

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

Haoyue Ran^{1,2}, Junguo Zhu^{1,2,†}, Hongyu Ge^{1,2}

¹Faculty of Information Engineering and Automation,
Kunming University of Science and Technology

²Key Laboratory of Artificial Intelligence in Yunnan Province,
Kunming University of Science and Technology
jg.zhu.hit@qq.com
ranhaoyue1@stu.kust.edu.cn

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 single-function benchmarks including BigCodeBench demonstrate that RA-CRF achieves significant performance gains across various open-source and closed-source models. Notably, RA-CRF lifts DeepSeek-V3 to 65.52% Pass@1, a new best on the official BigCodeBench leaderboard. 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 (Rozière et al., 2023), and GPT-5 (OpenAI,

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., 2024; Khoury et al., 2023; Lai et al., 2023). These failure modes still exist in modern benchmarks such as HumanEval (Chen et al., 2021), MBPP (Austin et al., 2021), APPS (Hendrycks et al., 2021), and BigCodeBench (Zhuo et al., 2025), 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., 2018; Chen et al., 2023), but this requires additional training overhead. Recent work has leveraged the inference capabilities of LLMs for self-debugging (Pan et al., 2024), iterating and improving based on execution feedback (Madaan et al., 2023; Chen et al., 2024; Shinn et al., 2023), and further integrating program-level signals (e.g., failed tests, runtime tracing) to guide code revisions (Le Goues et al., 2012; Liu et al., 2019; Xia et al., 2023), bringing more optimization possibilities to the code generated by LLMs.

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.

*Code and ERKG artifacts are available at <https://github.com/SweaterGuai/RA-CRF>.

[†]Corresponding author: jg.zhu.hit@qq.com.

