# From Bugs to Fixes: Self-Evolving Hierarchical Knowledge Graphs for LLM-Based Code Repair

**Anonymous ACL submission**

## Abstract

Large Language Models have shown remarkable potential in automated code generation, yet the programs they produce frequently fail due to logical flaws or unhandled edge cases. To address this, we propose RA-CRF, a Retrieval-Augmented Code Repair Framework that explicitly integrates a structured Error-Repair Knowledge Graph (ERKG). Unlike existing methods that rely on unstructured self-reflection or raw execution feedback, our framework constructs a hierarchical taxonomy connecting error types to specific repair strategies. By leveraging execution feedback, the framework retrieves relevant knowledge to generate structured repair documents, effectively decoupling error diagnosis from targeted code editing. Furthermore, we introduce an adaptive self-evolution mechanism that continuously refines the ERKG by persisting verified repair results. Experimental results on the BigCodeBench benchmark demonstrate that RA-CRF achieves significant performance gains across various open-source and closed-source models. Notably, DeepSeek-V3 enhanced by RA-CRF establishes a new state-of-the-art with a 65.52% Pass@1 score. Our findings indicate that integrating explicit and evolvable error-correction knowledge is a robust path to enhancing the reliability and interpretability of code generated by large language models.

## 1 Introduction

Large Language Models have demonstrated transformative capabilities in automated code generation, permeating domains such as software development, competitive programming, and intelligent tutoring.Recent models such as Codex (Chen et al., 2021), StarCoder (Li et al., 2023), Code Llama (Roziere et al., 2023), and GPT-5 (Wang et al., 2026)exhibit high fluency in translating natural language into executable programs. However, for complex algorithmic tasks requiring rigorous logical correctness and comprehensive test coverage, the reliability of LLM-generated code remains a critical bottleneck. Even the most advanced models frequently encounter silent logical flaws, improper handling of edge cases, or severe runtime exceptions. (Liu et al., 2023; Chen et al., 2023b; Khoury et al., 2023; Lai et al., 2023). These failure modes still exist in modern benchmarks such as HumanEval-V (Zhang et al., 2024b), MBPP(Austin et al., 2021), APPS (Hendrycks et al., 2021), and BigCodeBench (Zhuo et al., 2024),highlighting the pressing need to improve the reliability of LLM-generated code.

Prior work has pursued several directions to enhance code reliability. One approach trains specialized neural repair models to correct generated code, yielding performance gains on various tasks (Gupta et al., 2020; Wang et al., 2017; Fu et al., 2019; Chen et al., 2023a), but this requires additional training overhead. Recently, we have been leveraging the inference capabilities of LLM for self-debugging(Pan et al., 2024), iterating and improving based on execution feedback (Koyuncu et al., 2020; Madaan et al., 2023; Chen et al., 2023b; Shinn et al., 2023), and further integrating program-level signals (e.g., failed tests, runtime tracing) to guide code revisions (Le Goues et al., 2011; Liu et al., 2019; Xia et al., 2023), bringing more optimization possibilities to the code generated by LLM.

Despite these advances, existing approaches are hindered by two fundamental limitations:

- Lack of structured knowledge. Current systems rely solely on the LLM's implicit reasoning over raw error logs or self-generated rationales, without incorporating structured knowledge. This often yields superficial or inconsistent repairs and can even introduce new bugs or regressions.

- Inability to capture systematic error types. While LLM consistently exhibit recurring fail-

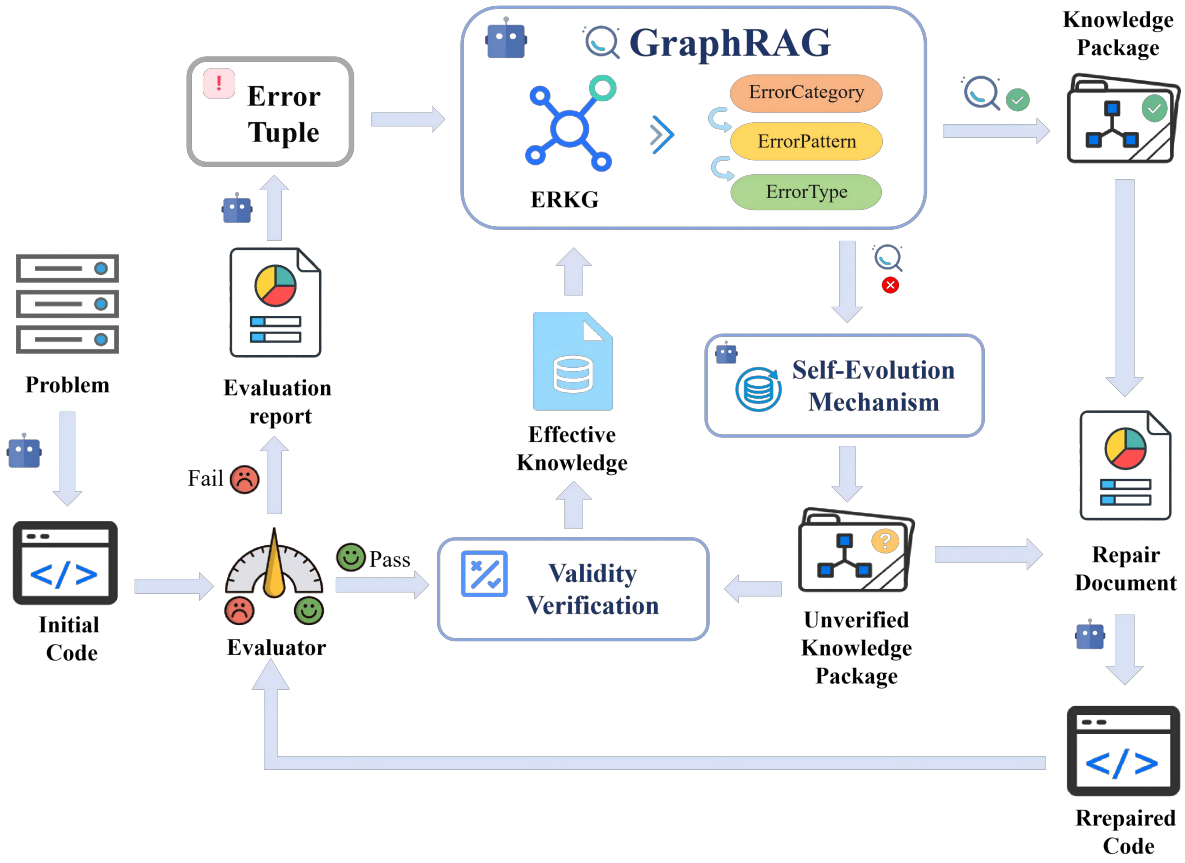


Figure 1: Overview of the RA-CRF framework. The robot icons indicate processes driven by the LLM. In our system, all these tasks are executed by a single unified model instance to ensure consistency.

