ERKG: Error–Repair Knowledge Graph for LLM-Based Code Repair

Anonymous ACL submission

Abstract

Large language models have demonstrated strong capabilities in automatic code generation tasks, but the programs they generate still often fail in actual execution due to logical errors or missing boundary conditions. To address this issue, we propose a retrieval-augmented code repair framework that explicitly incorporates structured error–repair knowledge to improve the execution reliability of code generated by large language models. Unlike previous methods that mainly relied on unstructured self-reflection or original execution feedback, this paper constructs a hierarchical knowledge graph from error to repair operations. Based on the execution feedback, the framework retrieves relevant knowledge and generates structured repair documents, guiding the model to make minimal and targeted code modifications. Experimental results on BigCodeBench show that the proposed method can absolute improve the PASS@1 index by 5.0-13.5 percentage points on a variety of open-source and closed-source large language models. It is indicated that explicitly introducing error-repair knowledge graph into the code repairing process can significantly enhance the reliability and stability of the repairs.

1 Introduction

Large language models (LLMs) have demonstrated strong capabilities in automatic code generation, enabling applications such as competitive programming, software development, and intelligent tutoring. Recent models such as Codex (Chen, 2021), StarCoder (Li et al., 2023), Code Llama (Roziere et al., 2023), and GPT-4 (Achiam et al., 2023a)exhibit high fluency in translating natural language into executable programs. However, LLM-generated code remains unreliable for complex algorithmic tasks that require rigorous logical correctness and comprehensive test coverage. Even state-of-the-art models frequently fail hidden tests, mishandle edge cases, or encounter runtime errors (Liu et al., 2023; Chen et al., 2023b; Khoury et al., 2023; Lai et al., 2023). These failure modes persist across modern benchmarks—including HumanEval-V (Zhang et al., 2024), MBPP, APPS (Hendrycks et al., 2021), and BigCodeBench (Zhuo et al., 2024),highlighting the pressing need to improve the reliability of LLM-generated code. This challenge is further compounded by limitations in current benchmarks, such as inadequate test coverage and ambiguously specified problems.

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), albeit requiring additional training. Others leverage LLMs for self-debugging(Pan et al., 2024), where execution feedback is used to iteratively refine outputs (Koyuncu et al., 2020; Madaan et al., 2023; Chen et al., 2023b; Shinn et al., 2023). Execution-guided methods (Chen et al., 2018; Haque et al., 2025) further incorporate program-level signals (e.g., failed tests, runtime traces) to steer revision. More recently, neural patch generation techniques (Le Goues et al., 2011; Liu et al., 2019; Xia et al., 2023) frame LLMs as end-to-end repair agents.

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.
- Failure to capture systematic error patterns. LLMs consistently exhibit recurrent fault patterns across tasks but existing approaches treat

