DOTA: Intelligent Debugging via Delta of Thought Agents

Rui Yang, Rajiv Gupta, Qian Zhang, Ashish Kundu
{ryang088,rajivg,qzhang}@ucr.edu, ashkundu@cisco.com

Abstract

Large language models (LLMs) have shown exceptional capabilities in code understanding and generation. However, they still face significant challenges in analyzing and debugging code. Most existing works rely on a single model, which often struggles to detect and fix bugs in complex program structures and semantic logic. This paper presents DOTA, a novel debugging framework that enhances LLMs' debugging capabilities through multi-agent collaboration. Our key innovation is two-fold. First, we enhance code understanding through automated hierarchical documentation analysis, enabling more effective bug detection and localization based on comprehensive program context. Second, we leverage a delta-of-thought process where multi LLM agents analyze different aspects of program correctness and iteratively contribute complementary insights to identify bugs. Our experiments show that combining different agents with enriched documentation context significantly improves LLMs' debugging capabilities. DOTA has been extensively evaluated on the DebugBench dataset of 4,253 debugging instances. It achieves an average improvement of 15% in bug detection accuracy across languages compared to GPT-3.5 with task background prompting. The framework shows particular strength in handling complex logical errors (+18.3%) and multiple bugs (+18.9%). On open-source models, DoTA enhances bug detection capabilities by 10.4–13.5%.

1 Introduction

The rapid development of large language models (LLMs) has shown promising potential in code understanding and analysis. While models such as GPT-4 [1] and Codex [2] demonstrate significant capabilities in code comprehension across various programming languages [3], their ability to accurately analyze code, detect bugs, and provide reliable debugging suggestions remains a challenge [4]. These limitations stem from the following:

- *Single Model Dependence*: Most existing work utilize a single model to interpret and analyze code. Their effectiveness is limited and often biased due to a singular perspective [5, 6]. For example, even state-of-the-art models show significant performance drops when dealing with complex semantic bugs that require multi-faceted understanding beyond simple pattern matching.
- *Inconsistent Code Analysis*: Effective debugging requires reliable, deep code understanding [7]. However, recent studies have highlighted that even advanced LLMs show inconsistent reasoning capabilities and often miss subtle semantic bugs [8]. This limitation becomes particularly evident when dealing with complex programming logic and multi-module dependencies [9].
- *Limited Context Understanding*: Existing LLM-based techniques often focus on low-level source code snippets but overlook the high-level context of desired functionality, which is critical for understanding system-wide invariants and identifying logical inconsistencies [10].

To address these limitations, we introduce DOTA (Delta of Thought Agents), a framework that enhances code debugging through multi-agent collaboration and automated documentation enrichment. Our key insight is that combining specialized agents' analyses with hierarchically augmented code documentation enables LLMs to understand program behavior better and thus detect bugs more accurately.

The name *DoTA* highlights the framework's core innovation: the *delta* represents the incremental differences and complementary insights obtained from multiple agent perspectives. Unlike single-model approaches that provide a monolithic analysis, our framework captures the *delta*—the unique contributions each specialized agent provides beyond what others detect. This delta-based approach allows us to aggregate diverse an-

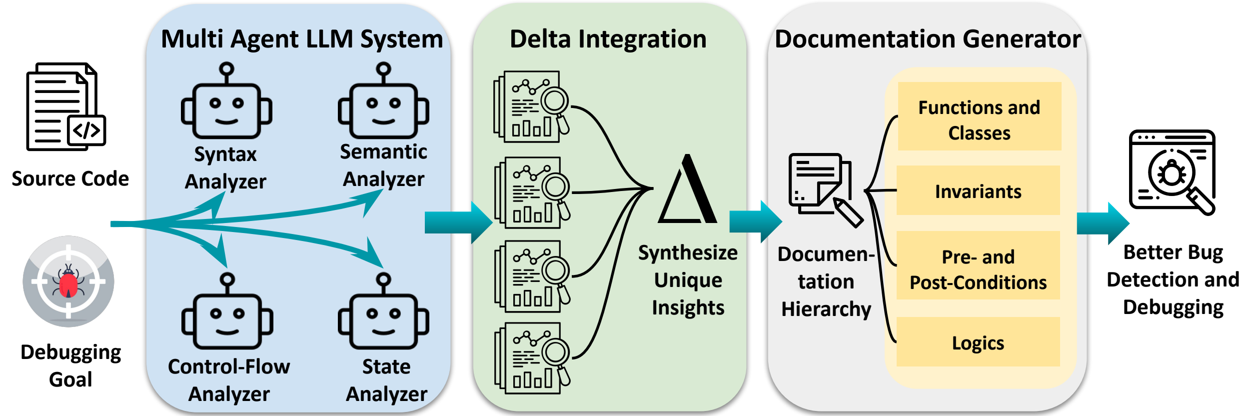

Figure 1: Overview of the DoTA framework. It combines a Multi-Agent LLM System (Section 3.1), a Delta Integration Module (Section 3.2), and an Automated Documentation Generator (Section 3.3) in a closed loop. Specialized agents analyze the code from different perspectives, the delta module synthesizes their complementary insights, and the documentation generator injects these insights back into the code as hierarchical annotations, enabling refinement of debugging quality.

alytical viewpoints, where each agent's delta contributes novel information about potential bugs, code patterns, or semantic issues that are overlooked by other agents. By systematically collecting and synthesizing these deltas, DoTA builds a more comprehensive understanding of the code than any individual model can achieve.