ure patterns across different tasks, existing methods treat each failure in isolation, ignoring cross-task patterns that could provide information and constraints for the repair process.

This motivates our central research question: **How can we effectively incorporate explicit structured repair knowledge into LLM-based code repair to enable more robust, accurate, and interpretable rectifications?** To address this, we propose RA-CRF as shown in Figure 1, a novel Retrieval-Augmented Code Repair Framework. RA-CRF integrates structured repair expertise into the LLM-based debugging process by retrieving context-aware patterns from a Hierarchical Error-Repair Knowledge Graph (ERKG) and transforming them into actionable repair guidance. This approach enables a more robust, precise, and interpretable code rectification mechanism, facilitating a transition from stochastic trial-and-error to knowledge-driven inference.

Our contributions are as follows:

1. We define and construct a hierarchical Error–Repair Knowledge Graph (ERKG) equipped with a self-evolution mechanism, which explicitly models multi-level mappings from error categories to precise repair templates, thereby filling the vacancy in structured knowledge representation for LLM-based code repair.

2. We develop the RA-CRF framework, which leverages a hierarchical retrieval mechanism and structured repair document generation to guide models from stochastic trial-and-error toward logic-driven precision repair, significantly enhancing the stability and interpretability of the debugging process.

3. We empirically demonstrate that structured knowledge provides an effective inductive bias for code rectification. Empowered by RA-CRF, DeepSeek-V3 achieves a new state-of-the-art performance of 65.52% (Pass@1) on the challenging BigCodeBench benchmark, establishing a robust baseline for future research in autonomous code repair.

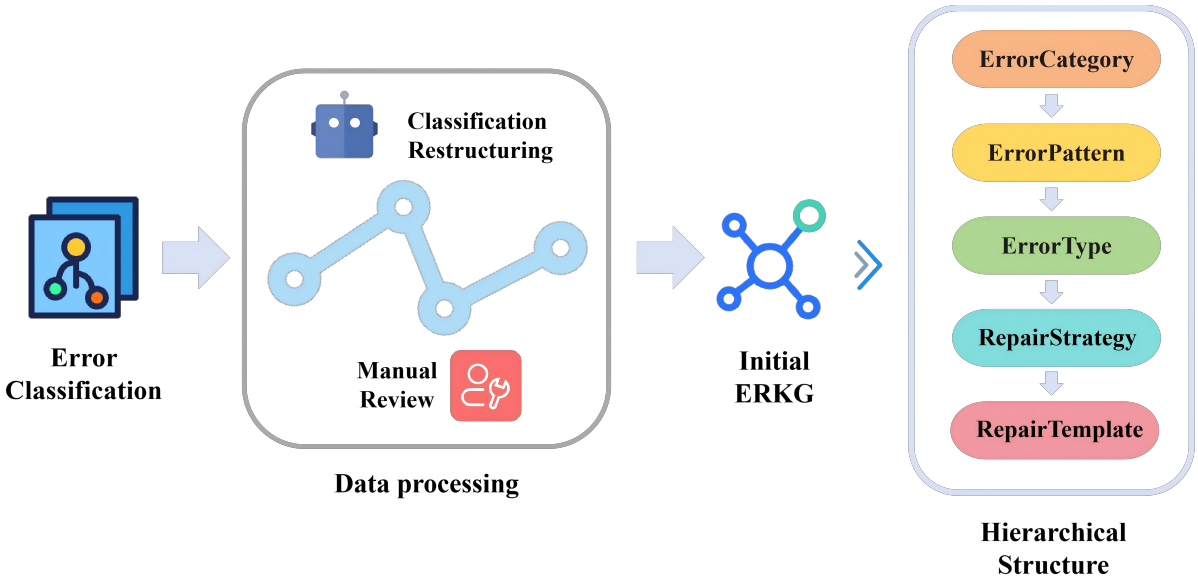


Figure 2: ERKG Construction Workflow. Detailed reconstruction prompts and node definitions are provided in Figure 6 and Table 4.

## 2 Related Work

### 2.1 LLM for Code Generation and Repair

Although code-centric LLMs (Chen et al., 2021; Roziere et al., 2023; Li et al., 2023) excel at generation, they often produce erroneous programs in algorithmic or boundary-sensitive tasks (Seo et al., 2021; Liu et al., 2023). To mitigate this, existing feedback-driven refinement strategies employ either text-based self-debugging (Madaan et al., 2023; Chen et al., 2023b) or test-case-guided localization inspired by traditional APR paradigms (Xia et al., 2023; Arslan et al., 2024). Recently, execution-guided and structural refinement frameworks have emerged. For instance, RESCUE (Shi and Zhang, 2025) utilizes dynamic execution traces for anomaly localization, while CODEXITY (Kim et al., 2024) leverages structural complexity metrics for refactoring. However, they either rely on flat, unorganized contextual feedback streams that induce stochastic trial-and-error, or prioritize structural aesthetics over core functional correctness.