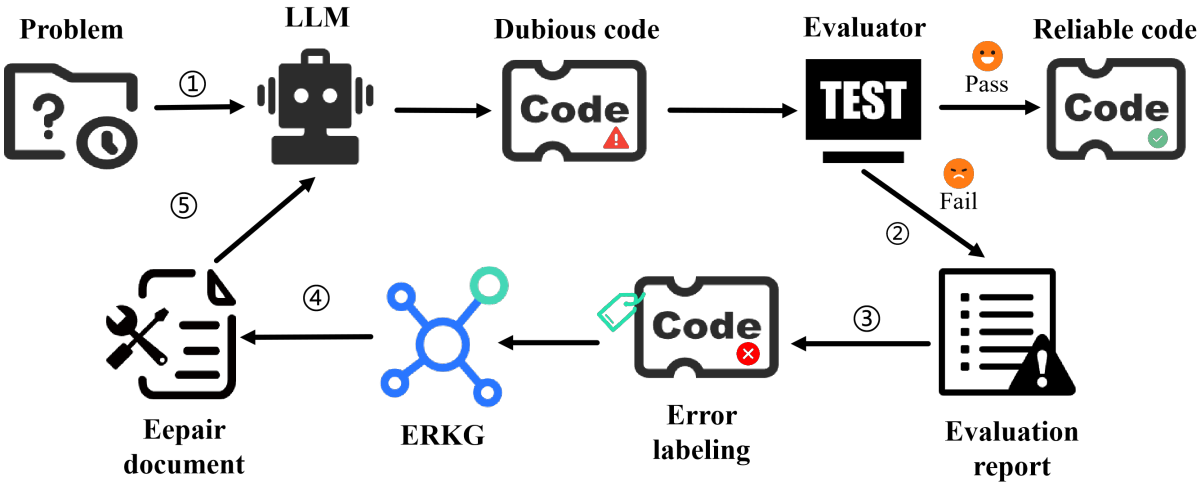

Figure 1: RA-SRF Overview: Given a programming task, an LLM first generates an initial solution from the natural-language description ①. The generated code is then executed under the benchmark's evaluation protocol to produce an evaluation report ②. Based on the initial code and execution feedback, the error is parsed and normalized into a standardized error label ③. The framework subsequently queries the error knowledge graph using the normalized label, retrieves relevant repair strategies and templates, and synthesizes a structured repair document ④. Guided by this document, the LLM performs targeted modifications to the initial solution, yielding the repaired program ⑤.

each failure in isolation, ignoring cross-task regularities that could inform and constrain 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 corrections ?** To address this, we propose a novel Retrieval Enhanced Code Repair Framework (RA-CRF), which integrates error repair knowledge into LLM-based code repair by retrieving error repair knowledge from a hierarchical error knowledge graph and converting it into actionable repair guidelines. This results in more robust, accurate, and interpretable code correction.As shown in Figure 1.

Our contributions are as follows: 1) We introduce a hierarchical error repair knowledge graph that explicitly models the relationships between error categories, error types, repair strategies, and repair templates, which has not been explicitly formalized in previous LLM-based code repair work. 2) We propose a retrieval-augmented structured repair pipeline that integrates knowledge retrieval with structured repair document generation to enable stable, targeted, and interpretable code repairs. 3) We show that incorporating structured error–repair knowledge provides an effective inductive bias for LLM-based code repair, improving generalization across error types and enabling more controllable and verifiable repairs.

2 Related Work

2.1 LLM for Code Generation and Repair

LLMs such as Codex(Chen, 2021), Code Llama(Roziere et al., 2023), and StarCoder(Li et al., 2023) excel at code generation but often produce incorrect programs, especially on algorithmic or boundary-sensitive tasks(Seo et al., 2021; Liu et al., 2023; Khoury et al., 2023; Lyu et al., 2025; Hasan et al., 2025). To address this, recent work employs self-debugging(Koyuncu et al., 2020; Madaan et al., 2023; Chen et al., 2023b; Shinn et al., 2023) using error feedback to iteratively refine outputs or proposed CodeTrans(Elnaggar et al., 2021), will repair is modeled as a task sequence to sequence.However, these methods rely solely on the LLM's implicit reasoning over raw error signals, lacking explicit knowledge of error types or repair strategies. This leads to inconsistent, superficial repairs and occasional regressions.

2.2 Automated Program Repair (APR)

Traditional automatic Program repair (APR) methods typically rely on test cases to locate and repair defects(Arslan et al., 2024). Including search-based strategies (such as GenProg(Le Goues et al., 2011; Zhou et al., 2022)), semantic constraint meth-