At the heart of DoTA is a two-fold innovation. First, DoTA implements a specialized *delta-of-thoughts* mechanism, where different LLM agents are assigned distinct analytical roles. A high-level agent focuses on architectural patterns and module interactions, while specialized agents examine type consistency, control flow, and state management. The coordinator agent identifies novel insights by analyzing the semantic differences (deltas) between agent outputs, filtering redundant information, and synthesizing complementary perspectives [11].

The second innovation lies in DoTA's automated multi-level documentation system. This system generates a hierarchical set of code annotations spanning multiple abstraction levels, from architectural patterns and module relationships to functional specifications and implementation constraints [10]. These structured annotations make explicit the complex code relationships and system invariants that are often implicit in raw code, significantly improving LLMs' ability to detect semantic bugs and logical inconsistencies [12].

We have extensively evaluated DoTA on DebugBench [13] with 4,253 debugging instances, using GPT-3.5 with task background prompting as our baseline model. Our framework demonstrates substantial improvements in bug detection accuracy, particularly for complex logical errors (+18.3%) and scenarios with multiple interacting bugs (+18.9%). Ablation studies show that both components are crucial: removing either the multi-agent collaboration or documentation system reduces the overall accuracy by approximately 5%, from 89.2% to 84.8% and 84.5% respectively. These results validate that the integration of collaborative agents with enhanced documentation represents a step forward.

2 Related Work

Code Generation with LLMs. In recent years, Large Language Models (LLMs) such as Codex [2] and GPT-4 [1] have shown significant promise for automatic code generation. These models are trained on large-scale code corpora and adapted to programming tasks ranging from basic algorithms to complex system designs [14]. Recent advances include StarCoder [15], which demonstrates enhanced multilingual code generation, and CodeGeeX [16], which achieves competitive performance with specialized architectures. Through architectural improvements, quality of code generated has improved considerably, with models producing functional code that sometimes exceeds human performance on CodeXGLUE [6] and mHumanEval [17].

Nevertheless, several significant limitations persist. Models may generate incorrect or unnecessarily complex code [4]. Moreover, while these models excel at generating independent code segments, integrating such segments into larger systems with multiple dependencies remains problematic [9]. Recent studies [9] indicate substantial challenges in cross-language code generation.

Traditional Debugging Techniques. Traditional debugging methods, including breakpoint-based debugging, static analysis, and dynamic program tracing, form the

Algorithm 1 Protocol for Agent Coordination

```
1:  procedure COORDINATEAGENTS(code, agents)
2:      Input: code - source code to analyze; agents - list of specialized LLM agents
3:      Output: consensus - integrated bug detection result with identified deltas
4:      responses ← []                                   ▷ List stores agent analyses and their deltas
5:      for all agent in agents do                                   ▷ Process each specialized agent
6:          analysis ← agent.analyzeCode(code)      ▷ Agent-specific analysis based on its specialization
7:          weight ← calculateWeight(agent.history)              ▷ Weight based on historical accuracy
8:          responses.append((analysis, weight))               ▷ Store analysis and reliability score
9:      end for
10:     consensus ← buildConsensus(responses)                ▷ Integrate analyses, extracting deltas
11:     updateAgentWeights(responses, consensus)       ▷ Adjust weights based on contribution quality
12:     return consensus                             ▷ Return synthesized result with unique insights
13: end procedure
```

foundation of software development practices. Tools like GDB [18] and Valgrind [19] monitor program execution, while static analyzers detect bugs before runtime. Advanced techniques such as symbolic execution facilitate bug detection in complex systems [20]. Modern surveys of fault localization techniques highlight the transition from spectrum-based methods to learning-based approaches [7]. However, these methods suffer from scalability issues, requiring exhaustive path exploration that becomes intractable for large systems.

AI-Assisted Debugging. AI-based debugging is an active area of research. Early systems like BugFix [21] and CoCoNuT [22] demonstrated that machine learning models can effectively identify and repair software bugs. Notable advances include DeepDebug [23], which leverages transformer architectures for comprehensive code analysis. Most current systems rely on single AI models, limiting their effectiveness on complex bugs. CriticGPT [8] pioneered the use of LLMs to criticize and improve code, showing models can examine complementary aspects to improve bug detection.

One cluster of the research efforts long this direction leverage multi-agent frameworks. Such systems employ specialized AI agents to examine different code aspects—syntax, semantics, control flow, and state management. Recent systems like AgentCoder [11] and ChatDev [24] demonstrate the effectiveness of collaborative agent-based approaches in software engineering tasks. Unlike single-model approaches, these frameworks utilize specialized roles to verify and refine code iteratively. DoTA distinguishes itself by implementing a delta-based collaboration protocol. While existing multi-agent learning systems [25] support the use of specialized roles, DoTA focuses on capturing unique contributions (deltas) from each agent.

Notably, code documentation plays a crucial role in this direction by clarifying intended functionality. Tools like CodeT5 [5] demonstrate that pre-trained models for code-to-text tasks enhance code understanding. High-quality documentation significantly improves debugging accuracy by providing semantic context [10]. This multi-level structure, inspired by literate programming principles [26], provides a richer context for both human developers and AI debugging agents.

