MCTS-Refined CoT: High-Quality Fine-Tuning Data for LLM-Based Repository Issue Resolution

Yibo Wang
Northeastern University
Shenyang, China
yibowangcz@outlook.com

Zhihao Peng
Northeastern University
Shenyang, China
2471378@stu.neu.edu.cn

Ying Wang*
Northeastern University
Shenyang, China
wangying@swc.neu.edu.cn

Zhao Wei*
Tencent
Beijing, China
zachwei@tencent.com

Hai Yu
Northeastern University
Shenyang, China
yuhai@mail.neu.edu.cn

Zhiliang Zhu
Northeastern University
Shenyang, China
ZHUZhiLiang_NEU@163.com

***Abstract*—LLMs demonstrate strong performance in automated software engineering, particularly for code generation and issue resolution. While proprietary models like *GPT-4o* achieve high benchmarks scores on *SWE-bench*, their API dependence, cost, and privacy concerns limit adoption. Open-source alternatives offer transparency but underperform in complex tasks, especially sub-100B parameter models. Although quality Chain-of-Thought (CoT) data can enhance reasoning, current methods face two critical flaws: (1) weak rejection sampling reduces data quality, and (2) inadequate step validation causes error accumulation. These limitations lead to flawed reasoning chains that impair LLMs' ability to learn reliable issue resolution.**

The paper proposes MCTS-REFINE, an enhanced Monte Carlo Tree Search (MCTS)-based algorithm that dynamically validates and optimizes intermediate reasoning steps through a rigorous rejection sampling strategy, generating high-quality CoT data to improve LLM performance in issue resolution tasks. Key innovations include: (1) augmenting MCTS with a reflection mechanism that corrects errors via rejection sampling and refinement, (2) decomposing issue resolution into three subtasks—*File Localization*, *Fault Localization*, and *Patch Generation*—each with clear ground-truth criteria, and (3) enforcing a strict sampling protocol where intermediate outputs must exactly match verified developer patches, ensuring correctness across reasoning paths.

Experiments on *SWE-bench Lite* and *SWE-bench Verified* demonstrate that LLMs fine-tuned with our CoT dataset achieve substantial improvements over baselines. Notably, *Qwen2.5-72B-Instruct* achieves 28.3%(*Lite*) and 35.0%(*Verified*) resolution rates, surpassing SOTA baseline *SWE-Fixer-Qwen-72B* with the same parameter scale, which only reached 24.7%(*Lite*) and 32.8%(*Verified*). Given precise issue locations as input, our fine-tuned *Qwen2.5-72B-Instruct* model achieves an impressive issue resolution rate of 43.8%(*Verified*), comparable to the performance of *Deepseek-v3*. We open-source our MCTS-REFINE framework, CoT dataset, and fine-tuned models to advance research in AI-driven software engineering.

***Index Terms*—MCTS, CoT, Fine-Tuning, Issue Resolution.**

I. INTRODUCTION

In recent years, large language models (LLMs) have demonstrated remarkable capabilities in tackling complex tasks, particularly achieving significant progress in code generation and issue resolution within the field of software engineering [1]–[3]. By comprehending and reasoning about large codebases, LLMs can efficiently perform tasks such as code completion [4]–[6], issue patching [11], and code refactoring [7]. State-of-the-art automated software development tools primarily rely on closed-source models (e.g., *GPT-4o* [8] and *Claude-3.5-Sonnet* [9]), which achieve strong performance on repository-level issue resolution benchmarks like *SWE-bench Lite* and *SWE-bench Verified* [11]. However, the dependence on external APIs, high invocation costs, and potential privacy concerns associated with closed-source models limit their widespread adoption in real-world scenarios. In contrast, open-source LLMs have emerged as a crucial alternative for advancing automated software development due to their flexibility, transparency, and controllability. However, open-source LLMs (<100B parameters) demonstrate significantly inferior performance in issue resolution tasks compared to their closed-source counterparts, with this performance gap substantially constraining their practical utility [23].

Recent advances demonstrate that generating high-quality Chain-of-Thought (CoT) data and employing them for fine-tuning represents an effective methodology for enhancing open-source LLMs' complex task performance [12], [13]. Training LLMs with CoT data containing explicit reasoning steps enables superior contextual comprehension and stepwise solution derivation. Research [14], [15], [21] indicates that **providing LLMs with correct reasoning steps—even minimally critical information—can substantially enhance performance, whereas erroneous reasoning steps significantly impair their capabilities. This demonstrates that high-quality CoT data is essential for advancing the reasoning performance of LLMs.**

Existing approaches [23], [25]–[27] for repository-level issue resolution typically employ single-turn Q&A prompting to generate CoT data. This involves synthesizing complete reasoning paths by providing issue descriptions and repository context, followed by performance enhancement through supervised fine-tuning (SFT) [16] or reinforcement learning (RL) [17]. However, these methods still face two key limitations:

- ***Inadequate Rejection Sampling Mechanisms for CoT Generation:*** Existing approaches to CoT data generation

* Ying Wang and Zhao Wei are the corresponding authors.

2643-1572/25/$31.00 ©2025 IEEE
DOI 10.1109/ASE63991.2025.00154

frequently employ permissive rejection sampling criteria - typically using validation metrics like *Jaccard Similarity* against developer-approved issue patch records (*ground truth*). While this strategy enhances generation efficiency, it risks compromising dataset quality by allowing defective reasoning chains to persist (see § II-A).

- ***Lack of Systematic Validation for Intermediate Reasoning Steps in CoT:*** Existing approaches typically employ "*Single-turn Q&A prompting*"—providing LLMs with issue descriptions and repository context to generate issue resolution CoT data—while only validating the final output and neglecting intermediate reasoning steps. When logical errors or omissions occur in these intermediate steps, they propagate through subsequent reasoning, ultimately compromising the quality of CoT data (see § II-B).

The recent successful applications of Monte Carlo Tree Search (MCTS) in multiple domains offer a promising solution to the aforementioned challenges [18]–[20]. By exploring multiple reasoning paths, MCTS can effectively enhance the efficiency of solving complex tasks. Although MCTS has demonstrated significant potential in machine translation and data reasoning tasks, its application in generating high-quality CoT data and improving LLMs' performance in repository-level issue resolution remains underexplored.