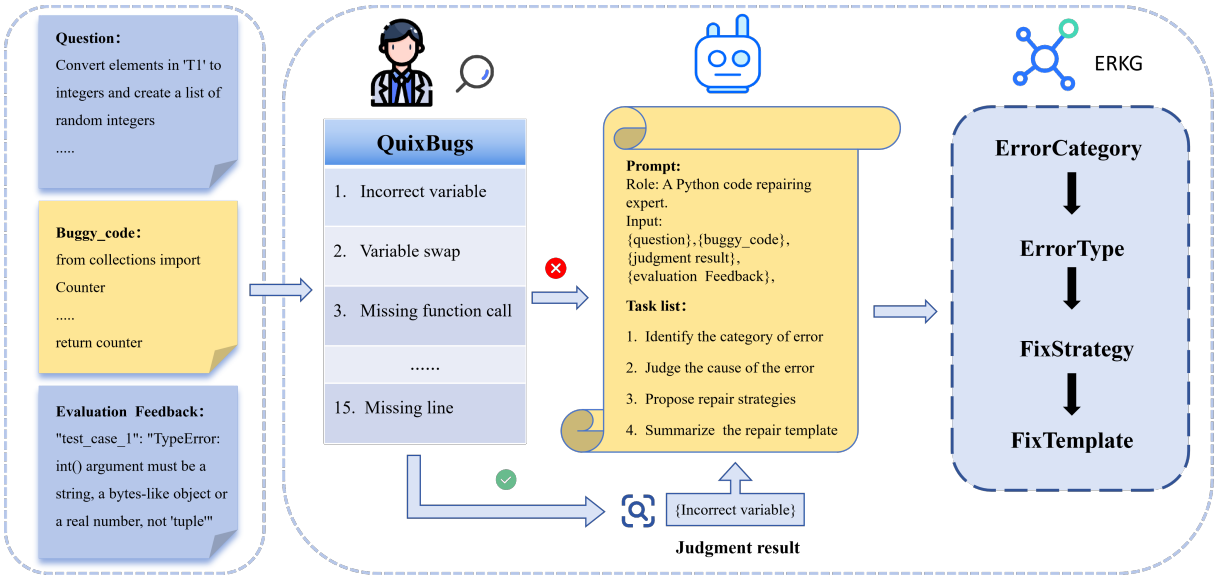

Figure 2: We construct the ERKG through a combination of human error annotation and LLM-assisted inference. For each buggy program, we first manually map its failure based on the problem description, source code, and execution feedback to one of the 15 predefined error types in QuixBugs, which serves as a reference signal for subsequent LLM reasoning. When no suitable match is identified, the LLM directly infers the error category, underlying cause, repair strategy, and fix template, thereby completing the four-level knowledge graph construction.

ods (such as SemFix(Nguyen et al., 2013; Wang et al., 2017)), and template-driven methods (such as TBar(Liu et al., 2019)) Recently, there have also been works introducing neural networks to repair(Tufano et al., 2019). These methods assume the existence of clear test oracles and error locations. Similarly, recent LLM will be used for repairing agent work(Xia et al., 2023; Gupta et al., 2020) is still using this premise, the modification of execution feedback or test failure.However, the code generated by LLMS often fails due to logical errors or the absence of boundary handling. The lack of clear oracle and location signals makes the traditional APR method of dependency testing difficult to apply.

2.3 Knowledge-Augmented LLMs for Code

Knowledge-augmented large language models for code typically rely on retrieving unstructured or weakly structured textual resources to assist code generation, such as technical documentation and code repositories on GitHub, discussion threads from programming Q&A communities like Stack Overflow and functionally or semantically similar code examples (Liu et al., 2019; Svyatkovskiy et al., 2020; Huang et al., 2024; Wang et al., 2025; Shaka et al., 2024). However, most of this knowledge is task-independent and lacks a structured organization oriented towards repair, making it difficult to provide fine-grained error location and repair guidance.

2.4 Error Analysis and Taxonomies in Code Generation

Recent research has analyzed the failures of LLM code on benchmark tests and the problems that occur in real programming scenarios, and proposed a classification of errors(Martin et al., 2011; Just et al., 2014; Lin et al., 2017). However, these analyses remain largely descriptive to a large extent, and their error categories are rarely manipulated into actionable repair mechanisms. Our work Bridges this gap by mapping error types to specific repair strategies and templates in structured knowledge graphs, transforming diagnostic insights into 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 fixing. Unlike previous methods that rely on unstructured self-reflection or raw execution feedback, our approach introduces a hierarchical bug fixing knowledge graph (ERKG) to provide systematic and controlled guidance throughout the repair process. The overall framework is shown in Figure 1.

Node Type	Count	Description	Example
ErrorCategory	5	High-level conceptual groups of LLM-specific code errors, such as exceptions, assertion violations, output errors, and timeouts.	"LogicError"
ErrorType	41	Concrete manifestations of failures within each category.	"Incorrect data type return"
RepairStrategy	41	Abstract repair actions that address a given error type.	"Cast the result to `float()` or use `0.0` instead of `0` to satisfy the return type requirement."
RepairTemplate	41	Standardized code templates for repairs.	"return float({result})"

Table 1: The definitions of each node in the Error–Repair Knowledge Graph and an example of an error under "LogicError".

3.1 Error-Repair Knowledge Graph (ERKG) Construction

3.1.1 Motivation and Design Principles

Most existing LLM-based code repair methods treat execution feedback as unstructured text. While useful, such feedback offers only limited guidance: similar error symptoms often stem from different root causes and require distinct repairs. This ambiguity obscures the mapping from observed failures to actionable repair strategies, diminishing the role of structured knowledge in the repair process.

To bridge this gap , we have constructed a hierarchical error repair knowledge graph (ERKG), which explicitly encodes the structured relationships among program errors and their corresponding repair mechanisms , particularly linking error categories, specific error types, repair strategies, and standardized repair templates. As a structured intermediary between the original execution feedback and code-level editing, ERKG enables LLMS to perform systematic and controllable reasoning during the repair process.