3 DoTA Methodology

We propose the DoTA framework that leverages multiple LLM agents for a delta-based reasoning process. Figure 1 illustrates the overall framework. The core innovation lies in the delta-of-thought mechanism, where *delta* refers to the unique, complementary insights contributed by each specialized agent during collective debugging. The framework combines ① delta-based multi-agent collaboration with ② automated hierarchical documentation generation, enabling comprehensive code analysis from multiple perspectives.

The DoTA framework consists of three interconnected components that form a closed feedback loop: (1) a *Multi-Agent LLM System* that performs specialized code analysis through different agents; (2) a *Delta Integration Module* that identifies and synthesizes unique insights from each agent; and (3) an *Automated Documentation Generator* that creates multi-level code annotations. The feedback loop enables continuous refinement, where documentation enhances subsequent agent analysis, which in turn improves bug detection accuracy.

3.1 Multi-Agent LLM System

DoTA leverages a multi-agent system for specialized code analysis. The system instantiates agents with distinct analytical roles: syntax analysis agents detect grammatical and type-related errors, semantic analysis agents examine program logic, control flow agents trace execution behavior, and state management agents verify data

Algorithm 2 Framework for LLM Integration and Agent Instantiation

```
1:  procedure INTEGRATELLM(code, focus)
2:      Input: code - source code segment; focus - analytical focus (e.g., syntax, semantics, control flow)
3:      Output: parsed_analysis - structured bug detection results with confidence scores
4:      prompt ← generateSpecializedPrompt(code, focus)              ▷ Create focus-specific prompt
5:      context ← extractCodeContext(code)                           ▷ Extract relevant code context
6:      model ← selectAppropriateModel(focus)                        ▷ Select specialized agent
7:      response ← model.generate(prompt, context)                   ▷ Generate analysis
8:      parsed_analysis ← parseResponse(response)                    ▷ Structure the output
9:      validateAnalysis(parsed_analysis)   ▷ Verify analysis quality return parsed_analysis   ▷ Return validated
    analysis
10: end procedure
```

consistency. Each agent contributes a partial analysis that is later compared against the others to identify informative deltas for collective analysis.

Algorithm 1 presents the coordination protocol. It takes as input the source code together with a set of specialized agents, such as syntax, semantic, and control-flow analyzers. In Lines 5-7, each agent is preconfigured or fine-tuned for a particular bug detection task, and the `agent.history` field in Line 7 maintains performance signals from previous analyses. During execution, each agent independently analyzes the input and produces a structured assessment of potential defects within its area of expertise.

To combine these assessments, DoTA estimates agent reliability from historical performance using a sigmoid weighting function:

$$w_i = \frac{1}{1 + e^{-k(p_i - p_{\text{avg}})}} \tag{1}$$

where w_i is agent i's weight, p_i is its performance score, p_{avg} is the average performance across all agents, and k controls weight sensitivity (empirically set to 2.0). It then synthesizes the analyses by preserving non-overlapping insights as explicit delta contribution:

$$\text{Consensus} = \sum_{i=1}^{n} w_i \cdot \text{Analysis}_i + \Delta_{\text{unique}} \tag{2}$$

where Δ_{unique} denotes the non-overlapping findings contributed by individual agents. After consensus construction, agents weights are updated according to contribution quality so that agents that consistently produce accurate and useful deltas receive greater influence.

Instantiation of Specialized Agents. DoTA comprises *four* specialized agents: a syntax analyzer for syntax errors, type mismatches, and compilation-related issues; a semantic analyzer for logical consistency, variable usage, and semantic correctness; a control-flow analyzer for execution-path reasoning and flow anomalies such as unreachable code; and a state analyzer for state consistency, memory management, and resource handling. This design preserves a clear separation of analytical roles while remaining easy to adapt and deploy. In our implementation, such multi-agent system is instantiated from a single base model, GPT-3.5, using task-specific background prompting and prompt-based role specialization rather than model fine-tuning.

Algorithm 2 details the flexible multi-LLM architecture that enables specialized agents to collaborate effectively. It generates role-specific prompts in Line 4 that steer each agent toward its intended debugging perspective. For example, the syntax agent emphasizes grammatical correctness, whereas the semantic agent focuses on logical consistency. The algorithm also extracts local declarations in Line 5, function signatures, and dependency information from a configurable context window; in our implementation, this window spans 50 lines.

Agent selection is based on both role suitability and historical performance:

$$\text{agent} = \arg\max_{a \in A} (\text{Capability}(a, \text{focus}) \times \text{Performance}(a)) \tag{3}$$

where A denotes the available agent pool, Capability measures alignment between an agent and the requested analytical focus, and Performance captures prior reliability. Each selected agent then produces a structured response containing the predicted bug location, defect type, severity, and confidence score. Before these results are forwarded to the delta integration module, the system validates them for consistency, completeness, and adherence to the expected schema.