To address these challenges, we propose MCTS-REFINE, an enhanced *Monte Carlo Tree Search*-based algorithm. Our approach dynamically validates and optimizes intermediate reasoning steps through a rigorous CoT sampling strategy, effectively constructing high-quality CoT data MCOT to significantly improve LLM performance in issue resolution tasks. The key innovations of MCTS-REFINE include:

- ***MCTS with Reflective Mechanism:*** We enhance standard MCTS (***Selection***, ***Expansion***, ***Simulation***, and ***Backpropagation***) with a reflection mechanism that validates intermediate reasoning steps against ground truth. By integrating rejection sampling and refinement phases, our approach dynamically corrects errors and optimizes reasoning paths, significantly improving CoT data quality.
- ***Subtask Decomposition for Issue Resolution:*** We decompose the issue resolution process into three distinct subtasks: ***File Localization***, ***Fault Localization***, and ***Patch Generation***. This hierarchical decomposition not only reduces task complexity but also establishes clear ground truth criteria for each subtask. By integrating with the MCTS-REFINE algorithm, our approach generates high-quality CoT data at each subtask level, ensuring that every intermediate output effectively contributes to the final solution.
- ***Rigorous Chain-of-Thought Sampling Protocol:*** For each of the three subtasks, we implement a strict rejection sampling mechanism based on their respective ground truth criteria. Specifically: (1) ***File Localization*** requires reasoning paths to produce file paths that exactly match those in developer-verified patches. (2) ***Fault Localization*** demands precise identification of classes/methods that completely align with ground truth. (3) ***Patch Generation*** enforces generation of modified code that is identical to the actual developer patches. This rigorous sampling protocol enables MCTS-REFINE to guarantee both correctness and consistency throughout the reasoning paths.

To validate the effectiveness of MCTS-REFINE, we conducted supervised fine-tuning on open-source LLMs: *Qwen2.5-Coder-7B-Instruct*, *Qwen2.5-Coder-32B-Instruct*, and *Qwen2.5-72B-Instruct* [22] using the MCTS-REFINE-generated CoT data, followed by experimental evaluation on the *SWE-bench* benchmark. Our evaluation results demonstrate significant performance improvements across ***File Localization***, ***Fault Localization***, and ***Patch Generation*** subtasks for our fine-tuned LLMs. Our key findings are:

(1) Fine-tuned *Qwen2.5-**72B**-Instruct* achieved resolution rates of 28.3% and 35.0% on *SWE-Bench-Lite* and *SWE-Bench-Verified*, respectively, surpassing SOTA baseline *SWE-Fixer-Qwen-**72B*** with the same parameter scale, which only reached 24.7% and 32.8%.

(2) Fine-tuned *Qwen2.5-Coder-**32B**-Instruct* achieved resolution rates of 25.7% and 32.4% on *SWE-Bench-Lite* and *SWE-Bench-Verified*, respectively, surpassing the SOTA baseline *SoRFT-Qwen-**32B*** with the same parameter scale, which scored 24.0% and 30.8%.

(3) Fine-tuned *Qwen2.5-Coder-**7B**-Instruct* achieved resolution rates of 16.3% and 22.6% on *SWE-Bench-Lite* and *SWE-Bench-Verified*, respectively, outperforming the SOTA baseline *SoRFT-Qwen-**7B*** with the same parameter scale, which scored 14.0% and 21.4%.

(4) Given precise issue locations as input, our fine-tuned *Qwen2.5-72B-Instruct* model achieves an impressive issue resolution rate of 43.8%, comparable to the performance of *Deepseek-v3*.

This paper makes the following contributions:

- ***MCTS-Refine Framework:*** We present MCTS-REFINE, a framework for automated generation of high-quality CoT data for issue resolution. Our method integrates MCTS with a reflection mechanism to dynamically optimize reasoning steps, while decomposing the task into three structured subtasks with rigorous ground-truth alignment, ensuring superior data quality.
- ***Substantial Performance Improvements:*** Fine-tuning LLMs with our MCTS-REFINE-generated dataset yields significant enhancements in reasoning performance across all subtasks. Experimental results demonstrate state-of-the-art performance on both *SWE-bench Lite* and *SWE-bench Verified* benchmarks, establishing a new paradigm for applying LLMs to real-world software engineering challenges.
- ***Open-sourced Artifacts:*** We open-source both the MCOT dataset generated by MCTS-REFINE and our fine-tuned LLMs on our website (***https://mcts-refine.github.io/***), to facilitate further research.

II. LIMITATIONS OF EXISTING WORK

To address complex reasoning challenges in repository-level issue resolution, recent studies [23], [25]–[28] employ a two-stage approach: (1) Leveraging state-of-the-art LLMs (e.g., *GPT-4o* [8], *Claude-3.5-Sonnet* [9]) to generate phased CoT data for issue localization and patch generation; (2)

Fine-tuning parameter-efficient open-source foundation models (e.g., *LLaMA3-7B* [10], *Qwen2.5-7B* [22]) with the synthesized CoT data to enhance task-specific reasoning performance. Table I provides a comparison of existing work [23], [25]–[28], highlighting their distinctive features. Although these approaches have achieved notable progress in repository-level issue resolution, they exhibit two critical limitations in CoT data synthesis: ① ***inadequate rejection sampling mechanisms for CoT generation***, and ② ***lack of systematic validation for intermediate reasoning steps in CoT***. **When exposed to CoT datasets contaminated by defective reasoning steps during fine-tuning, LLMs cannot acquire the essential logical schemata for reliable issue resolution.**

A. *Limitation 1: Inadequate Rejection Sampling Mechanisms for CoT Generation*

To ensure the high-quality of CoT data generation, prior studies [23], [25]–[27] implements **rejection sampling with validation metrics** to filter out CoT instances that deviate from developer-approved issue patch records (***ground truth***):

- ***Lingma-SWE GPT*** [23]: Utilizing *Jaccard Similarity* and *CodeBLEU metrics* to eliminate low-quality CoT data that exhibits significant semantic deviation from ground truth in both issue localization and patch generation sub-tasks.
- **SORFT** [25]: Filtering out CoT data that exhibit no overlap with ground truth specifications in either: (1) the actual patch locations (file names and line numbers), or (2) the code content of gold patches.
- ***SWE-RL*** [26]: Filtering out CoT data that violates code diff formatting requirements, without verifying the actual correctness of the reasoning chains.
- ***SWE-Gym*** [27]: Validating CoT correctness through unit test execution and eliminating failing instances. While this improves CoT data quality, the per-instance test environment requirement makes it prohibitively expensive to scale.
- ***SWE-Fixer*** [28]: The CoT data received no validation of its reasoning steps or final outputs.

However, the rejection sampling mechanisms proposed by the existing issue resolution approaches [23], [25]–[27] cannot ensure the correctness of filtered CoT data. As illustrated in Figure 1(a), even when prompting *GPT-4o* with *issue descriptions*, *target code*, and *developers' gold patches*, the generated CoT data still contain errors. Although the CoT contains the correct modification steps (e.g., changing `offset_dims` to a `tuple` type), it also includes incorrect operations not present in the gold patch (such as converting the type of `axes` to a `tuple`). Additionally, the CoT omits a critical modification from the ground truth (assigning axes to `start_index_map` after converting it to a `tuple` type). Existing rejection sampling mechanisms that use validation metrics like *Jaccard Similarity*, *CodeBLEU*, *overlap with ground truth patches*, or *code diff formatting requirements* fail to effectively filter out such defective CoT data.

