# LDB: A Large Language Model Debugger via Verifying Runtime Execution Step by Step

**Li Zhong Zilong Wang[†] Jingbo Shang[†]**
University of California, San Diego
{lizhong, zlwang, jshang}@ucsd.edu

## Abstract

Large language models (LLMs) are leading significant progress in code generation. Beyond one-pass code generation, recent works further integrate unit tests and program verifiers into LLMs to iteratively refine the generated programs. However, these works consider the generated program as an indivisible entity, which falls short for LLMs in debugging the programs, especially when the programs contain complex logic flows and data operations. In contrast, when human developers debug programs, they typically set breakpoints and selectively examine *runtime execution* information. The execution flow and the intermediate variables play a crucial role in the debugging process, yet they are underutilized in the existing literature on code generation. In this study, we introduce Large Language Model Debugger (LDB), a novel debugging framework that enables LLMs to refine their generated programs with the runtime execution information. Specifically, LDB segments programs into basic blocks and tracks the values of intermediate variables after each block throughout runtime execution. This allows LLMs to concentrate on simpler code units within the overall execution flow, verify their correctness against the task description block by block, and effectively pinpoint any potential errors. Experiments demonstrate that LDB consistently enhances the baseline performance by up to 9.8% across the HumanEval, MBPP, and TransCoder benchmarks, archiving new state-of-the-art performance in code debugging for various LLM selections.

## 1 Introduction

Code generation is a critical yet challenging task that has various downstream applications, such as text-to-code generation (Chen et al., 2021; Yin and Neubig, 2017; Li et al., 2022), code translation (Roziere et al., 2020), and code autocompletion (Li et al., 2018; Raychev et al., 2014). Recent progress in large language models (LLMs) (Li et al., 2023; Roziere et al., 2023; Achiam et al., 2023; Zhou et al., 2023a; Muennighoff et al., 2023) significantly boosts the performance of code generation and demonstrates a promising potential to be generally applied in different requirements and tasks (Shinn et al., 2023; Gu, 2023; Yuan et al., 2023). However, generating correct programs is not a one-time effort. Existing works suggest enhancing code generation through multiple sampling (Zhang et al., 2023b; Shinn et al., 2023), self-consistency (Le et al., 2023; Huang et al., 2023a; Chen et al., 2022), and candidates ranking (Shi et al., 2022; Ni et al., 2023; Zhang et al., 2023a). Despite these advanced approaches, they still fall short on basic programming questions from the HumanEval and MBPP datasets. This underscores the limitations of *single-pass* program generation.

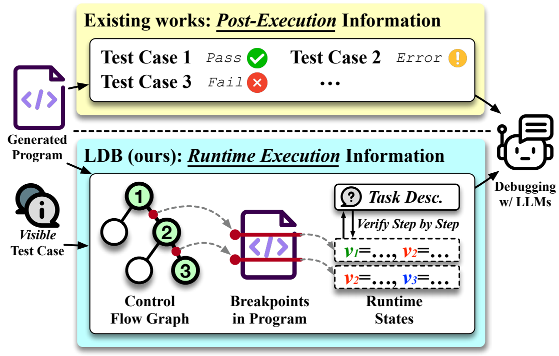


Figure 1: Comparison of LDB and existing debugging works. Existing works treat the programs as an indivisible entity and depend on the post-execution feedback for debugging, while LDB leverages the runtime execution information, tracking the values of intermediate variables and verifying basic blocks against the task description step by step.

Recognizing this, a series of works have been proposed to refine the programs generated in a single pass, based on feedback from either human annotator (Chen et al., 2023a; Wu et al., 2023) or LLMs themselves (Tang et al., 2023; Chen et al.,

[†] Corresponding authors.

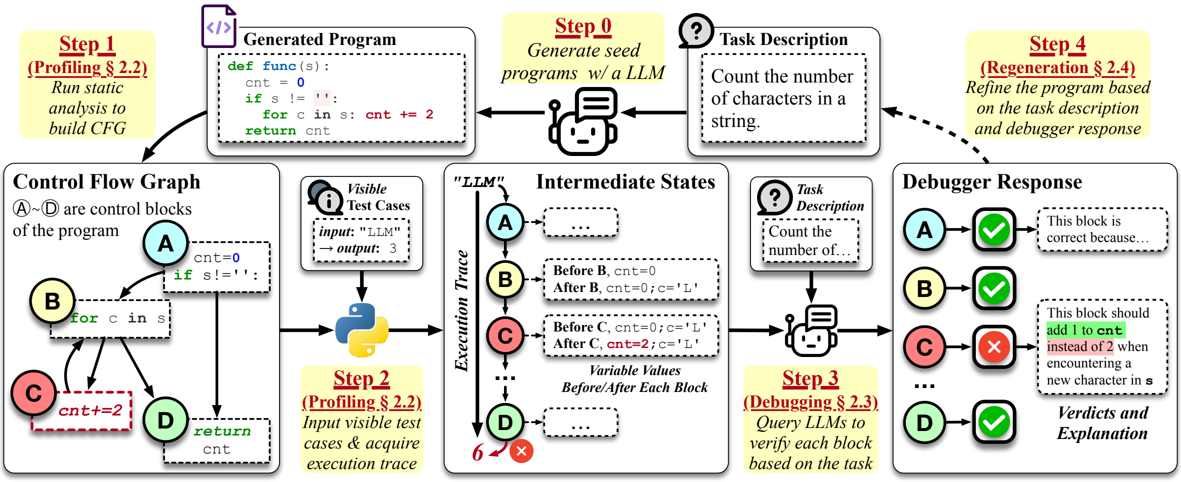


Figure 2: Illustration of the debugging workflow of LDB. A code generator is prompted to generate the seed programs (Step 0). **Profiling (§ 2.2):** LDB decomposes the seed program into basic blocks based on the control flow graph (Step 1), and feeds in a failed visible test case to acquire the execution trace (Step 2). **Debugging (§ 2.3):** LDB further inspects the runtime states of variables after each basic block during the runtime execution. Gathering the runtime execution information, LDB queries a LLM for verdicts on the correctness of the blocks in the relation to the task description (Step 3). **Regeneration (§ 2.4):** Finally, the LLM regenerates a refined program with the debugging feedbacks by LDB (Step 4).

2023c). This refinement process is akin to *debugging* in programming practices by human developers. Chen et al. (2023c); Jiang et al. (2023) introduce unit test results and error messages to LLMs. These approaches allow LLMs to reflect on potential mistakes and generate corrected programs. Nevertheless, considering the debugging process by human developers, it is sub-optimal to solely depend on these *post-execution information* to debug the program, especially in cases involving complex data structures and control flows. In fact, when human developers encounter a buggy program, they do more than just collect the program's outputs. They delve into the runtime execution to observe the execution traces[1] and examine the intermediate variables by setting breakpoints. When the intermediate execution states deviate from their intention, developers pinpoint the bugs and make the corrections. This is a common workflow for well-known interactive debuggers such as GDB (Stallman et al., 1988) and PDB (Foundation, 2001).

To this end, we propose LDB, a *large language model debugger* that refines programs generated by LLMs using runtime execution information, emulating the debugging practices of human developers. As shown in Figure 1, feeding in a visible test case, LDB segments the execution trace into basic blocks[1] based on the control flow graph[1]. LDB tracks the intermediate variables at the end of each basic block, similar to the breakpoints set by developers. After gathering runtime execution information, LDB queries LLMs for verdicts on each code block's correctness and explanations of the execution flow in relation to the coding task. This approach allows language models to concentrate on simpler code units, verify intermediate states against the task description, and pinpoint potential bugs. Consequently, it effectively debugs the program and improves the quality of code generation.

We validate LDB on three code generation benchmarks, including HumanEval (Chen et al., 2021) and MBPP (Austin et al., 2021) for text-to-code generation, and TransCoder (Roziere et al., 2020) for code translation. We conduct experiments using the proprietary model, GPT-3.5 (Achiam et al., 2023), and the open-sourced models, StarCoder (Li et al., 2023) and CodeLlama (Roziere et al., 2023). Experiments demonstrate that LDB consistently improves code generation accuracy across various LLM backbones and achieves state-of-the-art performance in debugging programs. Worth mentioning, even with the programs generated by more powerful code generators, such as GPT-4 (Achiam et al., 2023) and Reflexion (Shinn et al., 2023), LDB can still detect errors overlooked by previous advanced methods, thereby enhancing the capabilities of code generation even further. We summarize our contribution as follows:

- We propose a large language model debugger, LDB, which takes the very first step on incorpo-

[1]Refer to Appendix A for detailed introduction of execution trace, control flow graph, and basic block.

rating runtime execution information into LLMs to debug generated programs.
- We leverage basic blocks in the execution traces to properly segment the programs into smaller, simpler code units, allowing LLMs to verify them step by step against users' intention and effectively pinpoint the potential bugs.
- Extensive experiments on three code generation benchmarks demonstrate the effectiveness of LDB in debugging generated programs across different LLM backbones.