3.2 Delta Integration Module

DoTA uses a delta integration module to consolidate the outputs of the specialized agents. Its role is to identify observations that are complementary rather than redundant. In practice, the module compares the candidate findings produced by each agent, preserves high-value

Algorithm 3 Protocol for Agent Weight Adjustment

```
1: procedure UPDATEAGENTWEIGHTS(responses, outcome)
2:     Input: responses - agent analyses with current weights; outcome - consensus result
3:     Output: Updated agent weights reflecting contribution quality
4:     mean_performance ← calculateMeanPerformance(responses)      ▷ Calculate baseline performance
5:     for all (agent, response) in responses do
6:         performance ← evaluatePerformance(response, outcome)     ▷ Measure contribution quality
7:         delta ← α× (performance − mean_performance)              ▷ Calculate adjustment
8:         agent.weight ← agent.weight + delta                      ▷ Update weight
9:         agent.history.append(performance)                        ▷ Maintain performance history
10:    end for
11:    normalizeWeights(agents)                                     ▷ Ensure weights sum to 1
12: end procedure
```

non-overlapping evidence, and incorporates reliability-aware weighting when building a consensus diagnosis. This design is intended to retain minority but valuable signals that might otherwise be lost in a simple majority-vote scheme.

Weight Adjustment. The delta integration module supports adaptation over time. Agent weights are updated according to how well each agent's findings align with the final debugging outcome, allowing the system to emphasize agents that repeatedly contribute useful information while still preserving diversity across perspectives. This mechanism is central to DoTA's feedback loop, because the consensus result becomes a stronger input to downstream documentation generation and later debugging iterations.

Algorithm 3 depicts that DoTA evaluates each agent's contribution using three signals: the number of unique bugs it contributes, the correctness of those findings with respect to the final outcome, and its consistency with other reliable agents. The update magnitude is controlled by a learning-rate parameter α in Line 7:

$$\Delta w_i = \alpha(r_i - \bar{r}) \quad (4)$$

where $\alpha = 0.1$ (empirically determined), r_i is agent i's performance score, and $\bar{r}$ is the mean performance. To support adaptation over time, `agent.history` in Line 9 maintains a sliding window of the last 100 analyses, enabling long-term performance tracking and identification of agent specialization patterns across code types. After the update, weights are normalized to maintain numerical stability:

$$w_i = \frac{w_i}{\sum_{j=1}^{n} w_j} \quad (5)$$

In summary, DoTA uses $k = 2.0$ in the sigmoid weighting function, $\alpha = 0.1$ for weight updates, a history window of 100 analyses, and a minimum weight threshold of 0.05 to prevent an agent from being eliminated entirely.

3.3 Automated Document Generation

Once DoTA identified these analytical deltas, it automatically augments the code with hierarchical annotations that incorporate this information. Algorithm 4 describes this process. Rather than producing a single plat explanation, the generator organizes the resulting annotations across multiple abstraction levels so that later analysis can use structural, behavioral, and implementation-level information relevant to debugging.

On Line 1, the documentation algorithm takes as input the source code together with background project context, including design artifacts, commit history, and code dependencies. It produces annotations at four abstraction levels: the architecture level captures system-wide patterns and module interactions, the component level focuses on class- or module-level specifications, the function level models method contracts and behaviors, and the implementation level describes line-level logic and constraints.

For each level, DoTA extracts features appropriate to that abstraction in Lines 5-11, such as class hierarchies for component level or control flow information at the function level. It then incorporates the outputs of multi-agent analysis through a weighted contextual representation:

$$\text{Integrated_context} = \sum_{i=1}^{n} w_i \cdot C_i \quad (6)$$

where C_i is agent i's contextual analysis and w_i is its reliability weight. The background parameter enriches this representation with project-specific knowledge, improving semantic understanding. Finally, DoTA synthesizes level-specific documentation by combining code evidence, agent analysis, and project background context in Lines 9-10.

Documentation Hierarchy. The four-level documentation hierarchy provides the DoTA access to complimentary forms of context. Figure 2 illustrates this design on a binary-tree traversal example by showing how the same

Algorithm 4 Protocol for Documentation Generation

```
1:  procedure GENERATEDOCUMENTATION(code, background)
2:      Input: code - source code; background - existing code context and metadata
3:      Output: doc_levels - hierarchical documentation from L1 to L4
4:      doc_levels ← []                                   ▷ Initialize multi-level documentation storage
5:      for level in [ARCHITECTURE, COMPONENT, FUNCTION, IMPLEMENTATION] do
6:          context_vector ← getContext(code, level)                  ▷ Extract level-specific features
7:          agent_inputs ← collectAnalysis(context_vector)            ▷ Gather specialized agent analyses
8:          weighted_context ← computeWeightedContext(agent_inputs)   ▷ Integrate analyses with weights
9:          documentation ← generateDoc(weighted_context, background) ▷ Generate level-specific documentation
10:         doc_levels.append(documentation)                          ▷ Store in hierarchy
11:     end forreturn integrateDocLevels(doc_levels)      ▷ Return integrated multi-level documentation
12: end procedure
```

code fragment is annotated across the four documentation levels, from architectural intent to line-level behavior. Level 1 captures system-wide invariants, data-flow patterns, module dependencies, and architectural constraints. Level 2 describes class responsibilities, state management, interface contracts, and component interactions. Level 3 records method preconditions and postconditions, parameter constraints, exception handling, and behavioral specifications. Level 4 provides line-by-line logic explanations, algorithmic details, state modifications, and error-handling information.

This hierarchical structure helps the agents reason about code at multiple granularities, improving their ability to detect bugs that span abstraction levels. It also closes the feedback loop introduced in the overview: the documentation produced by this component informs later analysis, and improved analysis in turn yields more accurate documentation.