B. *Limitation 2: Lack of Systematic Validation for Intermediate Reasoning Steps in CoT*

Existing works [23], [25]–[27] typically adopt a "single-turn Q&A" approach, where LLMs are prompted with information such as *issue descriptions* and *repository context* to generate CoT for issue resolution. However, these works only validate the final outputs while failing to detect and correct errors in the intermediate reasoning steps of the CoT. This approach may cause the LLMs to exhibit "error propagation" during CoT generation, where mistakes in earlier steps lead to subsequent reasoning deviating from the correct trajectory. Particularly in repository-level issue resolution, such errors tend to accumulate and amplify, ultimately compromising the quality of the CoT data.

As illustrated in Figure 1(b), even when the LLM's output perfectly matches the developer's gold patch, errors may persist in the CoT's intermediate reasoning steps. The absence of dynamic validation and correction mechanisms for intermediate reasoning steps can significantly reduce the success rate of generating accurate CoT data. More critically, enforcing a strict "*exact match with gold patch*" validation metric to filter "single-turn Q&A"-generated CoT data—without intermediate reasoning corrections—would severely limit valid data retention, undermining fine-tuning effectiveness.

III. MCTS-REFINE APPROACH

Insight: "Single-turn Q&A prompting"—providing LLMs with issue descriptions and repository context to generate issue resolution CoT data—lacks validation mechanisms for intermediate reasoning steps. The absence of rigorous rejection sampling compromises intermediate reasoning step quality, while requiring *exact ground truth alignment* severely limits retained high-quality CoT instances under "single-turn Q&A prompting" approach, ultimately undermining model training effectiveness. Monte Carlo Tree Search (MCTS) [29] presents a viable alternative by efficiently navigating complex solution spaces. Unlike "single-turn QA-based CoT generation", MCTS enables (1) hierarchical decision tree exploration and (2) dynamic selection of optimal reasoning paths through iterative simulation. Yet traditional MCTS faces two key challenges: (1) error propagation from unverified intermediate steps, and (2) a reward-maximization strategy may fail to identify correct reasoning paths when all candidate paths contain flaws.

Architecture: To address the aforementioned issues, we propose MCTS-REFINE, an enhanced approach that integrates **CoT sampling** and **intermediate step reflection mechanism** into MCTS. Figure 2 illustrates the architecture of MCTS-REFINE, which consists of five core components: (1) ***Selection***: Chooses the current optimal node using the Upper Confidence Bound (UCB) strategy. (2) ***Expansion***: Generates new reasoning steps and assigns reward values. (3) ***Rejection Sampling***: Validates whether the reasoning path can correctly solve the task. (4) ***Refinement***: Provides feedback and corrects erroneous steps in the reasoning path. (5) ***Backpropagation***: Updates node reward values and visit counts to optimize the entire reasoning path. It is worth emphasizing that:

- ***Rejection Sampling*** component employs a strict "exact match" mechanism against ground truth to verify the validity of the current reasoning path.
- ***Refinement*** component performs two tasks: (1) detecting deviations from ground truth in current reasoning steps

TABLE I: A comparison of existing works' distinctive features

Approach	*Sub-Task CoT*		*Validation for CoT Steps*	*Rejection Sampling for CoT*	
	Localization	*Generation*		*Localization*	*Generation*
Lingma-SWE GPT [23]	✓	✓	✗	*Jaccard Similarity*	*CodeBLEU*
SORFT [25]	✓	✓	✗	*Overlap with the locations of gold patch*	*Overlap with the content of gold patch*
SWE-RL [26]	✓	✓	✗	*Include the file of gold patch*	*Correct code diff format*
SWE-Gym [27]	✗	✓	✗	✗	Unit Test
SWE-Fixer [28]	✗	✓	✗	✗	✗

