Think-Search-Patch: A Retrieval-Augmented Reasoning Framework for Repository-Level Code Repair

Bojian Xiong[1], **Yikun Lei**[2], **Xikai Liu**[2], **Shaowei Zhang**[1], **Pengyun Zhu**[1], **Yan Liu**[1], **Yongqi Leng**[1], **Ling Shi**[1], **Meizhi Zhong**[2], **Yurong Zhang**[2], **Yan Gao**[2], **Yi Wu**[2], **Yao Hu**[2], **Deyi Xiong**[1]*
[1]TJUNLP Lab, College of Intelligence and Computing, Tianjin University, Tianjin, China
[2]Xiaohongshu Inc.
{xbj1355, dyxiong}@tju.edu.cn

Abstract

Large language models usually suffer from multiple-file coding scenarios where strong inter-file dependencies manifest, typically demonstrated in SWE-bench. To mitigate this issue, we propose Think-Search-Patch (TSP), a retrieval-augmented reasoning framework for repository-level code repair. At the Think stage, our system breaks down a coding task and creates clear search query. Next, at the Search stage, it retrieves relevant code snippets using models like E5. At the final Patch stage, it generates standardized patches based on the key snippets. In addition the proposed framework, we enhance system reliability through a two-stage training process. At the first stage, the system undergoes supervised fine-tuning (SFT) on our TSP dataset. At the subsequent stage, we employ rejection sampling with correction to generate preference pairs for Direct Preference Optimization (DPO) training, thereby reducing errors in the intermediate phases. Experimental results demonstrate that TSP framework enhances retrieval accuracy and repair success on SWE-bench Lite, even surpassing models with a larger size in managing extensive code contexts and successfully addressing bugs spanning across multiple files. All data and code available at https://github.com/tjunlp-lab/TSP-framework.

1 Introduction

Large language models (LLMs) have demonstrated significant capabilities in code generation, achieving strong performance on single-file scenario benchmarks, such as code completion in HumanEval (Chen et al., 2021) and MBPP (Austin et al., 2021). However, code-specific LLMs, exhibit inadequate performance in multi-file scenarios, which are prevalent in real-world software development and characterized by high complexity and intricate cross-file dependencies.

The SWE-bench (Jimenez et al., 2024) accurately captures this challenge. It constructs test cases based on code snippets and issue descriptions from real open source projects, covering various repair tasks ranging from basic syntax errors to complex logical flaws. This benchmark has two main features, which collectively make it hard for models to handle multi-file repair tasks. **Task complexity**: Tasks demand multi-dimensional capabilities simultaneously, e.g. code semantic parsing, flaw localization, and precise patch generation. **Context scale**: Test samples provide an average context of hundreds of lines of code. Within such a vast context, models struggle to effectively filter out irrelevant information. Consequently, crucial code snippets relevant to the issue may be obscured by an overwhelming volume of unrelated code, thereby severely hindering accurate flaw localization.

To address the aforementioned challenges, we propose a **Think-Search-Patch (TSP)** framework. Our framework adopts a three-stage process and incorporates a two-stage training strategy, enabling it to autonomously analyze issues, identify root causes, and retrieve necessary code snippets before ultimately completing the repair. This approach mitigates the issue of context overload and enhances code repair performance at the repository level. Our framework involves the following steps: **Think stage** guides the model to decompose the problem into three sequential steps: (i) issue analysis, (ii) task decomposition, and (iii) search query construction. At this stage, semantic analysis is utilized to generate precise search queries. **Search stage** employs the E5 (Wang et al., 2022) model to search the repository, retrieving semantically relevant code snippets based on similarity metrics. **Patch stage** focuses on the retrieved key code snippets, performing code localization and code editing to generate well-formatted patch, thus completing the overall Think-Search-Patch process. The diagram of our framework is illustrated in Figure 1.

*Corresponding author.

Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing: Industry Track, pages 1555–1566
November 4-9, 2025

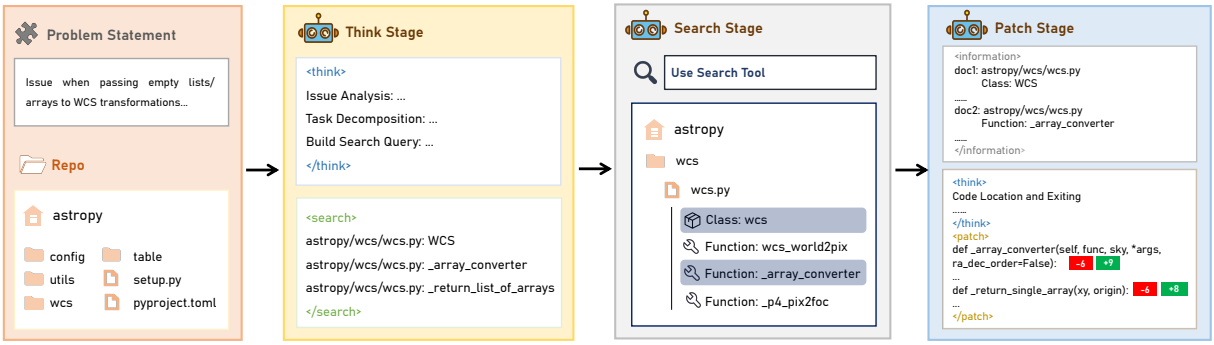

Figure 1: Overview of the framework for Think-Search-Patch.

