  **and**  $\tau < \tau^*$  **then**

$(\tau^*, j^*) \leftarrow (\tau, j)$

**if**  $\tau^* < +\infty$  **then**

$r.\text{tolerance} \leftarrow \tau^*$ ;

$r.\text{effective\_starter} \leftarrow j^*$ ;

$\mathcal{M}_\epsilon \leftarrow \mathcal{M}$

**return**  $\mathcal{M}_\epsilon$

---

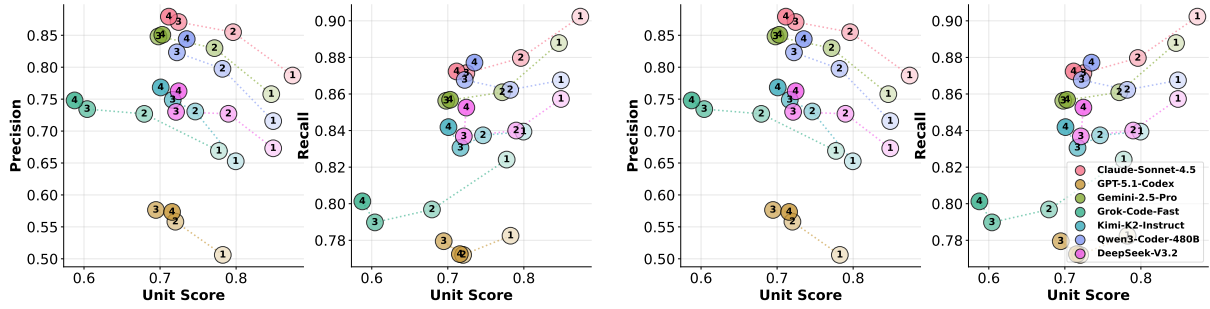


Figure 10: Correlation between precision, recall, and unit-test score across bug counts. Results are shown on subsets of PDB-FULL from BigCodeBench (left) and LiveCodeBench (right), with bug counts indicated by numbers. As the number of bugs increases, precision generally exhibits a negative correlation with unit-test score, while recall displays dataset-dependent behavior.

ODC Category	Sub-category	Brief Description
Assignment	Mutability Trap	Mutable default arguments cause unintended shared state across calls.
	Late Binding in Closures	Loop variables captured by reference, yielding unexpected final values.
	List Multiplication Surprise	List multiplication creates multiple references to the same inner object.
	Built-in Shadowing	Assigning to names like <code>list</code> or <code>sum</code> hides built-ins.
	Variable Shadowing	Inner-scope variables obscure outer-scope references.
	Name Error	Variable is used before being assigned or defined.
Checking	Off-by-One Error	Boundary condition is shifted by exactly one element or unit.
	Negation Error	Boolean condition is logically inverted.
	Missing or Incomplete Checks	Absent validation leads to runtime errors (e.g., <code>KeyError</code> , <code>TypeError</code> ).
	Overwriting Built-in Names	Built-in identifiers are reassigned, breaking later function calls.
	Variable Shadowing	Confusing variable scope leads to incorrect condition evaluation.
	Chained Boolean Comparison Logic	Misparsed chained comparisons yield unintended logic.
	Implicit Boolean Conversion	Empty collections and <code>None</code> are conflated in boolean context.
	Membership Logic Flaws	Misunderstanding how membership tests behave for data types.
Algorithm	Wrong Math Expression	Mathematical formula or operands are incorrectly specified.
	Modifying While Iterating	Collection is altered during iteration, skipping or misprocessing elements.
	Function Algorithm Misunderstanding	Function behavior is misunderstood (e.g., substring vs. set semantics).
	Function Argument Misunderstanding	Incorrect interpretation of function arguments or defaults.
	Infinite Loop / Recursion	Termination condition is missing or unreachable.
	Other Logical Errors	Deeper algorithmic invariants are violated during execution.
Build/Package/Merge	Invalid API Call	Method is invoked on an unsupported data type or abstraction.
	Dependency Version Conflicts	Code relies on APIs removed or changed across library versions.
Timing/Serialization	Serialization Issue	Non-serializable objects are passed to pickle or JSON encoders.
	Async Blocking	Blocking calls inside async code stall the event loop.

Table 5: ODC-style taxonomy of common programming defects with summarized descriptions. These are used as in-context examples.

and solution pairs, it can lower the cost of creating large synthetic corpora that contain intentional buggy programs with minimal-edit transformations. If used outside the intended research context, these data could be silently employed to bias training toward behaviors that facilitate code degradation, or to contaminate downstream datasets used for model deployment and benchmarking. Even when the immediate artifacts are non-sensitive, the potential, silent shift in model behavior raises concerns.

**Risks of malicious use of PDB-HARD.** PDB-HARD concentrates challenging debugging instances derived from benchmark-style programming tasks (LiveCodeBench and BigCodeBench). Using such data to train code-editing or debugging

models is a natural extension of its intended role, which includes training models that can both repair and introduce faults under different objectives. The potential risk arises from how this capability is used and framed: if the PDB-HARD (or derivatives) is used to optimize for fault insertion or to condition models toward producing plausible bugs with localized edits, it could support misuse in settings where code integrity matters.