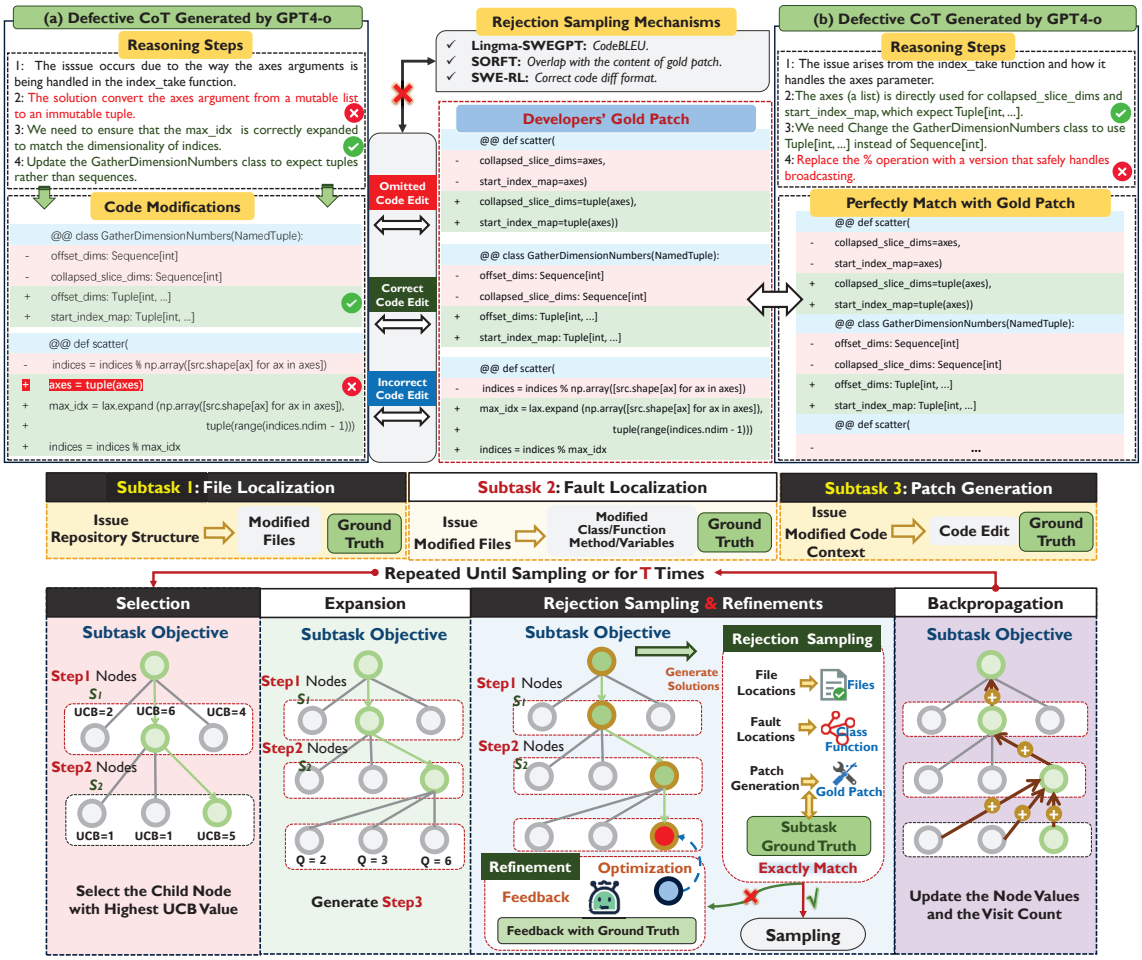

Fig. 2: An Overall Architecture of MCTS-REFINE

while providing corrective suggestions, and (2) optimizing erroneous steps by incorporating these modifications.

To address the complex issue resolution tasks, we adopt the AGENTLESS framework [30] by decomposing the process into three subtasks: ***File Localization***, ***Fault Localization***, and ***Patch Generation***. This structured decomposition enables: (1) ***clear definition of input-output specifications for each subtask***; (2) ***systematic construction of ground truth datasets for each subtask phase***; (3) ***generate high-quality CoT data specifically tailored for each subtask leveraging*** **MCTS-REFINE**. This approach enhances the model's reasoning performance across all subtasks in the issue resolution pipeline.

A. Task Decomposition for Issue Resolution

Task decomposition in issue resolution effectively reduces complexity while clarifying objectives for individual components. Our approach partitions the issue resolution process into three distinct subtasks, employing MCTS-REFINE to generate high-quality CoT reasoning for each. In contrast to conventional relaxed sampling mechanism adopted by existing approaches [23], [25]–[28], MCTS-REFINE implements a rigorous stepwise generation process with a ground truth-aligned rejection sampling criterion that guarantees reasoning path validity.

File Localization: This phase focuses on identifying target files for modification based on *issue descriptions* and *repository structure*. During CoT data sampling, we process both the issue description and repository structure as inputs. MCTS-REFINE progressively generates and explores reasoning paths, accepting CoT sequences only when their resulting file localization matches the actual files modified by developers.

Fault Localization: This phase focuses on identifying issue-introducing code elements (e.g., classes, methods, functions, and global variables), building upon ***File Localization***. The

CoT generation process takes the issue description and previously localized file structure as inputs, retaining only those reasoning paths whose predicted fault locations exactly match the actual code positions modified in developers' gold patches.

Patch Generation: This phase empowers the LLM to generate precise code fixes for the identified issues by processing both the issue description and fault-localized code elements along with their relevant contextual code segments from the previous ***Fault Localization*** phase. During CoT construction, the system validates a reasoning path only when the LLM-generated patch achieves exact match with developers' gold patch. To eliminate confounding factors, we: (1) integrate the LLM's code edits back into the original repository context; (2) normalize the output by removing comments, whitespace, and line breaks; (3) perform strict differential comparison against the gold patch. This rigorous verification protocol ensures the training data's authenticity by enforcing syntactic and functional equivalence with developer solutions.

B. MCTS With Rejection Sampling and Refinement

Current "single-turn QA-based CoT generation" approaches lack systematic validation of intermediate reasoning steps, preventing dynamic path adjustment and often yielding suboptimal solutions with defective CoT data. While traditional MCTS algorithms can dynamically narrow the search space by simulating multiple reasoning trajectories (through *Selection*, *Expansion*, *Simulation*, and *Backpropagation* phases), they exhibit two critical flaws: (1) During simulation, MCTS only estimates node potential without verifying step correctness, allowing error propagation in multi-step reasoning. (2) The threshold-based path selection cannot guarantee the final solution's validity. MCTS-REFINE addresses these limitations by: (1) Introducing a CoT sampling strategy for real-time correct path identification; (2) Incorporating a reflection mechanism to rectify erroneous reasoning steps.

MCTS-REFINE fundamentally models the reasoning process as a search tree structure, where: ***the root node encapsulates the subtask objective, and each subsequent node represents an individual reasoning step.*** The algorithm iteratively refines solutions through five core phases: ***Selection***, ***Expansion***, ***Rejection Sampling***, ***Refinement***, and ***Backpropagation***. Below we elaborate on the phase-specific implementations.

B.1. Selection

During the ***selection phase***, MCTS-REFINE balances exploration and exploitation by traversing the tree using a predefined policy until it reaches a not fully explored node. This policy weighs both the node's reward value and visit count, ensuring a preference for high-reward nodes while still allowing exploration of less-visited ones—thus preventing convergence to local optima.

In the issue resolution task, the **root node** corresponds to the subtask objective (e.g., precisely identifying target files in *File Localization*), while **child nodes** encode all potential reasoning steps. The **leaf nodes** constitute the search boundary, comprising either: unexplored nodes awaiting expansion, or terminal nodes representing verified solutions. Each iteration initiates at the root, then recursively applies *UCB*-based selection to navigate through child nodes until encountering a leaf node.

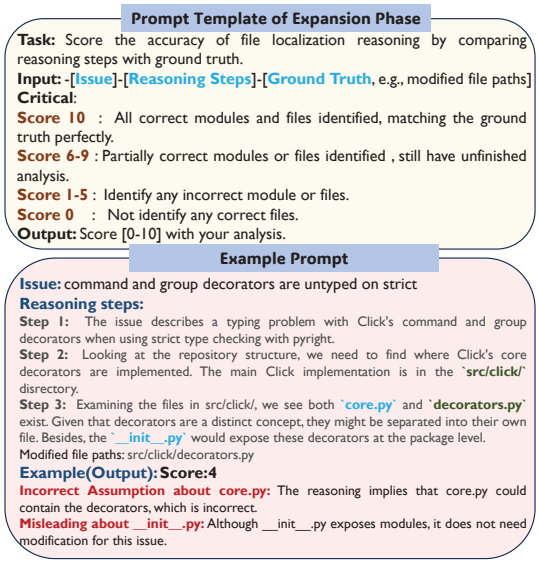

Fig. 3: An illustrative Prompt Template of ***Expansion Phase*** for ***File Localization*** Subtask

This traversal path's node sequence then provides contextual grounding for the subsequent ***expansion phase***.

$$UCB(S) = Q_S + \epsilon \cdot \sqrt{\frac{\ln N_{\text{parent}}}{N_S}} \tag{1}$$

In this formulation, the reward score Q_S for node S is assigned by the LLM during node expansion, where N_{parent} and N_S represent the visit counts of the parent node and current node S respectively. The exploration parameter ϵ is set to 0.5 following established practice [31], balancing exploration of new nodes against exploitation of known high-reward paths.