To systematically enhance model capabilities throughout our three-stage process, we design a two-stage training strategy. The first stage focuses on process learning, where supervised fine-tuning (SFT) is performed on TSP dataset. This phase establishes fundamental capabilities in adhering to task specifications and constructing the core process. Subsequently, the second stage implements preference alignment optimization: We utilize the model trained at stage 1 to perform rollout, generating a batch of data. Then, we use GPT-4o as an LLM-judge to invoke to assess and amend the quality of these data. Corrected samples serve as positive examples, while uncorrected erroneous outputs constitute negative examples. These preference pairs then facilitate Direct Preference Optimization (Rafailov et al., 2023) training. Through this two-stage training paradigm, we trained a series of models to verify the feasibility of the TSP framework, which effectively increased the proportion of key code recall, while achieving repair success rates that markedly outperform RAG methods.

In summary, our contributions are as follows:

1. We propose the Think-Search-Patch (TSP) framework. To effectively train the reasoning process of TSP, we built the TSP dataset, comprising 60K samples specifically designed to optimize collaborative reasoning across its three stages.

2. We designed a fine-grained indexing construction strategy that retrieves code at the Class and Function levels, effectively addressing the redundancy issues associated with traditional file-level code retrieval.

3. On the SWE-bench Lite benchmark, models trained based on the TSP framework demonstrate superior performance among models of similar scale, thereby validating the effectiveness of our framework and its technical strategies in code repair tasks.

2 Related Work

Recent developments in code-specific LLMs have significantly advanced code generation capabilities (Guo et al., 2023). Closed-source models, such as GPT-4.5[1], Claude-4[2] excel in coding tasks, while open-source models, including DeepSeek-Coder (Guo et al., 2024) and Qwen-Coder (Hui et al., 2024), offer competitive performance at smaller scales. Specialized LLMs address specific engineering challenges, and many studies focus on improving program generation (Liu et al., 2024, 2023; Lin et al., 2024; Zheng et al., 2023; Yang et al., 2025b). Despite this, effectively using LLMs to address complex code generation remains a challenge. Two primary paradigms are currently harnessing the potential of LLMs: agent-based frameworks (Wang et al., 2024; Ma et al., 2024; Yang et al., 2024; Zhang and Xiong, 2025; Leng et al., 2025; Li et al., 2025), which facilitate iterative interactions with tools and environments for decision making, and pipeline-based approaches (Wei et al., 2025; Xie et al., 2025), which follow predefined sequences of steps. However, Agent-based frameworks facilitate iterative interactions with tools and environments for decision-making. This strategy incurs significant resource consumption, as it requires guiding the SWE-Agent to invoke appropriate tools, typically involving multiple rounds. The pipeline strategy, on the other hand, places greater emphasis on the inherent coding abilities, which are more dependent on the size of models. Our TSP framework belongs to the pipeline-based approach,

[1] https://openai.com/index/introducing-gpt-4-5/

[2] https://www.anthropic.com/news/claude-4

Method	Token Consumption
RAG	16933.77
TSP framework	3724.56

Table 1: Comparison of recall code tokens consumption and recall code tokens between RAG and TSP framework on SWE-bench Lite.

which emphasizes the rationality of task decomposition and the efficiency of search tool integration, enabling 7B-scale and 14B-scale models to possess certain repository-level code repair capabilities.

3 Think-Search-Patch

In this section, we elaborate our Think-Search-Patch framework and TSP dataset. Section 3.1 presents the dataset structure and index construction, while Section 3.2 details the first stage of the two-stage training approach: SFT for problem-solving. Section 3.3 then elaborates on the second stage, which enhances the problem-solving ability of the entire process through DPO.

3.1 TSP Dataset

Data structure. We extend the rigorous filtering logic of the SWE-Fixer dataset (Xie et al., 2025) to construct our TSP dataset. Our filtering strategy primarily involves removing two types of inefficient samples. The first type is issues containing non-textual information, for which we deleted problems including image links, external document links, or other content that cannot be interpreted by the model, to ensure the textual self-consistency of the issue descriptions. This is because the model cannot directly process the semantic information contained in images or links, which may lead to deviations in defect localization. The second type is issues with vague descriptions, for which we filtered out problems lacking specific defect localization or references to code entities, as vague descriptions usually cannot guide the model to locate specific code units, resulting in ineffective retrieval and repair. We adopted the practice of using the annotated reasoning chain of issue analysis, task decomposition, and code localization in the SWE-Fixer dataset as the core reasoning process. To accommodate retrieval-augmented scenarios, we embed a search module after task decomposition, utilizing Qwen-Max (Yang et al., 2025a) to generate search query and retrieve relevant code snippetss, thereby establishing the Think-Search-Patch workflow. To address the high overhead of full indexing, the target code snippets are directly extracted from annotated Oracle files, preserving SWE-Fixer's advantage in precise code localization. Furthermore, we establish a closed-loop verification system: 5% of the data is randomly sampled to build an E5 vector index that simulates real-world environments, with the top-2 recall rate quantifying search query quality, achieving up to 80% average accuracy. Through failure case analysis focusing on problem description deficiencies, this feedback optimizes the data cleaning process, ultimately forming a closed loop: 1) data cleaning, 2) search query construction, 3) Oracle extraction, 4) code localization verification, and 5) filter optimization. The pipeline fully inherits SWE-Fixer's strengths in reasoning coherence and localization precision, while enabling adaptive upgrades for real repository through the search query mechanism. All the prompt instructions used for building search query can be found in Appendix A.

