Controllable LLM Debugging: Knowing when to Stop Matters

Aryan Gulati

Abstract

Large Language Models (LLMs) demonstrate proficiency in coding tasks but tend to overcorrect when debugging, making excessive changes beyond what's necessary. This research evaluates five approaches to achieve controllable LLM debugging: monolithic prompting, code chunking, targeted test case injection, multi-agent frameworks, and AST-based code difference analysis. Using a modified DebugEVAL framework, we measure both bug identification accuracy and minimality of changes. Our findings reveal a fundamental tension: while multi-agent approaches achieve higher bug resolution rates, they typically make 40-60% more code changes than necessary. AST-based approaches better balance accuracy with minimal intervention. We demonstrate that effective LLM debugging requires not just fixing errors but knowing when to stop making changes—a critical consideration for educational and production environments.

1. Introduction

In fall 2024, tired of manually debugging student code, I wondered if LLMs could help streamline this process by identifying issues or suggesting where to look. While LLMs have demonstrated impressive capabilities in code generation, their effectiveness in debugging presented a more nuanced reality—they could debug, but only "kind of."

A common pattern quickly emerged: when asked to debug code, LLMs often proposed extensive changes that went well beyond addressing the actual problem. As many users and developers using coding assistants succinctly describe it - *AI is sometimes too eager to make changes. It constantly makes suggestions to change a code, although these changes are not necessary or partly wrong*. This observation reveals a fundamental challenge in LLM-assisted debugging—the lack of a "stop signal" that indicates when the code is "good enough."

The problem manifests in several ways. First, human-written code is typically shorter and more focused than LLM-generated code achieving the same functionality. Second, LLMs continue generating beyond what is necessary, whether in code or explanations. Most critically, when debugging, they often propose multiple changes for small syntax issues or flag new problems that aren't actually there, introducing false positives.

This overcorrection tendency creates significant challenges, particularly in educational settings where targeted, minimal fixes are pedagogically preferable. Once LLMs suspect a bug, they often perform extensive rewrites to ensure they're "fixing" the problem, sometimes introducing new code paths and complexity unnecessarily.

Our research aims to address this fundamental challenge by developing and evaluating methods that keep LLMs "on track"—enabling them to fix real bugs while stopping short of rewriting entire sections of code or generating new irrelevant issues. We investigate five distinct approaches to controlled debugging:

1. **Single Prompt (Monolithic)** - Providing the entire codebase with failing test cases in one prompt
2. **Code Chunking** - Breaking code into manageable segments for focused analysis
3. **Targeted Test Case Injection** - Sequentially introducing failing test cases to build incremental fixes
4. **Multi-Agent Frameworks** - Distributing debugging tasks across specialized LLM instances
5. **AST-Based Code Differences** - Using abstract syntax trees to precisely identify and align functional differences

Using a modified version of the DebugEVAL framework, we evaluate these approaches on their bug identification accuracy, minimality of changes compared to reference solutions, and computational cost. Our results demonstrate a fundamental tension between debugging effectiveness and controllability, with different approaches offering distinct trade-offs between these competing objectives.

This paper contributes: (1) a systematic analysis of controllability challenges in LLM debugging; (2) comparative evaluation of five approaches to address these challenges;

(3) a novel metric for debug efficiency that accounts for both correctness and minimality; and (4) practical guidelines for developing more controlled AI-assisted debugging tools for educational environments.

2. Related Work

Automated program repair has been a subject of active research for over a decade. Early approaches such as GenProg (Goues et al., 2013) and DeepFix (Gupta et al., 2017) laid the groundwork by demonstrating that search-based methods and deep learning could automatically localize and repair bugs. Although these systems achieved success on a variety of common bug patterns, they often lacked the ability to control the extent of code modifications, leading to solutions that were not minimal in nature.

With the advent of large language models (LLMs), the landscape of code synthesis and debugging has shifted dramatically. In (Chen et al., 2021), Chen et al. demonstrated that LLMs trained on vast code corpora can generate code completions and even basic fixes. However, as highlighted by Brown et al. (Brown et al., 2020), while these models excel at capturing contextual nuances, they tend to overcorrect—making excessive changes beyond what is strictly necessary to fix a bug. This overcorrection behavior has been observed in various studies and poses a significant challenge when minimal intervention is desired, particularly in educational contexts.