**Licensing landscape of evaluated models.** The governance of the evaluated LLMs and benchmarks reveals a sharp dichotomy between proprietary services and open-weight ecosystems. The proprietary tier includes GPT-5.1-Codex (OpenAI), Claude-Sonnet-4.5 (Anthropic), Gemini-2.5-Pro

1011 (Google DeepMind), and Grok-Code-Fast (xAI),  
1012 all of which are accessible exclusively via commer-  
1013 cial APIs. These systems are governed by restric-  
1014 tive Terms of Service that prohibit model weight  
1015 extraction, reverse engineering, and competitive  
1016 distillation, serving to protect their respective archi-  
1017 tectural innovations and agentic harnesses. In con-  
1018 trast, the open-weight landscape is characterized by  
1019 permissive licensing designed to commoditize rea-  
1020 soning capabilities: DeepSeek-V3.2 and its reason-  
1021 ing variant DeepSeek-V3.2-Thinking are released  
1022 under the MIT License, while Qwen3-Coder-480B  
1023 utilizes the Apache License 2.0, which includes an  
1024 explicit patent grant. A hybrid governance model  
1025 is observed in Kimi-K2-Thinking and Kimi-K2-  
1026 Instruct, which operate under a Modified MIT Li-  
1027 cense; this variant permits general commercial use  
1028 but mandates strictly visible attribution for entities  
1029 exceeding 100 million monthly active users or \$20  
1030 million in monthly revenue.

1031 **Licensing landscape of datasets.** Regarding  
1032 evaluation frameworks, BigCodeBench is gov-  
1033 erned by the Apache License 2.0, whereas Live-  
1034 CodeBench adopts a split licensing model with its  
1035 codebase under the MIT License and its dataset  
1036 artifacts available under the Creative Commons  
1037 Attribution 4.0 International License (CC-BY 4.0).

1038 **Use of LLM.** We use LLMs to generate buggy  
1039 programs and debug buggy programs in our experi-  
1040 ments, and to improve writing fluency and correct  
1041 grammatical errors.

**Task Description**

Archive a specified project directory into a ZIP file based on the configuration specified in a config file. This function reads a configuration file to determine the project directory and archives this directory into a ZIP file. The ZIP file's name will be the project directory's basename, stored in the specified archive directory. Configuration File Format: [Project] directory=path\_to\_project\_directory

**Exceptions:**  
FileNotFoundError: If the `config\_file\_path` does not exist or the specified project directory does not exist. Exception: If the ZIP archive cannot be created.

**Outputs:**  
bool: True if the ZIP archive is successfully created, otherwise an exception is raised.

**Buggy Code**

```

1. import configparser
2. import os
3. import shutil
4. def task_func(config_file_path, archive_dir = '/home/user/arch
   archive'):
5.
6.     config = configparser.ConfigParser()
7.     config.read(config_file_path)
8.
9.     project_dir = config.get('Project', 'directory')
10.
11.    if not os.path.isdir(project_dir):
12.        raise FileNotFoundError(f'Directory {project_dir} does
   not exist.')
13.
14.    archive_file = f'{archive_dir}/{project_dir}.z
   ip'
15.
16.    # Using shutil to create the zip archive
17.    shutil.make_archive(base_name=os.path.splitext(archive_fil
   e)[0], format='zip', root_dir=project_dir)
18.
19.    if not os.path.isfile(archive_file):
20.        raise Exception(f'Failed to create archive {archive_fi
   le}')
21.
22.    return True

```

■ Add: Line added to inject bug   
 ■ Delete: Line removed to inject bug  
■ Modify: Line changed to inject bug

Bug Injection: gpt-5.1-codex

**Model Generated Solution**

```


1. import configparser
2. import os
3. import shutil
4. def task_func(config_file_path, archive_dir = '/home/user/arch
   archive'):
5.
6.     if not os.path.isfile(config_file_path):
7.         raise FileNotFoundError(f"Config file {config_file_pat
   h} does not exist.")
8.
9.     config = configparser.ConfigParser()
10.    config.read(config_file_path)
11.
12.    project_dir = config.get('Project', 'directory')
13.
14.    if not os.path.isdir(project_dir):
15.        raise FileNotFoundError(f'Directory {project_dir} does
   not exist.')
16.
17.    archive_file = f'{archive_dir}/{os.path.basename(proj
   ect_dir)}.zip'
18.
19.    # Using shutil to create the zip archive
20.    shutil.make_archive(base_name=os.path.splitext(archive_fil
   e)[0], format='zip', root_dir=project_dir)
21.
22.    if not os.path.isfile(archive_file):
23.        raise Exception(f'Failed to create archive {archive_fi
   le}')
24.
25.    return True

```

■ Fixed bug   
 ■ Extra edit   
 ■ Introduced bug

Debug Model: gemini-2.5-pro

Figure 11: Redundant guard checks (9.8%): The model adds unnecessary defensive checks that don't affect correctness.