Index construction. During the index construction phase of the SWE-bench framework, we address context bloat induced by file-level retrieval by proposing an Abstract Syntax Tree (AST)-based approach. As shown in Figure 2, this approach utilizes precise locator identifiers formatted as path/to/file.py:Class or path/to/file.py:Function. This methodology enables fine-grained indexing through a structured technical pipeline. Based on Python's ast module, the code file is parsed to generate an AST. By traversing the AST nodes, we extract Class and Function nodes, enriched them with metadata such as file path, name, and line number, and assigned unique identifiers to each node. Finally, the nodes are transformed into structured indexing units. Experimental results indicate that this approach reduces the average recalled context length by 78%, as shown in Table 1, effectively filtering out irrelevant code noise. Our indexes serve two key purposes: during the DPO training phase, they dynamically construct preference datasets; during the evaluation phase, they are integrated with the TSP framework to support inference and address issues.

3.2 Process Foundation Training

At Stage 1, we establish closed-loop task processing through standardized training. This phase utilizes SFT on TSP dataset. At the think phase, we base issue analysis on identifying the key requirements and objectives for solving the issue, and then

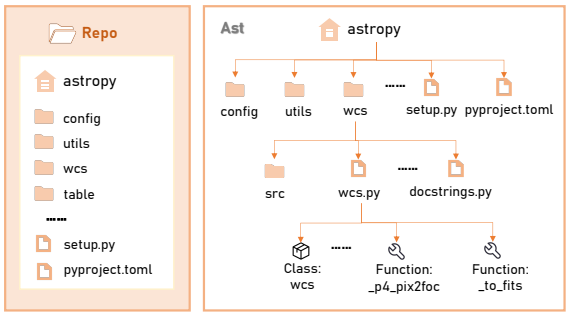

Figure 2: The construction process of the AST syntax tree.

decompose the issue into manageable subtasks. At the search phase, based on the analysis results, we generate a search query to locate the class or function in the target code snippets. At the patch phase, we utilize the retrieved results to perform code location and editing and subsequently generate a patch that conforms to the required format.

To ensure consistency with the reasoning process, we adopt a staged training strategy based on data partitioning, which is divided into the following two stages. (1) The first stage focuses on training the model's ability to generate search queries. At this stage, the model generates precise search query solely based on the issue, thus specifically cultivating its capability to map issue features to retrieval strategies. (2) emphasize training the model's ability to generate patches based on retrieved code snippets. The input for this stage consists of a combination of Issue Analysis, the retrieval process, and retrieved information, requiring the model to generate patches accordingly. This approach ensures that in real reasoning scenarios, the model is capable of integrating the retrieved results to complete closed-loop processing.

3.3 Preference Optimization Alignment

At stage 2 of preference optimization alignment training, we enhance index utilization by querying issue version information and employing an index reusability strategy for data from the same repository and version. Specifically, we first filter out data of the same version within a repository to avoid redundant index construction, thereby directly reusing existing indexes. Based on this mechanism, we sample nearly 4,000 entries and established almost 200 indexes for data from the same repository and version, thereby achieving efficient integration of the training data. Subsequently, we simulate a realistic retrieval scenario using a dynamic retrieved approach, allowing the model to generate a complete Think-Search-Patch process output for each issue. These samples contain various intermediate errors that occur naturally in the model's generation process, thus providing typical negative examples for subsequent optimization.

At the data filtering phase, an LLM-judge is introduced to conduct quality screening. Correctness is evaluated along three dimensions: **(1) search query dimension**, which assesses whether the recalled results include code snippets relevant to the issue, thereby ensuring the accuracy of search query; **(2) problem analysis logic dimension**, which requires the model to maintain logical consistency throughout the process in order to avoid haphazard fixes; **(3) code location dimension**, which enforces precise identification of files, classes, or functions, and line numbers. This allows for pinpointing of the dependency impact domain and minimizes the scope of modifications, thereby preventing interference with unrelated code areas. Ultimately, structured corrections are applied to non-compliant samples by preserving reasonable components of issue analysis, generating standardized reference outputs for the identified deficiencies, and forming revised positive sample cases.