**Reproducibility.** The code will be released on Github[2].

## 2 Large Language Model Debugger

**Problem Formulation.** We follow the problem formulation of program debugging in Chen et al. (2023c). In a code generation task, each sample can be represented as a triplet $(Q, T_v, T_h)$, where $Q$ represents the task description, $T_v$ and $T_h$ are visible and hidden test cases. In the text-to-code generation task, $Q$ consists of a brief paragraph in natural language that outlines the intended goal of the task. In the code translation task, $Q$ represents a program written in a language other than the target language, and the objective is to translate this program into the target language. A code generator is first provided with $Q$ and $T_v$ to generate the seed program $A_0$. In the debugging stage, a debugger further refines $A_0$ and finally outputs a new program $A^*$. $A^*$ will be tested using the hidden tests $T_h$ to evaluate whether it is correct or not. $T_h$ is only used in the evaluation stage and not visible during the code generation or debugging stages.

### 2.1 Overview

We show the workflow of LDB in Figure 2. We prompt a LLM to generate seed programs. This serves as the starting point of debugging if they fail any of the visible test cases. During debugging, LDB feeds in the failed visible test case to the seed program and collects the runtime execution information, including the execution trace and the runtime variable values after each basic block (**Section 2.2**). Then, LDB queries a LLM to verify the correctness of basic blocks step by step via comparing them with the task description (**Section 2.3**). Considering the block-wise runtime execution and the task description, LDB queries the LLM again to detect buggy blocks and regenerate the program. LDB repeats these three steps iteratively until the new solution pass all the visible tests, or the maximum debugging iteration is reached (**Section 2.4**).

### 2.2 Profiling

In the step of profiling, LDB collects the runtime execution information when running the generated program over the failed visible test cases. It collects the execution trace and then segments the trace into basic blocks to inspect intermediate variables.

**Execution Traces.** In static analysis, each program corresponds to a unique control flow graph (CFG) where each node in the graph is a code basic block, as shown in Figure 2. Each basic block is a straight-line sequence of code with only one entry point and one exit. The CFG represents all paths might be traversed through a program during its execution. After LDB feeding in a visible test case, the control flow goes through a sequence of basic blocks and the path is denoted as the execution trace, $[B_1, B_2, \ldots, B_n]$, where $B_i$ is a basic block in the trace and $n$ is the length of the trace.

**Intermediate States.** Given an execution trace $[B_1, B_2, B_3, \ldots, B_n]$, we execute the first $i$ blocks and collect all the variables in the scope along with their runtime values. We denote the state set as $V_i = \{v = \hat{v} | v \in \bigcup B_{\leq i}\}$, where $v$ is a variable used in the first $i$ blocks and $\hat{v}$ is its runtime value after the $i$-th block. We define the intermediate state after the first $i$ blocks as $(V_{i-1}, B_i, V_i)$, where $V_{i-1}$ represents the entry states of the block, $B_i$ is the current code block to execute, and $V_i$ provides the actual execution results after $B_i$.

### 2.3 Debugging

The block-wise intermediate states determined by profiling provide a comprehensive illustration for the runtime execution. In Debugging, we integrate the intermediate states into prompts and query a LLM to verify whether the basic blocks align with the intended semantics in the task description $Q$.

**Debugging Verdicts.** For each intermediate state throughout the trace, $(V_{i-1}, B_i, V_i)$, the LLM is acquired to make a verdict on its correctness $D_i \in \{\texttt{True}, \texttt{False}\}$, and elaborate the explanation $E_i$. If the LLM detects any buggy code block, LDB includes the message in the debugging response.

**Selective Debugging.** Loops and recursion are common in programming, potentially leading to extensive execution traces. If we include the lengthy

[2] https://github.com/FloridSleeves/LLMDebugger

traces directly in the prompt, it is highly likely to exceed the maximum token limit of LLMs. This is similar to what occurs with humans in program development. When the execution is lengthy, developers may only examine a few blocks and skip the other long and tedious execution traces when detecting bugs. Inspired by the human practice, LDB selectively samples $N_b$ blocks from program traces to ensure the total length of runtime information within the max token limit of LLMs.

**Batch Debugging** In our proposed LDB, the intermediate states following each basic block are determined during the execution of the seed program with test cases. Thus, LDB can batch these states together and query the LLM for debugging verdicts. This significantly improves the token efficiency of LDB and alleviates the pitfall of repeatedly sending lengthy context to LLMs in iterative refinement (Ge et al., 2023; Hu et al., 2023). Specifically, the batch debugging query process is as follows,

$$\{V_0, B_1, V_1, B_2, ..., B_n, V_n\}$$
$$\xrightarrow{\text{LLM}} \{(D_1, E_1), ..., (D_n, E_n)\}$$

where $V_i$ is the set of variables and their runtime values after the $i$-th blocks, $B_i$ is the $i$-th block in the trace, $D_i$ is the debugging verdict from the LLM, and $E_i$ is the corresponding explanation.

### 2.4 Regeneration

The runtime execution information helps accurately localize buggy code blocks, allowing LLMs to concentrate specifically on these areas during the regeneration process. In Regeneration, LDB collects the debugging verdicts $D$ and explanations $E$, and incorporate them along with the task description $Q$ into the prompt. Then, LDB queries the LLM again to generate the refined program. LDB iteratively runs *Profiling*, *Debugging*, and *Regeneration*, until the refined program passes all visible test cases, or the maximum debugging iteration is reached. We test the finalized solution $A^*$ using the hidden test cases $T_h$ to evaluate the performance.

## 3 Experiments

We evaluate LDB on three code generation benchmarks: HumanEval (Chen et al., 2021), TransCoder (Roziere et al., 2020), and MBPP (Austin et al., 2021). HumanEval and MBPP are for text-to-code generation, where the task description is a brief passage outlines the intended functionality of the program to be generated. TransCoder is for code translation which requires to translate a program from C++ into Python. The task description of TransCoder consists of a C++ program to be translated. We compute Pass@1 accuracy with hidden test cases for assesment. We conduct experiments with the proprietary LLM, GPT-3.5 (`turbo-0613`) (Achiam et al., 2023), and the open-source LLMs, CodeLlama (`34B-Instruct`) (Roziere et al., 2023) and StarCoder ($\sim$15B) (Li et al., 2023) as backbones.

### 3.1 Experiment Setup

We generate the seed programs following the same prompts and generation parameters used in our compared method Chen et al. (2023c). We set the maximum number of debugging iterations as 10. More detailed implementation details are reported in Appendix B. To obtain visible test cases for HumanEval, we extract the given visible test cases from the task description. For MBPP, we use the first test case of each problem as the visible test case and use the rest as hidden test cases. For TransCoder, we include all test cases from the dataset as visible test cases[3]. The experiment settings on MBPP and TransCoder are the same as the prior works (Chen et al., 2023c; Shi et al., 2022; Ni et al., 2023). After finalizing the code solution, we compute the Pass@1 accuracy with hidden test cases to evaluate the performance.

### 3.2 Compared Methods

We evaluate the seed programs and label the performance as **Baseline (w/o debugger)**. We compare LDB against two *rubber duck debugging* methods from Chen et al. (2023c): **Self-Debugging (+Expl.)** which prompts LLMs to explain generated programs line-by-line as feedback, and **Self-Debugging (+Trace)** which prompts LLMs to dry run generated programs as feedback.