 **Task Description**

Draw a circle with a given radius on the polar chart 'ax' and set radial ticks. This function manipulates plot data using matplotlib.

**Note:** If the radius is negative this function will raise ValueError. If 'ax' is not a polar plot this function will raise TypeError.

**Outputs:**  
matplotlib.axes.\_axes.Axes: The modified Axes object with the circle plotted.

**● Buggy Code**

```

1. import matplotlib.pyplot as plt
2. import numpy as np
3. def task_func(ax, radius):
4.
5.     if radius < 0:
6.         raise ValueError('Radius must be non-negative')
7.     if not isinstance(ax, plt.PolarAxes):
8.         raise TypeError('ax must be a polar plot')
9.
10.    theta = np.linspace(0, 2 * np.pi, 1000)
11.    ax.plot(theta, radius * np.ones_like(theta))
12.    ax.set_rlabel_position(radius * 45)
13.    ax.draw()
14.    return ax

```

■ Add: Line added to inject bug    ■ Delete: Line removed to inject bug  
■ Modify: Line changed to inject bug

Bug Injection: claude-sonnet-4.5

**● Model Generated Solution**

```

1. import matplotlib.pyplot as plt
2. import numpy as np
3. def task_func(ax, radius):
4.
5.     if radius < 0:
6.         raise ValueError('Radius must be non-negative')
7.     if ax.name != 'polar':
8.         raise TypeError('ax must be a polar plot')
9.
10.    theta = np.linspace(0, 2 * np.pi, 1000)
11.    ax.plot(theta, radius * np.ones_like(theta))
12.    ax.set_rlabel_position(45)
13.    return ax

```

■ Fixed bug    ■ Extra edit    ■ Introduced bug

Debug Model: Kimi-K2-Thinking

Figure 12: Additional modifications (66.8%): The model makes additional modifications to correct code blocks beyond what is required to fix the bug.

### Task Description

Organize files in a directory based on the first text that is not enclosed in square brackets. Move the files to subdirectories named after this text. If no matching text is found, the file is not moved.

#### Outputs:

tuple: str: The directory path with organized files. dict: A dictionary where keys are the created subdirectories and values are lists of files moved to them.

#### Buggy Code

```
1. import re
2. import os
3. import shutil
4. from datetime import datetime
5. def task_func(directory):
6.
7.     DATE_FORMAT = '%Y%m%d%H%M%S'
8.     moved_files = {}
9.     for filename in os.listdir(directory):
10.         with open(os.path.join(directory, filename), 'r') as f
11.             file:
12.                 content = file.read()
13.                 match = re.search('(.*?)\[.*?\]', content)
14.                 if match:
15.                     subdirectory = match.group(1).strip()
16.
17.                     if not os.path.exists(os.path.join(directory,
18. subdirectory)):
19.                         os.makedirs(os.path.join(directory, subdi
20. rectory))
21.
22.                         new_filename = f"{filename.split('.')[0]}_{dat
23. etime.now().strftime(DATE_FORMAT)}_{filename.split('.')[1]}"
24.                         shutil.move(os.path.join(directory, filename),
25. os.path.join(directory, subdirectory, new_filename))
26.
27.                         if subdirectory not in moved_files:
28.                             moved_files[subdirectory] = []
29.                             moved_files[subdirectory].append(new_filename)
30.
31.     return directory.resolve(), moved_files
```

- Add: Line added to inject bug
- Delete: Line removed to inject bug
- Modify: Line changed to inject bug

Bug Injection: claude-sonnet-4.5

#### Model Generated Solution

```
1. import re
2. import os
3. import shutil
4. from datetime import datetime
5. def task_func(directory):
6.
7.     DATE_FORMAT = '%Y%m%d%H%M%S'
8.     moved_files = {}
9.     for filename in os.listdir(directory):
10.         filepath = os.path.join(directory, filename)
11.         if os.path.isfile(filepath):
12.             with open(filepath, 'r') as file:
13.                 content = file.read()
14.                 match = re.search(r'^(.*?)\[.*?\](.*)$', con
15. tent, re.DOTALL)
16.                 if not match:
17.                     match = re.search(r'\[(.*?)\](.*)$', conte
18. nt, re.DOTALL)
19.                 if match:
20.                     before_brackets = content[:match.start
21. ()]
22.                     first_non_bracket_text = before_bracke
23. ts.split()[-1] if before_brackets.split() else ""
24.                     subdirectory = first_non_bracket_text
25.                     else:
26.                         continue
27.                     else:
28.                         before_first_bracket = match.group(1)
29.                         parts = before_first_bracket.strip().split
30. ()
31.                         subdirectory = parts[-1] if parts else ""
32.
33.                 if not subdirectory:
34.                     continue
35.
36.                 if not os.path.exists(os.path.join(directory,
37. subdirectory)):
38.                     os.makedirs(os.path.join(directory, subdi
39. rectory))
40.
41.                     name_part, ext_part = os.path.splitext(filename)
42.                     new_filename = f"{name_part}_{datetime.now().s
43. trftime(DATE_FORMAT)}_{ext_part}"
44.                     shutil.move(filepath, os.path.join(directory,
45. subdirectory, new_filename))
46.
47.                     if subdirectory not in moved_files:
48.                         moved_files[subdirectory] = []
49.                         moved_files[subdirectory].append(new_filename)
50.
51.     return os.path.abspath(directory), moved_files
```

- Fixed bug
- Extra edit
- Introduced bug

Debug Model: Qwen3-Coder-480B-A35B-Instruct-FP8

Figure 13: Complete rewrite (7.8%): The model completely regenerates the solution rather than making minimal targeted fixes.