During stage 1 optimization, we have observed a significant improvement in the model's ability to generate queries. However, practical evaluations have revealed that, despite the enhanced search query quality, the retrieved results based on Class and Function are far more concise than the file-level retrieved results. However, it remains difficult for the model to accurately identify the required code positions, resulting in suboptimal performance in the patch stage. To address this issue, we implement a targeted optimization strategy by reconstructing a dataset specifically for the patch stage. This part aims to precisely locate issue-relevant code snippets within the retrieved results and is used to reinforce the model's ability to generate accurate code locations and patches. Through data-level targeted optimization, we effectively compensate for the shortcomings in the retrieval and localization process during stage 1, thereby establishing a comprehensive TSP framework.

Based on the preference data pairs constructed using the aforementioned mechanism, we train the model using the DPO algorithm. This algorithm guides the model to learn the correct reasoning process by maximizing the conditional probability of positive examples relative to negative ones.

Specifically, during the retrieval phase, the DPO algorithm focuses on reducing the probability of generating erroneous queries, thereby enhancing query accuracy. During the patch phase, it focuses on improving the model's ability to accurately locate relevant recalled code and generate patches that address the identified issues. In this manner, the DPO algorithm effectively optimizes the model's performance in the search and patch phases, ultimately enhancing the overall capability of our framework.

4 Experiment

We conducted extensive experiments and in-depth analyses to evaluate the proposed entire framework, including evaluation results based on SWE-bench Lite, search query construction results, and the effects of our two-stage training.

4.1 Experiment Setup

In a hardware environment equipped with 16 NVIDIA H800 GPUs, to verify the effectiveness of the TSP framework, we selected multiple models, such as Qwen2.5-Coder-7B-Instruct, Qwen2.5-Coder-14B-Instruct (Hui et al., 2024), deepseek-coder-6.7b-instruct (Guo et al., 2024), Mistral-7B-Instruct-v0.3 (Jiang et al., 2023), etc., and implemented a two-stage training process. Specifically, we first conducted supervised fine-tuning (SFT) on the TSP dataset, followed by alignment with Direct Preference Optimization (DPO) using preference pairs. We employed the Verl framework (Sheng et al., 2025) for SFT training and the OpenRLHF framework (Hu et al., 2024) for DPO training. Additionally, we implemented a retrieved token loss masking mechanism. Although standard SFT and DPO typically compute loss across the entire unfolded sequence, the incorporation of externally retrieved code snippets in our TSP framework makes applying the same optimization to retrieved tokens potentially problematic, as it may induce unintended learning dynamics. To address this, we designed a specialized loss masking mechanism that explicitly excludes retrieved code snippets from policy gradient computations.

For the evaluation phase, we adopted SWE-bench Lite, a streamlined subset of the authoritative SWE-bench comprising 300 optimized representative instances. The final assessment framework employed a dual-dimensional validation mechanism. We strictly adhered to the official evaluation protocol of SWE-bench to verify whether the generated patches passed all test cases related to the issue. This standard is extremely rigorous, as it demands perfect remedying of all defects, potentially overlooking incremental improvements in partial fixes or semantic understanding. To address this, we introduced LLM-judge as a key supplementary evaluation dimension. Our LLM-judge supplements evaluation with three metrics: context relevance (accuracy of search queries), location accuracy (correctness of edit placements, including golden patch modifications), and repair effectiveness (resolution of underlying issues rather than superficial fixes).

To validate the consistency between LLM-judge and human evaluators, we selected 100 samples generated by TSP-Qwen2.5-Coder-7B from the SWE-bench Lite evaluation, invited five graduate students with backgrounds in software engineering and code repair, and used GPT-4o as the large language model evaluator. Both groups of evaluators independently scored the samples based on the three dimensions defined in the prompts—context relevance, position correctness, and core fix correctness. Detailed evaluation comparisons are provided in Appendix B. Additionally, since the TSP framework requires invoking retrieval during inference, we built indexes for each entity in the evaluation process. All prompt instructions used for scoring can be found in Appendix C.

4.2 Main Result

As shown in Table 2, the experimental results demonstrate that our Think-Search-Patch (TSP) framework exhibits significant structural paradigm advantages on SWE-bench Lite. First, during the search process, this framework, by retrieving code snippets at the class and function levels, comprehensively outperforms the traditional retrieval-augmented generation (RAG) methods that rely on file-level retrieval. In the rigorous unit tests on SWE-bench Lite, TSP-Qwen2.5-Coder-7B achieves a repair pass rate of 5.0%, and TSP-Qwen2.5-Coder-14B reaches a pass rate of 8.33%, while GPT-4 (1106) with the standard RAG process only has a pass rate of 2.67%. Moreover, compared with models of similar scale, the models trained via the TSP framework have significantly higher repair efficiency than SWE-Llama. This indicates that our finer-grained retrieval approach effectively addresses the persistent issue of long-code context interference, fully demonstrating that our finer-grained retrieval method successfully re-