### 2.2 Knowledge-Augmented LLMs for Code Intelligence

Knowledge-augmented LLMs typically retrieve unstructured or weakly structured external resources (e.g., Q&A threads, documentation, or code repositories) to assist generation (Svyatkovskiy et al., 2020; Huang et al., 2024). For example, SEC-CODER (Zhang et al., 2024a) retrieves whole-file secure code examples to facilitate in-context learning. Nevertheless, such knowledge remains task-independent and lacks a repair-oriented organization. Due to the flatness of injected entire-file contexts, these methods offer limited inductive bias for localized debugging, failing to provide fine-grained, self-evolving repair guidance.

### 2.3 Error Analysis and Taxonomies in Code Generation

Prior works have analyzed LLM execution failures and proposed various error taxonomies (Just et al., 2014; Lin et al., 2017; Bogaerts et al., 2024). However, these analyses remain largely descriptive, and their error categories are rarely translated into actionable, closed-loop repair mechanisms. Our work bridges this gap by explicitly mapping distinct error types to specific repair templates within a self-evolving Error-Repair Knowledge Graph (ERKG) topology, transforming static diagnostic insights into structured, actionable repair guidance.

## 3 Methodology

We propose a knowledge-guided code repair framework that improves the success rate of LLM-based code repair by explicitly modeling and leveraging structured knowledge of code bug repair. Unlike previous methods that rely on unstructured self-reflection or raw execution feedback, our approach introduces a hierarchical ERKG to provide systematic and controlled guidance throughout the repair

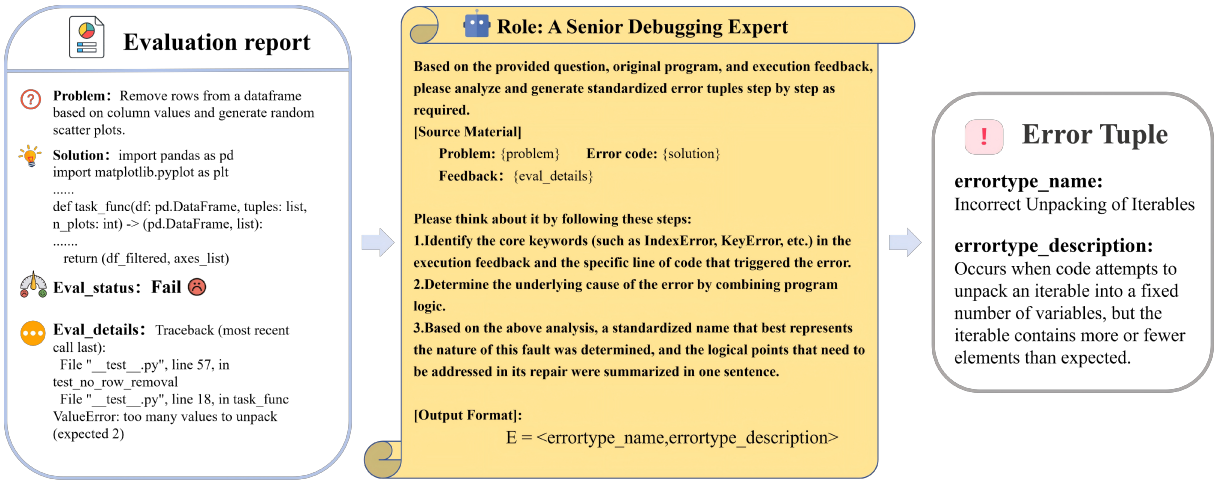


Figure 3: Example of generating an error tuple

process. The overall framework is shown in Figure 1.

### 3.1 Hierarchical Error-Repair Knowledge Graph (ERKG) Construction

Existing methods often treat execution feedback as unstructured text, resulting in an overly ambiguous mapping path from fault signals to repair decisions. To address this, we propose a hierarchical ERKG as a logical intermediary between execution feedback and repair strategies. By constructing structured associations from error features to repair strategies, we transform the black-box self-reflection of LLM (Learning-Based Learning) into a systematic and controllable knowledge reasoning process.

Inspired by the Python vulnerability taxonomy(Bogaerts et al., 2024) and QuixBugs(Lin et al., 2017), we guided the LLM to deeply deconstruct the classification logic of these two studies and reconstruct existing error patterns according to a pre-defined hierarchical system, thereby generating a preliminary structured graph. Subsequently, through rigorous manual cleaning and multiple rounds of verification, redundancy was eliminated and semantic accuracy was ensured, ultimately forming the baseline version of the ERKG. The specific construction process is shown in Figure 2.

### 3.2 Error extraction and Semantic Normalization

Effective knowledge-guided remediation relies on transforming raw execution feedback into a structured error representation. However, execution feedback generated by different benchmarks is highly heterogeneous in format and granularity. To address the challenge of heterogeneous execution feedback formats and varying granularities across different benchmarks, we introduce LLM as a semantic parser to extract core fault signals. LLM can accurately identify and remove environmental noise such as memory addresses and local paths while capturing key context such as fault location and execution trajectory.(Zhong et al., 2024; Xu et al., 2024; Zhang et al., 2023)

To support efficient knowledge retrieval, the parsed signals undergo semantic normalization via an LLM. As shown in Figure 3, in practice, we employ Chain-of-Thought prompting to guide the LLM through sequential stages of feature abstraction and contextual disambiguation. This approach enables the LLM to distill the core semantics of failures, transforming complex execution feedback into a standardized error tuple E = <errortype_name, errortype_description>. By filtering out environmental noise and mitigating semantic ambiguity, this process establishes a robust semantic foundation for precise downstream mapping with ERKG nodes.

### 3.3 Knowledge-Guided Retrieval and Self-Evolution Mapping

To achieve accurate conversion from raw fault signals to structured repair solutions, this framework designs a hierarchical retrieval and self-evolution mapping mechanism based on GraphRAG.As shown in Figure 4. This mechanism can not only retrieve known knowledge from ERKG but also dynamically expand the semantic space when facing unknown errors.