We reproduce the Self-Debugging methods following the instructions in Chen et al. (2023c) due to the unavailability of open-source code. When referring to "Self-Debugging", we default to the method with higher accuracy among the two methods unless otherwise specified. Throughout the evaluation process, we ensure that all debugging methods utilize the *same* LLM settings, visible test cases, seed programs, and prompts formats. This

[3]All test cases can be generated by running the original C++ programs, which are visible to LLMs.

| Model (# Param.) | Debugger | Dataset | | | | | |
|---|---|---|---|---|---|---|---|
| | | HumanEval | | TransCoder | | MBPP | |
| | | Acc. ↑ | Δ ↑ | Acc. ↑ | Δ ↑ | Acc. ↑ | Δ ↑ |
| GPT-3.5 (≥175B[†]) | Baseline (w/o debugger) | 73.8 | | 82.3 | | 67.6 | |
| | SD (+Expl.) (Chen et al., 2023c) | 81.1 | +7.3 | 85.9 | +3.6 | 74.4 | +6.8 |
| | SD (+Trace) (Chen et al., 2023c) | 80.5 | +6.7 | 86.1 | +3.8 | 72.6 | +5.0 |
| | **LDB (ours)** | **82.9** | **+9.1** | **87.7** | **+5.4** | **76.0** | **+8.4** |
| CodeLlama (34B) | Baseline (w/o debugger) | 49.4 | | 69.8 | | 51.2 | |
| | SD (+Expl.) (Chen et al., 2023c) | 53.0 | +3.6 | 79.4 | +9.6 | 55.6 | +4.4 |
| | SD (+Trace) (Chen et al., 2023c) | 54.3 | +4.9 | 76.4 | +6.6 | 57.2 | +6.0 |
| | **LDB (ours)** | **55.5** | **+6.1** | **79.6** | **+9.8** | **57.4** | **+6.2** |
| StarCoder (15B) | Baseline (w/o debugger) | 39.0 | | 61.8 | | 47.2 | |
| | SD (+Expl.) (Chen et al., 2023c) | 38.4 | -0.6 | 68.9 | +7.1 | 54.4 | +7.2 |
| | SD (+Trace) (Chen et al., 2023c) | 39.0 | +0.0 | 65.7 | +3.9 | 54.8 | +7.6 |
| | **LDB (ours)** | **39.6** | **+0.6** | **69.8** | **+8.0** | **55.4** | **+8.2** |

Table 1: Results of LDB and Self-Debugging (Chen et al., 2023c) (denoted as SD) on HumanEval, TransCoder, and MBPP with GPT-3.5, CodeLlama, and StarCoder. Accuracy is calculated based on Pass@1. The improvement (denoted as Δ) is measured against the baseline (w/o debugger). [†] We assume the parameter number in GPT-3.5 is larger than that of GPT-3 (175B).

ensures a fair comparison and eliminates potential disruptions caused by changes in prompt formats.

## 3.3 Main Results

We compare LDB with the baseline debugging methods on HumanEval, TransCoder, and MBPP, and present the result in Table 1. We observe that LDB consistently achieves improvements of up to 9.8% on all datasets across different LLM backbones. Specifically, compared to Self-Debugging which prompts LLMs to dry run or explain the program, LDB achieves higher and more stable performance gain over the baseline by introducing the actual runtime execution information.

We attribute the advantage of LDB to the fine-grained debugging feedback and the runtime information as external supplements to the LLM self-correction. The detailed block-level debugging responses help LLMs concentrate on the buggy areas in the program to better align the program with the task description. Moreover, as pointed out by Huang et al. (2023b), LLMs has limited self-correct reasoning abilities. Particularly in code generation, LLMs are prone to mistakes when reflecting on program execution. This is due to their inability to accurately calculate concrete variable values and predicting execution flow at branches or loops. The inaccurate feedback from LLMs could misguide the program debugging and refinement, which explains why Self-Debugging (+Expl.) and Self-Debugging (+Trace) fail to improve the seed programs on HumanEval with StarCoder. On the contrary, LDB generates the debugging verdicts and explanations based on accurate intermediate values and execution flows, guiding the generated programs towards the correct answer.

| Debugger | Code Generator | | |
|---|---|---|---|
| | GPT-3.5 | GPT-4 | Reflexion |
| *(w/o debugger)* | 73.8 | 75.0 | 91.5 |
| SD (GPT-3.5) | 81.1 (+7.3) | 82.3 (+7.3) | 92.1 (+0.6) |
| **LDB (GPT-3.5)** | **82.9 (+9.1)** | **82.9 (+7.9)** | **95.1 (+3.6)** |

Table 2: Results of LDB and Self-Debugging (denoted as SD) on HumanEval with seed programs from GPT-3.5, GPT-4, and Reflexion. We use GPT-3.5 as the debugging backbone. LDB can detect the subtle bugs overlooked by the powerful code generation method and improve the performance even further.

Worth mentioning, the visible test cases are also provided to the LLM during the initial seed program generation. However, their utility in code generation is limited, as evidenced by the suboptimal performance of Baseline (w/o debugger). This aligns with our assumption that actual runtime execution information significantly helps LLMs ground their reasoning, thereby improving their ability to generate better semantically aligned code.

## 3.4 Results on Advanced Code Generators

To further demonstrate the effectiveness of LDB, we apply LDB and Self-Debugging to debug the seed programs from advanced code generators, GPT-4 (Achiam et al., 2023) and Reflexion (Shinn et al., 2023). We conduct the analysis on HumanEval as an example. We query GPT-4

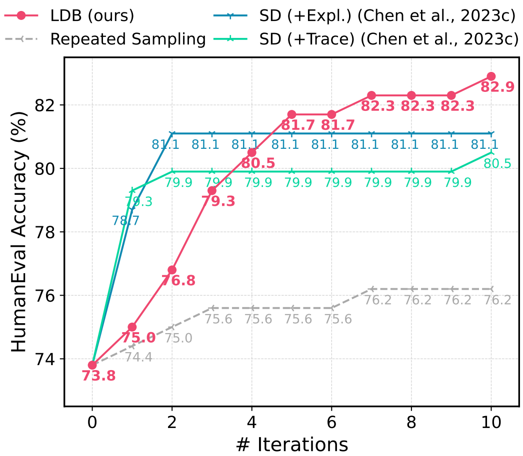

Figure 3: Performance at each debugging iteration on HumanEval with LDB, Repeated Sampling, and Self-Debugging using GPT-3.5 as the backbone. SD stands for Self-Debugging. LDB exhibits a continuing growth potential with the increasing iterations and achieves the best performance in debugging after 10 iterations.

(`1106-preview`) to generate seed programs in the same setting introduced in Section 3.1. As for Reflexion, we select the version based on GPT-3.5 and utilize the corresponding generated programs published in the official Github repository[4] as the seed programs. GPT-4 and Reflexion are considered as more powerful code generators that already achieve superior performance without any debuggers.

The results are shown in Table 2. The LLM backbone of Self-Debugging and LDB is GPT-3.5 (`turbo-0613`), which is weaker than the code generators, GPT-4 and Reflexion. We list the performance of GPT-3.5 in the table for reference. Despite the weaker LLM backbone, both Self-Debugging and LDB can refine the programs in the debugging process. This highlights the advantage of introducing a debugging stage in code generation with LLMs. Furthermore, LDB surpasses Self-Debugging in debugging and refining programs. It can improve performance on HumanEval even further and achieve a new state-of-the-art result (95.1%) in code generation by debugging the seed programs from Reflexion. This indicates that LDB is able to examine the runtime execution and correct bugs overlooked by the advanced code generators, serving as an orthogonal supplement to current code generation techniques.

### 3.5 Performance vs. Debugging Iterations

In Figure 3, we plot the performance of LDB, Repeated Sampling, Self-Debugging (+Expl.), and Self-Debugging (+Trace) across each iteration on HumanEval using GPT-3.5. We introduce Repeated Sampling as a straightforward comparison method, where we repeatedly sample coding solutions from the program generator until the solution passes the visible test. The performance at each iteration is computed in the same way as LDB. We run these methods up to 10 iterations to examine the performance tendency. We show the performance of LDB across 20 iterations in Appendix 7 to explore the continuous improvement trend.

**Continuous Debugging Potential of LDB** In Figure 3, with increasing debugging or resampling rounds, all methods refine the seed program and improve the performance. Particularly, LDB continuously improves the performance across the debugging iterations and achieves the best debugging performance despite the slightly slow rising speed. In contrast, Self-Debugging nearly stops improving the performance after 2 iterations, as also observed in Chen et al. (2023c).

**Necessity of Runtime Information.** From Figure 3, we observe that the performance of Self-Debugging presents a similar trend to Repeated Sampling after 3 rounds. They both stops effectively improving the performance at an early stage (around 2 $\sim$ 3 iterations). This phenomenon reveals a fundamental difference between LDB and Self-Debugging. We attribute it to the limited self-correcting ability of LLMs, as pointed out in Huang et al. (2023b). The feedback mechanisms in Self-Debugging (self-explaining and self-tracing) in fact enhance the initial understanding of coding tasks but fail to align the task to the specific code. Therefore, the debugging performance of these methods quickly converges and then hardly improves even given more rounds of debugging. In the contrast, LDB exhibits a continuing improvement with the growth of debugging iterations. The new information from runtime execution keeps moving the models towards correct programs, which closely resembles the human debugging process.