**Task Description**

Unzip a list of objects and their 3D coordinates, run PCA to reduce the dimensionality to 2D, and depending on the value of `save_plot` parameter, either save the plot to the provided path and return the 2D coordinates or return the 2D coordinates and the plot's Axes.

**Exceptions:**  
ValueError: If `save_plot` is True but `plot_path` is not provided.

**Outputs:**  
`coordinates_2d` (numpy.ndarray): The 2D coordinates after applying PCA. `ax` (matplotlib.axes.\_axes.Axes, optional): The plot's Axes if `save_plot` is True.

● **Buggy Code**

```

1. import numpy as np
2. from sklearn.decomposition import PCA
3. import matplotlib.pyplot as plt
4. def task_func(data, save_plot=False, plot_path=None):
5.
6.     items, x_values, y_values, z_values = zip(*data)
7.     coordinates = np.array(list(zip(x_values, y_values, z_valu
8. es)))
9.
10.    pca = PCA(n_components=2)
11.    coordinates_2d = pca.fit_transform(coordinates)
12.
13.    # Initialize a fresh plot
14.    plt.figure()
15.    fig, ax = plt.subplots()
16.    ax.scatter(*zip(*coordinates_2d))
17.
18.    if save_plot:
19.        if plot_path is not None == True:
20.            plt.savefig(plot_path)
21.            plt.close(fig)
22.            return coordinates_2d, ax
23.        else:
24.            raise ValueError("plot_path is required if save_pl
25. ot is True")
26.        else:
27.            return coordinates_2d

```

■ Add: Line added to inject bug   
■ Delete: Line removed to inject bug  
■ Modify: Line changed to inject bug

Bug Injection: claude-sonnet-4.5

● **Model Generated Solution**

```

1. import numpy as np
2. from sklearn.decomposition import PCA
3. import matplotlib.pyplot as plt
4. def task_func(data, save_plot=False, plot_path=None):
5.
6.     items, x_values, y_values, z_values = zip(*data)
7.     coordinates = np.array(list(zip(x_values, y_values, z_valu
8. es)))
9.
10.    pca = PCA(n_components=2)
11.    coordinates_2d = pca.fit_transform(coordinates)
12.
13.    # Initialize a fresh plot
14.    plt.figure()
15.    fig, ax = plt.subplots()
16.    ax.scatter(*zip(*coordinates_2d))
17.
18.    if save_plot:
19.        if plot_path is not None:
20.            plt.savefig(plot_path)
21.            plt.close(fig)
22.            return coordinates_2d, ax
23.        else:
24.            raise ValueError("plot_path is required if save_pl
25. ot is True")
26.        else:
27.            plt.close(fig)
28.            return coordinates_2d

```

■ Fixed bug   
■ Extra edit   
■ Introduced bug

Debug Model: Qwen3-Coder-480B-A35B-Instruct-FP8

Figure 14: Discovering bugs missed by ground-truth (1.9%): The model identifies and fixes bugs that were overlooked in the ground-truth solutions of the seed benchmark.

**Task Description**

Analyzes a given DataFrame containing article titles and content to identify articles with titles that include the words "how" or "what". It calculates the TF-IDF scores for the words in the content of these articles and visualizes these scores in a bar plot.

**Note:** If the DataFrame does not contain 'Title' and 'Content' columns, the function returns an empty plot. If no articles have titles containing "how" or "what," the function also returns an empty plot. Set the name of the y-axis to 'TF-IDF Score'. Set xticks to display the feature names vertically.

**Outputs:**  
Axes: A matplotlib Axes object displaying a bar plot of the TF-IDF scores.

**Buggy Code**

```

1. import re
2. import matplotlib.pyplot as plt
3. from sklearn.feature_extraction.text import TfidfVectorizer
4. import numpy as np
5. def task_func(df):
6.
7.     pattern = re.compile(r'(how|what)', re.IGNORECASE)
8.
9.     # Check if the DataFrame has the required columns
10.    if not set(['Title', 'Content']).issubset(df.columns):
11.        fig, ax = plt.subplots()
12.        return ax
13.
14.    interesting_articles = df[df['Title'].apply(lambda x: bool
(pattern.search(x)))]
15.
16.    fig, ax = plt.subplots()
17.
18.    # If there are no interesting articles, return an empty pl
ot
19.    if interesting_articles is None: A (Modify)
20.        return ax
21.
22.    vectorizer = TfidfVectorizer()
23.    X = vectorizer.transform(interesting_articles B (Modify)
['Content'])
24.    tfidf_scores = np.array(X.sum(axis=0))[0]
25.
26.    ax.bar(vectorizer.get_feature_names_out(), tfidf C (Delete)
_scores)
27.    ax.set_ylabel('TF-IDF Score')
28.    plt.xticks(rotation='vertical')
29.    return ax

