TraceCoder: A Trace-Driven Multi-Agent Framework for Automated Debugging of LLM-Generated Code

Jiangping Huang
Chongqing University of Posts and Telecommunications
Chongqing, China
huangjp@cqupt.edu.cn

Wenguang Ye
Chongqing University of Posts and Telecommunications
Chongqing, China
s231231083@stu.cqupt.edu.cn

Weisong Sun*
Nanyang Technological University
Singapore, Singapore
weisong.sun@ntu.edu.sg

Jian Zhang
Beihang University
Beijing, China
zhangj3353@buaa.edu.cn

Mingyue Zhang
Southwest University
Chongqing, China
myzhangswu@swu.edu.cn

Yang Liu
Nanyang Technological University
Singapore, Singapore
yangliu@ntu.edu.sg

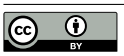
Abstract

Large Language Models (LLMs) often generate code with subtle but critical bugs, especially for complex tasks. Existing automated repair methods typically rely on superficial pass/fail signals, offering limited visibility into program behavior and hindering precise error localization. In addition, without a way to learn from prior failures, repair processes often fall into repetitive and inefficient cycles. To overcome these challenges, we present TraceCoder, a collaborative multi-agent framework that emulates the observe-analyze-repair process of human experts. The framework first instruments the code with diagnostic probes to capture fine-grained runtime traces, enabling deep insight into its internal execution. It then conducts causal analysis on these traces to accurately identify the root cause of the failure. This process is further enhanced by a novel Historical Lesson Learning Mechanism (HLLM), which distills insights from prior failed repair attempts to inform subsequent correction strategies and prevent recurrence of similar mistakes. To ensure stable convergence, a Rollback Mechanism enforces that each repair iteration constitutes a strict improvement toward the correct solution. Comprehensive experiments across multiple benchmarks show that TraceCoder achieves up to a 34.43% relative improvement in Pass@1 accuracy over existing advanced baselines. Ablation studies verify the significance of each system component, with the iterative repair process alone contributing a 65.61% relative gain in accuracy. Furthermore, TraceCoder significantly outperforms leading iterative methods in terms of both accuracy and cost-efficiency.

CCS Concepts

• **Software and its engineering → Software creation and management**; *Software verification and validation*; • **Computing methodologies → Multi-agent systems**; *Natural language processing*.

*Corresponding author.



ICSE '26, Rio de Janeiro, Brazil

ACM ISBN 979-8-4007-2025-3/26/04
https://doi.org/10.1145/3744916.3773187

Keywords

Code Generation, Multi-Agent Systems, Self-Debugging, Runtime Tracing, Historical Lesson Learning, Large Language Models

ACM Reference Format:
Jiangping Huang, Wenguang Ye, Weisong Sun, Jian Zhang, Mingyue Zhang, and Yang Liu. 2026. TraceCoder: A Trace-Driven Multi-Agent Framework for Automated Debugging of LLM-Generated Code. In *2026 IEEE/ACM 48th International Conference on Software Engineering (ICSE '26), April 12–18, 2026, Rio de Janeiro, Brazil*. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3744916.3773187

1 Introduction

Large Language Models (LLMs) [3, 32, 36] have become increasingly powerful tools for software engineering tasks such as code generation [8, 13, 30], code summarization [11, 12, 37], and program transformation [44]. Despite their impressive capabilities, LLMs often generate code that contains subtle yet critical bugs—particularly in complex or logic-intensive scenarios [6, 17]. This challenge has given rise to an emerging research direction focused on the automated repair of LLM-generated code, aiming to improve the reliability, correctness, and usability of LLM-assisted development [8, 33].

Recent work in this emerging area has explored diverse strategies for repairing LLM-generated code [50]. These include using Chain-of-Thought (CoT) [43] prompting to guide repair reasoning [48], leveraging natural language as an intermediate representation for debugging [52], integrating statistical fault localization to improve fault awareness [10, 45], proposing end-to-end multi-agent synergy for unified debugging [25], and fine-tuning LLMs for bug fixing [16]. Other approaches refine localization via token-level reasoning [14] or auxiliary fault-identification modules [23].

However, most existing self-correction methods operate as "black-boxes", relying solely on pass/fail feedback from a test suite. This approach, which lacks insight into the program's internal execution, suffers from significant limitations. As illustrated in Figure 1, a simple execution-feedback model can easily fall into a degenerative cycle. In **Round 1**, deprived of detailed runtime information about *why* a test failed, LLMs may make incorrect assumptions and apply faulty patches. This results in previously correct functionality being broken—referred to as **Performance Degradation**. Subsequently, in **Round 2**, the model may get stuck in a loop of incorrect local patches, leading to **Fixation & Stagnation**, where it

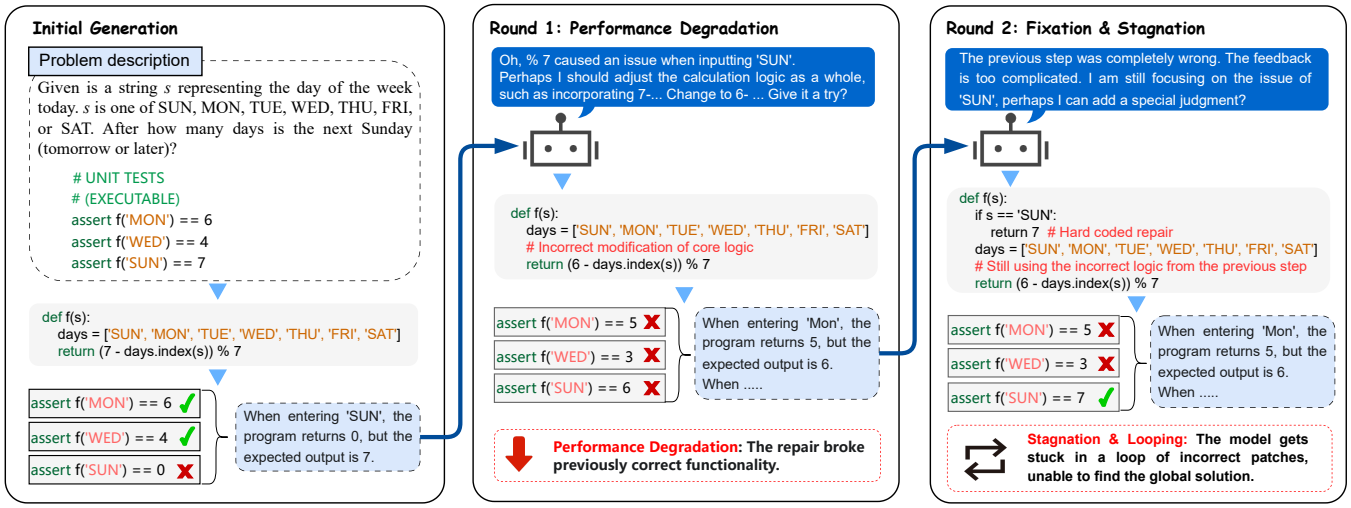


Figure 1: Limitations of simple execution feedback. Without runtime insights, the model repeatedly applies local patches that degrade the code's correctness, causing it to loop between incorrect versions rather than converging to a correct global solution.

fails to diagnose the root cause and find the correct global solution. This example highlights two fundamental limitations in current LLM-based self-debugging methods: 1) they rely on binary final execution results, ignoring the rich semantics in intermediate execution states, which leads to imprecise fault localization; and 2) they adopt a stateless repair paradigm, unable to learn from historical debugging knowledge to avoid repeating past mistakes.

To address these challenges, we propose TraceCoder, a multi-agent collaborative automated debugging framework that emulates the human debugging process. Grounded in established cognitive models of expert debugging, which describe a cycle of *information gathering*, *hypothesis formation*, and *repair* [31], TraceCoder operationalizes this workflow through three specialized agents to enhance modularity, reliability, and control. Specifically, the Instrumentation Agent mirrors the information-gathering stage by capturing fine-grained runtime traces. The Analysis Agent then emulates hypothesis formation, performing causal reasoning on these traces, informed by a novel Historical Lesson Learning Mechanism (HLLM) that learns from past failures. The Repair Agent executes the resulting repair plan to modify the code. To ensure stable convergence, a Rollback Mechanism (RM) reverts the system to its last known correct state after any failed attempt. This structured, human-inspired workflow establishes a cohesive and interpretable debugging loop with a clear separation of concerns.

The proposed multi-agent collaborative automated debugging architecture confers several distinct advantages. First, it enables fine-grained runtime tracing, allowing the system to capture semantically rich execution signals. Second, it facilitates experience-informed repair decisions through historical learning, and ensures a robust repair trajectory via integrated rollback and iterative repair mechanisms. Furthermore, by decomposing the debugging task into logically interpretable stages, TraceCoder enhances modularity, explainability, and extensibility. We evaluate TraceCoder on three widely used datasets, including BigCodeBench [53], ClassEval [9], and HumanEval+ [28], using diverse LLM backends. The results show that TraceCoder consistently outperforms existing automated debugging baselines, including Self-Debugging [5] and INTERVENOR [41], measured by standard metrics such as Pass@1. Notably, TraceCoder improves the Pass@1 accuracy in repairing LLM-generated code, reduces redundant repair attempts, and enhances cost-efficiency, especially on complex programming tasks where LLMs are most prone to failure.

In summary, this paper makes the following contributions:

- We present TraceCoder, a modular collaborative multi-agent framework that emulates the human debugging workflow to enable automated repair of LLM-generated code.
- We propose a novel HLLM that learns from prior failures to inform subsequent repairs and prevent recurring mistakes.
- Comprehensive evaluations demonstrate that TraceCoder substantially outperforms advanced baselines, achieving up to a 34.43% relative improvement in Pass@1 accuracy on challenging class-level code generation benchmarks.
- We release an open-source implementation of TraceCoder [47] to support reproducibility and facilitate future research.