### 3.6 Different Decomposition Levels of LDB

In Profiling (Section 2.2), we segment the runtime trace into basic blocks based on the control flow graph. A basic block only has one entry and one exit in the program execution, serving as an ideal basic unit in the runtime analysis (Sherwood et al., 2001). To explore the effectiveness and efficiency of block-level decomposition, we develop

[4] https://github.com/noahshinn/reflexion

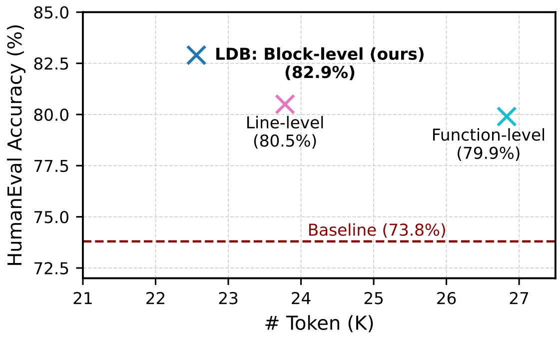

Figure 4: Performance vs. average token cost per program of LDB in different decomposition levels on HumanEval with GPT-3.5. LDB (block-level) achieves the best accuracy with the least token cost compared to LDB (line-level) and LDB (function-level).

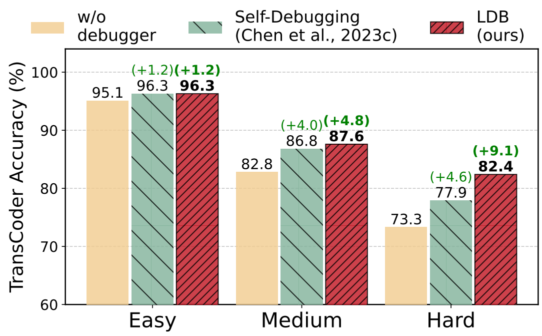

Figure 5: Performance of LDB on problems of different difficulty levels in TransCoder using GPT-3.5 as the backbone. LDB demonstrates the most improvement on Hard-level problems, indicating its capability in understanding the program execution and detecting non-trivial errors in program debugging.

two comparative methods, LDB (line-level) and LDB (function-level), which segment the runtime trace in the granularity of lines and functions respectively. We denote the original block-level debugging method as LDB (block-level). LDB in different decomposition levels share the same architecture. They collect runtime intermediate states at the end of each *code unit*. The code unit can be a line in LDB (line-level), a block in LDB (block-level), or a function in LDB (function-level). Similarly, we adopt Selective Debugging (Section 2.3) to fit prompts into the token limits of LLMs. Please refer to Appendix C for implementation details.

Figure 4 plots the performance and average token cost per program of LDB in different decomposition levels on HumanEval using GPT-3.5. The detailed statistics are listed in Table 4 in Appendix. All three debugging methods manage to enhance the performance, demonstrating the benefits from runtime execution information. Particularly, among the three decomposition levels, LDB (block-level) achieves the highest improvement.

LDB (line-level) performs worse than LDB (block-level) even if the line-level information is more fine-grained. This may arise because line-level decomposition leads to incomplete semantics in each code unit (i.e. a line of code). As a result, LLMs struggles to fully understand the code units and accurately identify bugs within the program.

LDB (function-level) provides the most coarse-grained information which largely preserves the complete programs. However, the intermediate states in the function-level fail to provide detailed runtime information. Consequently, LDB (function-level) is less effective and requires approximately 8.1 iterations on average to debug a program which are much more than the other levels (see Appendix C for statistics). This explains the highest token cost of LDB (function-level).

### 3.7 Performance of Different Difficulty Levels

To evaluate the capability of LDB in debugging programs, we dive into the problems it successfully debugs in TransCoder and categorize them into three difficulty levels, *Easy*, *Medium*, and *Hard*. The difficulty annotation is automatically performed by GPT-4 (`1106-preview`) based on the canonical solutions of each problem to avoid potential subjectiveness. Figure 5 shows the improved accuracy for each difficulty level from TransCoder dataset. We observe that the performance of GPT-3.5 in code generation decreases with the increasing problem difficulty while the improvement from the debugger increases in harder problems. Particularly, LDB shows the most improvement (9.1%) on the hard-level problems, which indicates that LDB is able to detect the non-trivial bugs and understand the complex execution flows in the harder problems.

### 3.8 Case Study

Figure 6 presents an example on HumanEval with GPT-3.5. In the example, LDB successfully fix the the program and enable it to pass the visible and hidden test cases. This case requires the program to check two conditions: (1) numbers sorted in ascending order. (2) list does not have more than 1 duplicate of the same number. In the seed program, it checks the first condition, while mistakenly presents the second condition as not having any duplicates (`lst.count(x) > 1`). In BLOCK-0 to BLOCK-4, LDB makes the verdicts that each block correctly checks the first condition. For BLOCK-5, LDB finds out the mistake in the condition and

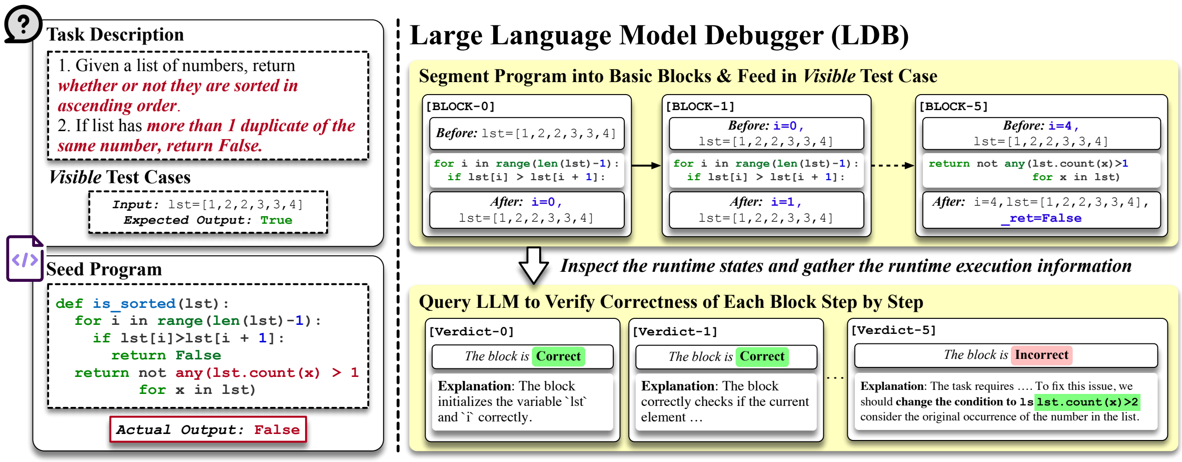


Figure 6: Debugging example of LDB on HumanEval with GPT-3.5. LDB accurately detects the bug in Block-5, and suggests tha correct fix to align the program to the task description.

locates the program bugs exactly in this block. It proposes to fix this condition into `lst.count(x) > 2` so that it aligns with the task description.

## 4 Related Work

**Augmented Code Language Models** Recent language models based on deep neural networks (Achiam et al., 2023; Roziere et al., 2023; Li et al., 2023; Nijkamp et al., 2022) demonstrate great potential for coding tasks. Despite their impressive capabilities, these models face challenges such as syntax correctness (Jin et al., 2023; Chen et al., 2023b), semantic alignment (Ni et al., 2023; Fan et al., 2023), and code reliability (Zhong and Wang, 2023). To address these challenges, some focus on enhancing initial code generation by leveraging multiple candidates (Shinn et al., 2023; Zhou et al., 2023a; Gu, 2023) or refining solutions based on better test cases (Zhang et al., 2023a) and self-consistency (Chen et al., 2022; Le et al., 2023), while others train verifiers on execution results to predict solution quality (Ni et al., 2023). In contrast, LDB enhances code generation without expanding sampling numbers complementing existing methods, while these existing code generation methods could provide better seeds for our debugger.