```

■ Add: Line added to inject bug    ■ Delete: Line removed to inject bug  
■ Modify: Line changed to inject bug

Bug Injection: gemini-2.5-pro

**Model Generated Solution**

```

1. import re
2. import matplotlib.pyplot as plt
3. from sklearn.feature_extraction.text import TfidfVectorizer
4. import numpy as np
5. def task_func(df):
6.
7.     pattern = re.compile(r'(how|what)', re.IGNORECASE)
8.
9.     # Check if the DataFrame has the required columns
10.    if not set(['Title', 'Content']).issubset(df.columns):
11.        fig, ax = plt.subplots()
12.        return ax
13.
14.    interesting_articles = df[df['Title'].apply(lambda x: bool
(pattern.search(x)))]
15.
16.    fig, ax = plt.subplots()
17.    if interesting_articles.empty: A
18.        if interesting_articles is None: A
19.        return ax
20.
21.    vectorizer = TfidfVectorizer()
22.    X = vectorizer.fit_transform(interesting_articles['Con B
tent'])
23.    tfidf_scores = np.array(X.sum(axis=0))[0]
24.    feature_names = vectorizer.get_feature_names_out()
25.
26.    ax.bar(feature_names, tfidf_scores) C
27.    ax.set_ylabel('TF-IDF Score')
28.    ax.set_xticks(range(len(feature_names)))
29.    ax.set_xticklabels(feature_names, rotation='vertical')
30.
31.    return ax

```

■ Fixed bug    ■ Extra edit    ■ Introduced bug

Debug Model: gpt-5.1-codex

Figure 15: Functionally correct but undetected (70% of recall < 1 cases): The model's fix is functionally correct but not detected due to over-edits and structural rewrites.

**Task Description**

Perform a linear regression between "age" and "score" in the DataFrame, excluding rows with duplicate names. Plot the regression line and the scatter plot of the data.

**Note:** The function use "Linear Regression" for the plot title. The function use "Age" and "Score" as the xlabel and ylabel respectively.

**Exceptions:**  
The function will raise a ValueError is input df is not a DataFrame.

**Outputs:**  
tuple: A tuple containing the matplotlib.pyplot object and the axes object.

**Buggy Code**

```

1. import pandas as pd
2. from scipy import stats
3. import matplotlib.pyplot as plt
4. def task_func(df):
5.
6.     if not isinstance(df, pd.DataFrame):
7.         raise ValueError("The input df is not a DataFrame")
8.
9.     df = df.drop_duplicates(subset='Name')
10.
11.    slope, intercept, r_value, _, _ = stats.linregress(df['Age'], df['Score'])
12.
13.    fig = plt.figure(figsize=(8, 6)) A (Delete)
14.    ax = fig.add_subplot(111)
15.    plt.scatter(df['Age'], df['Score'], label='Data')
16.
17.    plt.plot(df['Age'].values, df['Age_up'].values, B (Delete)
18.    ax, label='Fitted Line')
19.
20.    plt.xlabel('Age')
21.    plt.ylabel('Score')
22.    plt.title('Linear Regression')
23.    plt.legend()
24.    return plt, ax

```

■ Add: Line added to inject bug    ■ Delete: Line removed to inject bug  
■ Modify: Line changed to inject bug

Bug Injection: claude-sonnet-4.5

**Model Generated Solution**

```

1. import pandas as pd
2. from scipy import stats
3. import matplotlib.pyplot as plt
4. def task_func(df):
5.
6.     if not isinstance(df, pd.DataFrame):
7.         raise ValueError("The input df is not a DataFrame")
8.
9.     df = df.drop_duplicates(subset='Name')
10.
11.    slope, intercept, r_value, _, _ = stats.linregress(df['Age'], df['Score'])
12.
13.    fig = plt.figure(figsize=(8, 6))
14.    ax = fig.add_subplot(111)
15.    plt.scatter(df['Age'], df['Score'], label='Data')
16.    plt.plot(df['Age'], slope * df['Age'] + intercept, col=B
17.             or='red', label='Regression Line')
18.    plt.xlabel('Age')
19.    plt.ylabel('Score')
20.    plt.title('Linear Regression')
21.    plt.legend()
22.    return plt, ax

```

■ Fixed bug    ■ Extra edit    ■ Introduced bug

Debug Model: grok-code-fast-1

Figure 16: Multiple minimal fixes (20% of recall<1 cases): A single bug can have multiple minimal correct fixes, and the model chose a different valid fix than the ground-truth.

**Task Description**

Count the frequency of each letter in a list of lists. If a list is empty, fill it with a random sample from the alphabet, and then count the letters.