2 Related Work

2.1 Code Generation with LLMs

Recent advancements in LLMs have significantly propelled automated code generation. A growing body of work aims to improve the quality, correctness, and controllability of the generated code. For example, AMR-Evol [29] proposes a two-stage distillation framework to enhance generation fidelity, while ARCHCODE [13] leverages in-context learning to translate software requirements into code and corresponding test cases. Empirical investigations, such as DevGPT [20], reveal that LLM-generated code is often used for prototyping or conceptual illustration, rather than deployment. To support more rigorous evaluation, several benchmarks have been introduced: DA-Code [18] focuses on agent-based workflows, ClassEval [9] targets class-level programs with structural dependencies,

and EvalPlus [28] augments test suites to thoroughly assess functional correctness. On the algorithmic front, PG-TD [51] incorporates planning-guided decoding with lookahead search, while a self-planning framework [19] decomposes intent into subgoals to improve generation reliability. Despite these advances, most work focuses solely on improving generation itself, leaving post-generation debugging comparatively underexplored. LLM-generated code still frequently contains subtle logic errors that existing generation pipelines cannot reliably detect or correct. This gap motivates our work: we introduce a trace-driven, multi-agent framework that leverages runtime evidence and iterative refinement to not only diagnose but also systematically repair LLM-generated code.

2.2 Automated Program Repair

Automated Program Repair (APR) is a long-standing field in software engineering focused on automatically fixing bugs in source code. Traditional APR techniques often rely on search-based methods [24], which use genetic algorithms to evolve patches, or template-based approaches [22], which apply predefined fix patterns. These methods have been extensively evaluated on benchmarks like Defects4J [21], which contains real-world bugs from large-scale Java projects. However, they often struggle with complex logical errors that require a deep semantic reasoning about program behavior.

The advent of LLMs has opened new frontiers for APR. Recent works such as RepairAgent [2] and ThinkRepair [48] leverage LLMs to reason about bugs and synthesize human-like patches, often outperforming traditional methods. These approaches typically adopt an iterative feedback loop, where the model refines the code based on test outcomes. For instance, Self-Debugging [5] uses simple pass/fail signals, while INTERVENOR [41] employs a dual-agent "teacher-learner" framework to guide the repair process. A closely related work, AutoVerus [46], also uses a feedback loop for repairing Rust programs but focuses specifically on resolving formal verification errors, where feedback is structured and precise.

TraceCoder extends this line of work with three key distinctions. First, while most methods rely on black-box test outcomes, TraceCoder leverages fine-grained runtime traces, providing white-box visibility into program behavior for more precise fault localization. Second, unlike AutoVerus, which targets formal verification, TraceCoder is designed for general-purpose code and unstructured test feedback, a common and challenging scenario in practice. Most importantly, our novel HLLM addresses an overlooked gap by enabling the system to learn from past failures, preventing repeated mistakes and improving efficiency in repairing complex bugs.

2.3 LLM-Based Multi-Agent Systems

While monolithic LLMs have achieved strong performance across a variety of tasks, they often struggle in scenarios requiring complex strategic reasoning, iterative refinement, and dynamic adaptation. Limitations such as fixed context windows and unidirectional generation hinder their ability to perform trial-and-error reasoning, reflect on prior work, or plan over long horizons [3, 43]. To overcome these challenges, recent research has turned toward collaborative multi-agent systems (MAS), where multiple role-specialized agents interact to decompose complex problems and simulate human collaborative behavior [15, 42]. Systems such as AgentVerse [4] and OpenDevin [39] demonstrate how agent-based collaboration, when supported by robust communication protocols and task-aligned workflows, can lead to emergent capabilities, achieving results that surpass the performance of individual agents [35]. Advances in multi-agent alignment [27], reflective reasoning mechanisms [1], and agent-oriented planning strategies [26] further enhance reliability across multiple rounds of iterative decision-making. However, most MAS frameworks focus on task decomposition and static role assignment, with limited support for integrating dynamic runtime feedback or leveraging accumulated debugging history, both of which are critical for effective automated program repair. We extend this line of work by introducing a multi-agent architecture that fuses causal planning with collaborative repair to support runtime-aware, history-informed, and self-correcting debugging.

3 Methodology

3.1 Overview

We introduce TraceCoder, a trace-driven automated debugging framework that iteratively repairs LLM-generated code by emulating expert debugging workflows, as illustrated in Figure 2. The framework is organized around three specialized agents: the Instrumentation Agent inserts diagnostic probes to collect fine-grained runtime traces; the Analysis Agent performs causal reasoning over these traces to localize faults; and the Repair Agent synthesizes and applies concrete code modifications. To guide this process, TraceCoder integrates the HLLM to learn from past repair failures, and the RM to ensure stable convergence. When initial code fails its test suite, these agents are activated and iterate until all tests pass or a termination condition is reached.

3.2 Instrumentation Agent

The Instrumentation Agent is a core component of the proposed framework, responsible for collecting dynamic execution information. Its internal *thought process* involves lightweight reasoning over previous execution failures and current instrumentation suggestions to determine suitable probe locations. Based on this reasoning, the agent inserts diagnostic print statements into the code to reveal internal state transitions and control-flow behavior. These runtime insights serve as essential evidence for downstream causal analysis by the Analysis Agent. The Instrumentation Agent is invoked in two scenarios: when the initial execution fails its test suite, or when a subsequent repair attempt does not resolve the issue.

Its inputs are context-dependent, including: (1) the current code (C_{faulty}) to be instrumented, (2) the most recent test failure feedback (F_{error}) from a failed repair attempt, and (3) optionally, a set of fine-grained instrumentation suggestions (I_{sugg}) from the Analysis Agent. By synthesizing this contextual information, the Instrumentation Agent generates a new version of the code augmented with diagnostic probes. The resulting instrumented code (C_{inst}) faithfully preserves the original computational semantics but emits context-aware debug logs during execution, providing valuable insights into the program's dynamic behavior. The core operation of the Instrumentation Agent can be summarized as:

$$(C_{\text{faulty}}, F_{\text{error}}, I_{\text{sugg}}) \rightarrow C_{\text{inst}} \tag{1}$$

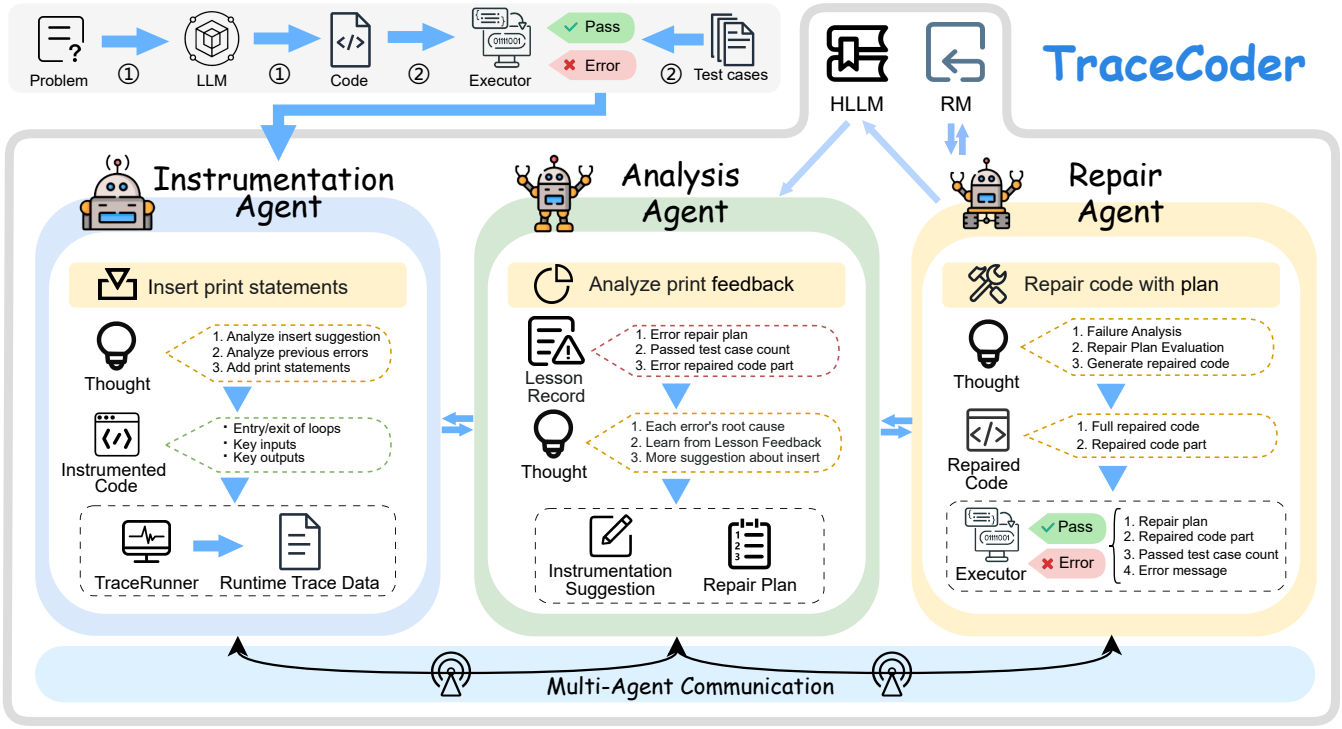


Figure 2: Overview of TraceCoder's workflow. ① An LLM generates an initial code solution. ② The code is executed and tested. A multi-agent debugging loop—comprising the Instrumentation, Analysis, and Repair Agents—emulates expert debugging behaviors by leveraging runtime tracing, HLLM, and RM to enable effective and stable repair. After each failed attempt, the HLLM logs the outcome and informs the Analysis Agent's strategy for the subsequent cycle.