**Feedback-based Code Refinement** Generating correct solutions could require iterative refinement due to model limitations. Interactive methods like using human feedback (Chen et al., 2023a; Wu et al., 2023) are effective but labor-intensive. Alternatively, code refinement techniques (Chen et al., 2023c; Jiang et al., 2023; Hu et al., 2024) based on language models have been proposed, utilizing interpreter outputs, self-generated explanation, and error messages. Some train additional models for bug fixing (Pearce et al., 2023; Huang et al., 2023c; Gupta et al., 2023), while LDB utilizes debugging capabilities of original large language models. LDB follow an similar iterative refinement paradigm as previous works (Madaan et al., 2023; Zhou et al., 2023b). It leverages execution results for debugging inspired by previous works on execution-guided code generation (Chen et al., 2018; Ni et al., 2022). Additionally, agent frameworks using reinforcement learning for coding tasks (Shinn et al., 2023; Zhou et al., 2023a; Hong et al., 2023; Rasheed et al., 2024; Le et al., 2022) incorporate feedback from environments to guide actions. While these frameworks excel in searching code generation space, LDB focuses on code refinement and consistently improves performance across various initial code.

**Decomposition in Reasoning** Prompting methods suggest that decomposing problems aids large language models in reasoning tasks (Wei et al., 2022; Zhou et al., 2022; Lightman et al., 2023; Dhuliawala et al., 2023; Wang et al., 2024; Cheng et al., 2022). Inspired by this, LDB decomposes programs into blocks, querying language models for debugging verdicts and explanation. Moreover, LDB further introduces batch debugging to improve the efficiency.

## 5 Conclusion

We present LDB, a debugging framework that help LLMs refine generated programs with runtime execution information. We empirically show that LDB significantly improves code generation accuracy and achieves state-of-the-art performance in program debugging, by segmenting the programs into basic blocks and tracking the intermediate values. Experiments also reveal its unique paradigm of program debugging by using runtime information.

## Limitation

LDB is a program debugging framework using large language models. Therefore, it is subjected to the limitation of existing debugging methods of human developers. The correct test cases are mandatory in LDB so that LDB can execute the program and compare the execution flow against the task description. It remains an open question in future study whether LLMs are able to do self-correct by simply looking at its intermediate execution without knowing whether the result is correct or not (a.k.a. test-case-free debugging).

## Ethic Statements

This paper focuses on debugging code generated by large language models. The architecture are built upon open-source models and publicly available proprietary models. All the datasets in this paper are available online. We did not hire any human annotators in our experiments. We will release the code and datasets on `https://github.com/FloridSleeves/LLMDebugger`. Therefore, we do not anticipate any major ethical concerns.

## References


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

Alfred V Aho and Ravi Sethi Jeffrey D Ullman. 2015. ,"compilers-principles, techniques, and tools", pearson education asia, 2007.

Frances E Allen. 1970. Control flow analysis. *ACM Sigplan Notices*, 5(7):1–19.

Glenn Ammons and James R Larus. 1998. Improving data-flow analysis with path profiles. In *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, pages 72–84.

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

Thomas Ball and James R Larus. 1994. Optimally profiling and tracing programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(4):1319–1360.

Thomas Ball and James R Larus. 1996. Efficient path profiling. In *Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture. MICRO 29*, pages 46–57. IEEE.

Angelica Chen, Jérémy Scheurer, Tomasz Korbak, Jon Ander Campos, Jun Shern Chan, Samuel R Bowman, Kyunghyun Cho, and Ethan Perez. 2023a. Improving code generation by training with natural language feedback. *arXiv preprint arXiv:2303.16749*.

Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. 2022. Codet: Code generation with generated tests. In *The Eleventh International Conference on Learning Representations*.

Hongqiao Chen, Kexun Zhang, Lei Li, and William Yang Wang. 2023b. Tooldec: Syntax error-free and generalizable tool use for llms via finite-state decoding. In *The 3rd Workshop on Mathematical Reasoning and AI at NeurIPS'23*.

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

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

Xinyun Chen, Chang Liu, and Dawn Song. 2018. Execution-guided neural program synthesis. In *International Conference on Learning Representations*.

Zhoujun Cheng, Tianbao Xie, Peng Shi, Chengzu Li, Rahul Nadkarni, Yushi Hu, Caiming Xiong, Dragomir Radev, Mari Ostendorf, Luke Zettlemoyer, et al. 2022. Binding language models in symbolic languages. In *The Eleventh International Conference on Learning Representations*.

John Cocke. 1970. Global common subexpression elimination. In *Proceedings of a symposium on Compiler optimization*, pages 20–24.

Shehzaad Dhuliawala, Mojtaba Komeili, Jing Xu, Roberta Raileanu, Xian Li, Asli Celikyilmaz, and Jason Weston. 2023. Chain-of-verification reduces hallucination in large language models. *arXiv preprint arXiv:2309.11495*.

Zhiyu Fan, Xiang Gao, Martin Mirchev, Abhik Roychoudhury, and Shin Hwei Tan. 2023. Automated repair of programs from large language models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 1469–1481. IEEE.

Python Software Foundation. 2001. pdb — the python debugger. https://docs.python.org/3/library/pdb.html.

Tao Ge, Jing Hu, Xun Wang, Si-Qing Chen, and Furu Wei. 2023. In-context autoencoder for context compression in a large language model. *arXiv preprint arXiv:2307.06945*.

Qiuhan Gu. 2023. Llm-based code generation method for golang compiler testing. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 2201–2203.

Priyanshu Gupta, Avishree Khare, Yasharth Bajpai, Saikat Chakraborty, Sumit Gulwani, Aditya Kanade, Arjun Radhakrishna, Gustavo Soares, and Ashish Tiwari. 2023. Grace: Language models meet code edits. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1483–1495.

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

Mengkang Hu, Yao Mu, Xinmiao Yu, Mingyu Ding, Shiguang Wu, Wenqi Shao, Qiguang Chen, Bin Wang, Yu Qiao, and Ping Luo. 2023. Tree-planner: Efficient close-loop task planning with large language models. *arXiv preprint arXiv:2310.08582*.

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

Baizhou Huang, Shuai Lu, Weizhu Chen, Xiaojun Wan, and Nan Duan. 2023a. Enhancing large language models in coding through multi-perspective self-consistency. *arXiv preprint arXiv:2309.17272*.

Jie Huang, Xinyun Chen, Swaroop Mishra, Huaixiu Steven Zheng, Adams Wei Yu, Xinying Song, and Denny Zhou. 2023b. Large language models cannot self-correct reasoning yet. *arXiv preprint arXiv:2310.01798*.

Kai Huang, Xiangxin Meng, Jian Zhang, Yang Liu, Wenjie Wang, Shuhao Li, and Yuqing Zhang. 2023c. An empirical study on fine-tuning large language models of code for automated program repair. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1162–1174. IEEE.

Shuyang Jiang, Yuhao Wang, and Yu Wang. 2023. Self-evolve: A code evolution framework via large language models. *arXiv preprint arXiv:2306.02907*.

Matthew Jin, Syed Shahriar, Michele Tufano, Xin Shi, Shuai Lu, Neel Sundaresan, and Alexey Svyatkovskiy. 2023. Inferfix: End-to-end program repair with llms. *arXiv preprint arXiv:2303.07263*.

James R Larus. 1999. Whole program paths. *ACM SIGPLAN Notices*, 34(5):259–269.

Hung Le, Hailin Chen, Amrita Saha, Akash Gokul, Doyen Sahoo, and Shafiq Joty. 2023. Codechain: Towards modular code generation through chain of self-revisions with representative sub-modules. *arXiv preprint arXiv:2310.08992*.

Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven Chu Hong Hoi. 2022. Coderl: Mastering code generation through pretrained models and deep reinforcement learning. *Advances in Neural Information Processing Systems*, 35:21314–21328.

Jian Li, Yue Wang, Michael R. Lyu, and Irwin King. 2018. Code completion with neural attention and pointer networks. In *Proceedings of the 27th International Joint Conference on Artificial Intelligence*, IJCAI'18, page 4159–25. AAAI Press.

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

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

Hunter Lightman, Vineet Kosaraju, Yura Burda, Harri Edwards, Bowen Baker, Teddy Lee, Jan Leike, John Schulman, Ilya Sutskever, and Karl Cobbe. 2023. Let's verify step by step. *arXiv preprint arXiv:2305.20050*.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, et al. 2023. Self-refine: Iterative refinement with self-feedback. *arXiv preprint arXiv:2303.17651*.

Niklas Muennighoff, Qian Liu, Armel Zebaze, Qinkai Zheng, Binyuan Hui, Terry Yue Zhuo, Swayam Singh, Xiangru Tang, Leandro Von Werra, and Shayne Longpre. 2023. Octopack: Instruction tuning code large language models. In *NeurIPS 2023 Workshop on Instruction Tuning and Instruction Following*.