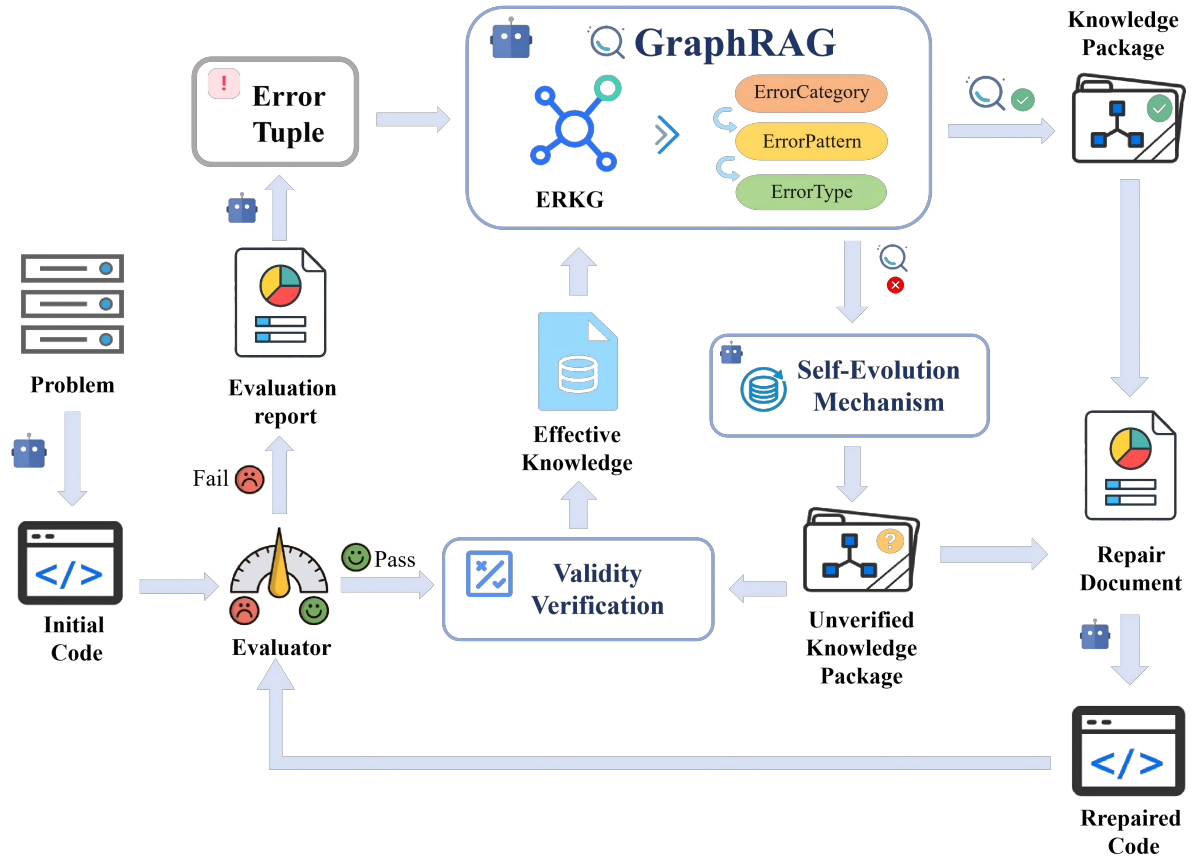


Figure 1: Overview of the RA-CRF framework. The robot icons indicate processes driven by the LLM. In our system, all these tasks are executed by a single unified model instance to ensure consistency.

- Inability to capture systematic error types. While LLMs consistently exhibit recurring failure 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 gap 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 attains 65.52% Pass@1 on BigCodeBench, exceeding the prior best

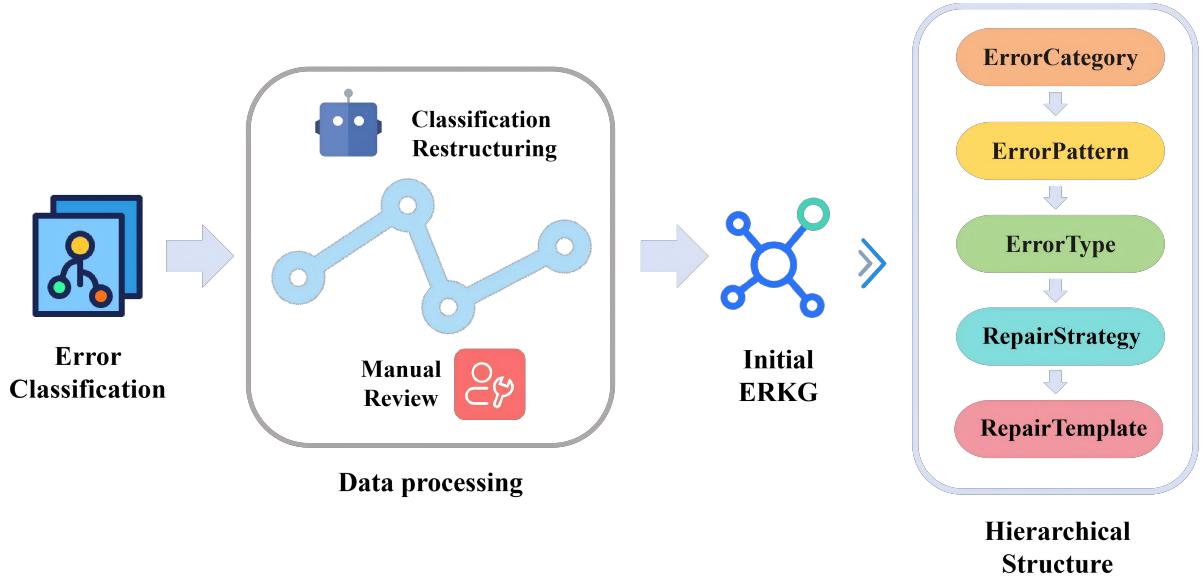


Figure 2: ERKG Construction Workflow. Detailed reconstruction prompts and node definitions are provided in Figure 6 and Table 4.

result on the official leaderboard and establishing a strong baseline for future research in autonomous code repair.

2 Related Work

2.1 LLM for Code Generation and Repair

Although code-centric LLMs (Chen et al., 2021; Rozière 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 work combines classic or LLM-based APR (Le Goues et al., 2012; Liu et al., 2019; Xia et al., 2023) with test-guided self-refinement (Madaan et al., 2023; Chen et al., 2024; Shinn et al., 2023), yet repair knowledge typically remains implicit in unstructured prompts rather than an explicit, reusable structure. Recently, security-oriented code generation frameworks have emerged. For instance, RESCUE (Shi and Zhang, 2025) retrieves hierarchical security knowledge to mitigate vulnerable generations, while CODEXITY (Kim et al., 2024) uses static-analysis feedback to iteratively patch insecure code. However, they target security properties rather than organizing functional execution failures into reusable error-repair knowledge.