The Instrumentation Agent employs a dedicated prompt that directs the LLM to strategically insert diagnostic probes into the faulty code. The prompt instructs the model to augment Python functions with probes guided by the test failure feedback, revealing both execution flow and key variable states. It explicitly encourages failure-aware instrumentation by prioritizing regions relevant to the observed failure, thereby avoiding indiscriminate logging and enhancing both the efficiency and informational value of runtime traces. To ensure consistency and effectiveness, the agent adheres to four core principles:

- **Logical Decomposition.** The code is decomposed into distinct logical units, such as function bodies, branches, and loops, that define the instrumentation scope.
- **State and Control Traceability.** Each unit is augmented with print statements that log key inputs, outputs, and intermediate values. Entry and exit points of major blocks are also traced to reveal control flow.
- **Instrumentation Purity.** Only non-invasive print statements may be inserted. The agent must not modify computational logic, comment out code, or introduce new variables, thus preserving semantic integrity.
- **Readable and Structured Output.** All logs must follow a clean and structured format to facilitate downstream analysis by both humans and automated tools.

Ensuring semantic preservation is critical for reliable instrumentation. Although formal guarantees of LLM behavior are inherently difficult, we enforce this constraint through strictly designed prompts that prohibit any modifications to the program's logic, control flow, or data structures, allowing only the insertion of 'print' statements. To assess the effectiveness of this enforcement, we conducted an empirical validation study (Section 4.6.1), which demonstrates a semantic preservation rate of over 99%.

After generating the instrumented code, the Instrumentation Agent proceeds to the dynamic execution phase by submitting the code and its associated test suite to *TraceRunner*, a custom execution wrapper. TraceRunner performs controlled execution by isolating runtime environments and capturing both diagnostic logs and exception traces in a structured format. The resulting runtime trace is forwarded to the Analysis Agent as a key diagnostic artifact, enabling it to perform causal reasoning and plan targeted repair actions for the next iteration.

3.3 Analysis Agent

The Analysis Agent diagnoses program failures by integrating runtime traces and historical failure data. It performs context-aware reasoning to identify root causes and derive actionable debugging insights across iterations, producing two complementary outputs: a repair plan for the Repair Agent and targeted instrumentation

suggestions for the next debugging cycle. Formally, it operates on a structured set of inputs:

- **Original Problem Description** (D_{prob}). A textual summary of the program's intended functionality and expected behavior, serving as a semantic reference for diagnosis.
- **Instrumented Code** (C_{inst}). The latest version of the source code augmented with diagnostic probes, generated by the Instrumentation Agent for runtime behavior analysis.
- **Runtime Trace Data** (T_{runtime}). Fine-grained execution logs captured by *TraceRunner*, including diagnostic outputs and all recorded runtime errors or exceptions.
- **Lesson Record** (L_{record}). A structured log of all failed repair attempts for the current problem, used to reflect on prior reasoning and avoid repeated mistakes.

The diagnostic capability of the Analysis Agent is driven by a structured *thought process*, implemented through a prompt-based reasoning schema. This prompt guides the agent to: (1) analyze runtime traces to identify the root cause of the error; (2) incorporate historical insights via the HLLM to avoid repeated mistakes; and (3) generate targeted instrumentation suggestions to guide future data collection. This diagnostic workflow is implemented as a two-stage procedure:

- **Diagnosis and Reflection.** The agent analyzes the Runtime Trace Data to identify the root cause of failure, while concurrently analyzing the Lesson Record to understand why previous attempts were unsuccessful. This reflective process enables the agent to learn from previously flawed reasoning steps, not just surface-level code issues.
- **Strategy Formulation.** Based on its diagnosis, the agent produces two complementary outputs: **(1)** *Repair Plan* (P_{repair}). A proposed code modification plan passed to the Repair Agent for implementation. **(2)** *Instrumentation Suggestion* (I_{sugg}). Targeted guidance for the next debugging cycle, specifying where new probes should be inserted to verify the repair or collect additional runtime signals if it fails.

This procedure is formalized as a function that maps diagnostic inputs to actionable repair and instrumentation outputs:

$$(D_{\text{prob}}, C_{\text{inst}}, T_{\text{runtime}}, L_{\text{record}}) \rightarrow (P_{\text{repair}}, I_{\text{sugg}}) \tag{2}$$

3.4 Repair Agent

The Repair Agent serves as the primary executor in the TraceCoder framework, responsible for translating high-level repair plans into concrete code modifications. Its core function is to implement the changes proposed by the Analysis Agent and participate in the feedback loop by reporting failure cases that guide future refinement. To support this task, the Repair Agent operates on four key inputs:

- **Original Problem Description.** A complete textual specification of the programming task, providing semantic and functional context for the intended behavior.
- **Code to Be Repaired.** The latest non-instrumented version of the source code from the current iteration that failed testing and still requires correction.
- **Test Failure Feedback.** Diagnostic feedback summarizing test case failures, including observed outputs, error messages, and assertion violations during execution.
- **Structured Repair Plan.** A detailed, step-by-step modification strategy generated by the Analysis Agent, explicitly specifying how the faulty code should be corrected.

The core mechanism of the Repair Agent centers on its interaction with LLMs, guided by a carefully designed prompt that embodies its internal *thought process*. This prompt defines the LLM's role, clarifies the repair objectives, and guides it to reason through the task in a systematic and controlled manner. To fulfill its function, the Repair Agent follows a structured three-step workflow:

- **Failure Analysis.** The LLM first performs a detailed analysis of the provided test failure feedback to precisely identify and diagnose the root cause of the error.
- **Repair Plan Evaluation.** It then rigorously evaluates the repair plan proposed by the Analysis Agent to assess whether the suggested modifications are both logically sound and sufficient to address the identified issue.
- **Code Repair Execution.** Finally, the LLM applies the validated repair plan to modify the code. This step requires strict adherence to the specified changes to ensure the defect is correctly resolved without introducing new errors.

To enhance the robustness and success rate of the repair process, the instructional prompt provides the LLM with a controlled degree of flexibility. If, after evaluating the repair plan, the LLM identifies minor omissions that do not compromise the core strategy, it is allowed to make localized, minor code modifications (e.g., correcting a variable name or adjusting a boundary condition) that do not contradict the core strategy of the Analysis Agent's repair plan. This ensures the issue can be effectively resolved while maintaining high fidelity to the Analysis Agent's intent. Formally, the operation of the Repair Agent can be modeled as:

$$(D_{\text{prob}}, C_{\text{faulty}}, F_{\text{error}}, P_{\text{repair}}) \rightarrow C_{\text{repaired}} \tag{3}$$

3.5 Multi-Agent Communication

The communication among TraceCoder's agents follows a structured, sequential pattern, mediated by shared artifacts rather than direct message passing. This design supports a disciplined, cyclical repair process. When a repair cycle begins, the Instrumentation Agent receives the faulty code along with its corresponding test failure feedback. It produces an instrumented version of the code, which is executed in the *TraceRunner* environment. The resulting runtime traces serve as the primary input to the Analysis Agent.

The Analysis Agent integrates these traces with the original problem description and historical repair experiences to diagnose the root cause of failure. It generates two outputs: a detailed repair plan for the Repair Agent, and instrumentation suggestions to guide the Instrumentation Agent in subsequent iterations. This feedback loop enables progressively refined analysis. Upon receiving the repair plan, the Repair Agent applies the specified modifications, and the new candidate solution is re-tested, initiating the next cycle.

This sequential, artifact-mediated communication model ensures that each agent operates with well-structured, contextually relevant information. We opt for this custom control loop over a dedicated

multi-agent system (MAS) framework (e.g., AgentVerse) because our workflow is linear and deterministic. For such a structured process, a full-fledged MAS framework would introduce unnecessary complexity and overhead without providing significant benefits. Our direct approach maintains full control and enhances reproducibility, which is crucial for scientific evaluation.

Importantly, after each failed repair attempt, the results are logged into the HLLM, which distills summarized lessons and provides them back to the Analysis Agent in the subsequent iteration. This mechanism closes the learning loop and enables history-informed, progressively refined debugging.

3.6 Historical Lesson Learning Mechanism

A key component of TraceCoder is the HLLM, which addresses the limitations of stateless repair by enabling the system to learn from past failures. Inspired by the Trial-and-Error Learning Theory [40], HLLM systematically records, retrieves, and reuses failed repair attempts from prior iterations, referred to as *Lesson Feedback*, on the same problem instance. This allows the Analysis Agent to avoid previously ineffective reasoning paths and refine its diagnostic approach across repair cycles. By leveraging these historical insights, HLLM improves debugging efficiency, reduces redundant attempts, and increases correction success rates in complex scenarios. The mechanism operates through the following three stages.

3.6.1 Lesson Record. Each time an iterative repair attempt fails to pass all predefined test cases, the system automatically captures key contextual information. This is recorded as an execution result (E_{result}) and a detailed execution message (E_{message}). If the execution fails, this message contains the specific repair plan that was attempted (P_{repair}), the resulting faulty code (C_{repaired}), the error feedback from the failed execution (F_{error}), and the passed test case count (S_{passed}). These records collectively constitute the Lesson Feedback for the current specific problem.

3.6.2 Lesson Feedback. Before generating a new repair plan, the Analysis Agent prompts the LLM to analyze the Lesson Record, which aggregates all failure records for the current problem instance. From this analysis, the LLM obtains a Lesson Feedback, allowing it to avoid previously ineffective strategies and make more informed repair decisions based on past diagnostic experiences.

3.6.3 Lesson-Informed Deliberation and Planning. The Analysis Agent is explicitly guided, via its structured prompt, to conduct deliberate reasoning over the retrieved Lesson Feedback before generating a new repair plan. This process involves three key tasks:

- Diagnosing the root causes of previous repair failures to understand why earlier strategies were ineffective;
- Summarizing recurring pitfalls or suboptimal repair patterns identified across multiple failed attempts;
- Formulating a revised repair plan that aims to address prior deficiencies and to explore alternative solutions.