4 Experimental Results

Experimental Setup. Our experiments were conducted on the DebugBench dataset [13], which comprises 4,253 debugging instances sourced from the LeetCode community. This dataset includes 1,438 C++ instances, 1,401 Java, and 1,414 Python, spanning syntax, reference, logic, and multiple error types. We evaluated DoTA, consisting of multi-agent LLM collaboration and automated inline documentation, against baseline approaches, including single-model reasoning without documentation. We use GPT-3.5 with task background prompting as baseline model.

The evaluation metrics focus on bug detection accuracy: whether identified error aligns with the ground-truth. We performed experiments on a high-performance cluster with NVIDIA A100 GPUs, ensuring standardized conditions. To verify that improvements are substantial and not coincidental, we performed manual verification.

```
/* L1:  Binary Tree Path Sum Module
* Flow:  Tree traversal with sum checks
* States:{exploring,validated,terminated}
* Invariant:  Complete path validation */
struct TreeNode {
    int val;
    TreeNode *left, *right;
};
/* L2:  PathSum Component
* Flow:  Node -> Path -> Validation
* States:  DFS traversal states
* Contract:  Leaf path completion */
class Solution {
public:
    /* L3:  Path Validation
     * Process:  Sum reduction per node
     * Cases:  null, leaf, internal node
     * Ensures:  Leaf-path completeness */
    bool hasPathSum(TreeNode* root, int
sum) {
        // L4:  Base case - empty tree
        if (!root) return false;
        // L4:  Leaf node validation
        if (!root->left && !root->right)
            return root->val == sum;
        // L4:  Path exploration
        return hasPathSum(root->left,sum -
root->val) || hasPathSum(root->right,sum
- root->val);
    }
};
```

Figure 2: Multi-level documentation example showing how DoTA automatically generates comprehensive understanding at multiple abstraction levels.

4.1 Results Across Languages

Table 1 reports performance across three programming languages. DoTA improves over the baseline in all three, with the largest gain in Python. A plausible explanation is that Python debugging often depends more heavily on

Table 1: Bug Detection Performance by Language.

Language	Baseline	DoTA	Gain
C++	74.2%	88.3%	+14.1%
Java	73.5%	87.8%	+14.3%
Python	75.1%	91.8%	+16.7%

Table 2: Bug Detection Performance by Error Type.

Error Type	Baseline	DoTA	Gain
Syntax	76.8%	90.2%	+13.4%
Reference	72.3%	88.1%	+15.8%
Logical	68.5%	86.8%	+18.3%
Multiple	61.2%	80.1%	+18.9%

implicit invariants and programmer intent, making the additional semantic context supplied by documentation and multi-agent reasoning especially useful.

4.2 Results on Error Types

Table 2 shows the largest gains in logic and multiple errors. This aligns with the complexity of these errors: logical bugs often require understanding multiple execution paths, and multi-errors combine syntactic/semantic flaws that interact in non-obvious ways. The data suggests that DoTA's approach is particularly effective at handling these complex error types where multiple aspects of code understanding must be combined.

4.3 Ablation Study

The ablation results in Table 3 show that multi-agent collaboration and inline documentation yield complementary benefits. Removing either reduces accuracy by more than four percentage points, indicating that the gains do not arise from a single dominant design choice.

4.4 Benefits of Inline Documentation

Table 4 focuses on the impact of inline documentation by error type. The data clearly shows that logical errors and multi-errors benefit most from documentation. This is likely because these error types depend heavily on understanding intended invariants, function roles, and permissible value ranges. The documentation provides crucial context that helps models better understand these complex relationships.

4.5 Performance on Open Source Models

To demonstrate the generalizability of our approach, we evaluated DoTA's effectiveness across widely-used open-source LLMs with task background prompting.

Table 3: Ablation Study Results.

Removed Component	Accuracy	Full System
Multi-agent collaboration	84.8%	89.2%
Inline documentation	84.5%	89.2%

Table 4: Impact of Automated Inline Documentation.

Error Type	Without Docs	With Docs	Gain
Syntax	84.7%	90.2%	+5.5%
Reference	82.9%	88.1%	+5.2%
Logical	78.9%	86.8%	+7.9%
Multiple	72.1%	80.1%	+8.0%

Table 5: Bug Detection Via Open Source Models.

Base Model	Original	+ DoTA	Gain
LLaMA2-70B	65.4%	78.9%	+13.5%
Mistral-7B	61.2%	72.8%	+11.6%
Code-LLaMA-13B	63.5%	76.2%	+12.7%
Phi-2	60.1%	70.5%	+10.4%

These models, while powerful, typically perform below proprietary models like GPT-4 or Claude 3 on complex debugging tasks. However, our framework shows consistent improvements across all tested models, suggesting that the combination of multi-LLM collaboration and multi-level documentation can enhance debugging capabilities regardless of the base model's performance.

As shown in Table 5, while the improvements are substantial across different model scales, the enhanced performance still remains below that of leading proprietary models. The larger LLaMA2-70B shows the most significant improvement, from 65.4% to 78.9%, while specialized models like Code-LLaMA-13B also demonstrate strong gains despite their smaller size. Even compact models like Phi-2 achieve improvements, though the enhancement varies with model scale and architecture.

This varying improvement pattern across different open-source models suggests that our method provides fundamental enhancements to the debugging process rather than merely amplifying existing model capabilities. The results also indicate that while our approach significantly improves open-source model performance, there remains a gap compared to proprietary models, highlighting the complementary nature of our enhancement with underlying model capabilities.