**Outputs:**  
Counter: A Counter object with the frequency of each letter.

**Buggy Code**

```

1. from collections import Counter
2. import itertools
3. import random
4. # Constants
5. ALPHABET = 'abcdefghijklmnopqrstuvwxyz'
6. def task_func(list_of_lists, seed=0):
7.
8.     random.seed(seed)
9.     flattened_list = list(itertools.chain(*list_of_lists))
10.     for list_item in list_of_lists:
11.         if list_item == []:
12.             flattened_list += random.sample(ALPHABET, 10)
13.
14.     counter = Counter(list_item)
15.
16.     return counter

```

■ Add: Line added to inject bug    ■ Delete: Line removed to inject bug  
■ Modify: Line changed to inject bug

Bug Injection: claude-sonnet-4.5

**Model Generated Solution**

```


1. from collections import Counter
2. import itertools
3. import random
4. # Constants
5. ALPHABET = 'abcdefghijklmnopqrstuvwxyz'
6. def task_func(list_of_lists, seed=0):
7.
8.     random.seed(seed)
9.
10.     for list_item in list_of_lists:
11.         if list_item == []:
12.             list_item.extend(random.sample(ALPHABET, 10))
13.
14.         flattened_list = list(itertools.chain.from_iterable(list_of_lists))
15.
16.         counter = Counter(flattened_list)
17.
18.     return counter

```

■ Fixed bug    ■ Extra edit    ■ Introduced bug

Debug Model: Kimi-K2-Instruct

Figure 17: Bug composition issue (10% of recall < 1 cases): Compounding bugs introduced during bug-composition stage where one injected bug changes program logic affecting other bugs.

 **Task Description**

Convert elements in 'T1' to integers and create a list of random integers where the number of integers is determined by the sum of the integers in 'T1'. Random integers are generated between 0 and 'RANGE' (default is 100). Count the occurrences of each number in the generated list using a Counter.

**Outputs:**  
Counter: A Counter object representing the count of each number appearing in the list of generated random integers.

**● Buggy Code**

```

1. from collections import Counter
2. import itertools
3. from random import randint
4. def task_func(T1, RANGE=100):
5.
6.     int_list = [list(map(int, x)) for x in T1]
7.     int_list = int_list[:1]
8.     flattened_list = list(itertools.chain(*int_list))
9.     total_nums = sum(flattened_list)
10.
11.     random_nums = [randint.random(0, RANGE) for _ i
n range(total_nums)]
12.     counts = Counter(random_nums)
13.
14.     return counts if total_nums else None

```

■ Add: Line added to inject bug    ■ Delete: Line removed to inject bug  
■ Modify: Line changed to inject bug

Bug Injection: gpt-5.1-codex

**● Model Generated Solution**

```

1. from collections import Counter
2. import itertools
3. from random import randint
4. def task_func(T1, RANGE=100):
5.
6.     int_list = [list(map(int, x)) for x in T1]
7.     flattened_list = list(itertools.chain(*int_list))
8.     total_nums = sum(flattened_list)
9.
10.    random_nums = [randint(0, RANGE) for _ in range(total_
nums)]
11.    counts = Counter(random_nums)
12.
13.    return counts if total_nums else None

```

■ Fixed bug    ■ Extra edit    ■ Introduced bug

Debug Model: claude-sonnet-4-5-20250929

Figure 18: Under-repair (31.4%): The model fixes some bugs without introducing unnecessary edits but fails to apply all required fixes (recall<1, precision=1).

**Task Description**

Create a numeric array from the "original" list, calculate Fast Fourier Transform (FFT) and record the original and FFT data. Additionally, plot the histogram of the magnitude of the FFT data and return the axes object of the plot. For an empty list, return an empty array for the FFT data and None for the axes object.

**Outputs:**  
 np.array: A numpy array for the original data. np.array: FFT data. plt.Axes: The axes object of the plot.

**Buggy Code**

```

1. import numpy as np
2. from scipy.fft import fft
3. from matplotlib import pyplot as plt
4. def task_func(original):
5.
6.     arr = np.array([b for (_, b) in original])
7.
8.     if arr.shape == 0:
9.         fft_data = np.array([])
10.        return arr, fft_data, None
11.
12.    fft_data = fft(arr, n=len(arr)*2)
13.    _, ax = plt.subplots()
14.    ax.hist(np.abs(fft_data))
15.
16.    return arr, fft_data, ax

```

■ Add: Line added to inject bug    ■ Delete: Line removed to inject bug  
■ Modify: Line changed to inject bug

Bug Injection: claude-sonnet-4.5

**Model Generated Solution**

```

1. import numpy as np
2. from scipy.fft import fft
3. from matplotlib import pyplot as plt
4. def task_func(original):
5.
6.     arr = np.array(original)
7.
8.     if arr.size == 0:
9.         fft_data = np.array([])
10.        return arr, fft_data, None
11.
12.    fft_data = fft(arr)
13.    _, ax = plt.subplots()
14.    ax.hist(np.abs(fft_data))
15.
16.    return arr, fft_data, ax

```

■ Fixed bug    ■ Extra edit    ■ Introduced bug

Debug Model: Kimi-K2-Thinking

Figure 19: Regressive repair (39.2%): The model fixes all original bugs (recall=1) but introduces new bugs that cause unit tests to fail.