To address these challenges, several more controlled debugging methodologies have emerged. Incremental debugging strategies such as targeted test case injection (Gupta et al., 2017; Liu et al., 2022) aim to limit the scope of modifications by addressing one failing test case at a time. Similarly, multi-agent frameworks (Yao et al., 2022) decompose the debugging task into specialized components (e.g., debugging, context analysis, and test evaluation) to better regulate code modifications. Although these approaches have been shown to improve bug detection accuracy, they still often result in more extensive code changes compared to the minimal reference fixes.

An alternative line of research leverages the structured representation of code through abstract syntax trees (ASTs). AST-based difference analysis (Mou et al., 2016) enables a fine-grained comparison between student and reference code, thereby facilitating more targeted interventions. This method inherently provides a form of a “stop signal,” as it confines modifications to only those areas where structural differences are detected.

More recent works have further explored controlled intervention techniques. For instance, Chen et al. (Chen et al., 2022) investigated transformer-based models with explicit mechanisms to limit overcorrection, while Wang et al. (Wang et al., 2023) focused on controlled code generation in debugging contexts. Additional efforts such as the work on AlphaCode (DeepMind, 2022) and Code Llama (AI, 2023) demonstrate ongoing advances in generating competitive-level code with attention to minimality and correctness.

In summary, while early work on automated program repair proved the feasibility of automatic debugging, the transition to LLM-based methods has introduced new challenges—particularly, the tendency to overcorrect. Our work builds on these foundations by systematically evaluating multiple debugging approaches and proposing strategies that enforce a more disciplined, minimal set of modifications.

3. Methodology

Our methodology follows a systematic comparative approach, evaluating five distinct LLM-based debugging strategies:

- Baseline monolithic approach
- Code chunking technique
- Targeted test case injection
- Multi-agent framework
- AST-based code difference analysis

For each methodology, we utilized identical LLM architectures, prompt templates (modified only to accommodate the specific approach), and test dataset to ensure fair comparison. The DebugEVAL dataset provided the foundation for our experiments, offering a standardized framework for evaluating debugging capabilities across varying complexity levels.

3.1. Large Language Models

For our evaluation, we utilized 3 leading closed-source LLMs accessible through their official APIs: GPT-4o (OpenAI), Claude 3.5 Sonnet (Anthropic) and Gemini 2.0 Pro (Google). These models were selected to represent the current state-of-the-art in commercial LLM capabilities for code understanding and manipulation.

To ensure experimental consistency, we maintained standard parameter settings across all models: a temperature of 0.6 for deterministic outputs and top-p of 0.95. We employed consistent system prompts across all model interactions, focusing on clear instructions for bug identification and minimal code corrections.

3.2. Dataset Preparation and Characteristics

3.2.1. DebugEVAL Dataset

We selected DebugEVAL as our primary dataset for this research due to its comprehensive evaluation of code debugging capabilities. DebugEVAL designs four task scenarios: BUG Localization, BUG Identification, Code Repair, and Code Recognition, providing a multifaceted assessment framework. The dataset consists of approximately 200 debugging problems across various programming languages and complexity levels.

A key advantage of DebugEVAL for our research is its structured approach to evaluating different aspects of the debugging process. Each problem in the original dataset is presented as a multiple-choice question, with LLMs tasked with selecting the correct option across four categories. This format enables precise evaluation of specific debugging capabilities while maintaining standardization across problems.

3.2.2. Data Adaptation

For our research, we adapted the DebugEVAL framework to better assess unguided debugging capabilities. Rather than using the multiple-choice format, we converted problems to a JSON identification task format. This modification removed the progressive hints inherent in multiple-choice options, requiring LLMs to identify bugs and propose fixes without guidance.

Our adaptation involved restructuring each problem to include:

- The original problem description
- Complete buggy code (without highlighting the problematic areas)
- Failing test cases and expected outputs
- A structured JSON format for responses that included fields for bug type, location, description, and proposed fix

This approach allowed us to evaluate not only accuracy in bug identification but also the minimality and precision of proposed fixes—a critical dimension for assessing controllability in debugging.

3.3. Implementation of Debugging Methodologies

3.3.1. Baseline Monolithic Approach

The baseline monolithic approach represents a straightforward implementation of LLM-based debugging, serving as a control condition against which we evaluated our more sophisticated methodologies. This approach employs a single prompt containing the entire student code along with a description of failing test cases, instructing the LLM to ”Identify the bug(s) and propose a minimal fix.”

For implementation, we provided the LLM with a standardized JSON response format that included fields for bug type, description, and proposed fix, matching the structure of DebugEVAL's evaluation criteria. This enforced consistency in response formatting while allowing sufficient freedom for the LLM to diagnose and resolve issues.