Method	Model	LLM-judge score	Pass rate
RAG (Jimenez et al., 2024)	SWE-Llama-7B	27.5	1.3
RAG (Jimenez et al., 2024)	SWE-Llama-13B	30.1	1
RAG (Jimenez et al., 2024)	GPT 4 (1106)	-	2.67
RAG (Jimenez et al., 2024)	Claude 3 Opus	-	4.33
RAG (Jimenez et al., 2024)	Qwen2.5-Coder-7B-Instruct	22.4	0
RAG (Jimenez et al., 2024)	Qwen2.5-Coder-14B-Instruct	30.1	0.33
RAG (Jimenez et al., 2024)	deepseek-coder-6.7b-instruct	20.5	0
RAG (Jimenez et al., 2024)	Mistral-7B-Instruct-v0.3	17.6	0
Think-Search-Patch	TSP-Qwen2.5-Coder-7B	48.1	5
Think-Search-Patch	TSP-Qwen2.5-Coder-14B	55.6	8.33
Think-Search-Patch	TSP-deepseek-coder-6.7b	40.1	4.33
Think-Search-Patch	TSP-Mistral-7B	40.5	3.33

Table 2: Performance comparison of different methods. LLM-judge score indicates model performance (0-100 scale), repair pass rate denotes task completion accuracy.

solves the challenge of long-code context interference. We also analyzed the reasons why models such as Qwen2.5-coder-7B-instruct and deepseek-coder-6.7b-instruct, which were not trained with the TSP framework, performed poorly. The main reasons are their insufficient search query generation capabilities and the lack of targeted training in the patch generation phase. This results in the models' inability to construct fine-grained retrieval queries, leading to a low recall rate of key code. In the entire pipeline, generating appropriate search queries is crucial because it directly determines whether problem-related code can be retrieved. Additionally, the absence of specialized training in the patch generation phase leads to insufficient code localization and modification capabilities.

The empirical study on the LLM-judge scoring system further reveals the core value of the two-stage training paradigm: TSP-Qwen2.5-Coder-7B achieves an overall score of 48.1, significantly surpassing Qwen2.5-Coder-7B-Instruct; TSP-Qwen2.5-Coder-14B even reaches a score of 55.6, greatly outperforming the corresponding model under RAG, and such a contrast is also reflected in the performance of other models. This structural advantage stems from a dual synergistic training mechanism: during the process foundation training phase, TSP framework models systematically establish a deep cognitive pattern for the Think-Search-Patch framework, enabling them to accurately grasp the rules of problem decomposition and the norms for generating retrieval instructions; subsequently, in the preference optimization alignment phase, through contrastive learning guided by LLM-judge corrections, the key ability — extracting problem-relevant effective content from retrieved code snippets for repair — is specifically enhanced. This result verifies the irreplaceability of the hierarchical training design in improving model performance.

Model	Search ACC
Qwen2.5-Coder-7B-Instruct	24.0
Qwen2.5-Coder-14B-Instruct	28.7
deepseek-coder-6.7b-instruct	25.1
Mistral-7B-Instruct-v0.3	19.5
TSP-Qwen2.5-Coder-7B	50.7
TSP-Qwen2.5-Coder-14B	64.8
TSP-deepseek-coder-6.7b	40.1
TSP-Mistral-7B	38.6

Table 3: Search accuracy comparison across models. Metrics reflect the precision of context retrieval in respective frameworks.

These findings collectively confirm that our framework, leveraging class and function-level retrieval, addresses the contextual redundancy inherent in file-based retrieval of RAG methods, thereby enhancing the model's capability to precisely locate issue-relevant code. By decomposing the complex problems in SWE-bench into issue analysis, retrieving relevant code snippets, and performing targeted repairs, the performance of the 7B-scale and 14B-scale models can be effectively enhanced. This demonstrates that our framework rectifies the core flaws of RAG methods by eliminating redundant file-level code while establishing the previously

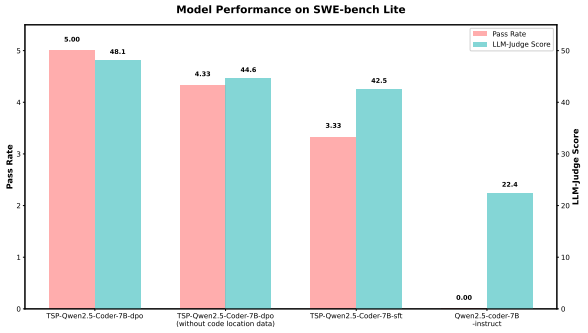

Figure 3: ablation study to validate the effectiveness of the DPO training.

missing scientific reasoning framework. A case study of our framework is included in the Appendix D.

4.3 Quality of Search Query

The quality of the generated search query plays a decisive role in the success of the task: an accurate search query can directly retrieve the code snippets related to the issue, thereby enabling model to perform comprehensive debugging on the target code snippets. To evaluate the retrieval capabilities of the models, we used the retrieved rate of key code snippets as metric to assess several models. Specifically, by extracting the function or class entity that needs modification from the golden patch, we verify whether the retrieved results encompass the relevant context code snippets for the target entity, thereby achieving an objective quantitative evaluation of the search query generation quality.