Ansong Ni, Jeevana Priya Inala, Chenglong Wang, Alex Polozov, Christopher Meek, Dragomir Radev, and Jianfeng Gao. 2022. Learning math reasoning from self-sampled correct and partially-correct solutions. In *The Eleventh International Conference on Learning Representations*.

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

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022. Codegen: An open large language model for code with multi-turn program synthesis. In *The Eleventh International Conference on Learning Representations*.

Hammond Pearce, Benjamin Tan, Baleegh Ahmad, Ramesh Karri, and Brendan Dolan-Gavitt. 2023. Examining zero-shot vulnerability repair with large language models. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 2339–2356. IEEE.

Reese T Prosser. 1959. Applications of boolean matrices to the analysis of flow diagrams. In *Papers presented at the December 1-3, 1959, eastern joint IRE-AIEE-ACM computer conference*, pages 133–138.

Zeeshan Rasheed, Muhammad Waseem, Mika Saari, Kari Systä, and Pekka Abrahamsson. 2024. Codepori: Large scale model for autonomous software development by using multi-agents. *arXiv preprint arXiv:2402.01411*.

Veselin Raychev, Martin Vechev, and Eran Yahav. 2014. Code completion with statistical language models. In *Proceedings of the 35th ACM SIGPLAN conference on programming language design and implementation*, pages 419–428.

Eric Rotenberg, Steve Bennett, and James E Smith. 1996. Trace cache: a low latency approach to high bandwidth instruction fetching. In *Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture. MICRO 29*, pages 24–34. IEEE.

Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*.

Baptiste Roziere, Marie-Anne Lachaux, Lowik Chanussot, and Guillaume Lample. 2020. Unsupervised translation of programming languages. *Advances in Neural Information Processing Systems*, 33:20601–20611.

Timothy Sherwood, Erez Perelman, and Brad Calder. 2001. Basic block distribution analysis to find periodic behavior and simulation points in applications. In *Proceedings 2001 International Conference on Parallel Architectures and Compilation Techniques*, pages 3–14. IEEE.

Freda Shi, Daniel Fried, Marjan Ghazvininejad, Luke Zettlemoyer, and Sida I Wang. 2022. Natural language to code translation with execution. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pages 3533–3546.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik R Narasimhan, and Shunyu Yao. 2023. Reflexion: Language agents with verbal reinforcement learning. In *Thirty-seventh Conference on Neural Information Processing Systems*.

Richard Stallman, Roland Pesch, Stan Shebs, et al. 1988. Debugging with gdb. *Free Software Foundation*, 675.

Zilu Tang, Mayank Agarwal, Alexander Shypula, Bailin Wang, Derry Wijaya, Jie Chen, and Yoon Kim. 2023. Explain-then-translate: an analysis on improving program translation with self-generated explanations. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pages 1741–1788.

Zilong Wang, Hao Zhang, Chun-Liang Li, Julian Martin Eisenschlos, Vincent Perot, Zifeng Wang, Lesly Miculicich, Yasuhisa Fujii, Jingbo Shang, Chen-Yu Lee, et al. 2024. Chain-of-table: Evolving tables in the reasoning chain for table understanding. *arXiv preprint arXiv:2401.04398*.

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

Zeqiu Wu, Yushi Hu, Weijia Shi, Nouha Dziri, Alane Suhr, Prithviraj Ammanabrolu, Noah A Smith, Mari Ostendorf, and Hannaneh Hajishirzi. 2023. Fine-grained human feedback gives better rewards for language model training. *arXiv preprint arXiv:2306.01693*.

Pengcheng Yin and Graham Neubig. 2017. A syntactic neural model for general-purpose code generation. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 440–450.

Zhiqiang Yuan, Junwei Liu, Qiancheng Zi, Mingwei Liu, Xin Peng, and Yiling Lou. 2023. Evaluating instruction-tuned large language models on code comprehension and generation. *arXiv preprint arXiv:2308.01240*.

Kexun Zhang, Danqing Wang, Jingtao Xia, William Yang Wang, and Lei Li. 2023a. Algo: Synthesizing algorithmic programs with generated oracle verifiers. *arXiv preprint arXiv:2305.14591*.

Tianyi Zhang, Tao Yu, Tatsunori Hashimoto, Mike Lewis, Wen-tau Yih, Daniel Fried, and Sida Wang. 2023b. Coder reviewer reranking for code generation. In *International Conference on Machine Learning*, pages 41832–41846. PMLR.

Li Zhong and Zilong Wang. 2023. A study on robustness and reliability of large language model code generation. *arXiv preprint arXiv:2308.10335*.

Andy Zhou, Kai Yan, Michal Shlapentokh-Rothman, Haohan Wang, and Yu-Xiong Wang. 2023a. Language agent tree search unifies reasoning acting and planning in language models. *arXiv preprint arXiv:2310.04406*.

Aojun Zhou, Ke Wang, Zimu Lu, Weikang Shi, Sichun Luo, Zipeng Qin, Shaoqing Lu, Anya Jia, Linqi Song, Mingjie Zhan, et al. 2023b. Solving challenging math word problems using gpt-4 code interpreter with code-based self-verification. *arXiv preprint arXiv:2308.07921*.

Denny Zhou, Nathanael Schärli, Le Hou, Jason Wei, Nathan Scales, Xuezhi Wang, Dale Schuurmans, Claire Cui, Olivier Bousquet, Quoc V Le, et al. 2022. Least-to-most prompting enables complex reasoning in large language models. In *The Eleventh International Conference on Learning Representations*.

# Appendix

## A Programming Language Concepts

**Basic Block.** A basic block is defined as a linear sequence of code containing a single entry point and a single exit point (Allen, 1970; Aho and Ullman, 2015). Upon executing the first instruction within a basic block, all subsequent instructions are guaranteed to execute exactly once and in sequential order. A sequence of instructions is considered a basic block if it satisfies two conditions: (1) each instruction in the sequence always executes before all subsequent instructions, and (2) there are no intervening instructions between any two instructions in the sequence. (Cocke, 1970; Allen, 1970)

**Control Flow Graph.** The control-flow graph (CFG) (Allen, 1970; Prosser, 1959) serves as a graphical depiction of all potential paths traversed during the execution of a program. Each node within the CFG corresponds to a basic block, with directed edges representing transitions in the control flow. Typically, two special blocks are identified: the entry block, which signifies the initiation of control flow into the graph, and the exit block, where all control flow exits the graph.

**Execution Traces.** In this paper, execution traces are control-flow traces of the whole program (Larus, 1999). An control-flow trace of a program is a sequence of consecutively executed basic blocks within the program. It also corresponds to a path in the control flow graph from the entry block to the exit block (Ball and Larus, 1994; Ammons and Larus, 1998; Ball and Larus, 1996; Ammons and Larus, 1998).

## B Implementation Details

In the debugging stage of LDB, we generate the debugging verdicts and explanation using greedy decoding with temperature $T = 0$ to improve the reproducibility of our experiment. The maximum number of debugging iterations is 10. We set the threshold for the number of sampled blocks and input tokens at 10 and 3,097, respectively.

## C Tradeoffs in Debugging in Different Decomposition Levels

We conduct different level debugging following this design: (1) For LDB (line-level) debugging, we collect the intermediate states before and after each line execution. We sample the first 25 lines and last 25 lines when the line number exceeds threshold $N_b = 50$, which is five times of the block-level threshold. We set this number based on the previous research on the average number of instructions in a basic block (Rotenberg et al., 1996). (2) For the original LDB (block-level) debugging, we sample the first 5 blocks and the last 5 blocks when the block number exceeds threshold $N_b = 10$. (3) For LDB (function-level) debugging, we decompose the program on function-level, namely we only collect the intermediate states at the entry and exit of the solution function. If the function trace exceeds the context length, we sample first 25 lines and last 25 lines to ensure same amount of code trace information with block-level and line-level. These three level of decomposition expand from fine to coarse granularity.