Our observations with this approach revealed several limitations:

- Overly Large Responses: The LLM often rewrote entire functions rather than isolating the one or two lines that needed fixing
- Occasional Irrelevant Changes: Even with simple bugs like semicolon omissions, the LLM sometimes ”improved” unrelated parts of the code
- Context Overload: For larger solutions, the LLM sometimes missed the real bug or got sidetracked by less critical warnings

This baseline established the fundamental challenge of controllability in LLM debugging and provided a reference point for measuring improvements in our more specialized approaches.

3.3.2. Code Chunking Technique

The code chunking approach addresses context window limitations by dividing large codebases into manageable segments. This method enables LLMs to process complex code structures through a divide-and-conquer strategy, preserving semantic coherence while reducing the token count required for each analysis phase.

Chunking Strategies: We implemented several distinct chunking strategies, each optimized for different code characteristics. Line-based chunking divided code into fixed-size segments while ensuring syntactic completeness. Function-based chunking extracted individual functions as independent units, preserving their complete internal logic while isolating their complexity. For our educational context, function-based chunking proved most effective as it aligned with students' natural organization of code.

For each chunk, we prompted the LLM with a consistent question: ”Any bugs here? If yes, highlight the lines and propose a fix.” This focused approach enabled more targeted analysis without the distraction of the entire codebase.

Context Management:

- Each chunk was processed sequentially with a consistent prompt template
- Chunk context (e.g., function headers, module imports) was preserved
- Inter-chunk references were maintained where necessary

Our observations with this approach showed:

- Better Focus: The LLM was more likely to zero in on the correct lines without distraction
- False Negatives: When bugs involved interactions between functions in different chunks, the LLM sometimes failed to connect the dots
- Time & Cost Trade-off: Multiple chunks led to more total queries, increasing token usage
- Limitations with Complex Code: For longer files with helpers distributed throughout, performance degraded

After processing individual chunks, we employed an aggregation mechanism to synthesize the distributed analyses into a coherent debugging output, resolving potential conflicts through confidence scoring based on evidence strength and context relevance.

3.3.3. Targeted Test Case Injection

Our targeted test case injection methodology represents a novel approach to controlling LLM debugging behavior by focusing on incremental problem-solving rather than comprehensive rewriting. Instead of providing all test cases at once, we systematically introduced failing test cases one at a time, asking the LLM to analyze and suggest fixes for each specific case before moving to the next.

This stepped approach mirrors how human teaching assistants guide students through debugging:

- Present a single failing test case with clear expected vs. actual output
- Ask for analysis and a minimal fix for just this test case
- Apply the suggested fix
- Introduce the next failing test case
- Repeat until all test cases pass

Our implementation carefully considered test case ordering, generally starting with simpler cases that required minimal changes and progressing to more complex ones. This sequencing proved critical, as adding functionality incrementally was more effective than requiring the LLM to refactor existing code.

Our observations with this approach revealed:

- Incremental Improvement: LLM focused on simpler fixes first, building toward more complex solutions
- Controlled Changes: Sequential approach prevented sweeping changes that affected multiple parts simultaneously
- Pedagogical Alignment: This approach better matched how students learn to debug their own code

By constraining the LLM's focus to one failing test at a time, we effectively implemented a natural ”stop signal” that prevented overcorrection, addressing a key challenge in controllable debugging.

3.3.4. Multi-Agent Framework

Our multi-agent framework distributes debugging responsibilities across specialized LLM instances, mimicking the incremental code debugging methodology human programmers use. This approach decomposes the debugging process into a structured workflow where specialized agents handle distinct aspects of code analysis and repair.

Agent Architecture: Our implementation featured an orchestrator agent managing the debugging process, supported by three specialized components:

- Debug Agent: Responsible for analyzing code and identifying potential bugs
- Context Agent: Maintains understanding of the overall codebase structure and relationships
- Test Agent: Evaluates proposed fixes against test cases and validates solutions

The orchestrator coordinated a structured workflow:

1. Understanding the code and test cases
2. Sequentially addressing failing tests in order of complexity
3. Making targeted code changes
4. Running tests to verify fixes
5. Iterating through this loop until all tests passed

Our observations with this approach showed:

- Highest Accuracy: This method achieved the best debugging success rate across all approaches
- Controllability Challenges: Despite success in fixing bugs, solutions often involved more extensive changes than necessary (e.g., 40 lines changed when reference solutions required only 24)
- Cost Considerations: Averaging approximately 7 LLM calls per problem significantly increased computational costs

While this approach demonstrated the highest bug resolution capability, it highlighted the fundamental tension between debugging effectiveness and controllability that motivated our research.