3.1.2 Graph Construction Process

Our Error Repair Knowledge Graph (ERKG) is inspired by the error classification method proposed in QuixBugs(Lin et al., 2017)and is built based on a systematic analysis of fault cases in programs generated by LLM on BigCodeBench(Zhuo et al., 2024). The specific process is shown in Figure 2.

At the Error Category level, we collect failure instances from execution feedback and abstract them into high-level error categories according to their semantic characteristics, such as exception types, failure locations, and observed runtime behaviors. At the Error Type level, we further analyze the underlying causes of these failures and consolidate recurring patterns into a unified set of fine-grained error types.At the Repair Strategy level, we associate each error type with one or more high-level repair strategies, distilled from prior code repair literature and common human debugging practices. Finally, at the Repair Template level, we operationalize each repair strategy as a parameterized code-editing template, enabling it to be instantiated across different programs and contexts.

Throughout the build process, we placed particular emphasis on designing repair templates to minimize interference with correct code regions while maintaining semantic correctness, thereby achieving reliable and reusable repairs across tasks. To ensure a clear graph structure, we performed redundancy elimination and syntax validation on the knowledge being built. The final ERKG contains 5 error categories, 41 error types, 41 repair strategies, and 41 repair templates, forming a compact and well-defined hierarchical structure, as shown in Table 1. Each error type is associated with a corresponding repair strategy, ensuring end-to-end coverage from error diagnosis to specific code-level editing.

3.2 Error extraction and normalization

Effective knowledge-guided remediation requires transforming raw execution feedback into a structured error representation so that it can be reliably mapped to an error remediation knowledge graph.However, execution outputs generated by different benchmarks and runtime environments are highly heterogeneous in format and granularity ranging from brief exception messages to detailed test logs.To address this challenge, we designed a unified error extraction and normalization process to transform heterogeneous feedback into structured remediation guidelines, as shown in Figure 3.

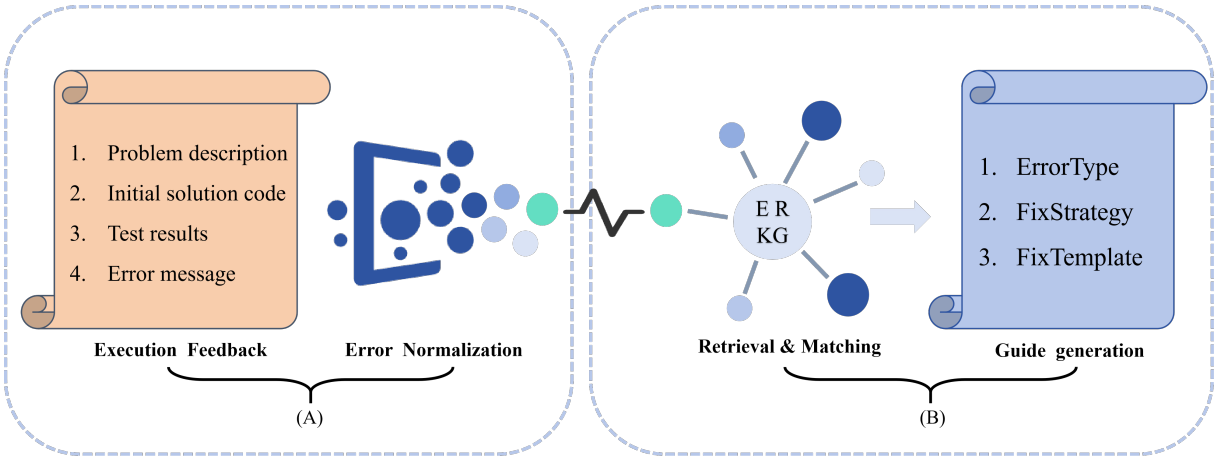

Figure 3: Transforming heterogeneous feedback into a structured remediation guide is implemented in two phases. In Phase A, we first collect the execution feedback generated during benchmarking and extract error information(3.2.1), normalizing it into standardized error labels(3.2.2). In Phase B, the standardized error label is used for node matching and retrieval in the Error-Remediation Knowledge Graph (ERKG) to obtain the corresponding error types, remediation strategies, and remediation templates(3.2.3,3.3.1).These are then integrated into a structured remediation knowledge package for subsequent code remediation(3.3.2).