4.6 Manual Verification of Results

To ensure that our improvements reflect genuine understanding rather than coincidental textual matches, we conducted manual verification on these instances. We

confirmed that in the majority of cases, the model's identified bugs corresponded to actual code defects. The discovered errors match the underlying semantics of the code, demonstrating that DoTA's improvements represent genuine enhancement in bug detection capabilities.

5 Case Study

5.1 Multi-Language Repository Analysis

To examine whether DoTA remains useful beyond benchmark-style bug-fixing tasks, we conducted a case study on a multi-language repository implementing a complete data-processing pipeline across three programming languages. This setup reflects a common software architecture in which different components exploit language-specific strengths while communicating across language boundaries.

The repository implements a data-processing workflow whose functionality is distributed across a Python data collector (`python/data_collector.py`), a c++ processing component (`cpp/processor.cpp`), a Java visualization module (`java/DataVisualizer.java`), a shared Python logging utility (`python/logger.py`), and a shell script (`run_demo.sh`) that compiles and executes the full pipeline. These components exchange data through files and command-line invocations, producing an inter-language dependency chain in which one component's assumptions directly influence the properties of the next.

We applied DoTA to this repository with the goal of analyzing both language-specific vulnerabilities and cross-component security issues that emerge only when the full pipeline is considered. The analysis proceeds in the same manner as in our main experiments: DoTA first generates hierarchical documentation from architectural to implementation levels, then invokes specialized agents for security, architecture, control-flow, and state-management reasoning, and finally combines the resulting evidence to trace vulnerabilities across component boundaries. This setting is particularly useful for evaluating whether the framework can move beyond isolated bug identification and instead reason about cross-language propagation paths.

5.2 Key Findings

DoTA identified multiple security vulnerabilities throughout the repository. In the C++ component, it detected a buffer overflow in `processArguments`, a use-after-free error in `demonstrateUseAfterFree`, and a memory leak in `processData`. In the Python code, it identified command injection in `call_cpp_processor`, path traversal in `save_data`, and SQL injection in `process_user_input`. In the Java module, it reported command injection in `logActivity`, path traversal and cross-site scripting in `generateHtmlReport`, and a resource leak in `readJsonData`. Taken together, these findings indicate that the framework is able to surface both conventional single-language vulnerabilities and issues that arise from the way components are connected.

One of the most important findings was the presence of vulnerability chains that propagate across language boundaries. In one representative example, the Python data collector concatenates user input into a command string that is then passed to the C++ processor. The C++ component further propagates the vulnerability through `system()` calls on tainted input, creating a path to arbitrary command execution.

One representative example is the buffer-overflow vulnerability in the C++ processor. DoTA identifies that the `processArguments` function copies user input directly into the fixed-size global buffer `g_buffer` by calling `strcpy` without validating the input length. From a security perspective, the intended requirement is straightforward: user input must not exceed buffer capacity. Because `g_buffer` has size `BUFFER_SIZE` (1024 bytes), any larger than threshold argument can overwrite adjacent memory and create a classic stack-based buffer-overflow condition. The framework therefore correctly connects the local code pattern to a broader security consequence, namely the possibility of arbitrary code execution, process crashes, or information disclosure.

5.3 DoTA's Effectiveness and Limitations

The case study also helps clarify where DoTA is strongest and where it remains limited. The framework performs well on vulnerabilities with strong static signatures, including memory-safety errors, command-injection risks, and path-traversal behavior. It is moderately effective on data-validation and resource-management problems, where correct diagnosis often depends on broader semantic context but still leaves observable source-level evidence. Its weakest performance arises from asynchronous behavior and timing-sensitive defects, for which static reasoning alone is insufficient.

Several limitations also became clear. Without execution capabilities, DoTA cannot directly validate exploitability or resolve runtime-dependent behaviors. Accuracy declines when reasoning must cross deeply nested call chains or multiple components, particularly in the recursive C++ code. In addition, some language-specific idioms and optimized implementation patterns remain difficult for the framework to interpret reliably, and concurrency-related bugs such as race conditions or deadlocks remain challenging because they depend on execution ordering rather than only source structure.

This repository-level analysis shows that DoTA can detect serious security vulnerabilities across language boundaries and can expose propagation chains that are difficult to recover with single-language analysis alone. The case study also reinforces the role of hierarchical documentation and multi-agent reasoning in understanding cross-component interactions and mismatches between intended and actual behavior. At the same time, it highlights a clear direction for future work: coupling the current static reasoning process with dynamic analysis and strengthening support for language-specific patterns, runtime behavior, and concurrency.