3.3.5. AST-Based Code Difference Analysis

The AST-based code difference approach represents our most structured method for controlled debugging. Unlike the previous approaches that analyze code directly as text, this method leverages abstract syntax trees to precisely identify and analyze functional differences between reference and student code.

Our implementation followed a systematic process:

1. Function Header Analysis: Using an LLM to parse code and systematically document function purposes, inputs, and outputs
2. AST Generation: Converting both reference and student code into Abstract Syntax Trees using language-specific parsers
3. Node Alignment: Aligning corresponding nodes between reference and student code based on function headers and signatures
4. Functionality Diffing: Creating a structured difference report focusing on critical elements such as initializations, conditions, loops, and return statements

For LLM integration, we developed a specialized prompt template that presented the structured difference report in a JSON format, with explicit instructions to:

- Analyze ONLY the specific differences highlighted
- Explain whether these differences could cause the observed test failure
- Suggest a minimal, focused fix addressing only the specific issue
- Avoid suggesting changes to other parts of the code
- Maintain the original structure of the function

This highly constrained approach provided clear boundaries for LLM debugging activity, effectively implementing the ”stop signal” missing in more general approaches. By focusing the LLM’s attention on specific functional differences rather than the entire codebase, we achieved a better balance between bug identification accuracy and minimality of changes.

4. Results

This section presents the key findings from our comparative evaluation of different LLM-based debugging approaches. We analyze performance across metrics that address both debugging accuracy and modification behavior.

4.1. Comparative Performance Analysis

Table 1 summarizes the bug detection accuracy across models for each methodology.

Table 1. Bug Detection Accuracy by Model (%)

Method	**Claude 3.5**	**GPT-4o**	**Gemini 2.0**
Baseline	59.8	57.2	48.4
Chunking	67.2	63.5	53.7
Test Case	71.4	68.7	58.9
Multi-Agent	78.6	76.3	65.1
AST-Based	74.8	72.6	61.8

Table 2 shows the comparison of key metrics across methodologies.

Table 2. Performance Metrics Comparison

Method	**Bug Detection Accuracy (%)**	**Code Changes (% of reference)**	**False Pos. Rate (%)**
Baseline	52.3	178	28.6
Chunking	59.8	156	24.1
Test Case	64.7	132	18.3
Multi-Agent	72.6	167	14.7
AST-Based	68.4	115	12.3

Our results demonstrate a clear pattern across debugging methodologies. The multi-agent framework achieved the highest bug detection accuracy (72.6%) but at significantly higher API costs and with excessive code modifications (167% of reference solution changes). The AST-based approach offered the best balance between accuracy (68.4%) and minimality of changes (115% of reference solution). The baseline monolithic approach performed worst in accuracy while paradoxically making the most extensive code modifications.

The Change Ratio represents how many changes each method makes compared to the reference solution. For example, the baseline approach makes 1.78x more changes than necessary (if the reference solution changes 10 lines, the baseline approach would change about 18 lines). The AST-based approach is the most efficient, making only 1.15x the necessary changes, while still maintaining good accuracy

False positive rates also varied significantly across methodologies. The AST-based approach achieved the lowest false positive rate (12.3%), meaning it was less likely to identify non-existent problems. This contrasts with the baseline approach which had more than double the false positive rate (28.6%), frequently flagging issues that weren't actually bugs.

4.2. Model-Specific Performance

As shown in Table 1, performance varied across models, but the relative advantages of each methodology remained consistent. Claude 3.5 Sonnet consistently outperformed other models across all debugging approaches, demonstrating superior code comprehension and reasoning capabilities. GPT-4o performed well but trailed Claude across all methods. Gemini 2.0 Pro showed significantly lower performance with a substantial gap compared to the other models, particularly in the more complex methodologies like multi-agent and AST-based approaches.

4.3. Code Change Analysis

A critical finding was the variation in modification behavior across approaches. While the multi-agent framework achieved high accuracy, it typically modified 40-60% more code than necessary compared to reference solutions. In contrast, the AST-based approach made more targeted modifications, with changes typically limited to the specific functions containing bugs. This selective modification behavior is reflected in its lower code change percentage (115%).

The targeted test case injection approach also demonstrated good minimality (132%), as its incremental nature prevented the sweeping changes observed in other approaches. Both the baseline and chunking approaches tended to make excessive modifications, often completely rewriting functions when only minor fixes were required.