3.2.1 Error Label Extraction

Given an initial candidate program, we execute it under the benchmark's evaluation protocol and extract structured error signals from the resulting execution feedback. For each failed test case, we record the corresponding exception or failure type. In addition, we capture contextual information including the location of the faulty code region and salient execution traces which together are used to construct a standardized error label for subsequent normalization and retrieval.

3.2.2 Error Normalization

The original error signal usually contains noise, redundancy or input-related details. To support efficient knowledge graph retrieval, we normalize it into a standardized error label, which retains the fundamental fault features while abstracted irrelevant details. Normalization consists of two steps: (1) Message simplification: Remove variable values, memory addresses, or input dependencies that have no repair value; (2) Context disambiglement: By using context cues such as the location of the fault or the point of assertion violation, semantically similar errors are distinguished and mapped to the most matching error type in the ERKG. The final error label is represented as a tuple [error type, context tag] , serving as an interface between execution feedback and structured knowledge retrieval.

3.2.3 Mapping to the Error Knowledge Graph

The standardized error labels are subsequently matched against nodes in the Error–Repair Knowledge Graph (ERKG). For labels that directly correspond to predefined error types, the mapping is performed deterministically. When an exact match is unavailable, candidate error types are identified using a set of heuristic rules and contextual features, including exception semantics, failure locations, and code-structure cues. If the error label cannot be uniquely resolved, multiple candidate mappings are retained and disambiguation is deferred to the downstream retrieval and repair planning stage. This design avoids premature error commitment and preserves flexibility during repair. By transforming heterogeneous execution feedback into structured, ERKG-compatible error labels, the normalization pipeline provides a reliable foundation for knowledge retrieval and subsequent ERKG-guided repair.

3.3 ERKG-Guided Retrieval via RAG

Given standardized error labels, we retrieve relevant repair knowledge from the Error-Repair Knowledge Graph (ERKG) and integrate it into the LLM repair process. Specifically, we adopt the Graph-RAG paradigm for retrieval enhancement generation, enabling repair decisions to be based on explicit structured error knowledge rather than free text context.

3.3.1 Graph-Based Knowledge Retrieval

Retrieval over the ERKG is performed through constrained graph traversal rather than unstructured similarity search. Starting from the matched Error Type node, we deterministically traverse the graph to collect the associated Repair Strategy and Fix Template nodes along the predefined hierarchy. By explicitly encoding error–repair relationships and relying on context-aware heuristic ranking, ERKG-based retrieval remains interpretable and controllable, in contrast to opaque embedding-based retrieval approaches.

3.3.2 Retrieved Knowledge Package

The output of the retrieval stage is a structured knowledge package that summarizes the retrieved content and is used for downstream reasoning. Each package includes: (i) the inferred error type, (ii) the corresponding fix strategy, and (iii) the corresponding fix template represented in an abstract code pattern with placeholders. This representation is deliberately compressed to minimize prompt overhead while retaining actionable repair guidance. Then, the retrieved knowledge package, along with the original program and execution feedback, is passed to the maintenance planning stage. By separating knowledge retrieval from generation, this framework ensures that subsequent LLM reasoning is based on explicit domain knowledge rather than relying solely on implicit model priors.

3.4 Controllable Code Repair via Structured Guidance

Directly prompting an LLMS to rewrite code based on error feedback and retrieval knowledge often leads to uncontrollable edits and inconsistent repairs. To address this issue, we introduced a structured repair document as an explicit intermediate representation, formalizing the repair intent before code modification. This document is generated by the LLM based on three parts of input: (1) the original program, (2) execution feedback and standardized error labels, and (3) the structured knowledge package retrieved from the ERKG. LLM generates structured repair documents strictly based on three parts of input. This mechanism effectively inhibits the repair of hallucinations and irrelevant changes. In the final stage, the repairing document guides the LLM to perform local code editing . Only modify the error areas marked in the document, rather than rewriting the entire program. This will separate the repair plan from the code editing, ensuring that the modifications are precise, controllable and minimized.

4 Experimental Setup

4.1 Datasets

We chose BigCodeBench (BCB)(Zhuo et al., 2024) as the main experimental dataset because it is specifically designed to evaluate the execution correctness and robustness of large language models in real programming scenarios, and is particularly suitable for code repair research. Unlike early benchmarks such as HumanEval(Zhang et al., 2024) or the original MBPP(Austin et al., 2021), which focused on short algorithmic problems and rarely involved library calls, BCB tasks originated from real code repositories, covering tool usage, API calls, and cross-library interactions, making them closer to actual development.