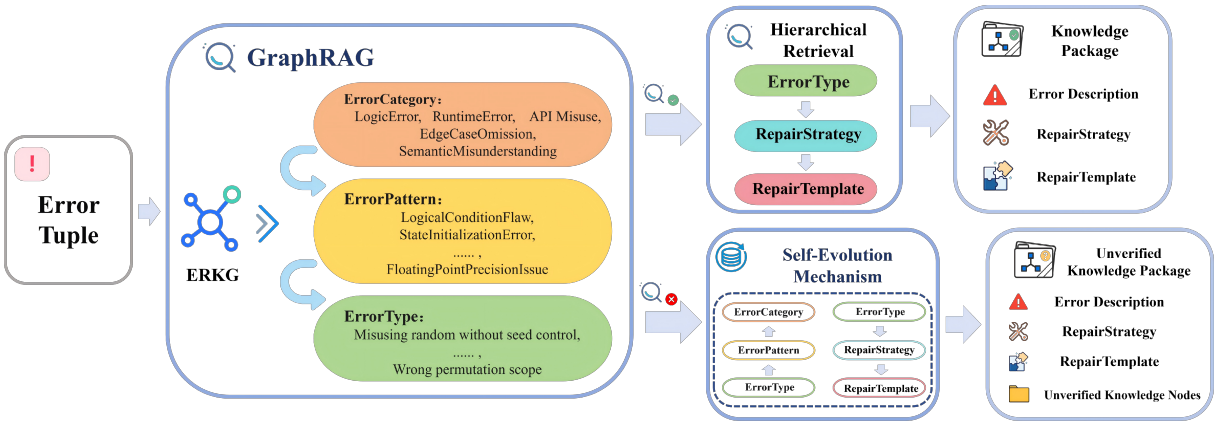


Figure 4: Illustration of the Hierarchical Retrieval and Self-Evolution Mechanism within the RA-CRF framework.

#### 3.3.1 Hierarchical Knowledge Mapping.

Based on the standardized error tuples generated in the previous stage E = <errortype_name, errortype_description>, the system executes a top-down hierarchical matching strategy. The LLM first performs semantic alignment in the top-level semantic space (ErrorCategory layer) of ERKG, then progressively searches down through the ErrorPattern layer to find candidate ErrorType nodes. The LLM determines whether the current error can be covered by existing Type nodes. If a match is successful, the system automatically extracts the associated RepairStrategy and RepairTemplate along the semantic path and encapsulates them into a knowledge package. Conversely, if existing nodes cannot achieve complete coverage, a self-evolution mechanism is triggered.

#### 3.3.2 Self-Evolution Mechanism

As shown in Alg 1. When faced with edge cases or novel faults that cannot be covered by existing ERKG nodes, the system dynamically expands its semantic space through a self-evolution mechanism. LLM performs source tracing based on the current error's feature description, comparing existing candidate ErrorPattern nodes to determine their potential to encompass new Type semantics. If a match is found, the system generates the corresponding repair strategy and template under the newly added Type node. If existing pattern libraries are incompatible, LLM induces the generation of a entirely new ErrorPattern node and automatically constructs a complete topological path from the pattern to the specific repair template. Ultimately, these dynamically generated semantic entities are uniformly encapsulated into a knowledge package, providing core content support and logical alignment for the subsequent generation of structured repair documents. To ensure the reliability of ERKG evolution, the system employs a delayed update mechanism based on verification results.Only repair logic that has been tested in practice will be formally accepted as part of the ERKG.

### 3.4 Structured Repair Documentation Generation

**Please generate a structured code repair document based on the following information.**

**Input Information:**
**1. Problem and Code :**{problem},{buggy_code}
**2. Evaluation results :**
**- eval_status:** {eval_status}
**- eval_details:** {eval_details}
**3.Relevant code repair knowledge :**
{knowledge_ package}

**Output structure:**
{{
"error_analysis": "...",
"repair_steps": ["...", "..."],
"code_template": "..."
}}

Figure 5: Prompt for Structured Repair Documentation

Directly guiding the LLM to rewrite code based on error feedback and retrieved knowledge often leads to uncontrollable semantic shifts or inconsistent repairs(Seo et al., 2021; Hasan et al., 2025). To address this issue, we introduce a structured repair document as an explicit intermediate representation to formalize the repair intent before performing physical modifications. As shown in Figure 5. This document is generated by the LLM by integrating three dimensions of input: (1) the

**Algorithm 1:** Hierarchical Retrieval and Self-Evolution Mechanism

**Input:** $E$(error tuple), $G$(ERKG)
**Output:** $K$(knowledge package)

`// Expression Statement:`

$EC$(`ErrorCategory`), $EP$(`ErrorPattern`), $ET$(`ErrorType`), $RS$(`RepairStrategy`), $RT$(`RepairTemplate`)

$**_{new}$(`new node`), $**_{known}$(`existing node`)

LLM follows the retrieval path $EC \rightarrow EP \rightarrow ET$ and;
executing a top-down hierarchical matching strategy.

**if** *Found* $ET_{matched}$ **then**
  Continue searching for $RS$ and $RT$ under the matching $ET_{matched}$, executes a top-down hierarchical matching strategy.
  $Encapsulate(\{ET_{matched}, RS, RT\}) \rightarrow K$
**else**
  Generate a $ET_{new}$, with corresponding $RS_{new}$ and $RT_{new}$. Then,establish index relationship:
  $ET_{new} \rightarrow RS_{new} \rightarrow RT_{new}$
  and perform upward Match($ET_{new}$,$EP_{known}$).
  **if** *Found* $EP_{matched}$ **then**
    Establish index relationship:
    $EP_{known} \rightarrow ET_{new}$
  **else**
    Generate a $EP_{new}$ Establish index relationship:
    $EP_{new} \rightarrow ET_{new}$ and $EC_{known} \rightarrow EP_{new}$
  **end**
  Collect all newly generated nodes and their index relationships then:
  $Encapsulate$(New_Nodes, Relationships) $\rightarrow K$
**end**
**return** $K$

problem and the code to be modified; (2) the execution feedback and standardized error tuples; and (3) the knowledge package generated in the previous stage.