As shown in Table 3, TSP-Qwen2.5-Coder-14B demonstrates superior performance with a retrieval accuracy of 64.8%, outperforming our baseline model Qwen2.5-coder-14B-instruct (28.7% accuracy). In contrast, SWE-Llama fails completely with an accuracy of 0.0%. This failure is attributed to SWE-Llama's excessive focus on patch generation during training, which leads to severe overfitting in the retrieval phase. Conversely, our proposed two-stage training framework effectively enhances the model's ability to construct high-quality search query, thereby achieving exceptional performance in real-world retrieval tasks.

4.4 Ablation Study

We designed a rigorous ablation study to validate the effectiveness of the DPO training mechanism. The experiment evaluated performance on the SWE-bench Lite benchmark by comparing models that underwent only the first-stage SFT, completed DPO training, and those with position-enhanced data removed during the second-stage training. Key evaluation metrics included the LLM-judge score and patch pass rate.

As shown in Figure 3, by comparing the training results of Qwen2.5-coder-7B-instruct after SFT and DPO, the ablation results confirmed that the performance of our Think-Search-Patch framework was significantly enhanced through a two-stage collaborative optimization approach: TSP-Qwen2.5-Coder-7B-dpo achieved a 5% patch pass rate and an LLM-judge score of 48.1 on SWE-bench Lite, outperforming TSP-Qwen2.5-Coder-7B-sft in both evaluation metrics. This notable advantage is attributed to the alignment of DPO training with human preference data, which effectively reinforces the model's end-to-end reasoning ability from issue analysis to code modification. Further analysis showed that TSP-Qwen2.5-Coder-7B-dpo without position data systematically declines in performance, validating that the targeted construction of position-enhanced data plays a crucial role in improving the spatial localization capability of retrieved results; we believe the absence of this module directly undermines the model's precision in identifying the correct location for code modifications.

5 Conclusion

We have presented the Think-Search-Patch (TSP) framework to address strong cross-file dependencies in multi-file coding, enhancing code repair accuracy. Using a fine-grained index at the class and function levels instead of file-level retrieval reduces redundancy and improves retrieval. Our TSP-coder models are trained in two stages, SFT on the TSP dataset and DPO on correction pairs. Experimental results show that TSP-Qwen2.5-Coder-14B achieves an 8.33% patch pass rate and an LLM-judge score of 55.6 on SWE-bench Lite. This not only outperforming larger general models within the RAG framework that relies on file-level retrieval, but also demonstrates the effectiveness and practical value of our TSP framework in code repair tasks.

Acknowledgments

The present research was supported by the Key Research and Development Program of China (Grant No. 2024YFE0203000). We would like to thank the anonymous reviewers for their insightful com-

ments.

Limitations

Although our approach outperforms the RAG method, it still falls short compared to agent-based and pipeline methods, primarily due to the limitation in model parameter scale which restricts complex reasoning abilities and makes it difficult to handle complex, cross-document repair tasks that require hierarchical inference. Future work may draw on the dynamic decision-making mechanism of agents to enable the model to adaptively adjust the Think-Search-Patch process based on task complexity.

References

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, and Quoc Le. 2021. Program Synthesis with Large Language Models. *arXiv preprint arXiv:2108.07732*.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, and Greg Brockman. 2021. Evaluating Large Language Models Trained on Code. *arXiv preprint arXiv:2107.03374*.

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y. Wu, Y. K. Li, Fuli Luo, Yingfei Xiong, and Wenfeng Liang. 2024. Deepseek-Coder: When the Large Language Model Meets Programming - The Rise of Code Intelligence. *CoRR*, abs/2401.14196.

Zishan Guo, Renren Jin, Chuang Liu, Yufei Huang, Dan Shi, Supryadi, Linhao Yu, Yan Liu, Jiaxuan Li, Bojian Xiong, and Deyi Xiong. 2023. Evaluating Large Language Models: A Comprehensive Survey. *CoRR*, abs/2310.19736.

Jian Hu, Xibin Wu, Weixun Wang, Xianyu, Dehao Zhang, and Yu Cao. 2024. OpenRLHF: An easy-to-use, scalable and high-performance RLHF framework. *CoRR*, abs/2405.11143.

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.

Albert Q. Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de Las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, Lélio Renard Lavaud, Marie-Anne Lachaux, Pierre Stock, Teven Le Scao, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. 2023. Mistral 7B. *CoRR*, abs/2310.06825.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R. Narasimhan. 2024. Swe-bench: Can Language Models Resolve Real-world Github Issues? In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net.

Yongqi Leng, Renren Jin, Yue Chen, Zhuowen Han, Ling Shi, Jianxiang Peng, Lei Yang, Juesi Xiao, and Deyi Xiong. 2025. Praetor: A Fine-Grained Generative LLM Evaluator with Instance-Level customizable evaluation criteria. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2025, Vienna, Austria, July 27 - August 1, 2025*, pages 10386–10418. Association for Computational Linguistics.