5.4 Relevant Code Listings

For completeness, we include excerpts of the key vulnerable sections of each component:

Listing 1: C++ Processor - Buffer Overflow

```
// Global buffer with fixed size
char g_buffer[BUFFER_SIZE];
// Process command line arguments -
// contains buffer overflow vulnerability
void processArguments(int argc, char* argv
    ↪ []) {
  if (argc < 2) {
    return;
  }
  // BUG: Buffer overflow vulnerability
  strcpy(g_buffer, argv[1]);
  std::cout << "C++: Processing argument: "
          << g_buffer << std::endl; }
```

Listing 2: Python - Command Injection

```
def call_cpp_processor(data_file):
  """Call C++ processor - command injection
     ↪ """
  print("Python: Calling C++ processor...")
  cpp_processor = os.path.join(
    os.path.dirname(os.path.dirname(__file__
       ↪ )),
    "cpp", "processor"
  )
  # BUG: Command injection vulnerability
  cmd = f"{cpp_processor} {data_file}"
  try:
    # BUG: shell=True enables injection
    result = subprocess.run(
      cmd, shell=True,
      capture_output=True, text=True
    )
    if result.returncode != 0:
      print(f"Error: {result.stderr}")
      sys.exit(1)
    return result.stdout.strip()
  except Exception as e:
    print(f"Error: {e}")
    sys.exit(1)
```

Listing 3: Java - XSS Vulnerability

```
private static void generateHtmlReport(
    List<DataItem> items, String outputPath
) throws IOException {
  System.out.println("Generating HTML report
      ↪ ...");
  // BUG: Path traversal - no validation
  File outputFile = new File(outputPath);
  try (FileWriter writer =
       new FileWriter(outputFile)) {
    writer.write("<html><head>" +
      "<title>Data Report</title></head>" +
      "<body>\n<h1>Processed Data</h1>\n");
    writer.write("<table border='1'>" +
      "<tr><th>ID</th><th>Name</th>" +
      "<th>Value</th></tr>\n");
    for (DataItem item : items) {
      writer.write("<tr>");
      writer.write("<td>" + item.id + "</td>"
          ↪ );
      // BUG: XSS - unescaped user data
      writer.write("<td>" + item.name + "</td
          ↪ >");
      writer.write("<td>" + item.value + "</
          ↪ td>");
      writer.write("</tr>\n");
    }
    writer.write("</table></body></html>");
        ↪ } }
```

6 Discussion

The quantitative and qualitative results suggest that DoTA's combination of multi-agent collaboration and inline documentation improves debugging performance on complex software tasks. These improvements appear to stem from two structural changes in the reasoning process: multiple agents contribute complementary analytical perspectives, and documentation provides semantic context that helps guide inference.

Insight into Multi-Agent Collaboration. Complex logic bugs remain difficult to resolve, especially when defects span multiple functions or components. Our results indicate that parallel analysis by specialized agents is useful because different agents surface different classes of evidence. An individual agent may identify a local anomaly in control flow, data use, or program state, but the main benefit arises when these observations are combined into a shared diagnosis. This effect is particularly visible in cases where failures emerge from interactions among components rather than from a single localized error, which is precisely the setting in which single-pass debugging methods often struggle.

Role of Inline Documentation. The results also suggest that inline documentation is most valuable when the defect is semantically subtle rather than syntactically obvious. Structural code analysis alone is often insufficient for such cases, whereas documentation can provide cues

about intended behavior, invariants, and expected interactions. In DoTA, the multi-level documentation scheme helps bridge this gap by exposing context from architectural design down to implementation details. More broadly, the results support the view that documentation is not only a maintenance aid for developers, but also a useful source of semantic information for automated debugging systems.

Language and Error-Type Variations. Performance varies across programming languages and defect categories. Improvements are more pronounced in dynamic-language settings, where documentation may compensate for the lack of explicit type information and weaker static guarantees. We also observe stronger gains on compound defects and interleaved logic errors, particularly when the debugging task requires integrating evidence across multiple abstraction levels. These trends suggest that DoTA is most useful in settings where program understanding is context-heavy and where a purely local analysis is unlikely to be sufficient.

Challenges and Limitations. Evaluation on DebugBench ($n = 4{,}253$ test cases) also exposed important limitations. Broadly, the failure cases fall into two categories: those that exceed the reasoning depth of the current analysis process and those that depend on runtime behavior that is difficult to capture statically.

Limited Reasoning Capabilities. Analysis of 458 unsuccessful cases (10.8% of the dataset) revealed recurring patterns associated with limited reasoning depth. Deep call hierarchies, especially those extending beyond five levels, reduced effectiveness because tracking data and control dependencies became substantially harder. This issue was particularly visible in recursive implementations and programs with high cyclomatic complexity. We also observed weaker performance on cases requiring long inference chains, where correct diagnosis depends on several linked logical deductions.

Static Analysis Issues. Investigation of runtime-dependent scenarios (384 cases, 9.0%) exposed limitations inherent to largely static analysis. Behaviors involving network communication, file I/O, dynamic type changes, or complex input transformations often could not be resolved reliably from source context alone. Concurrent execution and resource-management behaviors were also difficult to analyze, particularly when correctness depended on runtime ordering. We also observed substantial overlap between external dependency failures and dynamic behavior, indicating that these two sources of difficulty are often coupled rather than independent. Thus, future versions of DoTA would likely benefit from improved integration of execution traces, runtime instrumentation, or environment-aware analysis.

7 Conclusion

We presented DoTA, a novel debugging framework that blends multi-agent LLM reasoning with structured integration and automated inline documentation. Through extensive experiments on the DebugBench dataset, we demonstrated improved accuracy across all languages and error types, with notable gains in logic and multi-error scenarios. Our quantitative and qualitative analyses show that improved debugging outcomes can be achieved by combining multiple reasoning agents, semantic guidance, and thorough verification against real-world code conditions. Our results demonstrate the effectiveness of combining collaborative agent reasoning with systematic documentation enhancement to advance automated bug detection and program analysis. Although DoTA represents a significant step forward, intricate bugs remain a challenge.