To operationalize the synergy across the three stages, Algorithm 1 illustrates the core workflow of the HLLM. As shown, the algorithm processes a structured E_{message} object, which encapsulates the full context of a failed repair attempt. It then extracts failure patterns from this context to synthesize structured lessons, which in turn guide the formulation of subsequent repair strategies.

Algorithm 1 Historical Lesson Learning Mechanism

Require: Result of code execution, E_{result}
Message of code execution, E_{message}
The historical lesson record, $L_{\text{record\_in}}$
Ensure: The updated lesson record, $L_{\text{record\_out}}$

$L_{\text{record\_updated}} \leftarrow L_{\text{record\_in}}$
if E_{result} is a failure **then**
  Extract P_{repair} from E_{message}
  Extract F_{error} from E_{message}
  Extract C_{repaired} from E_{message}
  Extract S_{passed} from E_{message}
  new_record $\leftarrow (P_{\text{repair}}, F_{\text{error}}, C_{\text{repaired}}, S_{\text{passed}})$
  Add new_record to $L_{\text{record\_updated}}$
end if
$L_{\text{record\_out}} \leftarrow L_{\text{record\_updated}}$
return $L_{\text{record\_out}}$

3.7 Rollback Mechanism

To maintain both convergence and robustness throughout the iterative self-debugging process, TraceCoder incorporates the RM. This mechanism serves as a critical strategy for state management and recovery, designed to revert the system to a previously validated or superior state whenever a new repair attempt fails to yield progress or introduces regressions. By doing so, RM prevents the repair trajectory from deteriorating across iterations and anchors the search process around the best-known solutions. The operation of the RM is structured around the following two core procedures.

3.7.1 Key State Recording. Throughout the debugging process, TraceCoder continuously tracks several key state variables to inform its decisions. It maintains a record of the historically best-performing code so far (C_{best}), defined as the version that has passed the greatest number of test cases. Its corresponding score, the highest passed test count so far (S_{best}), is stored as the primary performance benchmark. To detect significant regressions, TraceCoder also records the passed test count of the previous attempt (S_{previous}). Finally, a stagnation counter (k) is maintained to track the number of consecutive attempts that have failed to improve the best score.

3.7.2 Progress Evaluation and Decision-Making. Each time the Repair Agent generates a new repair candidate ($C_{\text{attempted}}$), TraceCoder evaluates its performance by obtaining its passed test count ($S_{\text{attempted}}$). This score is then compared against the historically best score (S_{best}) and the previous attempt's score (S_{previous}) to make a decision, as formalized in Algorithm 2. The algorithm returns an updated state tuple $\langle C_{\text{best\_new}}, S_{\text{best\_new}}, C_{\text{next\_base}}, k_{\text{new}} \rangle$, where $C_{\text{next\_base}}$ specifies the baseline code (either $C_{\text{attempted}}$ or C_{best}) for the subsequent repair cycle. The decision process operates through three distinct scenarios.

- **Improvement.** If the new code passes more test cases than the current best, it is promoted as the new historically best, and the next iteration proceeds from it. The counter for non-improving attempts is reset.

- **Stagnation or Regression.** If the new code shows no improvement or performs worse, the system reverts to the previously recorded best version for the next attempt.
- **Prolonged Stagnation.** If no progress occurs after several iterations, the repair process is terminated to prevent wasted computation and potential overfitting.

Building on the two core processes above, the decision logic of the RM is formalized in Algorithm 2. It performs precise state management by quantitatively comparing the performance of the new candidate against the historically best version, while monitoring for repeated setbacks. This ensures that the repair process consistently progresses toward an optimal solution.

Algorithm 2 Rollback Mechanism

Require: The latest attempted code, $C_{\text{attempted}}$
The best-performing code so far, C_{best}
The highest passed test count so far, S_{best}
The passed test count of the previous attempt, S_{previous}
The stagnation counter, k
Ensure: The decision: $\langle \text{accept, continue, rollback} \rangle$
The updated state: $\langle C_{\text{best\_new}}, S_{\text{best\_new}}, C_{\text{next\_base}}, k_{\text{new}} \rangle$

Let $S_{\text{attempted}}$ be the number of tests passed by $C_{\text{attempted}}$
$\Delta \leftarrow S_{\text{attempted}} - S_{\text{best}}$
if $\Delta > 0$ **then**
  $C_{\text{next\_base}} \leftarrow C_{\text{attempted}}$ // Promote new code
  return $\langle \text{accept}, \langle C_{\text{attempted}}, S_{\text{attempted}}, C_{\text{next\_base}}, 0 \rangle \rangle$
else if $\Delta = 0$ **then**
  $C_{\text{next\_base}} \leftarrow C_{\text{attempted}}$ // Continue with current
  return $\langle \text{continue}, \langle C_{\text{best}}, S_{\text{best}}, C_{\text{next\_base}}, k+1 \rangle \rangle$
else
  if $S_{\text{attempted}} < S_{\text{previous}}$ **then**
    $C_{\text{next\_base}} \leftarrow C_{\text{best}}$ // Trigger Rollback
    return $\langle \text{rollback}, \langle C_{\text{best}}, S_{\text{best}}, C_{\text{next\_base}}, k+1 \rangle \rangle$
  else
    $C_{\text{next\_base}} \leftarrow C_{\text{attempted}}$ // Stagnation, keep trying
    return $\langle \text{continue}, \langle C_{\text{best}}, S_{\text{best}}, C_{\text{next\_base}}, k+1 \rangle \rangle$
  end if
end if

4 Evaluation

This section presents four research questions addressed by TraceCoder and details the experimental setup, including datasets, baselines, evaluation metrics, and implementation details. We conduct extensive experiments to answer these questions and provide a comprehensive evaluation of TraceCoder's effectiveness.

4.1 Research Questions

RQ1: How effective is TraceCoder at repairing LLM-generated code compared to advanced automated repair methods?
RQ2: How do TraceCoder's key hyperparameters affect its repair performance and stability?
RQ3: What is the contribution of each core component to TraceCoder's overall effectiveness?
RQ4: How does TraceCoder perform in practice, particularly compared to sampling-based strategies, in terms of reliability, cost efficiency, and failure modes?

4.2 Experimental Setup

4.2.1 Datasets. We conduct a comprehensive evaluation of TraceCoder on four widely adopted benchmark datasets: *HumanEval* [3], *HumanEval+* [28], *BigCodeBench* [53], and *ClassEval* [9]. The tasks in *HumanEval*, *HumanEval+*, and *BigCodeBench* are at the function level, where the goal is to generate a single correct Python function. *HumanEval+* extends *HumanEval* with broader and more robust test coverage. *BigCodeBench* offers a diverse set of realistic function-level tasks emphasizing complex instruction following. By contrast, *ClassEval* targets class-level code generation, evaluating LLMs on object-oriented constructs such as inter-method dependencies and class hierarchies. To mitigate test leakage, we ensured that *BigCodeBench* was released after the knowledge cut-off date of the evaluated LLMs, avoiding prior exposure during training.

4.2.2 Baselines. To validate its effectiveness, we compare TraceCoder with five representative baseline methods that reflect different paradigms in LLM-based code generation and repair.

- **Direct** [3]: Generates code directly from the problem description without additional reasoning or planning.
- **CoT** [43]: Encourages the LLM to produce intermediate reasoning steps before code generation.
- **Self-Planning** [19]: Decomposes complex problems into a series of subgoals, improving task structure and clarity.
- **Self-Debugging** [5]: Executes the generated code and uses its execution results to iteratively refine the solution.
- **INTERVENOR [41]**: Improves repair accuracy by alternating LLM roles as learner and teacher, simulating human-like interactions to produce a Chain-of-Repair.

4.2.3 Metrics. We evaluate functional correctness using the standard *Pass@K* metric [49], which measures the percentage of problems for which at least one of k generated solutions passes all benchmark test cases. In our experiments, we adopt the greedy **Pass@1** setting ($k = 1$), where only a single solution is evaluated per problem. This stricter metric better reflects real-world development scenarios, as developers typically do not sample multiple candidates but rely on the top-1 generated solution.

4.2.4 Implementation details. We evaluate TraceCoder using three representative LLMs: *Gemini-2.5-flash-0417* [38], *DeepSeek-V3-0324* [7], and *Qwen-Plus-2025-01-25* [34]. For any given run, a single LLM is used consistently across all agents and the initial code generation module. To ensure result reproducibility, we adopt default deterministic API configurations when invoking the models.

For a fair comparison, all baseline methods are implemented using the same base LLMs and API parameters as TraceCoder. We reproduce each baseline based on its official implementation or prompt design. During initial code generation, all methods, including TraceCoder, generate code based solely on the natural language problem description; the test suite is used exclusively for verification and feedback in subsequent repair stages. We limit all iterative methods, including TraceCoder and the iterative baselines, to a

maximum of 5 repair attempts. All generated code is executed and evaluated in a unified Python 3.10 environment.

4.3 Performance Evaluation (RQ1)

To address RQ1, we conduct a comprehensive evaluation of TraceCoder against several baseline methods across three foundation models and four benchmark datasets, with results summarized in Table 1. TraceCoder consistently achieves the highest Pass@1 accuracy, outperforming both non-iterative methods (e.g., `Direct` and `CoT`) and iterative ones (e.g., `Self-Debugging` and `INTERVENOR`). While latter incorporate feedback loops, they are limited by single-path repair strategies and lack mechanisms for learning from prior failures, often resulting in convergence to suboptimal solutions.

TraceCoder overcomes these limitations through multi-agent collaboration and runtime trace-driven analysis, enabling more precise and adaptive repairs. This advantage is especially evident on structurally complex benchmarks such as ClassEval and BigCodeBench. For example, using Gemini-2.5-Flash-0417 on ClassEval, TraceCoder achieves a Pass@1 score of 82.00%, surpassing the second-best baseline (61.00%) by a relative margin of 34.43%. A similar trend is observed on BigCodeBench. When averaged across all benchmarks, TraceCoder reaches 90.72% with Gemini, outperforming the strongest baseline (81.05%) by 11.93%, demonstrating robust performance across diverse programming tasks.