Zhigen Li, Jianxiang Peng, Yanmeng Wang, Yong Cao, Tianhao Shen, Minghui Zhang, Linxi Su, Shang Wu, Yihang Wu, Yuqian Wang, Ye Wang, Wei Hu, Jianfeng Li, Shaojun Wang, Jing Xiao, and Deyi Xiong. 2025. ChatSOP: An SOP-Guided MCTS Planning Framework for Controllable LLM dialogue agents. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2025, Vienna, Austria, July 27 - August 1, 2025*, pages 17637–17659. Association for Computational Linguistics.

Bo Lin, Shangwen Wang, Ming Wen, Liqian Chen, and Xiaoguang Mao. 2024. One Size Does Not Fit All: Multi-granularity Patch Generation for Better Automated Program Repair. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 1554–1566.

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.

Xiangyan Liu, Bo Lan, Zhiyuan Hu, Yang Liu, Zhicheng Zhang, Fei Wang, Michael Shieh, and Wenmeng Zhou. 2024. CodexGraph: Bridging Large Language Models and Code Repositories via Code Graph Databases. *arXiv preprint arXiv:2408.03910*.

Yingwei Ma, Rongyu Cao, Yongchang Cao, Yue Zhang, Jue Chen, Yibo Liu, Yuchen Liu, Binhua Li, Fei Huang, and Yongbin Li. 2024. Lingma SWE-GPT: An Open Development-Process-Centric Language Model for Automated Software Improvement. *arXiv preprint arXiv:2411.00622*.

Rafael Rafailov, Archit Sharma, Eric Mitchell, Christopher D Manning, Stefano Ermon, and Chelsea Finn. 2023. Direct preference optimization: Your language model is secretly a reward model. *Advances in*

Neural Information Processing Systems, 36:53728–53741.

Guangming Sheng, Chi Zhang, Zilingfeng Ye, Xibin Wu, Wang Zhang, Ru Zhang, Yanghua Peng, Haibin Lin, and Chuan Wu. 2025. Hybridflow: A flexible and efficient rlhf framework. In *Proceedings of the Twentieth European Conference on Computer Systems*, pages 1279–1297.

Liang Wang, Nan Yang, Xiaolong Huang, Binxing Jiao, Linjun Yang, Daxin Jiang, Rangan Majumder, and Furu Wei. 2022. Text Embeddings by Weakly-Supervised Contrastive Pre-training. *arXiv preprint arXiv:2212.03533*.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, and Jaskirat Singh. 2024. OpenHands: An Open Platform for AI Software Developers as Generalist Agents. *arXiv preprint arXiv:2407.16741*.

Yuxiang Wei, Olivier Duchenne, Jade Copet, Quentin Carbonneaux, Lingming Zhang, Daniel Fried, Gabriel Synnaeve, Rishabh Singh, and Sida I Wang. 2025. Swe-RL: Advancing LLM Reasoning via Reinforcement Learning on Open Software Evolution. *arXiv preprint arXiv:2502.18449*.

Chengxing Xie, Bowen Li, Chang Gao, He Du, Wai Lam, Difan Zou, and Kai Chen. 2025. Swe-Fixer: Training Open-Source LLMs for Effective and Efficient Github Issue Resolution. *arXiv preprint arXiv:2501.05040*.

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

John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. 2024. Swe-Agent: Agent-computer interfaces enable automated software engineering. *Advances in Neural Information Processing Systems*, 37:50528–50652.

Lei Yang, Renren Jin, Ling Shi, Jianxiang Peng, Yue Chen, and Deyi Xiong. 2025b. ProBench: Benchmarking Large Language Models in Competitive Programming. *arXiv preprint arXiv:2502.20868*.

Shaowei Zhang and Deyi Xiong. 2025. Debate4MATH: Multi-Agent Debate for Fine-Grained Reasoning in Math. In *Findings of the Association for Computational Linguistics, ACL 2025, Vienna, Austria, July 27 - August 1, 2025*, pages 16810–16824. Association for Computational Linguistics.

Lin Zheng, Jianbo Yuan, Zhi Zhang, Hongxia Yang, and Lingpeng Kong. 2023. Self-Infilling Code Generation. *arXiv preprint arXiv:2311.17972*.

A Prompt for building search queries

This is the prompt we use in SWE-Fixer's reasoning dataset to call Qwen-max and construct search queries.

Search Query Construction Prompt

Based on the provided issue: {issue}, and combined with the analysis of this problem: {reasoning_process}, complete the following tasks:

<task>
1. **Reasoning for constructing search queries**: Within `<think></think>`:
   - Analyze how to reasonably construct the search query based on the provided content
   - Align with the original analysis; avoid excessive repetition

2. **Generate search query**: Within `<search></search>`:
   - Identify classes corresponding to the issue's resolution context
   - Output based on the issue and analysis as: path/file.py:Class_name or function_name. If no reliable path exists, output only the filename.
   - Output as a Python list
</task>

Final deliverables:
- Reasoning for constructing the search query in `<think></think>`
- Search query within `<search></search>`

Table 4: Search Query Construction Prompt.

B The consistency between LLM-judge and human evaluator

We used the Pearson correlation coefficient to quantify the agreement between the evaluators. The results demonstrate a high level of consistency between the LLM-judge and human evaluators.