Under this mechanism, the retrieved repair strategies and repair templates constitute strict semantic constraints, forcing the LLM to generate a repair plan within a predetermined logical framework, thereby significantly suppressing irrelevant, redundant modifications or illusion-driven, spurious repairs. During the execution phase, this document serves as an instruction guide for the LLM to perform fine-grained local editing. By decoupling repair planning from code synthesis, the model is confined to the document-marked defect areas for precise fine-tuning, avoiding the uncertainties of a full program rewrite. This decoupling paradigm ensures high controllability of the repair process and achieves minimal intrusion into the original program logic.

### 3.5 Evolutionary Stability of the Knowledge Base

The system employs an adaptive iterative mechanism, guiding the LLM to execute precise local edits based on the repair document. After each iteration, the evaluator monitors the repair status via regression testing: successful repairs trigger an early exit to minimize computational costs, while failures capture error trajectories as negative feedback to refine subsequent repair schemes. As illustrated in Table 5. The convergence of Pass@1 across escalating repair iterations validates the system's ability to iteratively approximate the optimal repair through feedback learning.

To ensure the integrity of the ERKG, the system implements an execution-validated update mechanism. Knowledge nodes are persisted into the graph only after their corresponding repair paths pass verification. As illustrated in Table 5. Empirical results indicate that as the graph evolves, the increment of effective knowledge gradually reaches a steady state. This dual convergence serves as an inherent hallucination filter, effectively mitigating noise infiltration from erroneous attempts and facilitating a paradigm shift from scale-centric growth to quality-oriented refinement.

## 4 Experimental Setup

### 4.1 Datasets

This study adopts BigCodeBench (BCB)(Zhuo et al., 2024) as the primary benchmark. Unlike earlier datasets like HumanEval-V(Zhang et al., 2024b)or MBPP(Austin et al., 2021),which focus on isolated algorithms, BCB features real-world tasks involving complex tool calls, API usage, and cross-library interactions. This provides a more rigorous environment for evaluating the correctness, robustness, and repair capabilities of LLMs in practical development scenarios.

Furthermore, We further utilized a HumanEvalPack(Python subset)(Muennighoff et al., 2024) to assess the generalization of our method. While BCB emphasizes third-party API usage, HumanEvalPack focuses on core algorithmic logic and language-intrinsic features. This allows us to verify that ERKG remains effective for purely algorithmic errors, demonstrating robust fault-correction capabilities even without complex external library contexts.

| Models | BigCodeBench | | HumanEvalPack | |
|---|---|---|---|---|
| | Initial code | Modified code | Initial code | Modified code |
| **The closed-source models** | | | | |
| GPT4o | 51.66 | 65.87(↑ 14.21) | 92.07 | 96.34(↑ 4.27) |
| GPT-5 | 51.49 | **66.84(↑ 15.35)** | 90.24 | **96.95**(↑ 6.71) |
| **The open-source models** | | | | |
| Qwen2.5-14B-Instruct | 43.94 | 53.68(↑ 9.74) | 79.26 | 87.80(↑ **8.54**) |
| Qwen2.5-32B-Instruct | 39.12 | 52.1(↑ 12.98) | 85.97 | 88.41(↑ 2.44) |
| Qwen2.5-72B-Instruct | 47.54 | 58.85(↑ 11.31) | 83.53 | 89.63(↑ 6.1) |
| Qwen3-Coder-30B-A3B-Instruct | 48.94 | 61.14(↑ 12.2) | 92.07 | 93.29( ↑ 1.22) |
| llama-3.1-70b-instruct | 46.22 | 57.63(↑ 11.41) | 84.14 | 88.41(↑ 4.27) |
| llama-4-maverick | 45.96 | 55.26(↑ 9.3) | 85.36 | 91.46(↑ 6.1) |
| DeepSeek-V3 | 50.17 | **65.52**(↑ **15.35**) | 89.63 | **93.6**(↑ 4.27) |

Table 1: Experimental results are reported over three iterations on BigCodeBench (Full) and HumanEvalPack. For BigCodeBench, initial code is generated via the official instruct_prompt. Bold values highlight the Best Performance and Maximum Improvement for closed-source and open-source models, respectively.

### 4.2 Models

For model selection, we referenced the official BigCodeBench (BCB) benchmark results and selected a series of representative language models in terms of parameter size and performance to comprehensively evaluate the framework. Open-source models included the Qwen series(Hui et al., 2024; Yang et al., 2025), Llama series(Grattafiori et al., 2024; Shukla, 2025), and the best-performing DeepSeek-V3(Liu et al., 2024). Closed-source models included GPT-4o(Hurst et al., 2024) and GPT-5(Singh et al., 2025). All models were not fine-tuned, and consistent decoding configurations (such as temperature and maximum length) were maintained throughout all experiments.

### 4.3 Metrics

We adopt Pass@1 (Chen et al., 2021) as the primary metric. Compared to Pass@k(k>1), Pass@1 requires a correct repair in a single attempt, avoiding performance overestimation and ensuring a more rigorous evaluation of repair effectiveness.

## 5 Main Results

As shown in Table 1, the proposed RA-CRF framework consistently improves execution correctness over initial code generation across all evaluated models.On the BigCodeBench (BCB) dataset, compared to the highest Pass@1 performance on the current official leaderboard (as of May 2026), RA-CRF enables DeepSeek-V3 to achieve SOTA performance. Experimental results span both open-source and closed-source models of varying parameter scales,strongly verifying the universality and effectiveness of the framework in complex code repair tasks. As shown in Table 2, our method also has a significant competitive advantage compared to other methods.

| Method | Pass@1 |
|---|---|
| Self-Refine(Madaan et al., 2023) | 26.5 |
| SecCoder(Zhang et al., 2024a) | 61.7 |
| Codexity(Kim et al., 2024) | 60.9 |
| RESCUE(Shi and Zhang, 2025) | 60.9 |
| **RA-CRF(Ours)** | **65.5** |

Table 2: Performance comparison on BigCodeBench (detailed analysis in Appendix C).