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; Arslan et al., 2024). For example, SECCODER (Zhang et al., 2024) retrieves whole-file secure code examples for in-context learning, but such flat context lacks repair-oriented organization. In contrast, our hierarchical ERKG supports selective retrieval of error-repair mappings rather than undifferentiated chunk concatenation.

2.3 Error Analysis and Taxonomies in Code Generation

Prior work has analyzed LLM execution failures and proposed various error taxonomies (Just et al., 2014; Lin et al., 2017; Bogaerts et al., 2024), and fix-pattern mining extracts recurring patch templates from historical repositories (Koyuncu et al., 2020; Liu et al., 2019). However, these analyses remain largely descriptive and are rarely linked to runtime error signals in a closed-loop repair system. 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.

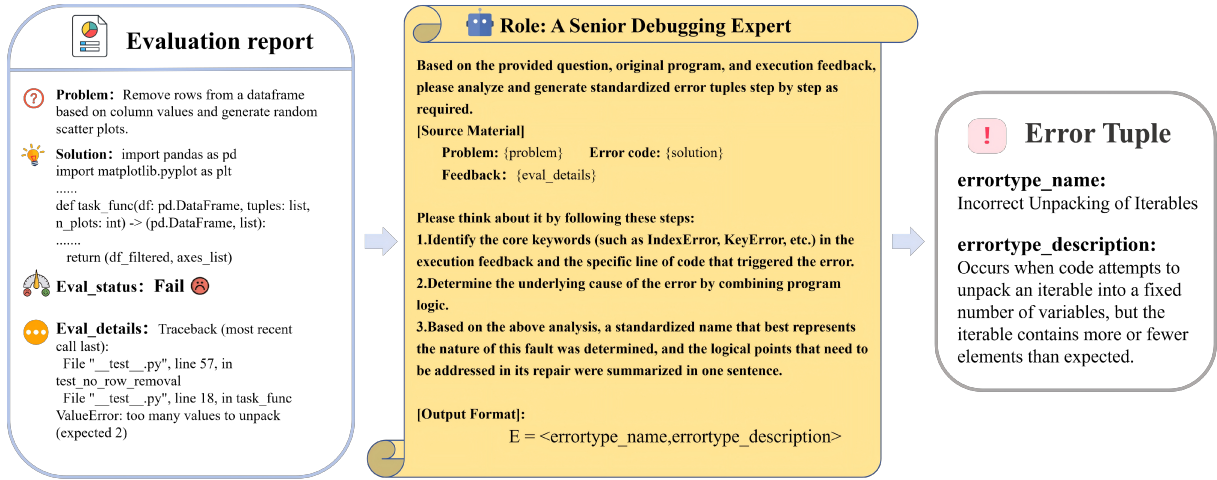


Figure 3: Example of generating an error tuple.

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 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 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 LLMs 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

seed graph stores abstract repair patterns only and does not include BigCodeBench or HumanEval-Pack task identifiers, canonical solutions, or full patches. Concrete node and template examples are given in Appendix A.1 (Example A). 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. To address the challenge of heterogeneous execution feedback formats and varying granularities across different benchmarks, we introduce an LLM as a semantic parser to extract core fault signals. The LLM is used to 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 $E = \langle \text{errortype_name}, \text{errortype_description} \rangle$. By filtering out environmental noise and mitigating semantic ambiguity, this process establishes a robust semantic foundation for precise downstream mapping with ERKG nodes.