Your task is to perform a deep analysis of a code snippet and intentionally introduce ONE bug.  
 You will be given two major components:  
 PART 1: A task description outlining the intended functionality.  
 PART 2: A solution to the task.  
 First, carefully read and understand both the task description and the provided solution.  
 Then, modify the solution by injecting a realistic programming error.  
 You will be asked to introduce one of the following bug types into the code: Assignment, Checking, Algorithm, Build/Package/Merge or Timing/Serialization.  
 You will be asked to perform exactly one action: Add one line, Delete one line or Modify one line.

Instructions for modifying the code:

- ONLY modify ONE selected line to induce a HARD bug to the task.
- Keep the other lines of the code solution EXACTLY the SAME.
- Do NOT add any new comments to the modified line.
- DO NOT introduce easy bugs such as referencing variable names before declaration, adding typos or commas.

You should output two things:

- The subtype of the introduced bug.
- A buggy code snippet with only a ONE line difference with the original code solution, and no comments on that modified line.

---

task\_prompt: {task\_prompt}  
 correct\_solution: {correct\_solution}  
 bug\_type: {bug\_type}  
 action\_on\_lines: {action\_on\_lines}

**task\_prompt**

The programming task description for context

**correct\_solution**

A correct Python code solution

**bug\_type**

The type of bug to add

**action\_on\_lines**

The action and the lines to choose from

Figure 20: Bug injection prompt for benchmark construction.

Debug the given Python code that contains errors. ONLY fix the bugs. Make minimal edits.  
Do NOT generate a new solution based on the problem description.  
Do NOT reformat lines that are already correct.  
Do NOT edit or add any comments.

The input consists of two parts:

- A problem description outlining the intended functionality.
- A buggy code that needs to be fixed.

Your response should include:

- A self-contained, corrected Python implementation, with minimal edits.

---

```
problem_description: {problem_description}  
buggy_solution: {buggy_solution}
```

**problem\_description**

The problem description

**buggy\_solution**

The buggy solution

Figure 21: Minimal debugging prompt with problem description and buggy code.

```
Debug the given Python code that contains errors. ONLY fix the bugs. Make minimal edits.
Do NOT generate a new solution based on the problem description.
Do NOT reformat lines that are already correct.
Do NOT edit or add any comments.

The input consists of three parts:
- A problem description outlining the intended functionality.
- A buggy code that needs to be fixed.
- A set of unit tests for the problem.

Your response should include:
- A self-contained, corrected Python implementation, only making minimal edits on the buggy code.

---

problem_description: {problem_description}
buggy_solution: {buggy_solution}
unit_tests: {unit_tests}
```

```
problem_description
The problem description
```

```
buggy_solution
The buggy solution
```

```
unit_tests
The unit tests
```

Figure 22: Minimal debugging prompt with unit tests.

```
Debug the given Python code that contains errors. ONLY fix the bugs. Make minimal edits.
Do NOT generate a new solution based on the problem description.
Do NOT reformat lines that are already correct.
Do NOT edit or add any comments.

The input consists of three parts:
- A problem description outlining the intended functionality.
- A buggy code that needs to be fixed.
- Previously failed attempts and optionally error feedback.

Your response should include:
- A self-contained, corrected Python implementation, only making minimal edits on the buggy code.

---

problem_description: {problem_description}
buggy_solution: {buggy_solution}
failed_attempts: {failed_attempts}
```

**problem\_description**

The problem description

**buggy\_solution**

The buggy solution

**failed\_attempts**

Previous attempts that failed unit tests, AVOID THEM!

Figure 23: Minimal debugging prompt with execution feedback.

```

Debug the given Python code that contains errors. ONLY fix the bugs. Make minimal edits.
Do NOT generate a new solution based on the problem description.
Do NOT reformat lines that are already correct.
Do NOT edit or add any comments.

The input consists of four parts:
- A problem description outlining the intended functionality.
- A buggy code that needs to be fixed.
- A set of unit tests for the problem.
- Previously failed attempts and optionally error feedback.

Your response should include:
- A self-contained, corrected Python implementation, only making minimal edits on the buggy code.

---

problem_description: {problem_description}
buggy_solution: {buggy_solution}
unit_tests: {unit_tests}
failed_attempts: {failed_attempts}

```

**problem\_description**

The problem description

**buggy\_solution**

The buggy solution

**unit\_tests**

The unit tests

**failed\_attempts**

Previous attempts that failed unit tests, AVOID THEM!