The overall trend indicates that the average absolute improvement of RA-CRF on BCB is significantly higher than on HumanEvalPack. We attribute this discrepancy to the inherent nature of the datasets: compared to the highly complex tasks in BCB, the initial generation accuracy of HumanEvalPack already averages over 80%. This ceiling effect objectively constrains the available

margin for further improvement.Furthermore, comparing general-purpose instruction-tuned models with code-specialized models, the latter possesses a natural advantage in initial generation (48.94%). However, with the introduction of RA-CRF, the general-purpose model effectively compensates for its lack of domain-specific programming knowledge through retrieved precise repair documents, achieving a notable 11.41% improvement. This proves that RA-CRF can serve as a powerful external knowledge plug-in, significantly narrowing the performance gap between general and specialized models in expert tasks and greatly enhancing the utility of general models in specific domains.

In summary, the experimental results demonstrate that RA-CRF can effectively correct programs that fail in the initial generation phase. Models of different sizes and sources show consistent performance improvements, reflecting its strong model independence. This further indicates that the improvement in code reliability primarily stems from the proposed cooperative pattern, rather than solely relying on the inherent capabilities of a specific model.

## 6 Analysis

To investigate the specific contributions of the key components within the RA-CRF framework to its repair performance, we conducted ablation experiments on DeepSeek-V3, the best-performing open-source model, using the BigCodeBench dataset. As shown in Table 3, we compared the full framework with the following variants:

| **Method Variant** | **BigCodeBench** |
|---|---|
| Initial Generation | 50.17 |
| - w/o Evaluation Report and ERKG | 53.24(↑ 3.07) |
| - w/o Evaluation Report | 53.59(↑ 3.42) |
| - w/o ERKG | 61.75(↑ 11.58) |
| Evaluation report + ERKG(Ours) | 65.52(↑ 15.35) |

Table 3: Ablation study of RA-CRF on BigCodeBench (Pass@1). All variants are evaluated using DeepSeek-V3 as the backbone model.

- Compared to the "- w/o Evaluation Report" variant, the performance gain is only 3.42%, which is significantly lower than that of the full framework. This underscores the fundamental role of execution reports in error localization and context awareness. Without execution feedback, the model struggles to accurately align abstract knowledge with specific runtime failures.

- Comparing the "- w/o ERKG" variant , the performance gain further increases from 11.58% to 15.35%. This additional 3.77% net growth validates the uniqueness of structured knowledge in transcending simple trial-and-error modes; it provides the model with explicit mapping guidance from error categories to repair strategies.

- The "- w/o Evaluation Report and ERKG" variant (relying only on prompt-based self-repair) shows the lowest gain of only 3.07%. This indicates that without systematic constraints and external knowledge guidance, models are highly prone to ineffectual attempts. Notably, while the model maintains an 11.58% improvement when only ERKG is removed, the performance drops sharply when execution feedback is also omitted. This reveals that execution feedback defines the repair target, whereas ERKG provides the repair pathway. Their combination not only improves the absolute success rate but also significantly enhances the stability of repair decisions.

In summary, the ablation study confirms the rationality of the RA-CRF module design and emphasizes that the synergy between feedback-based localization and structured knowledge guidance is pivotal for high-quality repair. Simultaneously, the results reveal the model's dependence on external assistance, suggesting that parameter scale alone is insufficient for handling extremely complex logical vulnerabilities. Therefore, future research will explore the internalization of external feedback mechanisms through data distillation or reinforcement learning, which is key to fundamentally enhancing the model's capacity for autonomous reasoning and zero-shot repair.

## 7 Conclusion

We have presented RA-CRF, which decouples error diagnosis from code editing via a structured ERKG. Fueled by an execution-driven self-evolution mechanism, the framework shifts the focus from graph scale to repair quality. Empirical results confirm that integrating explicit knowledge with iterative verification significantly enhances the reliability and interpretability of LLM-based code repair.

## Limitations

Despite its effectiveness, our framework has several limitations. First, the current language coverage is restricted to Python; the generalizability to other programming paradigms remains to be explored. Second, the system exhibits a heavy execution dependency, relying on high-quality test suites to provide feedback, which may not always be available in real-world legacy codebases. Finally, we observe knowledge sparsity for long-tail errors: as the repair process converges, the system encounters diminishing returns in acquiring generalized knowledge for rare, complex failure modes.

## References


Muhammad Arslan, Hussam Ghanem, Saba Munawar, and Christophe Cruz. 2024. A survey on rag with llms. *Procedia computer science*, 246:3781–3790.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and 1 others. 2021. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*.

Frédéric CG Bogaerts, Naghmeh Ivaki, and José Fonseca. 2024. A taxonomy for python vulnerabilities. *IEEE Open Journal of the Computer Society*, 5:368–379.

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

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé 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, and 39 others. 2021. Evaluating large language models trained on code. *CoRR*, abs/2107.03374.

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

Cheng Fu, Huili Chen, Haolan Liu, Xinyun Chen, Yuandong Tian, Farinaz Koushanfar, and Jishen Zhao. 2019. Coda: An end-to-end neural program decompiler. *Advances in Neural Information Processing Systems*, 32.

Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, and 1 others. 2024. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*.

Kavi Gupta, Peter Ebert Christensen, Xinyun Chen, and Dawn Song. 2020. Synthesize, execute and debug: Learning to repair for neural program synthesis. *Advances in Neural Information Processing Systems*, 33:17685–17695.

Mohammad Saqib Hasan, Saikat Chakraborty, Santu Karmaker, and Niranjan Balasubramanian. 2025. Teaching an old llm secure coding: Localized preference optimization on distilled preferences. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 26039–26057.

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

Tao Huang, Zhihong Sun, Zhi Jin, Ge Li, and Chen Lyu. 2024. Knowledge-aware code generation with large language models. In *Proceedings of the 32nd IEEE/ACM International Conference on Program Comprehension*, pages 52–63.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, and 1 others. 2024. Qwen2. 5-coder technical report. *arXiv preprint arXiv:2409.12186*.

Aaron Hurst, Adam Lerer, Adam P Goucher, Adam Perelman, Aditya Ramesh, Aidan Clark, AJ Ostrow, Akila Welihinda, Alan Hayes, Alec Radford, and 1 others. 2024. Gpt-4o system card. *arXiv preprint arXiv:2410.21276*.