8 Acknowledgment

The participants in this work are in part supported by Cisco Research and the National Science Foundation grants CCF-2226448, CCF-2426161, and CCF-2512416 to UC Riverside. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

[1] OpenAI. GPT-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023.

[2] Mark Chen, Jerry Tworek, Heewoo Jun, et al. Evaluating large language models trained on code. In *arXiv preprint arXiv:2107.03374*, 2021.

[3] Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, et al. Code Llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*, 2023.

[4] Carlos E. Jimenez, John Yang, Alexander Wettig, et al. SWE-bench: Can language models resolve real-world GitHub issues? In *International Conference on Learning Representations (ICLR)*, 2024.

[5] Yue Wang, Weishi Wang, Shafiq Joty, and Steven C. H. Hoi. CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In *Proc. of the Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 8696–8708, 2021.

[6] Shuai Lu, Daya Guo, Shuo Ren, et al. CodeXGLUE: A machine learning benchmark

dataset for code understanding and generation. In *Advances in NeurIPS Datasets and Benchmarks Track*, 2021.

[7] W. Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. A survey on software fault localization. *IEEE Transactions on Software Engineering*, 42(8):707–740, 2016.

[8] Nat McAleese, Rai Rai, et al. Llm critics help catch llm bugs. In *The Thirteenth International Conference on Learning Representations (ICLR)*, 2025.

[9] Yangruibo Ding, Ziyang Wang, Wasi Uddin Ahmad, Hantian Ding, Ming Tan, Nihal Jain, Murali Krishna Ramanathan, Ramesh Nallapati, Parminder Bhatia, Dan Roth, and Bing Xiang. Crosscodeeval: a diverse and multilingual benchmark for cross-file code completion. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.

[10] Antonio Mastropaolo, Matteo Ciniselli, Massimiliano Di Penta, and Gabriele Bavota. Evaluating code summarization techniques: A new metric and an empirical characterization. In *Proc. of the IEEE/ACM International Conf. on Software Engineering (ICSE)*, 2024.

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

[12] Sungmin Kang, Bei Chen, Shin Yoo, and Jian-Guang Lou. Explainable automated debugging via large language model-driven scientific debugging. *Empirical Software Engineering*, 30(2):26, 2025.

[13] Runchu Tian, Yining Ye, Yujia Qin, Xin Cong, Yankai Lin, Yinxu Pan, Yesai Wu, Haotian Hui, Weichuan Liu, Zhiyuan Liu, and Maosong Sun. Debugbench: Evaluating debugging capability of large language models. In *Findings of the Association for Computational Linguistics: ACL 2024*, pages 4173–4198, 2024.

[14] Yujia Li, David Choi, Junyoung Chung, et al. Competition-level code generation with AlphaCode. *Science*, 378(6624):1092–1097, 2022.

[15] Raymond Li, Loubna Ben Allal, Yang Zi, et al. StarCoder: May the source be with you! *Trans. on Machine Learning Research (TMLR)*, 2023.

[16] Qinkai Zheng, Xiao Xia, et al. CodeGeeX: A pretrained model for code generation with multilingual evaluations on HumanEval-X. In *Proc. of the ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pages 5521–5531, 2023.

[17] Nishat Raihan, Antonios Anastasopoulos, and Marcos Zampieri. mHumanEval: A multilingual benchmark to evaluate large language models for code generation. In *Proc. of the Conference of the North American Chapter of the Association for Computational Linguistics (NAACL)*, 2025.

[18] Richard Stallman, Roland Pesch, and Stan Shebs. *Debugging with GDB: The GNU Source-Level Debugger*. Free Software Foundation, 2000.

[19] Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 89–100, 2007.

[20] Roberto Baldoni et al. A survey of symbolic execution techniques. *ACM Computing Surveys*, 51(3):1–39, 2018.

[21] Dennis Jeffrey, Min Feng, Neelam Gupta, and Rajiv Gupta. Bugfix: A learning-based tool to assist developers in fixing bugs. In *International Conf. on Program Comprehension*, pages 70–79, 2009.

[22] Thibaud Lutellier, Hung Viet Pham, et al. CoCoNuT: Combining context-aware neural translation models using ensemble for program repair. In *Proc. of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pages 101–114, 2020.

[23] Dawn Drain, Colin B Clement, Guillermo Serrato, and Neel Sundaresan. Deepdebug: fixing Python bugs using stack traces, backtranslation, and code skeletons. In *Proc. of the IEEE/ACM International Conference on Software Engineering (ICSE)*, pages 735–747, 2021.

[24] Chen Qian, Wei Liu, Hongzhang Liu, Nuo Chen, Yufan Dang, Jiahao Li, Cheng Yang, Weize Chen, Yusheng Su, Xin Cong, Juyuan Xu, Dahai Li, Zhiyuan Liu, and Maosong Sun. ChatDev: Communicative agents for software development. In *Proc. of the Annual Meeting of the Association for Computational Linguistics*, pages 15174–15186, August 2024.

[25] Sirui Hong, Xiawu Zheng, et al. MetaGPT: Meta programming for multi-agent collaborative framework. In *The Twelfth International Conference on Learning Representations (ICLR)*, 2024.

[26] Donald E. Knuth. Literate programming. *The Computer Journal*, 27(2):97–111, 1984.