Metric	LLM-judge Score	Human Score	Pearson Correlation Coefficient
Context Relevance (20%)	11.29	11.92	86.78%
Position Correctness (40%)	21.89	17.62	83.09%
Core Fix Correctness (40%)	14.79	10.97	78.36%
Total	48.07	40.51	83.01%

Table 5: The consistency between LLM-judge and human evaluator on SWE-bench Lite.

C Prompt of LLM-judge for scoring

We present an example of a prompt for GPT-4o as an LLM-judge. Specifically, we require GPT-4o to score from three dimensions, namely Context Relevance, Position Correctness, and Core Fix Correctness. Table 6 shows an example of a prompt in LaTeX format.

As an SWE-bench task evaluation expert, you need to evaluate the model output based on the following materials:

1. `issue`: Bug description reported by the user
2. `golden patch`: Correct code repair solution
3. `model output`: Response generated by the model (including thinking process, search query, recall information, and model patch)

Scoring Dimensions and Weights:

1. **Context Relevance (20%)**
   - Check whether the search query accurately recalls the code context related to the issue

2. **Position Correctness (40%)**
   - Determine whether the model patch includes the modification content in the golden patch

3. **Core Fix Correctness (40%)**
   - Check whether the model patch solves the core problem of the issue

Finally, put the score in <reward></reward>.

Issue: {issue}
Golden Patch: {golden_patch}
Model Output: {model_output}

Table 6: The SWE-bench Task Evaluation Prompt.

D Case Study

The repair process of the TSP framework for the `django-12453` case proceeds as follows: First, it parses the problem description (when accessing `UserAdmin` via `to_field`, the link to `PasswordResetForm` returns a 404 error) and enters the **Think** stage: here, it identifies the root cause as flawed URL generation logic for the password reset link, decomposing the task into "locating the code responsible for URL generation". Next, in the **Search** stage, leveraging fine-grained indexing at the class and function levels, it precisely retrieves the code of the `__init__` method in the `UserChangeForm` class within `django/contrib/auth/forms.py` (this method formats the `help_text` of the password field, which embeds URL concatenation logic). Finally, in the **Patch** stage, analyzing the code logic reveals that the original format string (`"../password/"`) in `help_text` causes an invalid URL path. By modifying it to `"../../{self.instance.pk}/password/"` (constructing an absolute path using the instance's primary key) and generating a diff patch, the framework completes the repair, resolving the link redirection issue under cross-file dependencies.

instant id: django-12453

problem statement: Accessing UserAdmin via to_field leads to link to PasswordResetForm being broken (404) Description (last modified by Simon Kern) Accessing the UserAdmin via another model's Admin that has a reference to User (with to_field set, e.g., to_field=üuid̈)...

<think>**Issue Analysis:** The issue at hand involves a broken link to the PasswordResetForm when accessing the UserAdmin via a to_field in Django. Specifically, the URL generated for the password reset link is incorrect, leading to a 404 error...
Task Decomposition: To resolve this issue, we need to: 1. Identify the Code Responsible for URL Generation: Locate where the password reset link is being constructed in the 'UserChangeForm' class...
Build Search query: The relevant file is identified as 'django/contrib/auth/forms.py', where the 'UserChangeForm' class resides. This class contains the '__init__' method where the 'password.help_text' is modified... </think>

<search>django/contrib/auth/forms.py:UserChangeForm </search>

<information>
```
doc1: django/contrib/auth/forms.py
Class: UserChangeForm
147 class UserChangeForm(forms.ModelForm):
148    password = ReadOnlyPasswordHashField(
        label=_('Password'),
        help_text=_('Raw passwords are not stored...'))
149
150    class Meta:
151        model = User
152        fields = '__all__'
153        field_classes = {'username': UsernameField}
154
155    def __init__(self, *args, **kwargs):
156        super().__init__(*args, **kwargs)
157        password = self.fields.get('password')
158        if password:
159            password.help_text = password.help_text.format('../password/')
160        user_permissions = self.fields.get('user_permissions')
```
</information>

<think>
##Code Localization and Editing
Step 1: Identify the Code Responsible for URL Generation
- File Path: `django/contrib/auth/forms.py`
- Relevant Code: The `UserChangeForm` class, specifically the `__init__` method, where the password field's help text is being formatted with a URL.
- Code Snippet:
```
156        super().__init__(*args, **kwargs)
157        password = self.fields.get("password")
...
```
</think>
<patch>
```
diff -git a/django/contrib/auth/forms.py b/django/contrib/auth/forms.py
- a/django/contrib/auth/forms.py
+++ b/django/contrib/auth/forms.py
@@ -156,7 +156,7 @@ def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        password = self.fields.get("password")
        if password:
-           password.help_text = password.help_text.format("../password/")
+           password.help_text =
password.help_text.format(f"../../{self.instance.pk}/password/")
        user_permissions = self.fields.get("user_permissions")
        if user_permissions:
            user_permissions.queryset =
user_permissions.queryset.select_related("content_type")
```
</patch>

Table 7: Issue descriptions and analysis for Django UserAdmin problems.