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*, pages 437–440.

Raphaël Khoury, Anderson R Avila, Jacob Brunelle, and Baba Mamadou Camara. 2023. How secure is code generated by chatgpt? In *2023 IEEE international conference on systems, man, and cybernetics (SMC)*, pages 2445–2451. IEEE.

Sung Yong Kim, Zhiyu Fan, Yannic Noller, and Abhik Roychoudhury. 2024. Codexity: Secure ai-assisted code generation. *arXiv preprint arXiv:2405.03927*.

Anil Koyuncu, Kui Liu, Tegawendé F Bissyandé, Dongsun Kim, Jacques Klein, Martin Monperrus, and Yves Le Traon. 2020. Fixminer: Mining relevant fix patterns for automated program repair. *Empirical Software Engineering*, 25(3):1980–2024.

Yuhang Lai, Chengxi Li, Yiming Wang, Tianyi Zhang, Ruiqi Zhong, Luke Zettlemoyer, Wen-tau Yih, Daniel Fried, Sida Wang, and Tao Yu. 2023. Ds-1000: A natural and reliable benchmark for data science code generation. In *International Conference on Machine Learning*, pages 18319–18345. PMLR.

Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. 2011. Genprog: A generic method for automatic software repair. *Ieee transactions on software engineering*, 38(1):54–72.

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, and 1 others. 2023. Starcoder: may the source be with you! *arXiv preprint arXiv:2305.06161*.

Derrick Lin, James Koppel, Angela Chen, and Armando Solar-Lezama. 2017. Quixbugs: A multi-lingual program repair benchmark set based on the quixey challenge. In *Proceedings Companion of the 2017 ACM SIGPLAN international conference on systems, programming, languages, and applications: software for humanity*, pages 55–56.

Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, and 1 others. 2024. Deepseek-v3 technical report. *arXiv preprint arXiv:2412.19437*.

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. *Advances in Neural Information Processing Systems*, 36:21558–21572.

Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F Bissyandé. 2019. Tbar: Revisiting template-based automated program repair. In *Proceedings of the 28th ACM SIGSOFT international symposium on software testing and analysis*, pages 31–42.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, and 1 others. 2023. Self-refine: Iterative refinement with self-feedback, 2023. *URL https://arxiv.org/abs/2303.17651*.

Niklas Muennighoff, Qian Liu, Armel Zebaze, Qinkai Zheng, Binyuan Hui, Terry Yue Zhuo, Swayam Singh, Xiangru Tang, Leandro Von Werra, and Shayne Longpre. 2024. Octopack: Instruction tuning code large language models. In *International Conference on Learning Representations*, volume 2024, pages 7604–7623.

Liangming Pan, Michael Saxon, Wenda Xu, Deepak Nathani, Xinyi Wang, and William Yang Wang. 2024. Automatically correcting large language models: Surveying the landscape of diverse automated correction strategies. *Transactions of the Association for Computational Linguistics*, 12:484–506.

Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, and 1 others. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*.

HyeonTae Seo, Yo-Sub Han, and Sang-Ki Ko. 2021. Multifix: Learning to repair multiple errors by optimal alignment learning. In *Findings of the Association for Computational Linguistics: EMNLP 2021*, pages 4850–4855.

Jiahao Shi and Tianyi Zhang. 2025. Rescue: Retrieval augmented secure code generation. *arXiv preprint arXiv:2510.18204*.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. Reflexion: Language agents with verbal reinforcement learning. *Advances in Neural Information Processing Systems*, 36:8634–8652.

A Shukla. 2025. Llama 4: Meta's multimodal leap in ai-architecture, capabilities, and comparative analysis.

Aaditya Singh, Adam Fry, Adam Perelman, Adam Tart, Adi Ganesh, Ahmed El-Kishky, Aidan McLaughlin, Aiden Low, AJ Ostrow, Akhila Ananthram, and 1 others. 2025. Openai gpt-5 system card. *arXiv preprint arXiv:2601.03267*.

Alexey Svyatkovskiy, Shao Kun Deng, Shengyu Fu, and Neel Sundaresan. 2020. Intellicode compose: Code generation using transformer. In *Proceedings of the 28th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering*, pages 1433–1443.

Ke Wang, Rishabh Singh, and Zhendong Su. 2017. Dynamic neural program embedding for program repair. *arXiv preprint arXiv:1711.07163*.

Shansong Wang, Mingzhe Hu, Qiang Li, Mojtaba Safari, Richard LJ Qiu, and Xiaofeng Yang. 2026. Capabilities of gpt-5 on multimodal medical reasoning. In *Medical Imaging 2026: Imaging Informatics*, volume 13930, pages 77–86. SPIE.

Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. 2023. Automated program repair in the era of large pre-trained language models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 1482–1494. IEEE.

Junjielong Xu, Ruichun Yang, Yintong Huo, Chengyu Zhang, and Pinjia He. 2024. Divlog: Log parsing with prompt enhanced in-context learning. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pages 1–12.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, and 1 others. 2025. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*.

Boyu Zhang, Tianyu Du, Junkai Tong, Xuhong Zhang, Kingsum Chow, Sheng Cheng, Xun Wang, and Jianwei Yin. 2024a. Seccoder: Towards generalizable and robust secure code generation. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 14557–14571.

Fengji Zhang, Linquan Wu, Huiyu Bai, Guancheng Lin, Xiao Li, Xiao Yu, Yue Wang, Bei Chen, and Jacky Keung. 2024b. Humaneval-v: Evaluating visual understanding and reasoning abilities of large multimodal models through coding tasks. *CoRR*, abs/2410.12381.

Kechi Zhang, Zhuo Li, Jia Li, Ge Li, and Zhi Jin. 2023. Self-edit: Fault-aware code editor for code generation. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 769–787.

Aoxiao Zhong, Dengyao Mo, Guiyang Liu, Jinbu Liu, Qingda Lu, Qi Zhou, Jiesheng Wu, Quanzheng Li, and Qingsong Wen. 2024. Logparser-llm: Advancing efficient log parsing with large language models. In *Proceedings of the 30th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pages 4559–4570.