Figure 24: Minimal debugging prompt with unit tests and execution feedback.

```

Debug the given Python code that contains errors. Do NOT add any comments. The input consists of two parts:
- A problem description outlining the intended functionality.
- A buggy code that needs to be fixed.

Your response should include:
- A self-contained, corrected Python implementation.

---

problem_description: {problem_description}
buggy_solution: {buggy_solution}

```

**problem\_description**

The problem description

**buggy\_solution**

The buggy solution

Figure 25: Free-form debugging prompt without minimal edit constraint.

```

Debug the given Python code that contains errors. Do NOT add any comments. The input consists of three parts:
- A problem description outlining the intended functionality.
- A buggy code that needs to be fixed.
- A set of unit tests for the problem.

Your response should include:
- A self-contained, corrected Python implementation.

---

problem_description: {problem_description}
buggy_solution: {buggy_solution}
unit_tests: {unit_tests}

```

**problem\_description**

The problem description

**buggy\_solution**

The buggy solution

**unit\_tests**

The unit tests

Figure 26: Free-form debugging prompt with unit tests.

```

Debug the given Python code that contains errors. Do NOT add any comments. The input consists of three parts:
- A problem description outlining the intended functionality.
- A buggy code that needs to be fixed.
- Previously failed attempts and optionally error feedback.

Your response should include:
- A self-contained, corrected Python implementation.

---

problem_description: {problem_description}
buggy_solution: {buggy_solution}
failed_attempts: {failed_attempts}

```

**problem\_description**

The problem description

**buggy\_solution**

The buggy solution

**failed\_attempts**

Previous attempts that failed unit tests, AVOID THEM!

Figure 27: Free-form debugging prompt with execution feedback.

```
Debug the given Python code that contains errors. Do NOT add any comments. The input consists of four parts:  
- A problem description outlining the intended functionality.  
- A buggy code that needs to be fixed.  
- A set of unit tests for the problem.  
- Previously failed attempts and optionally error feedback.  
  
Your response should include:  
- A self-contained, corrected Python implementation.  
  
---  
  
problem_description: {problem_description}  
buggy_solution: {buggy_solution}  
unit_tests: {unit_tests}  
failed_attempts: {failed_attempts}
```

**problem\_description**

The problem description

**buggy\_solution**

The buggy solution

**unit\_tests**

The unit tests

**failed\_attempts**

Previous attempts that failed unit tests, AVOID THEM!

Figure 28: Free-form debugging prompt with unit tests and execution feedback.

```

Debug the given Python code that contains errors. ONLY fix the bugs. Make minimal edits.
Do NOT generate a new solution based on the problem description.
Do NOT reformat lines that are already correct.
Do NOT edit or add any comments.

The input consists of three parts:
- A problem description outlining the intended functionality.
- A buggy code that needs to be fixed.
- A set of unit tests for the problem.

Your response should include:
- A self-contained, corrected Python implementation, only making minimal edits on the buggy code.

PART 1: Problem Description
```text
{task_prompt}
```

PART 2: Buggy Code
```python
{buggy_code}
```

PART 3: Unit Tests (context only)
```python
{unit_tests_code}
```

Output format (follow *exactly*):
```python
[Corrected code here]
```

Corrected Code Output (use the format above):

```

```

task_prompt
Problem description text

```

```

buggy_code
Python code with bugs

```

```

unit_tests_code
Unit test code

```

Figure 29: External API template for minimal debugging.

```

Debug the given Python code that contains errors. Do NOT add any comments. The input consists of three parts:
- A problem description outlining the intended functionality.
- A buggy code that needs to be fixed.
- A set of unit tests for the problem.

Your response should include:
- A self-contained, corrected Python implementation.

PART 1: Problem Description
```text
{task_prompt}
```

PART 2: Buggy Code
```python
{buggy_code}
```

PART 3: Unit Tests (context only)
```python
{unit_tests_code}
```

Output format (follow *exactly*):
```python
[Corrected code here]
```

Corrected Code Output (use the format above):

```

**task\_prompt**

Problem description text

**buggy\_code**

Python code with bugs

**unit\_tests\_code**

Unit test code

Figure 30: External API template for free-form debugging.

```

Your task is to rewrite the solution code of a task with structural and stylistic perturbations.
You will be given two parts:
PART 1: A task description outlining the intended functionality.
PART 2: A correct solution to the task.

First, read both the task description and the provided solution to understand what the code is supposed to do.
Then, your task is to perform a deep rewriting (perturbation) of the solution code WITHOUT changing its
functionality.
Strictly adhere to the following rules:
  - Do NOT change the starter code as given in the task description.
  - Do NOT add any new comments.
  - Do NOT make shallow redundancy edits, such as a = 1 + 2 - 1.
  - Do NOT condense the code to very short format.
  - NEVER rename variables to very short names (e.g., sum_production -> p), but you can give variables wrong
names.
  - The rewritten code should resemble what human will write and should NOT be hard for human to read.
  - The rewritten code should be different enough from the original code.
You can probably use the following hints:
  - Change loop syntax (e.g., for to while or vice versa, if applicable)
  - Convert recursion to iteration or vice versa (if functionally equivalent)
  - Invert control flow where logical structure remains the same (e.g., replace `if not ...` with an inverted
block)
  - Merge or flatten adjacent or nested if blocks.

Your response should ONLY contain the rewritten Python code, which is a different but correct solution to the
task.

---

task_description: {task_description}
original_solution: {original_solution}

```

**task\_description**

The programming task description.

**original\_solution**

The original Python code solution.

Figure 31: Solution rewriting prompt for benchmark construction.