These results provide strong empirical evidence for the effectiveness of TraceCoder's framework design. By combining runtime instrumentation and historical lesson learning, TraceCoder enables the Analysis Agent to identify root causes rather than surface-level symptoms, leading to more accurate and targeted repairs, particularly for complex bugs that are difficult to resolve.

> **Answer to RQ1:** TraceCoder consistently outperforms baseline methods across all settings. Its advantage is particularly notable on complex benchmarks such as ClassEval and BigCodeBench, achieving a relative improvement of up to 34.43% over the strongest baselines.

4.4 Impact of Framework Parameters (RQ2)

We analyze RQ2 by examining the impact of two key hyperparameters: *max_attempts*, which defines the upper limit of repair iterations, and *patience*, which controls the early stopping threshold. Sensitivity analysis is conducted on BigCodeBench-Complete using Gemini-2.5-Flash-0417, with results shown in Figure 3.

Effect of max_attempts. We observe a consistent increase in Pass@1 accuracy as *max_attempts* increases, confirming the benefit of iterative refinement in TraceCoder's design. Allowing more repair cycles offers additional opportunities for diagnosis and repair.

Effect of patience. Similarly, increasing *patience* yields steady performance gains. A higher tolerance enables the framework to perform multi-step repairs and better utilize the LLM's stochastic exploration, especially when early attempts fail. Peak performance is reached at the highest tested *patience* level, suggesting that greater fault tolerance enhances effectiveness on complex tasks. However, this improvement comes at the cost of increased computational overhead, indicating a trade-off between accuracy and efficiency.

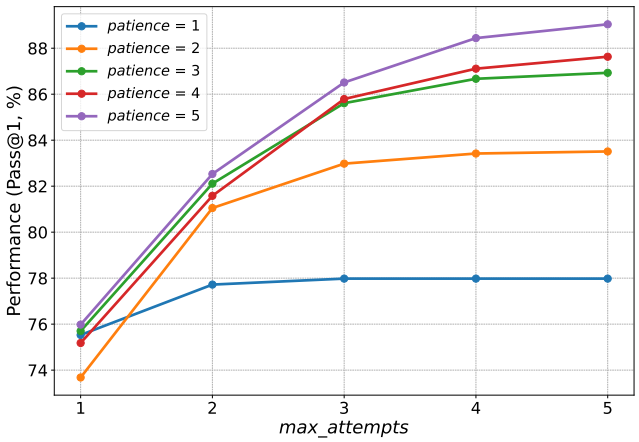


Figure 3: Sensitivity analysis of *max_attempts* and *patience* on Pass@1 performance. Results are reported on BigCodeBench-Complete using Gemini-2.5-Flash-0417.

> **Answer to RQ2:** TraceCoder's performance is highly sensitive to both *max_attempts* and *patience*. Increasing either improves accuracy by enabling the framework to escape local optima. For complex tasks, greater *patience* enables deeper exploration, leading to more successful repairs.

4.5 Ablation Study (RQ3)

To evaluate the individual contributions of TraceCoder's core components, we conduct an ablation study on the BigCodeBench-Complete dataset. The results are detailed in Table 2. We systematically disable key parts of the framework to assess their impact.

- **w/o Instrumentation**. Removes the Instrumentation Agent. The Analysis Agent must operate without runtime traces.
- **w/o Instrumentation & Analysis**. Excludes both Instrumentation and Analysis Agents, leaving the Repair Agent to act solely on raw test failure feedback.
- **w/o Iterative Repair**. Disables the entire agent-based repair framework, reverting the system to single-pass code generation without iterative reasoning.
- **w/o HLLM**. Deactivates HLLM, which eliminates the framework's ability to incorporate lessons from prior failures.
- **w/o RM**. Removes the RM, requiring the framework to proceed from the latest attempt regardless of its performance.
- **w/o HLLM & RM**. Disables both HLLM and RM to evaluate their joint contribution to overall performance.

The ablation results confirm that the full TraceCoder framework achieves the highest accuracy, with all components contributing positively. The most substantial degradation occurs when the entire iterative repair loop is eliminated (**w/o Iterative Repair**), resulting in a sharp accuracy drop from 89.04% to 53.77%.

Gradual removal of agents highlights their individual significance. Excluding the Instrumentation Agent alone causes a marked decline, underscoring the importance of runtime traces for precise diagnosis. Further excluding the Analysis Agent leads to an

Table 1: Comparison of Pass@1 (%) between TraceCoder and baseline methods across four benchmarks and three foundation models. "Ours" denotes the proposed TraceCoder. Bold numbers indicate the best performance in each column; values in parentheses denote the relative improvement (↑%) over the second-best result.

Models	Methods	HumanEval	HumanEval+	ClassEval	BigCodeBench-Complete	BigCodeBench-Instruct	Average
Gemini-2.5-Flash -0417	Direct	96.34	91.46	38.00	53.77	43.77	64.67
	CoT	93.90	91.46	41.00	53.86	43.68	64.78
	Self-Planning	94.51	90.85	36.00	55.61	43.15	64.02
	Self-Debugging	98.78	96.34	61.00	78.07	71.05	81.05
	INTERVENOR	**99.39**	95.12	61.00	75.88	69.82	80.24
	Ours	**99.39**	**98.17 (↑ 1.90%)**	**82.00 (↑ 34.43%)**	**89.04 (↑ 14.05%)**	**85.00 (↑ 19.63%)**	**90.72 (↑ 11.93%)**
DeepSeek-V3 -0324	Direct	94.51	90.24	41.00	38.25	46.67	62.13
	CoT	93.29	88.41	41.00	60.35	47.98	66.21
	Self-Planning	95.12	90.24	37.00	61.14	26.93	62.09
	Self-Debugging	**98.78**	**96.34**	61.00	82.37	74.56	82.61
	INTERVENOR	95.73	92.68	63.00	79.82	70.79	80.40
	Ours	**98.78**	**96.34**	**78.00 (↑ 23.81%)**	**88.33 (↑ 7.24%)**	**83.77 (↑ 12.35%)**	**89.04 (↑ 7.78%)**
Qwen-Plus -2025-01-25	Direct	90.85	86.59	31.00	50.09	41.49	60.00
	CoT	93.29	87.19	33.00	48.07	43.50	61.01
	Self-Planning	90.85	84.75	37.00	37.36	41.75	58.34
	Self-Debugging	**96.34**	**93.90**	49.00	70.96	63.77	74.80
	INTERVENOR	95.12	91.46	48.00	68.60	61.75	72.99
	Ours	**96.34**	**93.90**	**63.00 (↑ 28.57%)**	**71.93 (↑ 1.37%)**	**68.60 (↑ 7.57%)**	**78.75 (↑ 5.28%)**

Table 2: Ablation study of TraceCoder's components on the BigCodeBench-Complete dataset. The table shows the Pass@1 performance (%) after removing specific components.

Configuration	BigCodeBench-Complete
TraceCoder	**89.04**
w/o Instrumentation	78.51
w/o Instrumentation & Analysis	75.09
w/o Iterative Repair	53.77
w/o HLLM	86.75
w/o RM	84.55
w/o HLLM & RM	84.43

additional 4.36% drop, validating its essential role in formulating structured repair plans even in the absence of trace data.

The supporting mechanisms also play complementary roles. Disabling the RM results in a 5.31% relative decline, validating its effectiveness in preserving promising states and preventing convergence deterioration. Meanwhile, the HLLM contributes a 2.64% improvement by avoiding repeated failures and accelerating repair convergence. Removing both RM and HLLM confirms that their impacts are largely independent yet synergistic.

Overall, these findings align with the debugging strategies employed by experienced developers: addressing complex bugs requires not only repeated attempts, but also systematic analysis of runtime behavior (enabled by the Instrumentation and Analysis Agents), retention of prior failures (facilitated by HLLM), and disciplined rollback to stable baselines (enforced by RM).

Answer to RQ3: The ablation study confirms that all components of TraceCoder are essential for optimal performance. The foundational iterative repair framework provides the largest improvement, while the RM and HLLM contribute vital optimizations, adding 5.31% and 2.64% respectively for stability and learning efficiency.

4.6 Practical Evaluation: Reliability, Efficiency, Cost, and Failure Analysis (RQ4)

To assess TraceCoder's real-world applicability and understand its underlying behavior, we conduct a multi-faceted operational analysis. Specifically, we examine the reliability of its core instrumentation mechanism, measure computational cost, compare its efficiency against sampling-based strategies, diagnose its dominant failure modes, and illustrate its repair process on a challenging bug.

4.6.1 Validation of Instrumentation Purity. A core assumption of TraceCoder is that the Instrumentation Agent can insert diagnostic probes without altering the program's original semantics. To empirically validate this assumption, we conduct a targeted study as follows: (1) we collect all initially correct samples generated by the `Direct` baseline on HumanEval and ClassEval; (2) these correct samples are instrumented using our Instrumentation Agent; and (3) the full test suites are re-executed on the instrumented versions to check whether correctness was preserved.

As shown in Table 3, TraceCoder achieves a semantic preservation rate of 99.32%–100% across both evaluated LLMs, confirming the reliability of our instrumentation strategy. This provides strong empirical evidence that our prompt-guided instrumentation process is highly reliable and rarely introduces new bugs. To further corroborate these findings, we also employ formal verification using CrossHair's `diffbehavior` tool[1]. This analysis confirms that the

[1]https://github.com/pschanely/CrossHair

Table 3: Empirical validation of semantic preservation. The preservation rate measures the percentage of initially correct programs that remain correct after instrumentation.