Terry Yue Zhuo, Minh Chien Vu, Jenny Chim, Han Hu, Wenhao Yu, Ratnadira Widyasari, Imam Nur Bani Yusuf, Haolan Zhan, Junda He, Indraneil Paul, and 1 others. 2024. Bigcodebench: Benchmarking code generation with diverse function calls and complex instructions. *arXiv preprint arXiv:2406.15877*.

## A ERKG Create Details

Role: A Python code repairing expert.

**Please analyze and integrate the classification results from the Python Vulnerabilities Taxonomy and QuixBugs studies, and then reconstruct them according to the new classification requirements.**

**[Source Material]**

**Source A:** {Python Bug Taxonomy}

**Source B:** {QuixBugs Study}

**Requirements:**

**1.Analyze each error label in Source A and Source B.**

**2.Integrate the classification results from Source A and Source B.**

**3.Construct a classification system based on the new error definition** {new rule}**.**

**4.The relationships between nodes in the new category must strictly follow this structure:**

(:ErrorCategory) -[:HAS_PATTERN]->(:ErrorPattern) -[:HAS_SUBTYPE] -> (:ErrorType) -[:HAS_STRATEGY]-> (:FixStrategy) -[:HAS_TEMPLATE]-> (:FixTemplate)

**[Output Format]:**

- ErrorCategory:[name],[description]
- ErrorPattern:[name],[description]
- ErrorType:[name],[description]
- RepairStrategy:[name],[explanation]
- RepairTemplate:[code_snippet],[placeholders]

Figure 6: Classification Restructuring Prompt

| Node Type | Count | Description |
|---|---|---|
| ErrorCategory | 5 | High-level semantic clusters that partition the defect space into broad domains. |
| ErrorPattern | 13 | Structural archetypes that abstract recurring failure signatures and logical vulnerabilities across different tasks. |
| ErrorType | 41 | Fine-grained fault taxonomies that map specific programmatic errors to their precise root causes and symptoms. |
| RepairStrategy | 41 | Conceptual remediation blueprints that provide high-level algorithmic logic and strategic intent for repairing specific faults. |
| RepairTemplate | 41 | Parametric code skeletons that offer concrete, reusable implementation patterns for physical code rectification. |

Table 4: Detailed definitions and initial population counts for ERKG node types.

## B Validation of Iterative Convergence

| Repair Round | Pass@1 (%) | $\Delta_1$ | $\Delta_2$ | New Nodes | Valid Nodes |
|---|---|---|---|---|---|
| Initial code | 48.94 | 0 | 0 | 0 | 0 |
| 1 | 56.92 | 7.98 | 7.98 | 197 | 32 |
| 2 | 59.91 | 10.97 | 2.99 | 61 | 3 |
| 3 | 61.14 | 12.20 | 1.23 | 77 | 3 |
| 4 | 61.92 | 12.98 | 0.78 | 65 | 0 |
| 5 | 62.63 | 13.69 | 0.71 | 53 | 1 |
| 6 | 63.15 | 14.21 | 0.52 | 54 | 0 |
| 7 | 63.42 | 14.48 | 0.27 | 60 | 1 |
| 8 | 63.68 | 14.74 | 0.26 | 47 | 0 |
| 9 | 63.85 | 14.91 | 0.17 | 44 | 0 |
| 10 | 63.94 | 15.00 | 0.09 | 48 | 0 |
| 11 | 63.94 | 15.00 | 0 | 50 | 0 |

Table 5: This table details the iterative progression, where $\Delta_1$ and $\Delta_2$ denote cumulative and marginal gains, respectively. The synchronous convergence of both repair performance and valid node synchronization suggests that the self-evolution mechanism effectively constrains erratic LLM generation, ensuring steady knowledge growth.

## C Comparative Analysis with Baseline Methods

As summarized in Table 2, our proposed RA-CRF achieves a commanding Pass@1 score of **65.5%**, consistently outperforming all competitive baselines and establishing a new performance frontier. To elaborate on the empirical and theoretical advantages of RA-CRF, we systematically compare it against the baselines across three strategic dimensions:

- **vs. Introspective Self-Refinement (SELF-REFINE):** SELF-REFINE(Madaan et al., 2023) achieves an accuracy of only 26.5%. While its "generate-feedback-refine" closed loop offers strong generality without training, the feedback itself remains bounded by the model's intrinsic parametric knowledge. RA-CRF surpasses it by a massive 39.0%, confirming that relying solely on model-generated introspection inherently suffers from error propagation and secondary regressions ("repairing for the worse"). In contrast, our approach grounds the repair process in deterministic execution verification and structured ERKG guidance.

- **vs. Structurally-Oriented Optimization (CODEXITY):** CODEXITY(Kim et al., 2024) focuses on code complexity metrics (e.g., cyclomatic complexity, nesting depth) to guide refactoring, achieving 60.9%. Although minimizing complexity enhances code maintainability, structural simplification does not inherently guarantee algorithmic correctness. RA-CRF yields a 4.6% performance lift over it, demonstrating that resolving functional bugs requires an explicit understanding of error-to-repair causal dependencies rather than purely aesthetic or structural code reorganization.

- **vs. Static Execution Feedback or Specialized Mitigation (RESCUE & SECCODER):** Both RESCUE(Shi and Zhang, 2025) and SECCODER(Zhang et al., 2024a) represent stronger baselines, scoring 60.9% and 61.7% respectively. RESCUE incorporates dynamic execution traces, but its flat feedback stream lacks macroscopic abstraction, leading to high iterative computational costs. SECCODER utilizes a multi-stage feedback mechanism with pre-defined security rules, which inherently biases the model toward overly conservative code generation strategies. RA-CRF maintains a distinct advantage over both, proving that organizing dynamic execution feedback into a self-evolving knowledge topology (ERKG) successfully decouples error diagnosis from code editing, achieving higher repair quality without architectural restrictions or expensive trial-and-error overhead.