B.2. Expansion

The ***expansion phase*** creates new child nodes from the leaf node generated in the ***selection phase***, with each child node corresponding to a unique reasoning step. Formally, let S_k represent the current selected node (denoting the k-*th* reasoning step), and $P_k = (S_1, S_2, \cdots, S_k)$ define the complete reasoning path from the root node. MCTS-REFINE extends path P_k by creating b new child nodes S_{k+1}, where we adopt the conventional branching factor of b=3 as established in prior MCTS literature [31].

The LLM assigns a reward score to each newly generated child node to evaluate the validity of its corresponding reasoning step. This assessment follows a rigorous process: (1) inputting the current reasoning path and subtask ground truth to the LLM, (2) analyzing the deviation between them using predefined scoring metrics in the prompt template, and (3) generating a reward score on a **0-10 scale**, where higher values indicate closer alignment with the correct solution. We have developed specialized prompt templates for each subtask to ensure consistent quality assessment of reasoning paths.

As depicted in Figure 3, during ***File Localization***, the LLM assigned a reward score of 4 to the current reasoning path based on the provided inputs. This suboptimal score resulted

from a critical error in *Step 3*, where the LLM incorrectly identified both `core.py` and `_init_.py` files, leading to substantial deviation from the expected reasoning path.

*B.3. **Rejection Sampling***

In the classical MCTS algorithm, after ***expansion phase***, a node is selected for simulation based on its *UCB* value. While this simulation phase aims to estimate the node's potential value, it cannot verify whether the current reasoning path can fully solve the subtask—particularly when the path contains flawed reasoning steps. This limitation arises because each node only stores partial reasoning steps toward task completion without generating the actual subtask solution, potentially leading to unnecessary exploration of incorrect nodes in later iterations. To overcome this issue, we introduce CoT sampling and reflection immediately after node expansion. This enhancement ensures that: (1) proactively identifies valid reasoning paths, and (2) detects and corrects flawed reasoning steps during exploration.

The mechanism evaluates reasoning paths from the root node to its expanded optimal nodes by using an LLM to generate corresponding subtask solutions. These generated solutions are then compared against the ground truth to determine whether the current reasoning path should be sampled. The specific sampling rules are as follows:

- ***File Location Subtask:*** Samples reasoning paths where the identified files exactly match the developer-modified files.
- ***Fault Localization Subtask:*** Samples reasoning paths where the identified class signatures, method signatures, function signatures, and global variables completely align with the developer modifications.
- ***Patch Generation Subtask:*** Integrates the generated edit code (based on the reasoning path) into the original repository and compares the modified files with the developer-edited versions. Our tool samples reasoning paths where the generated edits exactly match the ground truth. It should be noted that the comparison ignores whitespace, line breaks, and comments. We intentionally avoid direct comparisons between the generated edits and the gold patch. This is because the gold patch only contains modified sections with limited context, whereas the LLM's output typically includes both modifications and additional contextual code. This approach effectively prevents potential false negatives in our evaluation arising from the inherent structural disparities between these two distinct formats.

*B.4. **Refinement***

The ***refinement phase*** serves to identify and rectify potential errors in current reasoning steps through two coordinated sub-phases: the ***Feedback sub-phase*** detects reasoning errors and generates diagnostic feedback, while the ***Optimization sub-phase*** subsequently performs corrective modifications and optimizations based on this feedback.

$$F_{k+1} = Feedback(P_k, GT) \tag{2}$$

During the ***Feedback sub-phase***, the model generates feedback F_{k+1} by evaluating the current reasoning path $P_k = \{S_1, S_2, \cdots, S_{k+1}\}$ against the ground truth (GT), as formalized in Equation (2). This phase executes two critical functions:

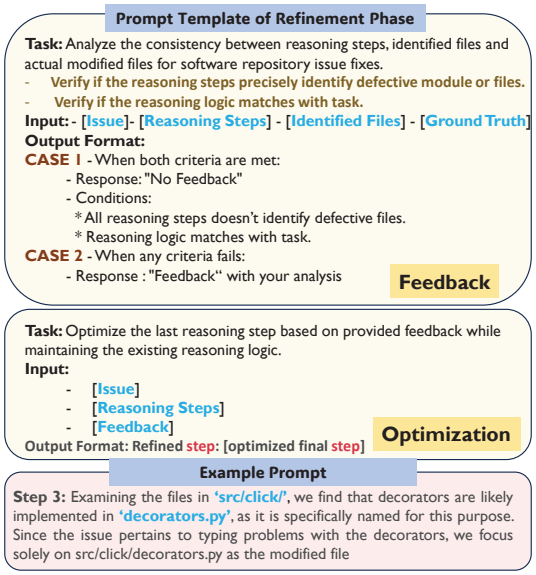

Fig. 4: An illustrative Prompt Template of ***Refinement Phase*** for ***File Localization*** Subtask

(1) assessing the alignment between the reasoning path and ground truth, and (2) generating specific corrective suggestions for identified deviations. Figure 4 shows the prompt template of ***Feedback sub-phase***, which produces two distinct outputs:

- ***No-Feedback:*** Indicates full path correctness requiring no modification.
- ***Feedback:*** Provides specific corrective instructions when errors are detected. For instance, in ***File Localization*** subtasks, the feedback mechanism evaluates whether the current path includes directories or files erroneously identified by the developer (as specified in the ground truth). As demonstrated in Figure 4's prompt template, when the path incorrectly locates both `core.py` and `_init_.py` files, the model not only flags these inaccuracies but also generates concrete revision guidance. Similarly, for ***Patch Generation*** subtasks, the feedback verifies both the syntactic correctness of modified code segments and their semantic alignment with the ground truth's modification strategy.

Upon receiving corrective ***Feedback***, the system initiates the ***Optimization sub-phase*** to refine the reasoning path. The LLM dynamically adjusts the current reasoning step S_{k+1} using the diagnostic feedback and optimizes the reasoning path. This process can be formally represented as:

$$\tilde{S}_{k+1} = Optimization(S_{k+1}, F_{k+1}) \tag{3}$$

where $\tilde{S}_{k+1}$ represents the refined reasoning step after optimization. This phase focuses exclusively on improving the current step while deliberately restricting next-step generation. As illustrated in Figure 4, the LLM successfully rectifies flawed reasoning steps by integrating reflective analysis with current step evaluation. This constrained correction approach ensures: (1) ***Localized Modification Scope:*** Each adjustment is precisely bounded to the target step. (2) ***Global Path Stability:*** Preserves the structural integrity of the overall reasoning chain. (3) ***Controlled Optimization:*** Prevents cascading disruptions

across the reasoning path.