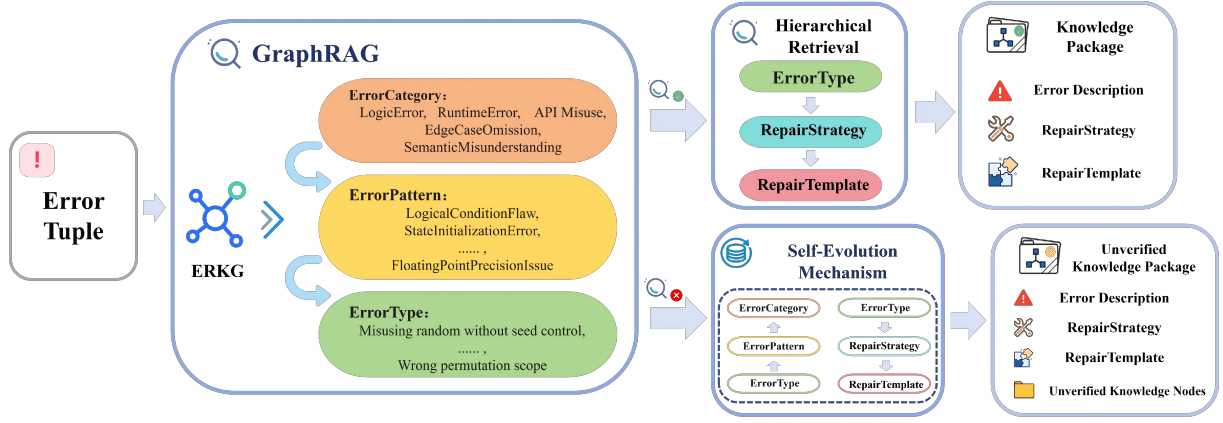


Figure 4: Illustration of the Hierarchical Retrieval and Self-Evolution Mechanism within the RA-CRF framework.

3.3 Knowledge-Guided Retrieval and Self-Evolution Mapping

To achieve accurate conversion from raw fault signals to structured repair solutions, we design a hierarchical graph retrieval and self-evolution mapping mechanism, 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.

3.3.1 Hierarchical Knowledge Mapping

Based on the standardized error tuples generated in the previous stage $E = \langle \text{errortype_name}, \text{errortype_description} \rangle$, 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 Algorithm 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. The 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, the LLM induces the generation of an 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 the retrieved knowledge 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

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 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,

Algorithm 1: Hierarchical Retrieval and Self-Evolution Mechanism.

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

// Notation:
 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;
executes a top-down hierarchical matching strategy.

if Found $ET_{matched}$ then
Continue searching for RS and RT under the
matching $ET_{matched}$ and execute 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

thereby significantly suppressing irrelevant, redundant modifications or hallucination-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 local edits, 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

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.

failures capture error trajectories as negative feedback to refine subsequent repair schemes. As illustrated in Table 5 for Qwen3-Coder-30B-A3B, 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. Accepted vs. rejected graph updates, including conflict/dedup handling, are illustrated in Appendix A.1 (Examples B–C). 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., 2025) as the primary benchmark. Unlike earlier datasets like HumanEval (Chen et al., 2021) 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

Models	BigCodeBench			HumanEvalPack		
	Initial	R1	R3	Initial	R1	R3
The closed-source models						
GPT-4o	51.66	62.63	65.87(↑14.21)	92.07	93.29	96.34(↑4.27)
GPT-5	51.49	60.70	66.84 (↑15.35)	90.24	93.90	96.95 (↑6.71)
The open-source models						
Qwen2.5-14B-Instruct	43.94	50.17	53.68(↑9.74)	79.26	81.09	87.80(↑8.54)
Qwen2.5-32B-Instruct	39.12	47.28	52.10(↑12.98)	85.97	86.58	88.41(↑2.44)
Qwen2.5-72B-Instruct	47.54	54.21	58.85(↑11.31)	83.53	86.58	89.63(↑6.10)
Qwen3-Coder-30B-A3B-Instruct	48.94	56.92	61.14(↑12.20)	92.07	92.68	93.29(↑1.22)
Llama-3.1-70B-Instruct	46.22	54.82	57.63(↑11.41)	84.14	87.19	88.41(↑4.27)
Llama-4-Maverick	45.96	51.05	55.26(↑9.30)	85.36	89.63	91.46(↑6.10)
DeepSeek-V3	50.17	59.91	65.52 (↑15.35)	89.63	92.07	93.90 (↑4.27)

Table 1: Experimental results on BigCodeBench (Full) and HumanEvalPack. For BigCodeBench, initial code is generated via the official `instruct_prompt`. Initial, R1, and R3 report Pass@1 before repair and after the first and third repair rounds, respectively. Bold marks the best result in each column within closed-source and open-source models, respectively.

development scenarios.