We show the average token cost per program and debugging turns of LDB with different granularity debugging levels in Table 3 and Table 4. Using GPT-3.5, LDB (line-level) has less token cost than LDB-Function due to less debugging turns. However, using CodeLlama, an open source model, LDB has higher token cost in the line-level debugging than the function-level debugging. The debugging turns of line-level debugging is not significantly lower than function-level debugging, which shows that CodeLlama has worse reasoning ability for line-level debugging even with more runtime execution information. Besides, line-level debugging has higher token costs for each debugging turns. Therefore, it has the highest token cost. Both GPT-3.5 and CodeLlama demonstrate better efficiency and accuracy in the block-level debugging. Based on these observations, we choose block-level debugging in LDB.

| **HumanEval** | **GPT-3.5** | **Token** | **CodeLlama** | **Token** |
|---|---|---|---|---|
| (w/o debugger) | 73.8 | - | 49.4 | - |
| LDB (line-level) | 80.5 (+6.7) | 24K | 53.7 (+4.3) | 72K |
| LDB (block-level) | 82.9 (+9.1) | 23K | 55.5 (+6.1) | 52K |
| LDB (function-level) | 79.9 (+6.1) | 27K | 53.7 (+4.3) | 54K |

Table 3: Accuracy vs average token number per problem on HumanEval. For both GPT-3.5 and CodeLlama, LDB with block-level debugging achieves the highest accuracy and least token cost.

## D Complexity Analysis of Batch Debugging

LDB can batch runtime information of all selected blocks together and query language models for debugging verdicts. This significantly improves the

| HumanEval | GPT-3.5 | Avg. Turn | CodeLlama | Avg. Turn |
|---|---|---|---|---|
| (w/o debugger) | 73.8 | - | 49.4 | - |
| LDB (line-level) | 80.5 | 6.4 | 53.7 | 9.1 |
| LDB (block-level) | 82.9 | 6.2 | 55.5 | 7.8 |
| LDB (function-level) | 79.9 | 8.1 | 53.7 | 9.3 |

Table 4: Accuracy vs debugging turns on HumanEval. For both GPT-3.5 and CodeLlama, LDB with block-level debugging achieves the highest accuracy and fewest debugging turns.

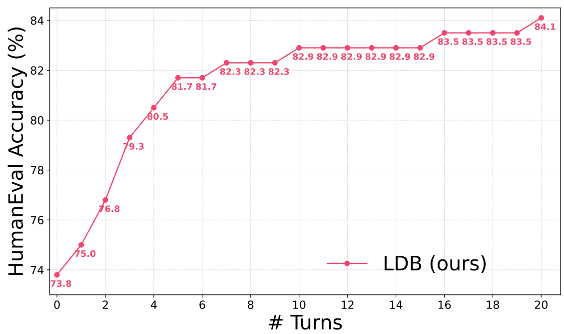


Figure 7: Performance of LDB on HumanEval with GPT-3.5 in 20 debugging iterations. The final performance after 20 iterations is 84.1%.

token efficiency of LDB and alleviates the pitfall of repeatedly sending context to language models in iterative refinement (Hu et al., 2023). Assume the average token numbers is $\mathcal{N}$ for debugging each block, average block number is $\mathcal{B}$ for debugging, and average debugging iteration number is $\mathcal{D}$. Without batch debugging, for debugging $i$-th block, the context length for debugging is $i{*}\mathcal{N}$. Therefore, debugging $\mathcal{N}$ blocks consumes $\sum_{i=0}^{B}(i * \mathcal{N})$, approximately $O(\mathcal{B}^2 * \mathcal{N})$ tokens. As a comparison, batch debugging only send one debugging message in each turn, which has context length of $O(\mathcal{B} * \mathcal{N})$ tokens.

## E Debugging 10 More Iterations

As LDB shows a different trend of performance regarding debugging turns compared to the state-of-the-art methods, it is interesting to see whether the performance continues growing after 10 debugging turns. We conduct a experiment that continue debugging until 20 debugging turns to explore the characteristic of LDB, as shown in Figure 7. The accuracy still grows after 10 debugging turns and achieves 84.1%.

## F Error Analysis of LDB

We analyze the debugging feedback of LDB in two perspectives, the bug localization accuracy and the bug categorization. We evaluate the performance of LDB in localizing the bug accurately and further specify the types of bugs detected by LDB. We use the results with GPT-3.5 as an example.

| | HumanEval | MBPP | TransCoder |
|---|---|---|---|
| **Accuracy** | 93.7% | 95.3% | 86.7% |
| **Syntax Error** | 18.8% | 23.2% | 20.0% |
| **Semantic Error** | 81.2% | 76.8% | 80.0% |

Table 5: Debug correctness of LDB and detected program error types.

**Accuracy of Bug Localization.** To evaluate the correctness of debugging verdicts made by LDB, we collect the cases that LDB successfully debugs and query GPT-4 (`1106-Preview`) to automatically verify whether LDB correctly identifies the buggy basic block. In the prompt, we provide the task description, the seed program, the runtime information, debugging responses, and the final programs for GPT-4. Table 5 shows the analysis results on HumanEval, MBPP and TransCoder. We observe that LDB achieves an accuracy over 85% over all datasets. It has the highest accuracy on MBPP (95.5%) while performs worst on TransCoder (86.7%). This may arise because TransCoder consists of more complicated programs where the original C++ programs outlines the intention implicitly.

**Bug Categorization.** We further categorize the bugs detected and fixed by LDB into two categories: Syntax Errors and Semantic Errors. Syntax errors refer to the cases that violates the syntax rules of its programming language. Semantic error occurs when the code runs without producing any syntax errors, but it does not behave as intended, thus fails the test cases. We annotate the categories through GPT-4 (`1106-preview`) to avoid subjectiveness. Across three datasets, LDB fixes semantic errors with around 80% of the time. The reason is that the syntax errors are less frequent given the code generation ability of current LLMs, and LDB can better debug semantic errors by introducing the detailed runtime execution information.

## G Prompt Example in LDB

We list the prompts used in LDB in detail as follow,

- The prompts used in the chat mode are shown in Figure 8.
- The prompts used in the text completion mode are shown in Figure 9.

```
==================================== System ====================================
You are an expert programming assistant.
```

```
===================================== User =====================================
Complete the following task in Python. Please respond with code only.
def is_sorted(lst):
    '''
    Given a list of numbers, return whether or not they
    are sorted in ascending order. If list has more than
    1 duplicate of the same number, return False.
    Assume no negative numbers and only integers.

    Examples
    is_sorted([5]) = True
    is_sorted([1, 2, 3, 4, 5]) = True
    is_sorted([1, 2, 3, 4, 5, 6, 7]) = True
    is_sorted([1, 2, 2, 2, 3, 4]) = False
    '''
```

```
=================================== Assistant ==================================
    for i in range(len(lst) - 1):
        if lst[i] > lst[i + 1]:
            return False
    return not any(lst.count(x) > 1 for x in lst)
```

```
===================================== User =====================================
The code above fails the given unit test:
assert is_sorted([1, 2, 2, 3, 3, 4]) == True # Real Execution Output: False.
Help me debug this.

Here is the code execution trace block by block with the intermediate variable
values to debug your code. You need to answer FOR EACH BLOCK whether this code
block is correct or not. If not, give an explanation on what is wrong. Please
wrap your response into a JSON object that contains keys `block` with the name of
 each block, key `correct` with value False or True, and key `explanation` with
an explanation on the bug.

Example Answers:
{"block": "BLOCK-1", "correct": "True", "explanation": "The block initializes
variable `a` and `b`."}
{"block": "BLOCK-2", "correct": "False", "explanation": "The block is incorrect
because the code does not add the two integers together, but instead subtracts
the second integer from the first. To fix this issue, we should change the
operator from `-` to `+` in the return statement. This will ensure that the
function returns the correct output for the given input."}

[BLOCK-0]
    # lst=[1, 2, 2, 3, 3, 4]
    for i in range(len(lst) - 1):
        if lst[i] > lst[i + 1]:
    # i=0       lst=[1, 2, 2, 3, 3, 4]
[BLOCK-1]
    # i=0       lst=[1, 2, 2, 3, 3, 4]
    for i in range(len(lst) - 1):
        if lst[i] > lst[i + 1]:
    # i=1       lst=[1, 2, 2, 3, 3, 4]
[BLOCK-2]
    # i=1       lst=[1, 2, 2, 3, 3, 4]
    for i in range(len(lst) - 1):
        if lst[i] > lst[i + 1]:
    # i=2       lst=[1, 2, 2, 3, 3, 4]
[BLOCK-3]
    # i=2       lst=[1, 2, 2, 3, 3, 4]
    for i in range(len(lst) - 1):
        if lst[i] > lst[i + 1]:
    # i=3       lst=[1, 2, 2, 3, 3, 4]
```

```
[BLOCK-4]
    # i=3       lst=[1, 2, 2, 3, 3, 4]
    for i in range(len(lst) - 1):
        if lst[i] > lst[i + 1]:
    # i=4       lst=[1, 2, 2, 3, 3, 4]
[BLOCK-5]
    # i=4       lst=[1, 2, 2, 3, 3, 4]
    for i in range(len(lst) - 1):
    return not any(lst.count(x) > 1 for x in lst)
    # i=4       lst=[1, 2, 2, 3, 3, 4]  _ret=False
```

```
=================================== Assistant ===================================
{"block": "BLOCK-0", "correct": true, "explanation": "The block initializes the
variable `lst` correctly."}
{"block": "BLOCK-1", "correct": true, "explanation": "The block correctly checks
if the current element is greater than the next element in the list."}
{"block": "BLOCK-2", "correct": true, "explanation": "The block correctly checks
if the current element is greater than the next element in the list."}
{"block": "BLOCK-3", "correct": true, "explanation": "The block correctly checks
if the current element is greater than the next element in the list."}
{"block": "BLOCK-4", "correct": true, "explanation": "The block correctly checks
if the current element is greater than the next element in the list."}
{"block": "BLOCK-5", "correct": false, "explanation": "The block is incorrect
because it returns the opposite of the condition `lst.count(x) > 1` for any
element `x` in the list. This means that if any element has more than 1 duplicate
, the function will return False. However, the task requires that if there are
more than 1 duplicate of the same number, the function should return False. To
fix this issue, we should change the condition to `lst.count(x) > 2` to account
for the original occurrence of the number in the list."}
```

```
===================================== User =====================================
Please fix the Python code.
```

```
=================================== Assistant ===================================
def is_sorted(lst):
    '''
    Given a list of numbers, return whether or not they
    are sorted in ascending order. If list has more than
    1 duplicate of the same number, return False.
    Assume no negative numbers and only integers.

    Examples
    is_sorted([5]) = True
    is_sorted([1, 2, 3, 4, 5]) = True
    is_sorted([1, 2, 3, 4, 5, 6, 7]) = True
    is_sorted([1, 2, 2, 2, 3, 4]) = False
    '''
    for i in range(len(lst) - 1):
        if lst[i] > lst[i + 1]:
            return False
    return not any(lst.count(x) > 2 for x in lst)
```