To comprehensively assess the generalization ability of the method in terms of natural language instruction understanding and semantic alignment, we supplement the use of MBPP-sanitized(Austin et al., 2021). This version systematically cleans the original MBPP: it corrects ambiguous or ambiguous descriptions, inconsistent test cases, and enhances boundary scenario coverage, thereby providing more reliable and discriminative execution feedback.

4.2 Models

We evaluate our framework across different code capabilities LLMS (7B-480B), spanning both open-source and closed-source families. For open-source models, we fine-tune the versions using instructions from Qwen (Hui et al., 2024; Yang et al., 2025)and DeepSeek(Liu et al., 2024, 2025): Qwen2.5-coder (7B, 32B), Qwen2.5 (72B), Qwen3-Coder (30B, 480B) Qwen3-Next (80B) and Deepseeking v3 / v3.2 both run in a fully replicable environment to analyze how scale and architecture affect the repairs. For closed-source models, we conducted tests on the OpenAI system: gpt-4o(Hurst et al., 2024), GPT-4 Turbo (2024-04-09)(Achiam et al., 2023b), GPT-4.1(Fachada et al., 2025), and GPT-5.1, representing the most advanced proprietary code generators.

In all the experiments, the same model was used for initial generation and repair to ensure that improvements only came from our knowledge guidance framework, rather than model switching or capability stacking. All models run in inference

Models	BigCodeBench		MBPP-sanitized	
	Initial code	Modified code	Initial code	Modified code
The closed-source models				
GPT4o	51.66	61.14(↑ 9.48)	86.18	87.58(↑ 1.40)
GPT-4-turbo-2024-04-09	49.64	61.14(↑ 11.50)	85.01	89.22(↑ 4.21)
GPT-4.1-2025-04-14	50.43	61.14(↑ 10.71)	84.77	**89.46(↑ 4.69)**
GPT-5.1-chat-2025-11-13	50.87	**64.29(↑ 13.42)**	83.84	88.52(↑ 4.68)
The open-source models				
Qwen2.5-Coder-7B-Instruct	39.12	45.17(↑ 6.05)	81.49	83.37(↑ 1.88)
Pro-Qwen2.5-Coder-7B-Instruct	38.77	44.21(↑ 5.44)	82.66	84.07(↑ 1.41)
Qwen2.5-Coder-32B-Instruct	39.12	48.50(↑ 9.38)	81.49	83.37(↑ 1.88)
Qwen2.5-72B-Instruct	47.54	55.70(↑ 8.16)	85.48	88.29(↑ 2.81)
Qwen3-Coder-30B-A3B-Instruct	48.94	56.75(↑ 7.81)	83.37	85.94(↑ 2.57)
Qwen3-Coder-480B-A35B-Instruct	39.82	52.80(↑ **12.98**)	85.24	88.05(↑ 2.81)
Qwen3-Next-80B-A3B-Instruct	50.26	60.17(↑ 9.91)	84.54	90.16(↑ **5.62**)
DeepSeek-V3	50.17	**61.92**(↑ 11.75)	87.11	88.75(↑ 1.64)
DeepSeek-V3.2	46.92	58.50(↑ 11.58)	86.18	**90.86**(↑ 4.68)

Table 2: The experimental results of each model using RA-SRF on the BigCodeBench and MBPP-sanitized datasets are presented. The bolded sections indicate the best performance and the largest improvement in the closed-source and open-source models, respectively.

mode without fine-tuning and have the same decoding Settings (e.g., temperature, maximum length) for fair comparison in both phases.

4.3 Metrics

We use PASS@1 (Chen et al., 2021)as the primary evaluation metric because code repairs start from a repaired failed program and aim to generate the only correct repair version. Compared with PASS@K (K>1), PASS@1 can avoid overestimating performance by allowing multiple results thereby providing a stricter and more reliable evaluation of the repair effect.

5 Main Results

As shown in Table 2, on all evaluation models, the method proposed in this paper can significantly improve the execution correctness on the basis of direct code generation, verifying its overall effectiveness in code repair tasks.