Model	Dataset	Initial Correct Samples	Correct After Instrumentation	Preservation Rate
Gemini-2.5-Flash	HumanEval	161	161	100.0%
	ClassEval	44	44	100.0%
Qwen-Plus	HumanEval	147	146	99.32%
	ClassEval	30	30	100.0%

Table 4: Token consumption comparison of different methods on Gemini-2.5-Flash-0417. Values indicate the average number of tokens used per problem.

Method	HumanEval+		ClassEval	
	Input tokens	Output tokens	Input tokens	Output tokens
Direct	168.35	401.59	638.46	2,264.56
CoT	176.70	489.14	679.80	2,249.65
Self-Planning	584.98	544.25	1,892.09	2,363.0
Self-Debugging	246.67	945.26	25,906.64	**26,744.15**
INTERVENOR	**921.87**	919.23	22,785.2	15,182.29
Ours	900.45	**970.60**	**29,771.90**	16,264.34

instrumented code is behaviorally equivalent to the original in over 97% of cases, providing strong formal evidence for the semantic integrity of our instrumentation process. This validation further suggests that TraceCoder's prompt-engineering strategy can be generalized to other programming languages and LLMs.

4.6.2 Cost Analysis. We first evaluate the computational cost of TraceCoder and all baselines on the *Gemini-2.5-Flash-0417*, measuring the average token usage per problem. As shown in Table 4, non-iterative methods like `Direct` and `CoT` consume the fewest tokens, but this comes at the cost of lower performance on complex tasks. Iterative baselines demonstrate improved accuracy but incur substantially higher token usage, establishing a clear trade-off between cost and effectiveness. TraceCoder operates in this high-cost, high-performance regime, leveraging its budget for targeted, intelligent repair rather than unguided attempts.

4.6.3 Efficiency Comparison with Sampling. To ensure a fair comparison against breadth-based search (sampling), we conduct additional experiments under equal token and attempt budgets. As detailed in Table 5 and Table 6, when baselines like `CoT` are allowed to generate multiple samples to match TraceCoder's total attempts or token usage, TraceCoder's Pass@1 still significantly outperforms their Pass@k on complex tasks like ClassEval. This demonstrates that TraceCoder's feedback-guided, depth-first repair strategy is more effective than blind sampling for resolving deep logical flaws, as it uses feedback to intelligently navigate the solution space rather than relying on chance.

4.6.4 Error Analysis. To gain deeper insights into TraceCoder's remaining failure modes, we analyze the types of errors it encounters on the BigCodeBench dataset. As shown in Table 7, outcomes are categorized into Pass, Runtime Error (RE), Wrong Answer (WA), and Time Limit Exceeded (TLE). The data confirms that **WA** is the dominant failure mode, reaching up to 8.86%. This indicates that

Table 5: Performance Comparison under Equal Attempt Setting (Pass@6 for baselines vs. Pass@1 for TraceCoder).

Model	Method	HumanEval	HumanEval+	ClassEval
Gemini-2.5-Flash	Direct (Pass@6)	98.78%	96.34%	54.00%
	CoT (Pass@6)	98.78%	96.95%	55.00%
	Ours (Pass@1)	**99.39%**	**98.17%**	**82.00%**
Qwen-Plus	Direct (Pass@6)	94.51%	90.24%	39.00%
	CoT (Pass@6)	96.34%	93.29%	41.00%
	Ours (Pass@1)	**96.34%**	**93.90%**	**63.00%**

Table 6: Performance Comparison under Equal Token Budget Setting (Pass@k for baselines vs. Pass@1 for TraceCoder).

Model	Method	HumanEval+	ClassEval
Gemini-2.5-Flash	Direct	93.90% (Pass@3)	58.00% (Pass@15)
	CoT	92.68% (Pass@3)	58.00% (Pass@15)
	Ours	**98.17% (Pass@1)**	**82.00% (Pass@1)**
Qwen-Plus	Direct	88.41% (Pass@3)	46.00% (Pass@15)
	CoT	93.29% (Pass@3)	53.00% (Pass@15)
	Ours	**93.90% (Pass@1)**	**63.00% (Pass@1)**

while our trace-driven framework is effective at resolving explicit runtime errors, the remaining challenge lies in correcting subtle logical defects that produce incorrect outputs without crashing.

Table 7: Error analysis on BigCodeBench subsets.

Metric	BigCodeBench-Complete	BigCodeBench-Instruct
Pass	89.04%	85.00%
Runtime Error (RE)	4.23%	5.00%
Wrong Answer (WA)	6.34%	8.86%
Time Limit Exceeded (TLE)	0.39%	1.14%

4.6.5 Case Study: Resolving a Semantic Bug. Given that WA errors are the main challenge, this case study illustrates how TraceCoder is uniquely equipped to diagnose and fix such semantic issues.

Task. Write a function `get_positives(numbers)` that returns a list of strictly positive numbers from the input list.

Incorrect LLM-Generated Code. The initial code has a common off-by-one logical error, incorrectly including zero in the output.

```
def get_positives(numbers):
    return [x for x in numbers if x >= 0]  # Bug:
        Should be > 0
```

Test Case and Baseline's Dilemma. The bug is exposed by the test case `assert get_positives([0, 1, -1]) == [1]`. The buggy code returns "[0, 1]", causing an `AssertionError: [0, 1] != [1]`. For a black-box method that only sees this final error message, the root cause is ambiguous. It might try various incorrect fixes, such as changing the list order or modifying the numbers, without understanding that the core issue lies in the filtering logic itself.

TraceCoder's Resolution Process. To showcase TraceCoder's semantic debugging capability, we illustrate how it resolves a subtle logical error that standard sampling-based methods fail to detect.

(a) **Smart Instrumentation**: Upon detecting an assertion error, the Instrumentation Agent inserts diagnostic probes into the list comprehension's filtering logic. These probes are purposefully placed to track the evaluation of each element *without altering the program's semantics*, providing fine-grained visibility into decision-making behavior.

(b) **Execution & Tracing**: Running the test case with the instrumented code yields a fine-grained runtime trace.

```
DEBUG: Checking num=0. Condition 0 >= 0
    is True. Appending.
DEBUG: Checking num=1. Condition 1 >= 0
    is True. Appending.
DEBUG: Checking num=-1. Condition -1 >= 0
     is False. Skipping.
```

(c) **Analysis & Localization**: The Analysis Agent inspects the trace and observes that `0` satisfies the predicate and is appended to the list. By aligning this observation with the task specification ("strictly positive numbers"), it infers the root cause: the condition `x >= 0` is semantically incorrect, since `0` is not a positive value.

(d) **Targeted Repair**: Based on this precise analysis, the Repair Agent receives an unambiguous plan: the filtering predicate for positive numbers is incorrect. The condition should be `x > 0` rather than `x >= 0`. The agent then applies this fix and generates the correct patched code.

This case study demonstrates how TraceCoder leverages internal execution visibility to diagnose and fix semantic bugs that are often opaque to methods relying solely on pass/fail signals.

Answer to RQ4: TraceCoder achieves a strong cost–performance trade-off, outperforming baselines even under equal token and attempt budgets. Its guided repair strategy is significantly more efficient than unguided sampling, and our analysis shows that WA remains the principal failure mode. The case study further demonstrates TraceCoder's ability to resolve these subtle logical errors through fine-grained runtime tracing.

5 Threats to Validity

In this section, we assess potential threats to validity to ensure a rigorous and balanced interpretation of the results.

External Validity. A primary threat to external validity lies in the high computational cost of TraceCoder. Its detailed, trace-driven framework, while effective, incurs substantial token usage. Although our experiments confirm a superior cost-performance trade-off, its absolute token consumption could hinder its adoption in resource-constrained environments. Designing more lightweight, token-efficient agents is a crucial direction for future work.

Construct Validity. Our evaluation relies on the provided test suites to measure program correctness. TraceCoder's ability to repair bugs is fundamentally constrained by test coverage; inadequate tests may lead to overfitting, where hidden flaws remain unaddressed. Consequently, passing all tests does not guarantee true program correctness, and complementary validation methods may be required to mitigate this threat.

Internal Validity. A potential threat to internal validity is data contamination, where evaluation benchmarks may have been included in the LLMs' pre-training data. We mitigated this risk by using the BigCodeBench dataset, which was released after the knowledge cut-off dates of our selected models. While this substantially reduces the likelihood of direct leakage, indirect contamination cannot be entirely ruled out. Nevertheless, all methods were subject to the same experimental conditions, ensuring a fair comparison.

6 Conclusion

This paper introduces *TraceCoder*, a trace-driven multi-agent framework that emulates expert debugging behavior to automatically repair LLM-generated code. Through the integration of runtime instrumentation, coordinated agent collaboration, and iterative refinement, TraceCoder enables precise fault localization and targeted correction. Its HLLM prevents redundant failures by reusing past insights, while the RM stabilizes progress across iterations. Extensive evaluations confirm substantial improvements over state-of-the-art baselines, particularly on complex programming tasks.

Future work will focus on improving token efficiency and extending TraceCoder's capabilities. A key direction is scaling the framework to repository-level debugging, which raises challenges such as managing large contexts, mitigating instrumentation explosion, and handling cross-file dependencies. To address these, we plan to explore solutions such as static analysis for coarse-grained localization and retrieval-augmented generation to construct minimal execution contexts for repair. Furthermore, inspired by the trade-off between depth- and breadth-first exploration, we will investigate hybrid strategies that combine initial sampling with parallel TraceCoder instances to efficiently explore a broader solution space. Another promising direction is enhancing the HLLM with structured knowledge representations, enabling agents to generalize repair strategies across different tasks and domains.