Figure 8: Prompt for chat mode.

```
# Write Python function to complete the task and pass the assertion tests.
### Task Start ###
# These are the assertions for your function:
assert find_char_long('Please move back to stream') == ['Please', 'move', 'back',
 'stream']

def find_char_long(text):
    """
    Write a function to find all words which are at least 4 characters long in a
    string by using regex.
    """
    if text == "":
        return []
    pat = r"\b\w{4}\b"
    res = re.findall(pat, text)
    return res

With the above function, the assertion is `find_char_long('Please move back to
stream') == ['Please', 'move', 'back', 'stream']` but the real execution output
is `['move', 'back']`.

Debug the program trace block by block until find the incorrect block. Every
block should have different feedbacks:
[BLOCK-0]
    # text="Please move back to stream"
    if text == "":
[BLOCK-1]
    # text="Please move back to stream"
    pat = r"\b\w{4}\b"
    res = re.findall(pat, text)
    # text="Please move back to stream" pat="\b\w{4}\b"  res=['move', 'back']
[debug]
[BLOCK-0]
Feedback: CORRECT. This block is correct. It checks if the input text is empty.
If the input text is empty, it returns an empty list without do regex match.
[BLOCK-1]
Feedback: INCORRECT. This block defines a regular expression pattern `pat` with
value r"\b\w{4}\b". However, there's an issue with the regular expression pattern
. It only matches words that are exactly 4 characters long. Therefore, the return
 value `_ret` is `['move', 'back']`. In the task description, it asks for words
which are at least 4 characters long. To fix the code, we should change`pat=r"\b\
w{4}\b"` into `pat = r"\b\w{4,}\b"`.
[/debug]
Please fix the Python code.
[python]
import re
def find_char_long(text):
    """
    Write a function to find all words which are at least 4 characters long in a
    string by using regex.
    """
    if text == "":
        return []
    pat = r"\b\w{4,}\b"
    res = re.findall(pat, text)
    return res
[/python]
### Task End ###

### Task Start ###
# These are the assertions for your function:
assert solution([3, 3, 3, 3, 3]) == 9

def solution(lst):
    """
    Given a non-empty list of integers, return the sum of all of the odd elements
     that are in even positions.
```

```
    Examples
    solution([5, 8, 7, 1]) ==> 12
    solution([3, 3, 3, 3, 3]) ==> 9
    solution([30, 13, 24, 321]) ==>0
    """

    result = 0
    for i in range(len(lst)):
        if lst[i] % 2 == 0 and i % 2 == 0:
            result += lst[i]
    return result

With the above function, the assertion is `assert solution([3, 3, 3, 3, 3]) == 9
` but the real execution output is ` 0`.

Debug the program trace block by block until find the incorrect block. Every
block should have different feedbacks:
[BLOCK-0]
    # lst=[3, 3, 3, 3, 3]
    result = 0
    for i in range(len(lst)):
        if lst[i] % 2 == 0 and i % 2 == 0:
    # lst=[3, 3, 3, 3, 3] result=0 i=0
[BLOCK-1]
    # lst=[3, 3, 3, 3, 3] result=0 i=0
    for i in range(len(lst)):
        if lst[i] % 2 == 0 and i % 2 == 0:
    # lst=[3, 3, 3, 3, 3] result=0 i=1
[BLOCK-2]
    # lst=[3, 3, 3, 3, 3] result=0 i=1
    for i in range(len(lst)):
        if lst[i] % 2 == 0 and i % 2 == 0:
    # lst=[3, 3, 3, 3, 3] result=0 i=2
[BLOCK-3]
    # lst=[3, 3, 3, 3, 3] result=0 i=2
    for i in range(len(lst)):
        if lst[i] % 2 == 0 and i % 2 == 0:
    # lst=[3, 3, 3, 3, 3] result=0 i=3
[BLOCK-4]
    # lst=[3, 3, 3, 3, 3] result=0 i=3
    for i in range(len(lst)):
        if lst[i] % 2 == 0 and i % 2 == 0:
    # lst=[3, 3, 3, 3, 3] result=0 i=4
[BLOCK-5]
    # lst=[3, 3, 3, 3, 3] result=0 i=4
    for i in range(len(lst)):
    return result
    # lst=[3, 3, 3, 3, 3] result=0 i=4 _ret=0
[debug]
```

```
================================== Completion ==================================
[BLOCK-0]
Feedback: CORRECT. This block is correct. It initializes the result variable to
0.
[BLOCK-1]
Feedback: INCORRECT. This block has an issue. It checks if the element at index `
i` is even and if the index `i` is even. However, the task description asks for
the sum of all of the odd elements that are in even positions. To fix the code,
we should change the line `if lst[i] % 2 == 0 and i % 2 == 0:` into `if lst[i] %
2 != 0 and i % 2 == 0:`.
[BLOCK-2]
Feedback: INCORRECT. This block has an issue. It checks if the element at index `
i` is even and if the index `i` is even. However, the task description asks for
the sum of all of the odd elements that are in even positions. To fix the code,
we should change the line `if lst[i] % 2 == 0 and i % 2 == 0:` into `if lst[i] %
2 != 0 and i % 2 == 0:`.
[BLOCK-3]
Feedback: INCORRECT. This block has an issue. It checks if the element at index `
i` is even and if the index `i` is even. However, the task description asks for
the sum of all of the odd elements that are in even positions. To fix the code,
we should change the line `if lst[i] % 2 == 0 and i % 2 == 0:` into `if lst[i] %
2 != 0 and i % 2 == 0:`.
[BLOCK-4]
Feedback: INCORRECT. This block has an issue. It checks if the element at index `
i` is even and if the index `i` is even. However, the task description asks for
the sum of all of the odd elements that are in even positions. To fix the code,
we should change the line `if lst[i] % 2 == 0 and i % 2 == 0:` into `if lst[i] %
2 != 0 and i % 2 == 0:`.
[BLOCK-5]
Feedback: CORRECT. This block is correct. It returns the result variable.
[/debug]
Please fix the Python code.
[python]
def solution(lst):
    """
    Given a non-empty list of integers, return the sum of all of the odd elements
     that are in even positions.

    Examples
    solution([5, 8, 7, 1]) ==> 12
    solution([3, 3, 3, 3, 3]) ==> 9
    solution([30, 13, 24, 321]) ==>0
    """
    result = 0
    for i in range(len(lst)):
        if lst[i] % 2 != 0 and i % 2 == 0:
            result += lst[i]
    return result
[/python]
### Task End ###
```

Figure 9: Prompt for completion mode.