Furthermore, we used the Python subset of HumanEvalPack (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.

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; Qwen Team, 2025), Llama series (Grattafiori et al., 2024; Meta AI, 2025), and DeepSeek-V3 (DeepSeek-AI, 2024), the strongest open-source model in our suite. Closed-source models included GPT-4o (OpenAI, 2024) and GPT-5 (OpenAI, 2026). None of the models were fine-tuned. Decoding used temperature 0.0 for all calls. The maximum number of generated tokens was set per pipeline stage and kept consistent across models for the same stage. Inference cost and call budgets for the main BCB run are reported in Appendix D.

4.3 Metrics

We adopt Pass@1 (Chen et al., 2021) as the sole evaluation metric: for each task, one program is

submitted and scored by the official unit-test harness. We report Pass@1 at three protocol stages on the same task set. Initial is Pass@1 of the first generated program before repair. R1 and R3 are Pass@1 after at most one and at most three repair rounds, respectively, with early exit when tests pass. The maximum number of repair rounds is three. For every task we submit exactly one final program (the earliest passing candidate if repair succeeds, otherwise the last repaired attempt). Thus R1/R3 measure cumulative repair progress under Pass@1 rather than separate sampling-based Pass@ k .

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 BigCodeBench, RA-CRF lifts DeepSeek-V3 to 65.52% Pass@1, a new best on the official leaderboard (as of May 2026). 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.

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 Hu-

Method	Pass@1
<i>Matched repair setting</i>	
Self-Refine (Madaan et al., 2023)	26.5
RA-CRF (Ours)	65.52
<i>Original setting</i>	
SecCoder (Zhang et al., 2024)	61.7
Codexity (Kim et al., 2024)	60.9
RESCUE (Shi and Zhang, 2025)	60.2

Table 2: Performance comparison on BigCodeBench (DeepSeek-V3). Matched: same repair protocol; Original: prior methods’ original settings. See Appendix C.

manEvalPack 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 possess a natural advantage in initial generation. 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 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 strongest open-source model in our suite, using the BigCodeBench dataset. Table 3 reports R1 ablations and a final full-system row at R3. Table 3 first tests whether repair gains require execution feedback and struc-

tured prompting via the Evaluation Report. Without that report, performance stays near Initial even when ERKG is present, so abstract graph patterns alone cannot infer the true failure mode. With feedback available, removing ERKG still yields a large gain at R1, confirming that execution signals anchor repair; structured ERKG knowledge and further rounds up to R3 then reach 65.52%, beyond feedback-guided trial-and-error.

Method Variant	Pass@1
Initial Generation	50.17
<i>Ablations (R1)</i>	
- 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)
- w/o Normalization	63.59(↑ 13.42)
- w/o Repair Document	60.96(↑ 10.79)
- w/o Hierarchical Retrieval	60.70(↑ 10.53)
- w/o Graph Updates	59.91(↑ 9.74)
<i>Full system (R3)</i>	
RA-CRF (Ours)	65.52(↑ 15.35)

Table 3: Ablation on BigCodeBench (Pass@1, DeepSeek-V3). Ablation variants are at R1; the final row is the full system at R3. All differences are relative to Initial Generation.

We further ablate normalization, repair-document formatting, hierarchical retrieval, and graph updates while keeping evaluation feedback and ERKG. This leave-one-out design at R1 asks whether gains rely only on the feedback-ERKG coupling, or also on these organizational choices. Because the core pathway remains intact, every variant stays far above Initial (Table 3), showing that ERKG knowledge remains useful even from raw evaluation reports and is not an artifact of any single formatting, retrieval, or update choice. Removing normalization yields the strongest score among these variants, suggesting that normalization mainly prepares feedback for hierarchical retrieval in later rounds rather than boosting accuracy at R1 alone; removing the structured repair document lowers performance more noticeably, indicating that explicit formatting helps translate retrieved patterns into edits. Removing hierarchical retrieval cuts LLM calls while remaining competitive at R1, consistent with the modest inference budget in Appendix D. Removing graph updates matches the full system at R1 in Table 1, as verified writes occur between rounds; the gain from R1 to R3 therefore reflects

iterative repair and validated graph evolution.