B.5. ***Backpropagation***

During the ***Backpropagation phase***, we perform a bottom-up update from the selected node S_{k+1} back to the root node. This sequential upward propagation ensures that all nodes along the path are adjusted according to the latest reward values. The key objective of this phase is to update: (1) the visit count N_S, and (2) the reward value Q_S, for each reasoning step node S_i in the path $P_k = \{S_1, S_2, \cdots, S_{k+1}\}$. The update process involves: (1) incrementing the visit count N_{S_i} of each node S_i by 1. (2) updating the reward value Q_{S_i} based on the reward contributions from b child nodes S_{i+1} of S_i.

$$Q_{increment} = \frac{\sum_{i=1}^{b} N_{S_{i+1}} \times Q_{S_{i+1}}}{\sum_{i=1}^{b} N_{S_{i+1}}} \tag{4}$$

$$Q_{S_i} = \alpha \times Q_{S_i} + (1 - \alpha) \times Q_{increment} \tag{5}$$

The weighting factor α is set to 0.5 to balance between: (1) the parent node's current reward value Q_{S_i}, and (2) the reward increments $Q_{increment}$ from its b child nodes S_{i+1}. Through this update rule, Q_{S_i} progressively incorporates child-node rewards, resulting in a more accurate representation of the node's importance in the reasoning path P_k.

Repeat the process (***selection*** $\rightarrow$ ***expansion*** $\rightarrow$ ***rejection sampling*** $\rightarrow$ ***refinement*** $\rightarrow$ ***backpropagation***) until a reasoning path and result consistent with the ground truth are sampled, or terminate after T iterations.

C. Implementation

Dataset Construction: To construct MCoT dataset for the issue resolution task, we leverage the *SWE-Fixer-Train-110K* dataset collected by SWE-Fixer [28], comprising 110,000 high-quality Issue and Pull Request (PR) pairs sourced from GitHub repositories (excluding those overlapping with SWE-Bench). For effective subtask decomposition, we utilized the GitHub REST API to enrich each Issue-PR pair with contextual information including: (1) repository directory structures, and (2) pre-patch code file contents, thereby creating comprehensive inputs for each subtask's processing pipeline.

Inference Framework: We leverage our constructed CoT data MCoT to fine-tune LLMs across multiple parameter scales and integrate the resulting models into the Agentless 1.0 [30] framework. This framework operates through two distinct phases: (1) *Fault Localization*: The LLM hierarchically narrows down potential issue-introducing candidates (files $\rightarrow$ classes $\rightarrow$ functions) to pinpoint issue-relevant code segments. (2) *Code Edit Generation*: The LLM then produces syntactically valid code modifications to resolve the identified issues. The Agentless 1.0 framework's architecture is generally compatible with our CoT data, enabling end-to-end evaluation of MCTS-Refine's *Rejection Sampling* and *Refinement* methodology.

IV. Evaluation

In evaluation section, we propose three research questions:

RQ1: (*Overall Effectiveness of Issue Resolution*): *To what extent does the fine-tuned model improve overall performance in issue resolution tasks compared to SOTA baselines?*

RQ2: (*Effectiveness of File Location and Fault Localization*): *What is the performance of the fine-tuned model on the File Localization and Fault Localization subtasks?*

RQ3: (*Effectiveness of Patch Generation*): *What is the performance of the fine-tuned model on the Patch Generation subtask?*

A. Experiment Setup

1) ***CoT Data Synthesis****:* To reduce computational overhead and experimental costs, we randomly selected 20,000 Issue-PR pairs from *SWE-Fixer-Train-110K* [28] to build our CoT dataset using MCTS-Refine, for three core issue resolution subtasks—***File Localization***, ***Fault Localization***, and ***Patch Generation***. The data synthesis employed *DeepSeek-v3*, which provides three advantages over *GPT-4o* and *Claude-3.5-Sonnet*: (1) *lower computational costs*, (2) *complete open-source availability*, and (3) *full local deployability*.

The refinement process iterates through six MCTS phases—***selection***, ***expansion***, ***rejection***, ***sampling***, ***refinement***, and ***backpropagation***—terminating when either: (1) the derived solution perfectly matches the ground truth, or (2) reaches our empirically set maximum of 50 iterations (T = 50) based on the existing MCTS approach [31].The final curated fine-tuning dataset comprises 52,068 high-quality samples with the following distribution: (1) ***File Localization***: 18,340 samples; (2) ***Fault Localization***: 17,338 samples; (3) ***Patch Generation***: 16,390 samples.

2) ***Training Configuration****:* We fine-tuned three open-source LLMs of varying parameter scales using our constructed CoT data: *Qwen2.5-Coder-7B-Instruct*, *Qwen2.5-Coder-32B-Instruct*, and *Qwen2.5-72B-Instruct* [22]. These models achieve SOTA performance at their respective parameter scales and demonstrate robust code processing capabilities. For the 7B and 32B models, we conducted full-parameter supervised fine-tuning (SFT) using *Llama-Factory* [41], completing training on eight `NVIDIA H800 80GB GPUs`. The training configuration included: (1) *Context window length*: 32k tokens; (2) *Batch size*: 128; (3) *Training duration*: 2 epochs; (4) *Learning rate*: 5e-6 initial rate with cosine decay; (5) *Warmup ratio*: 3%. Due to computational constraints, we adapted the 72B model using LoRA [42] fine-tuning via the *Llama-Factory* framework while maintaining consistent hyperparameters.

3) ***Baselines****:* Our evaluation systematically compares the fine-tuned model against both proprietary and open-source LLMs across different issue resolution frameworks.

- ***Framework Selection:*** Our comparison considers two types of issue resolution frameworks: (1) ***Agent-based frameworks*** (e.g., LingmaAgent-SWE-GPT [23], OpenHands [33], SWE-Agent [32]) that dynamically integrate external tools like code analyzers and compilers to enable adaptive, context-aware repair processes. Notably, SWE-Search [44] utilizes MCTS to enhance agent decision-making in dynamic environments. (2) ***Pipeline-based frameworks*** (e.g., SWE-Fixer [28]'s RAG-to-patch generation workflow, Agentless [30]'s decomposition methodology) that employ structured, modular task sequencing with well-defined interfaces - with the former excelling in flexibility through on-demand tool invocation while the latter provides

reproducible evaluation frameworks particularly valuable for benchmarking LLM capabilities.