In the closed-source model, GPT-5.1 achieved the highest PASS@1 performance on BigCodeBench (BCB) and achieved the optimal absolute improvement; On MBPP-sanitized, the overall performance of GPT-4.1 was slightly better than that of GPT-5.1. In the open-source model, Deepseek-V3 achieved a maximum PASS@1 of 61.92 on BCB, while Qwen3-Coder-480B-A35B-Instruct achieved a maximum absolute improvement of 12.98 percentage points. On MBPP-sanitized, DeepSeek-V3.2 achieved the best performance with 90.86 Pass@1, while Qwen3-Next-80B-A3B-Instruct achieved the best absolute improvement of 5.62 percentage points. From the overall trend, the average absolute improvement of the method proposed in this paper on BCB is 9.85 percentage points, and the maximum improvement reaches 13.42 percentage points. On MBPP-sanitized, the average absolute increase was 3.10 percentage points, and the maximum increase was 5.62 percentage points. We believe that the difference in the extent of improvement between the two mainly stems from the inherent nature differences of the datasets themselves. Compared with BCB, MBPP-sanitized is more likely to have been incorporated into the training data of some models, and its initial generation accuracy rate is relatively high, thereby limiting the improvement space that can be brought by further repair. It is worth noting that the optimal absolute performance and the maximum performance improvement do not always occur in the same model. To some extent, this indicates that the repair effect of the method proposed in this paper does not strictly depend on the initial performance level of the model, but rather stems more from the proposed repair mechanism itself.

Overall, the above experimental results indicate that the framework proposed in this paper can ef-

fectively correct programs that fail in initial generation, verifying the effectiveness of explicitly introducing structured error-repair knowledge in code repair tasks. It is worth noting that this performance improvement is consistent on both open-source and closed-source models, indicating that the method proposed in this paper has strong model independence. This phenomenon further indicates that the performance gain mainly stems from the proposed repair mechanism itself rather than the improvement of the specific model's capabilities.

6 Analysis

The experimental results show that the method proposed in this paper can achieve stable performance improvement on language models of different scales and architectures. It is worth noting that the optimal absolute performance and the maximum performance improvement do not always occur in the same model. This phenomenon further indicates that the improvement of the repair effect does not mainly depend on the parameter scale or initial performance level of the model, but rather stems from the knowledge-guided repair mechanism itself proposed. To further illustrate the effectiveness of this method, we conducted an ablation study on the key components of the repair framework, as shown in Table 3.

Method Variant	**BigCodeBench**
Initial Generation	39.28
- w/o Execution Feedback and ERKG	42.45(↑ 3.17)
- w/o ERKG	43.85(↑ 4.57)
- w/o Execution Feedback	37.28(↓ 2.00)
Evaluation report + ERKG(Ours)	52.80(↑ 12.98)

Table 3: To obtain representative and highly distinguishable ablation results, we conducted ablation experiments on BigCodeBench (BCB). To minimize the impact of differences in the model's context understanding ability on the results, all ablation experiments were repaired to use Qwen3-Coder-480B-A35B-Instruct as the basis for generating and repairing models. The experiment still used PASS@1 as the evaluation index and compared the results of each ablation variant uniformly with the corresponding initial generation performance to quantify the contribution of each component to the repair effect.

The results of the ablation experiment show that the performance improvement brought about by relying solely on direct prompts for repair or simply based on execution feedback is relatively limited. This indicates that in the absence of systematic repair constraints, the model is difficult to stably convert local error feedback into effective code modifications. It is worth noting that when only the external error knowledge graph is used as the basis for repair without considering the execution feedback, the repair performance not only fails to improve but also shows a negative gain. Our analysis suggests that the main reason lies in the fact that in the absence of a specific execution context, the model has difficulty accurately aligning the abstract error types with the actual fault locations and manifestations in the current code, which may trigger inappropriate repair strategies and introduce new errors. This phenomenon, from the side, emphasizes the significant role of execution feedback in error location and repair decision-making during the code repair process. In contrast, a complete framework, under the premise of uniformly integrating execution feedback and structured error-repair knowledge, can simultaneously provide precise error location information and clear repair strategy guidance, thereby significantly enhancing the stability and effectiveness of the repair. The overall ablation results verified the rationality and effectiveness of the knowledge-guided code repair framework proposed in this paper in terms of design.

Despite the overall effectiveness of our approach, there remain a small number of tasks where the generated code cannot be successfully repaired.These observations suggest that expanding the coverage of error–repair knowledge and incorporating richer semantic and contextual modeling will be important directions for future work to further improve the robustness of knowledge-guided code repair.