In summary, the ablations show a clear division of roles. Execution feedback is a prerequisite for meaningful repair. Once failures are localized, ERKG provides an actionable knowledge pathway, while normalization, document formatting, hierarchical retrieval, and verified graph updates refine this coupling rather than replace it. The full system at R3 delivers the best Pass@1 (65.52%). Even with this strong backbone, execution feedback and ERKG remain necessary for the observed gains, motivating future work on internalizing such external feedback through distillation or reinforcement learning.

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, our evaluation is confined to single-function, benchmark-style repair tasks with provided unit tests; extending RA-CRF to repository-level, multi-file issue fixing remains unvalidated future work. Second, the current language coverage is restricted to Python; the generalizability to other programming paradigms remains to be explored. Third, 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.

Acknowledgments

This work is supported by the Regional Fund Project of the National Natural Science Foundation of China (62166022), the Yunnan Xingdian Talent Support Plan Project (KKXX202403023), the Guangdong Province Basic and Applied Basic Research Fund (Guangdong-Foshan Joint

Funds, 2024A1515140026), and the Social Science Foundation of Guangdong Province, China (No. GD24CZY02).

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 I. Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie J. Cai, Michael Terry, Quoc V. Le, and Charles Sutton. 2021. [Program synthesis with large language models](#). *CoRR*, abs/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. 2023. [Improving code generation by training with natural language feedback](#). *CoRR*, abs/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 Schaerli, and Denny Zhou. 2024. [Teaching large language models to self-debug](#). In *International Conference on Learning Representations*, volume 2024, pages 8746–8825.
- DeepSeek-AI. 2024. [Deepseek-v3 technical report](#). *CoRR*, abs/2412.19437.
- Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, and Alex Vaughan. 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 Niranjana 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 Jacob Steinhardt. 2021. [Measuring coding challenge competence with APPS](#). In *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks 1, NeurIPS Datasets and Benchmarks 2021, December 2021, virtual*.
- 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, Kai Dang, An Yang, Rui Men, Fei Huang, Xingzhang Ren, Xuancheng Ren, Jingren Zhou, and Junyang Lin. 2024. [Qwen2.5-coder technical report](#). *CoRR*, abs/2409.12186.
- 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](#). *CoRR*, abs/2405.03927.
- Anil Koyuncu, Kui Liu, Tegawendé F Bissyandé, Dong-sun 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. 2012. 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, Qian Liu, Evgenii Zheltonozhskii, Terry Yue Zhuo, Thomas Wang, Olivier Dehaene, Mishig Davaadorj, Joel Lamy-Poirier, João Monteiro, Oleh Shliazhko, and 48 others. 2023. [StarCoder: may the source be with you!](#) *Trans. Mach. Learn. Res.*, 2023.
- 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.
- 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 Wiegrefe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. 2023. [Self-refine: Iterative refinement with self-feedback](#). In *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*.
- Meta AI. 2025. The Llama 4 herd: The beginning of a new era of natively multimodal AI innovation. <https://ai.meta.com/blog/llama-4-multimodal-intelligence/>. Accessed: 2026-08-26.
- 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.
- OpenAI. 2024. [Gpt-4o system card](#). *CoRR*, abs/2410.21276.
- OpenAI. 2026. [Openai GPT-5 system card](#). *CoRR*, abs/2601.03267.
- 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.
- Qwen Team. 2025. [Qwen3 technical report](#). *CoRR*, abs/2505.09388.
- Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, Artyom

- Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton-Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, and 6 others. 2023. [Code llama: Open foundation models for code](#). *CoRR*, abs/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.
- 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. 2018. [Dynamic neural program embeddings for program repair](#). In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net.
- 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.
- Boyu Zhang, Tianyu Du, Junkai Tong, Xuhong Zhang, Kingsum Chow, Sheng Cheng, Xun Wang, and Jianwei Yin. 2024. Seccoder: Towards generalizable and robust secure code generation. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 14557–14571.
- 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 Widayarsi, 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 Kadour, Ming Xu, Zhihan Zhang, and 2 others. 2025. [Bigcodebench: Benchmarking code generation with diverse function calls and complex instructions](#). In *The Thirteenth International Conference on Learning Representations, ICLR 2025, Singapore, April 24-28, 2025*. OpenReview.net.

A ERKG Construction 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.