Acknowledgments

This work was supported by the Science and Technology Research Program of the Chongqing Municipal Education Commission (Grant No. KJQN202500631), the Humanities and Social Sciences Fund of the Ministry of Education (Grant No. 20YJCZH047), the National Research Foundation Singapore and DSO National Laboratories under the AI Singapore Programme (AISG Award No. AISG4-GC-2023-008-1B), the Fundamental Research Funds for the Central Universities (Grant No. KG16426001), the National Natural Science Foundation of China (Grant No. 62402400), and the National Key Research and Development Program of China (Grant No. 2024YFF0908000).

References

[1] Xiaohe Bo, Zeyu Zhang, Quanyu Dai, Xueyang Feng, Lei Wang, Rui Li, Xu Chen, and Ji-Rong Wen. 2024. Reflective Multi-Agent Collaboration based on Large Language Models. In *Advances in Neural Information Processing Systems*, A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang (Eds.), Vol. 37. Curran Associates, Inc., 138595–138631. doi:10.52202/079017-4397

[2] Islem Bouzenia, Premkumar Devanbu, and Michael Pradel. 2025. RepairAgent: An Autonomous, LLM-Based Agent for Program Repair. In *Proceedings of the IEEE/ACM 47th International Conference on Software Engineering* (Ottawa, Ontario, Canada) *(ICSE '25)*. IEEE Press, 2188–2200. doi:10.1109/ICSE55347.2025.00157

[3] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating Large Language Models Trained on Code. arXiv:2107.03374 [cs.LG] https://arxiv.org/abs/2107.03374

[4] Weize Chen, Yusheng Su, Jingwei Zuo, Cheng Yang, Chenfei Yuan, Chi-Min Chan, Heyang Yu, Yaxi Lu, Yi-Hsin Hung, Chen Qian, Yujia Qin, Xin Cong, Ruobing Xie, Zhiyuan Liu, Maosong Sun, and Jie Zhou. 2024. AgentVerse: Facilitating Multi-Agent Collaboration and Exploring Emergent Behaviors. In *International Conference on Representation Learning*, B. Kim, Y. Yue, S. Chaudhuri, K. Fragkiadaki, M. Khan, and Y. Sun (Eds.), Vol. 2024. 20094–20136. https://proceedings.iclr.cc/paper_files/paper/2024/file/578e65cdee35d00c708d4c64bce32971-Paper-Conference.pdf

[5] Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. 2024. Teaching Large Language Models to Self-Debug. In *The Twelfth International Conference on Learning Representations*. https://openreview.net/forum?id=KuPixIqPiq

[6] Zhi Chen and Lingxiao Jiang. 2025. Evaluating Software Development Agents: Patch Patterns, Code Quality, and Issue Complexity in Real-World GitHub Scenarios. In *Proceedings of the 30th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 657–668. doi:10.1109/SANER64311.2025.00068

[7] DeepSeek-AI, Aixin Liu, Bei Feng, Bing Xue, and Bingxuan Wang et al. 2025. DeepSeek-V3 Technical Report. arXiv:2412.19437 [cs.CL] https://arxiv.org/abs/2412.19437

[8] Yihong Dong, Xue Jiang, Zhi Jin, and Ge Li. 2024. Self-Collaboration Code Generation via ChatGPT. *ACM Transactions on Software Engineering and Methodology* 33, 7 (Sept. 2024), 1–38. doi:10.1145/3672459

[9] Xueying Du, Mingwei Liu, Kaixin Wang, Hanlin Wang, Junwei Liu, Yixuan Chen, Jiayi Feng, Chaofeng Sha, Xin Peng, and Yiling Lou. 2024. Evaluating Large Language Models in Class-Level Code Generation. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. ACM, Lisbon Portugal, 1–13. doi:10.1145/3597503.3639219

[10] Zhiyu Fan, Xiang Gao, Martin Mirchev, Abhik Roychoudhury, and Shin Hwei Tan. 2023. Automated Repair of Programs from Large Language Models. In *Proceedings of the 45th IEEE/ACM International Conference on Software Engineering (ICSE)*. IEEE, Melbourne, Australia, 1469–1481. doi:10.1109/ICSE48619.2023.00128

[11] Jian Gu, Pasquale Salza, and Harald C. Gall. 2022. Assemble Foundation Models for Automatic Code Summarization. In *Proceedings of the 29th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 935–946. doi:10.1109/SANER53432.2022.00112

[12] Rajarshi Haldar and Julia Hockenmaier. 2024. Analyzing the Performance of Large Language Models on Code Summarization. In *Proceedings of the 2024 Joint International Conference on Computational Linguistics, Language Resources and Evaluation (LREC-COLING 2024)*, Nicoletta Calzolari, Min-Yen Kan, Veronique Hoste, Alessandro Lenci, Sakriani Sakti, and Nianwen Xue (Eds.). ELRA and ICCL, Torino, Italy, 995–1008. https://aclanthology.org/2024.lrec-main.89

[13] Hojae Han, Jaejin Kim, Jaeseok Yoo, Youngwon Lee, and Seung-won Hwang. 2024. ArchCode: Incorporating Software Requirements in Code Generation with Large Language Models. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, Lun-Wei Ku, Andre Martins, and Vivek Srikumar (Eds.). Association for Computational Linguistics, Bangkok, Thailand, 13520–13552. https://aclanthology.org/2024.acl-long.730

[14] Soneya Binta Hossain, Nan Jiang, Qiang Zhou, Xiaopeng Li, Wen-Hao Chiang, Yingjun Lyu, Hoan Nguyen, and Omer Tripp. 2024. A Deep Dive into Large Language Models for Automated Bug Localization and Repair. *Proceedings of the ACM on Software Engineering* 1, FSE (July 2024), 1471–1493. doi:10.1145/3660773

[15] Jiangping Huang, Dongming Jin, Weisong Sun, Yang Liu, and Zhi Jin. 2025. Envisioning Intelligent Requirements Engineering via Knowledge-Guided Multi-Agent Collaboration. In *Proceedings of the 40th IEEE/ACM International Conference on Automated Software Engineering (ASE) (ASE '25)*.

[16] Kai Huang, Xiangxin Meng, Jian Zhang, Yang Liu, Wenjie Wang, Shuhao Li, and Yuqing Zhang. 2023. An Empirical Study on Fine-Tuning Large Language Models of Code for Automated Program Repair. In *Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, Luxembourg, Luxembourg, 1162–1174. doi:10.1109/ASE56229.2023.00181

[17] Kai Huang, Jian Zhang, Xinlei Bao, Xu Wang, and Yang Liu. 2025. Comprehensive Fine-Tuning Large Language Models of Code for Automated Program Repair. *IEEE Transactions on Software Engineering* 51, 4 (2025), 904–928. doi:10.1109/TSE.2025.3532759

[18] Yiming Huang, Jianwen Luo, Yan Yu, Yitong Zhang, Fangyu Lei, Yifan Wei, Shizhu He, Lifu Huang, Xiao Liu, Jun Zhao, and Kang Liu. 2024. DA-Code: Agent Data Science Code Generation Benchmark for Large Language Models. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, Yaser Al-Onaizan, Mohit Bansal, and Yun-Nung Chen (Eds.). Association for Computational Linguistics, Miami, Florida, USA, 13487–13521. doi:10.18653/v1/2024.emnlp-main.748

[19] Xue Jiang, Yihong Dong, Lecheng Wang, Zheng Fang, Qiwei Shang, Ge Li, Zhi Jin, and Wenpin Jiao. 2024. Self-Planning Code Generation with Large Language Models. *ACM Transactions on Software Engineering and Methodology* 33, 7, Article 182 (Sept. 2024), 30 pages. doi:10.1145/3672456

[20] Kailun Jin and York University. 2024. Can ChatGPT Support Developers? An Empirical Evaluation of Large Language Models for Code Generation. In *Proceedings of the 21st International Conference on Mining Software Repositories*.

[21] René Just, Darioush Jalali, and Michael D Ernst. 2014. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*. 437–440.

[22] Dongsun Kim, Jaechun Nam, Jaewoo Song, and Sunghun Kim. 2013. Automatic patch generation learned from human-written patches. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 802–811.

[23] Rrezarta Krasniqi and Hyunsook Do. 2023. A Hierarchical Topical Modeling Approach for Recommending Repair of Quality Bugs. In *Proceedings of the 30th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, Taipa, Macao, 37–48. doi:10.1109/SANER56733.2023.00014

[24] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. 2011. Genprog: A generic method for automatic software repair. In *Proceedings of the 19th international symposium on Foundations of software engineering*. 317–327.

[25] Cheryl Lee, Chunqiu Steven Xia, Longji Yang, Jen-tse Huang, Zhouruixing Zhu, Lingming Zhang, and Michael R. Lyu. 2025. UniDebugger: Hierarchical Multi-Agent Framework for Unified Software Debugging. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, Christos Christodoulopoulos, Tanmoy Chakraborty, Carolyn Rose, and Violet Peng (Eds.). Association for Computational Linguistics, Suzhou, China, 18248–18277. doi:10.18653/v1/2025.emnlp-main.921

[26] Ao Li, Yuexiang Xie, Songze Li, Fugee Tsung, Bolin Ding, and Yaliang Li. 2025. Agent-Oriented Planning in Multi-Agent Systems. In *Proceedings of the International Conference on Learning Representations*. https://openreview.net/forum?id=EqcLAU6gyU

[27] Yi-Cheng Lin, Kang-Chieh Chen, Zhe-Yan Li, Tzu-Heng Wu, Tzu-Hsuan Wu, Kuan-Yu Chen, Hung-yi Lee, and Yun-Nung Chen. 2025. Creativity in LLM-based Multi-Agent Systems: A Survey. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, Christos Christodoulopoulos, Tanmoy Chakraborty, Carolyn Rose, and Violet Peng (Eds.). Association for Computational Linguistics, Suzhou, China, 27572–27595. doi:10.18653/v1/2025.emnlp-main.1403

[28] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and LINGMING ZHANG. 2023. Is Your Code Generated by ChatGPT Really Correct? Rigorous Evaluation of Large Language Models for Code Generation. In *Advances in Neural Information Processing Systems*, A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine (Eds.), Vol. 36. Curran Associates, Inc., 21558–21572. https://proceedings.neurips.cc/paper_files/paper/2023/file/43e9d647ccd3e4b7b5baab53f0368686-Paper-Conference.pdf

[29] Ziyang Luo, Xin Li, Hongzhan Lin, Jing Ma, and Lidong Bing. 2024. AMR-Evol: Adaptive Modular Response Evolution Elicits Better Knowledge Distillation for Large Language Models in Code Generation. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, Yaser Al-Onaizan, Mohit Bansal, and Yun-Nung Chen (Eds.). Association for Computational Linguistics, Miami, Florida, USA, 1143–1166. doi:10.18653/v1/2024.emnlp-main.66

[30] Noble Saji Mathews and Meiyappan Nagappan. 2024. Test-Driven Development and LLM-based Code Generation. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering (ASE) (ASE '24)*. Association for Computing Machinery, New York, NY, USA, 1583–1594. doi:10.1145/3691620.3695527

[31] Renée McCauley, Sue Fitzgerald, Gary Lewandowski, Laurie Murphy, Beth Simon, Linda Thomas, and Carol Zander. 2008. Debugging: a review of the literature from an educational perspective. *Computer Science Education* 18, 2 (2008), 67–92.