7 Conclusion

This paper proposes a Retrieval-Augmented Structured Repair Framework(RA-SRF), which explicitly introduces structured error-repair knowledge and decouples the error analysis from the code modification process with the help of repair documentation, thereby achieving more stable and controllable code repair. The experimental results verified the effectiveness of the framework on different models and benchmarks, indicating that combining execution feedback with explicit error knowledge is a feasible path to enhance the code reliability of large language models.

Limitations

Although the knowledge-guided code repair framework proposed in this paper has achieved stable performance improvements on multiple models and benchmarks, it still has certain limitations. Firstly, the repair effect is to some extent limited by the coverage of the error knowledge graph. Analysis shows that when tasks contain implicit external constraints or complex semantic assumptions that have not yet been modeled, existing graphs are difficult to provide precise enough guidance for repair. Secondly, the method proposed in this paper relies on the feedback information generated during the execution phase to trigger and guide the repair process. In scenarios where test coverage is insufficient or tasks cannot be executed, the scarcity of error signals may affect the accuracy of error normalization and repair knowledge retrieval. Finally, compared with direct code generation, the multi-stage repair process introduces additional computational overhead, which may pose limitations in application scenarios that are sensitive to inference latency. Future work can further enhance the applicability and efficiency of the method by expanding the coverage of error knowledge, introducing richer semantic modeling, and exploring more lightweight repair strategies.

References

Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, and 1 others. 2023a. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*.

Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, and 1 others. 2023b. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*.

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*.

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. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

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*.

Xinyun Chen, Chang Liu, and Dawn Song. 2018. Execution-guided neural program synthesis. In *International Conference on Learning Representations*.

Ahmed Elnaggar, Wei Ding, Llion Jones, Tom Gibbs, Tamas Feher, Christoph Angerer, Silvia Severini, Florian Matthes, and Burkhard Rost. 2021. Codetrans: Towards cracking the language of silicon's code through self-supervised deep learning and high performance computing. *arXiv preprint arXiv:2104.02443*.

Nuno Fachada, Daniel Fernandes, Carlos M Fernandes, Bruno D Ferreira-Saraiva, and João P Matos-Carvalho. 2025. Gpt-4.1 sets the standard in automated experiment design using novel python libraries. *Future Internet*, 17(9):412.

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.

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.

Mirazul Haque, Petr Babkin, Farima Farmahinifarahani, and Manuela Veloso. 2025. Towards effectively leveraging execution traces for program repair with code llms. *arXiv preprint arXiv:2505.04441*.

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.

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*.

Aixin Liu, Aoxue Mei, Bangcai Lin, Bing Xue, Bingxuan Wang, Bingzheng Xu, Bochao Wu, Bowei Zhang, Chaofan Lin, Chen Dong, and 1 others. 2025. Deepseek-v3. 2: Pushing the frontier of open large language models. *arXiv preprint arXiv:2512.02556*.

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.

Michael R Lyu, Baishakhi Ray, Abhik Roychoudhury, Shin Hwei Tan, and Patanamon Thongtanunam. 2025. Automatic programming: Large language models and beyond. *ACM Transactions on Software Engineering and Methodology*, 34(5):1–33.

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*.

Bob Martin, Mason Brown, Alan Paller, Dennis Kirby, and Steve Christey. 2011. 2011 cwe/sans top 25 most dangerous software errors. *Common Weakness Enumeration*, 7515:2011.

Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. 2013. Semfix: Program repair via semantic analysis. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 772–781. IEEE.

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.

Martha Shaka, Diego Carraro, and Kenneth Brown. 2024. Error tracing in programming: a path to personalised feedback. In *Proceedings of the 19th Workshop on Innovative Use of NLP for Building Educational Applications (BEA 2024)*, pages 330–342.

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.

Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. 2019. An empirical study on learning bug-fixing patches in the wild via neural machine translation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 28(4):1–29.

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

Zora Zhiruo Wang, Akari Asai, Xinyan Velocity Yu, Frank F. Xu, Yiqing Xie, Graham Neubig, and Daniel Fried. 2025. Coderag-bench: Can retrieval augment code generation? In *Findings of the Association for Computational Linguistics: NAACL 2025, Albuquerque, New Mexico, USA, April 29 - May 4, 2025*, pages 3199–3214. Association for Computational Linguistics.

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.

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*.

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

Shuyan Zhou, Uri Alon, Frank F Xu, Zhengbao Jiang, and Graham Neubig. 2022. Docprompting: Generating code by retrieving the docs. In *The Eleventh International Conference on Learning Representations*.

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*.