5. Limitations and Future Work

5.1. Limitations

Our study has several limitations that should be considered when interpreting the results:

- **Dataset Constraints:** Although DebugEVAL provides a standardized framework for evaluation, it may not represent the full diversity of bugs encountered in professional software development. The dataset focuses primarily on common bug patterns, potentially overlooking more complex or domain-specific issues.

- **Language Coverage:** Our evaluation concentrated primarily on Python and JavaScript code samples. The effectiveness of these approaches may vary across different programming languages, particularly those with more complex type systems or memory management requirements.

- **Scale Limitations:** While we tested on educational code samples of varying complexity, we did not evaluate these approaches on very large production codebases with hundreds of thousands of lines of code, where different challenges might emerge.

5.2. Future Work

Several promising directions for future research emerge from our findings:

- **Incremental Change Path Exploration:** Developing methods that only allow for one change at a time and incrementally evaluate multiple possible modification paths. Such an approach would check how each incremental change affects test outcomes, then self-reflect on whether pairs of changes might be more effective, continuing this process until reaching the minimal sufficient set of modifications. This would enforce a principled ”stopping point” based on empirical results rather than LLM judgment.

- **Enhanced Prompting Techniques:** Developing more sophisticated prompting strategies specifically designed to limit overcorrection. This could include explicit ”change budgets” or requiring justification for each modification.

The long-term goal remains developing LLM-based debugging tools that not only identify and fix bugs correctly but do so with the minimal necessary intervention—knowing when to stop is indeed as important as knowing what to fix.

References

AI, M. Code llama: Open and efficient code generation. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing (EMNLP 2023)*, pp. 23–34, Toronto, Canada, 2023. ACL.

Brown, T., Mann, B., Ryder, N., Subbiah, M., et al. Language models are few-shot learners. In *Advances in Neural Information Processing Systems (NeurIPS) 2020*, pp. 1877–1901, Virtual, 2020. NeurIPS.

Chen, M., Tworek, J., Jun, H., Yuan, Q., de Oliveira Pinto, P., Kaplan, M., Li, N., Edwards, M., Burda, P., Hesse, D., et al. Evaluating large language models trained on code. In *Proceedings of the Neural Information Processing Systems (NeurIPS) Workshop on Machine Learning for Code*, pp. 1–10, Virtual, 2021. NeurIPS.

Chen, Y., Guo, Z., Ma, H., and Li, S. Recent advances in code generation using transformer models. In *Proceedings of the 45th International Conference on Machine Learning (ICML 2022)*, pp. 110–120, Virtual, 2022. PMLR.

DeepMind. Competition-level code generation with alphacode. In *Proceedings of the 39th International Conference on Artificial Intelligence (AAAI 2022)*, pp. 512–523, San Francisco, CA, 2022. AAAI Press.

Goues, C. L., Nguyen, T., Forrest, S., and Weimer, W. Genprog: A generic program repair approach for software. In *Proceedings of the 31st International Conference on Software Engineering (ICSE 2013)*, pp. 120–130, New York, NY, 2013. ACM.

Gupta, R., Kanade, A., and Pande, P. S. Deepfix: Fixing c programs with deep learning. In *Proceedings of the 39th International Conference on Software Engineering (ICSE 2017)*, pp. 1–10, Austin, TX, 2017. IEEE.

Liu, H., Zhao, A., Chen, Y., and Li, X. Incoder: A generative model for code infilling. In *Proceedings of the 39th International Conference on Machine Learning (ICML 2022)*, pp. 123–134, Vienna, Austria, 2022. PMLR.

Mou, L., Li, G., Zhang, L., Wang, T., Jin, D., and Li, Z. Convolutional neural networks over tree structures for programming language processing. In *Proceedings of the 24th International Conference on Artificial Intelligence (AAAI 2016)*, pp. 169–178, Phoenix, AZ, 2016. AAAI Press.

Wang, L., Zhao, X., and Li, M. Controlled code generation: Limiting overcorrection in automated debugging. In *Proceedings of the 48th International Conference on Machine Learning (ICML 2023)*, pp. 230–240, Vienna, Austria, 2023. PMLR.

Yao, Y., Zhou, A., Huang, S., Xie, J., et al. React: Synergizing reasoning and acting in language models. In *Proceedings of the 40th International Conference on Machine Learning (ICML 2022)*, pp. 1–10, Honolulu, HI, 2022. PMLR.