A.1 Concrete ERKG Walkthroughs

Example A (BCB/23): hit seed retrieval	
Setting	Round-1 retrieval on the <i>initial</i> ERKG; no in-round write
Failure	Empty l1/l2; tests report None != 10 and missing ValueError
Logged path	Type: Missing empty input guard Strategy: Add guard clause for empty input Template: if not {input_var}: return 0.0 Seed parent: EdgeCaseOmission (Pattern: see B–C)
Instantiation	if not l1 and not l2: raise ValueError("Both lists empty.")
Outcome	Initial: fail; Round 1: pass
Note	<i>The seed template is a parametric skeleton (return 0.0); instantiation adapts it to the task (raise ValueError) rather than copying it verbatim. Seed retrieval follows Type→Strategy→Template in Round 1 (fail→pass) without writing the graph.</i>
Example B (BCB/35): rejected update / failure	
Setting	Round 1 proposal logged but not persisted
Failure	TypeError: 'Axes' object is not subscriptable
Proposed path	Category: RuntimeError Pattern: CollectionOperationError Type: Incorrect subplot axes handling Strategy: Properly handle subplot axes Template: wrap single-subplot axes as a list
Decision	Skip (pending_expansions_skipped_1): fail→fail; semantic overlap ⇒ merge, no duplicate Type
Outcome	Round 1: still fail
Note. Failed or near-duplicate proposals are skipped and do not pollute ERKG.	
Example C (BCB/828): accepted update	
Setting	Write only after verified fail→pass
Failure	Round 1 pre-repair: FileNotFoundError on missing source file
Proposed path	Category: RuntimeError Pattern: FileOperationError Type: NonExistentFileOperation Strategy: Check file existence before operation Template: if not os.path.exists({file_path}): raise ...
Instantiation	if not os.path.exists(src): raise FileNotFoundError(...)
Decision	Apply (pending_expansions_applied_1) after Round 1 pass; abstract nodes only
Outcome	Round 1: pass → applied
Note. An abstract file-existence pattern is written only after Round 1 fail→pass.	

B Validation of Iterative Convergence

Repair Round	Pass@1 (%)	Δ_1	Δ_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: Iterative progression on Qwen3-Coder-30B-A3B. Δ_1 and Δ_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 65.52% Pass@1, outperforming the listed baselines under these settings. Table 2 groups Self-Refine with RA-CRF under a matched repair setting, and reports SecCoder, Codexity, and RESCUE under their original evaluation settings. To elaborate on these empirical differences, 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 39.02 percentage points, 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. Static-Analysis Security Repair (CODEXITY):** CODEXITY (Kim et al., 2024) integrates LLMs with static analyzers to iteratively reduce security vulnerabilities, achieving 60.9%. Although such feedback can suppress insecure patterns, static warnings do not guarantee functional correctness under unit tests. RA-CRF yields a 4.62 percentage-point lift over it, showing that resolving functional bugs requires explicit error-to-repair mappings rather than security-oriented static-analysis loops.
- **vs. Security-Oriented Retrieval and Specialized Mitigation (RESCUE & SECCODER):** Both RESCUE (Shi and Zhang, 2025) and SECCODER (Zhang et al., 2024) represent stronger baselines, scoring 60.2% and 61.7% respectively. RESCUE constructs a hierarchical security knowledge base and retrieves vulnerability-related guidance, but this knowledge is organized for secure generation rather than functional fail-to-pass repair. 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, indicating that organizing functional execution feedback into ERKG yields higher Pass@1 than security-oriented retrieval or static-analysis loops.

D Computational Cost and Inference Budget

This appendix reports the inference cost of RA-CRF on the main BigCodeBench experiment with DeepSeek-V3 (DeepSeek-AI, 2024) under the same decoding settings as Section 4. For the Full RA-CRF run on 1,140 tasks at R3, each failing task uses execution feedback from the BCB harness and at most six LLM calls per round: feedback normalization, up to three hierarchical retrieval calls, repair-document synthesis, and localized editing, with early exit on pass. API usage for this run is about 7.79M tokens and \$3.31 in total. Averaged over 1,140 tasks, this corresponds to about 6.8K tokens and \$0.003 per task. Wall-clock time is about 19.5 h. Relative to the benchmark scale and the multi-round repair setting, this overhead is modest.