[32] OpenAI. 2023. ChatGPT: Language model (Mar 14 version). https://chat.openai.com/. Accessed: 2025-05-28.

[33] Chanathip Pornprasit, Chakkrit Tantithamthavorn, Patanamon Thongtanunam, and Chunyang Chen. 2023. D-ACT: Towards Diff-Aware Code Transformation for Code Review Under a Time-Wise Evaluation. In *Proceedings of the 30th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, Taipa, Macao, 296–307. doi:10.1109/SANER56733.2023.00036

[34] Qwen, :, An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, Huan Lin, Jian Yang,

Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jingren Zhou, Junyang Lin, Kai Dang, Keming Lu, Keqin Bao, Kexin Yang, Le Yu, Mei Li, Mingfeng Xue, Pei Zhang, Qin Zhu, Rui Men, Runji Lin, Tianhao Li, Tianyi Tang, Tingyu Xia, Xingzhang Ren, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yu Wan, Yuqiong Liu, Zeyu Cui, Zhenru Zhang, and Zihan Qiu. 2025. Qwen2.5 Technical Report. arXiv:2412.15115 [cs.CL] https://arxiv.org/abs/2412.15115

[35] Christoph Riedl. 2025. Emergent Coordination in Multi-Agent Language Models. arXiv:2510.05174 [cs.MA] https://arxiv.org/abs/2510.05174

[36] Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. 2024. Code Llama: Open Foundation Models for Code. arXiv:2308.12950 [cs.CL] https://arxiv.org/abs/2308.12950

[37] Weisong Sun, Yun Miao, Yuekang Li, Hongyu Zhang, Chunrong Fang, Yi Liu, Gelei Deng, Yang Liu, and Zhenyu Chen. 2025. Source Code Summarization in the Era of Large Language Models . In *Proceedings of the 47th IEEE/ACM International Conference on Software Engineering (ICSE)*. IEEE Computer Society, Los Alamitos, CA, USA, 1882–1894. doi:10.1109/ICSE55347.2025.00034

[38] Gemini Team, Petko Georgiev, Ving Ian Lei, Ryan Burnell, Libin Bai, Anmol Gulati, Garrett Tanzer, Damien Vincent, Zhufeng Pan, and Shibo Wang et al. 2024. Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context. arXiv:2403.05530 [cs.CL] https://arxiv.org/abs/2403.05530

[39] OpenDevin Core Team. 2024. OpenDevin: A Community-Driven Platform for Autonomous AI Software Engineers. GitHub repository. https://github.com/OpenDevin/OpenDevin

[40] Edward Lee Thorndike. 1913. *Educational Psychology...* Vol. 2. Teachers college, Columbia university.

[41] Hanbin Wang, Zhenghao Liu, Shuo Wang, Ganqu Cui, Ning Ding, Zhiyuan Liu, and Ge Yu. 2024. INTERVENOR: Prompting the Coding Ability of Large Language Models with the Interactive Chain of Repair. In *Findings of the Association for Computational Linguistics: ACL 2024*, Lun-Wei Ku, Andre Martins, and Vivek Srikumar (Eds.). Association for Computational Linguistics, Bangkok, Thailand, 2081–2107. doi:10.18653/v1/2024.findings-acl.124

[42] Lei Wang, Chen Ma, Xueyang Feng, Zeyu Zhang, Hao Yang, Jingsen Zhang, Zhiyuan Chen, Jiakai Tang, Xu Chen, Yankai Lin, Wayne Xin Zhao, Zhewei Wei, and Jirong Wen. 2024. A survey on large language model based autonomous agents. *Frontiers of Computer Science* 18, 6 (March 2024), 186345. doi:10.1007/s11704-024-40231-1

[43] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, brian ichter, Fei Xia, Ed Chi, Quoc V Le, and Denny Zhou. 2022. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. In *Advances in Neural Information Processing Systems*, S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh (Eds.), Vol. 35. Curran Associates, Inc., 24824–24837. https://proceedings.neurips.cc/paper_files/paper/2022/file/9d5609613524ecf4f15af0f7b31abca4-Paper-Conference.pdf

[44] Zixiang Xian, Rubing Huang, Dave Towey, Chunrong Fang, and Zhenyu Chen. 2024. TransformCode: A Contrastive Learning Framework for Code Embedding via Subtree Transformation. *IEEE Transactions on Software Engineering* 50, 6 (June 2024), 1600–1619. doi:10.1109/TSE.2024.3393419 Conference Name: IEEE Transactions on Software Engineering.

[45] Junjielong Xu, Ying Fu, Shin Hwei Tan, and Pinjia He. 2025. Aligning the Objective of LLM-Based Program Repair . In *Proceedings of the 47th IEEE/ACM International Conference on Software Engineering (ICSE)*. IEEE Computer Society, Los Alamitos, CA, USA, 2548–2560. doi:10.1109/ICSE55347.2025.00169

[46] Chenyuan Yang, Xuheng Li, Md Rakib Hossain Misu, Jianan Yao, Weidong Cui, Yeyun Gong, Chris Hawblitzel, Shuvendu Lahiri, Jacob R. Lorch, Shuai Lu, Fan Yang, Ziqiao Zhou, and Shan Lu. 2025. AutoVerus: Automated Proof Generation for Rust Code. *Proceedings of the ACM on Programming Languages* 9, OOPSLA2, Article 396 (Oct. 2025), 29 pages. doi:10.1145/3763174

[47] Wenguang Ye and Jiangping Huang. 2025. TraceCoder: A Trace-Driven Multi-Agent Framework for Automated Debugging of LLM-Generated Code. https://github.com/CSMA-Research-Group/TraceCoder. Accessed: 25-Nov-2025.

[48] Xin Yin, Chao Ni, Shaohua Wang, Zhenhao Li, Limin Zeng, and Xiaohu Yang. 2024. ThinkRepair: Self-Directed Automated Program Repair. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, Vienna Austria, 1274–1286. doi:10.1145/3650212.3680359

[49] Hao Yu, Bo Shen, Dezhi Ran, Jiaxin Zhang, Qi Zhang, Yuchi Ma, Guangtai Liang, Ying Li, Qianxiang Wang, and Tao Xie. 2024. CoderEval: A Benchmark of Pragmatic Code Generation with Generative Pre-trained Models. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering* (Lisbon, Portugal) *(ICSE '24)*. Association for Computing Machinery, New York, NY, USA, Article 37, 12 pages. doi:10.1145/3597503.3623316

[50] Quanjun Zhang, Chunrong Fang, Yuxiang Ma, Weisong Sun, and Zhenyu Chen. 2023. A Survey of Learning-based Automated Program Repair. *ACM Transactions on Software Engineering and Methodology* 33, 2, Article 55 (Dec. 2023), 69 pages. doi:10.1145/3631974

[51] Shun Zhang, Zhenfang Chen, Yikang Shen, Mingyu Ding, Joshua B. Tenenbaum, and Chuang Gan. 2023. Planning with Large Language Models for Code Generation. In *Proceedings of the International Conference on Learning Representations*. https://openreview.net/forum?id=Lr8cOOtYbfL

[52] Weiming Zhang, Qingyao Li, Xinyi Dai, Jizheng Chen, Kounianhua Du, Weiwen Liu, Yasheng Wang, Ruiming Tang, Yong Yu, and Weinan Zhang. 2025. NL-Debugging: Exploiting Natural Language as an Intermediate Representation for Code Debugging. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, Christos Christodoulopoulos, Tanmoy Chakraborty, Carolyn Rose, and Violet Peng (Eds.). Association for Computational Linguistics, Suzhou, China, 1533–1549. doi:10.18653/v1/2025.emnlp-main.80

[53] Terry Yue Zhuo, Vu Minh Chien, Jenny Chim, Han Hu, Wenhao Yu, Ratnadira Widyasari, Imam Nur Bani Yusuf, Haolan Zhan, Junda He, Indraneil Paul, Simon Brunner, Chen GONG, James Hoang, Armel Randy Zebaze, Xiaoheng Hong, Wen-Ding Li, Jean Kaddour, Ming Xu, Zhihan Zhang, Prateek Yadav, Naman Jain, Alex Gu, Zhoujun Cheng, Jiawei Liu, Qian Liu, Zijian Wang, David Lo, Binyuan Hui, Niklas Muennighoff, Daniel Fried, Xiaoning Du, Harm de Vries, and Leandro Von Werra. 2025. BigCodeBench: Benchmarking Code Generation with Diverse Function Calls and Complex Instructions. In *The Thirteenth International Conference on Learning Representations*. https://openreview.net/forum?id=YrycTjllL0