- ***Model Selection:*** We comprehensively evaluated our fine-tuned LLMs against ***cutting-edge proprietary models*** (including `OpenAI`'s *GPT-4* and *GPT-4o* [8] - renowned for their exceptional natural language understanding and efficient inference capabilities, and `Anthropic`'s *Claude 3 Opus* and *Claude 3.5 Sonnet* [9] - particularly strong in coding tasks and multi-turn interactions) as well as ***leading open-source models*** (*Qwen* series models like *Qwen2.5-Coder-7B-Instruct* [22] that we fine-tuned, and *Deepseek* models including the general-purpose *Deepseek-V3* [4] and reasoning-specialized *Deepseek-R1* [17]), with all comparison models carefully selected from prominent research works and the *SWE-bench* leaderboard to ensure a rigorous benchmark of our approach's effectiveness across different model categories and capabilities.

4) ***Benchmark and Metrics****:* We conducted rigorous performance assessments using the industry-standard ***SWE-bench*** benchmark suite, specifically leveraging both its ***Verified*** (500 samples) and ***Lite*** (300 samples) subsets comprising real-world GitHub issues [11]. *SWE-bench* has become the authoritative benchmark for issue resolution tasks, now serving as the standard evaluation framework for leading AI coding systems.

We analyzed model performance following the AGENTLESS [30] framework's workflow, focusing on four metrics: **(1)** ***%Resolved:*** *Success rate of model-generated patches passing all tests.* **(2)** ***%FileHit:*** *Precision in file-level localization.* **(3)** ***%FuncHit:*** *Precision in class/method-level localization.* **(4)** ***%Line-Hit:*** *Precision in code line-level localization.*

B. RQ1: Overall Effectiveness of Issue Resolution

Methodology. To comprehensively evaluate the performance of the fine-tuned model in resolving real-world GitHub issues, we conducted comparative experiments with baselines based on *SWE-bench Verified* and *SWE-bench Lite*. Our evaluation protocol strictly provided only: (1) the original issue description and (2) corresponding repository context as model inputs, with patch correctness being definitively determined through test-case verification. We measured the model's performance by quantifying the number of issues resolved (***%Resolved***).

Results. Table II and Figure 5 present the comparative performance of our fine-tuned model against existing approaches on both *SWE-Bench Lite* and *SWE-Bench Verified* benchmarks. The results are categorized into two distinct groups: proprietary model-based solutions and open-source model approaches. We observe that the highest resolution rates are predominantly achieved by closed-source model-based approaches. For instance, the *OpenHands* method, which utilizes the *Claude-3.5-Sonnet* model, achieves resolution rates of 41.7% on *SWE-Bench Lite* and 53% on *SWE-Bench Verified*.

Comparison with Proprietary Model Frameworks. Notably, our fine-tuned 72B model outperforms several proprietary approaches on both *SWE-Bench Lite* and *Verified* benchmarks, including methods based on *GPT-4*, *GPT-4o*, and *Claude-3-Opus*. For instance, the SWE-AGENT (using *Claude-3-Opus*) achieves 33.60% (*Verified*) and 23.00% (*Lite*) resolution rates.

TABLE II: Overall Effectiveness of Issue Resolution

Framework	*Models*	*SFT*	*Type*	*Verified*	*Lite*
Proprietary Models					
OpenHands [33]	*GPT4-o*	*N/A*	*Agent*	-	22.0%
OpenHands [33]	*Claude-3.5-Sonnet*	*N/A*	*Agent*	53.0%	41.7%
SWE-Agent [32]	*Claude-3-Opus*	*N/A*	*Agent*	18.2%	11.7%
SWE-Agent [32]	*GPT-4o*	*N/A*	*Agent*	23.0%	18.3%
SWE-Agent [32]	*Claude-3.5-Sonnet*	*N/A*	*Agent*	33.6%	23.0%
SWE-Search [44]	*GPT-4o*	*N/A*	*Agent*	32.6%	31.0%
Agentless [30]	*GPT-4o*	*N/A*	*Pipeline*	38.8%	32.0%
Agentless [30]	*Claude-3.5-Sonnet*	*N/A*	*Pipeline*	50.8%	40.7%
Open-source Frameworks & Open-source Models (<10B)					
Openhands [33]	*SWE-Gym-Qwen-7B*	*SFT*	*Agent*	10.6%	10.0%
Agentless [30]	*SoRFT-Qwen-7B*	*RL*	*Pipeline*	21.4%	14.0%
Lingma-SWE-GPT [23]	*Lingma-SWE-GPT-7B*	*SFT*	*Agent*	18.4%	12.0%
Open-source Frameworks & Open-source Models (>10B)					
Openhands [33]	*SWE-Gym-Qwen-14B*	*SFT*	*Agent*	16.4%	12.7%
Agentless [30]	*SoRFT-Qwen-32B*	*RL*	*Pipeline*	30.8%	24.0%
Lingma-SWE-GPT [23]	*Lingma-SWE-GPT-72B*	*SFT*	*Agent*	30.2%	22.0%
SWE-Search [44]	*Qwen-2.5-72b-Instruct*	*N/A*	*Agent*	23.4%	24.7%
SWE-GPT [23]	*SWE-Gym-Qwen-32B*	*SFT*	*Agent*	20.6%	15.3%
SWE-Fixer [28]	*SWE-Fixer-Qwen-72B*	*SFT*	*Pipeline*	32.8%	24.7%
Agentless [30]	*Deepseek-V3-671B*	*SFT*	*Pipeline*	42.0%	-
Agentless [30]	*Deepseek-R1-671B*	*RL*	*Pipeline*	49.2%	-
Fine-tuned Models based on our CoT Dataset					
Agentless [30]	***Fine-tuned Qwen2.5-7B-Instruct***	*SFT*	*Pipeline*	**22.6%**	**16.3%**
Agentless [30]	***Fine-tuned Qwen2.5-32B-Instruct***	*SFT*	*Pipeline*	**32.4%**	**25.7%**
Agentless [30]	***Fine-tuned Qwen2.5-72B-Instruct***	*SFT*	*Pipeline*	**35.0%**	**28.3%**

'-' means the corresponding technique did not report ***%Resolved*** results in its paper. '*N/A*' denotes the unknown training technique.
SFT denotes supervised fine-tuning; ***RL*** denotes reinforcement learning.
Baselines' ***%Resolved*** values are from the technique's original paper.

In contrast, our approach yields higher performance: The fine-tuned 72B model achieves 35.0% (*Verified*) and 28.3% (*Lite*). Even the 32B variant reaches 32.4% (*Verified*) and 25.7% (*Lite*), demonstrating competitive efficiency.

Comparison with Open-Source Model Frameworks. *Lingma SWE-GPT* (based on *Qwen2.5-72B-Instruct*) achieves resolution rates of 22.0% (*Lite*) and 30.2% (*Verified*). Additionally, *SWE-Search*, which is also based on *Qwen2.5-72B-Instruct*, achieves a resolution rate of 24.7% (Lite) and 23.4% (*Verified*) [45]. In comparison, our fine-tuned *Qwen2.5-72B-Instruct* improves performance by ⇑6.3% (*Lite*) and ⇑4.8% (*Verified*) over *Lingma SWE-GPT*, as well as a ⇑3.6% (*Lite*) and ⇑11.6% (*Verified*) over *SWE-Search*. For the 32B model tier, existing approaches like SORFT [25] (*Qwen2.5-Coder-32B-Instruct*) report 30.8% (*Lite*) and 24.0% (*Verified*), whereas our fine-tuned variant boosts resolution rates by ⇑1.6% and ⇑1.7%, respectively. For models under 10B parameters, SOTA approach SORFT [25] achieves maximum resolution rates of 21.4% (*Lite*) and 14.0% (*Verified*). Our fine-tuned *Qwen2.5-Coder-7B-Instruct* attains higher performance at ⇑1.2% and ⇑2.3%, respectively. The results demonstrate that our fine-tuned models achieve significant improvements in issue resolution capability, establishing new SOTA performance within their respective parameter scales.

As shown in Figure 6, we conducted a Venn diagram analysis to examine the overlap of successfully solved instances among different models on the *SWE-bench Verified* benchmark. The results reveal that our fine-tuned 7B, 32B, and 72B models uniquely resolved 5, 7, and 16 issues respectively - cases that other models failed to address. Additionally, each model shares a substantial number of commonly solved instances with other models. These findings demonstrate that our approach not only achieves superior performance in terms of resolution quantity, but also exhibits unique issue-solving

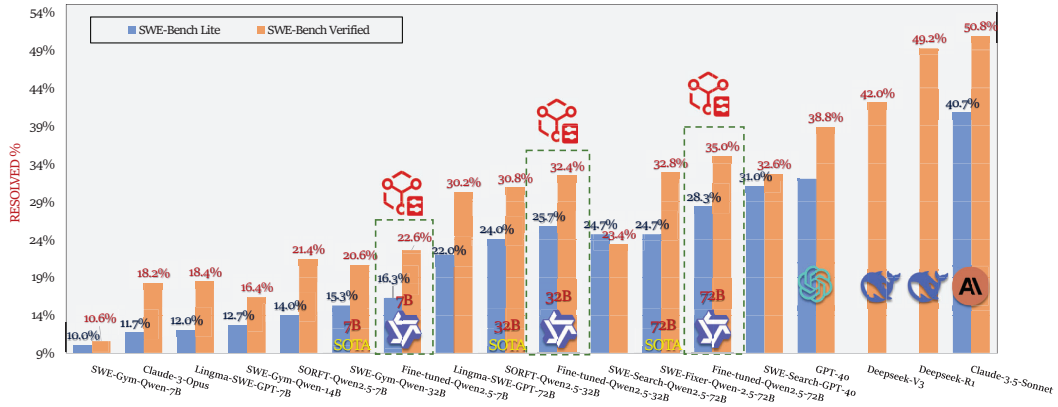

Fig. 5: Comparison of issue resolution rates across baselines

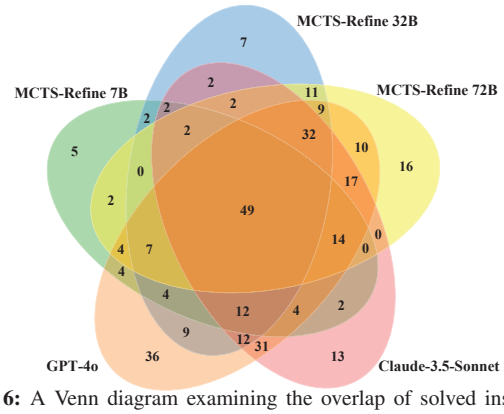

Fig. 6: A Venn diagram examining the overlap of solved instances among different models on the *SWE-bench Verified* benchmark

capabilities in terms of coverage scope.

Due to computational constraints, we adopted LoRA [42] for fine-tuning *Qwen2.5-72B-Instruct*, which achieved SOTA issue resolution rates at its parameter scale. However, empirical evidence shows that full-parameter fine-tuning consistently surpasses LoRA's performance given equivalent model capacity and training data [43]. Our analysis suggest that replacing LoRA with full-parameter fine-tuning on our CoT dataset could further improve the model's issue resolution capability.

Conclusion: *Our **7B**, **32B**, and **72B** models, fine-tuned using MCoT data, demonstrate significant improvements in issue resolution, achieving state-of-the-art resolution rates for their respective parameter scales. On SWE-bench Verified, they achieve **16.3%**, **25.7%**, and **28.3%** resolution rates, while on SWE-bench Lite, they reach **22.6%**, **32.4%**, and **35.0%**.*

C. RQ2: Effectiveness of File Location and Fault Localization

Methodology. Precise file and fault localization is fundamental to successful issue resolution, enabling both automated patch generation and developer debugging. To rigorously evaluate localization performance of our fine-tuned models, we leverage the *SWE-Bench-Verified* dataset, comparing model-predicted patch locations against ground-truth patches across three granularity levels:

- ***File Localization:*** A prediction is considered correct if the model's suggested file list includes the file actually modified by developers.
- ***Function Localization:*** After integrating the patch into the codebase, we use Abstract Syntax Tree (AST) analysis to identify the modified functions or classes. A prediction is successful if the model's output contains the exact function/class altered in the ground-truth patch.
- ***Line Localization:*** Following *Lingma-SWE-GPT* [23]'s approach, we extract the patched line along with its three surrounding lines as the localization scope. A match is confirmed if this range includes the actual modified lines.

TABLE III: Effectiveness of *File Localization* and *Fault Localization*

LLM	*Size*	*FileHit*	*FunHit*	*LineHit*
Qwen2.5-Coder-Instruct	7B	19.2%	13.9%	13.2%
Lingma SWE-GPT	7B	58.8%	42.2%	39.1%
Fine-tuned Qwen2.5-7B-Instruct	7B	64.4%	47.2%	41.8%
Qwen2.5-Coder-Instruct	32B	33.2%	28.8%	25.4%
Fine-tuned Qwen2.5-32B-Instruct	32B	72.0%	59.2%	49.4%
Qwen2.5-Instruct	72B	61.3%	45.7%	42.8%
Lingma SWE-GPT	72B	80.9%	66.2%	61.6%
Fine-tuned Qwen2.5-72B-Instruct	72B	81.8%	68.0%	63.0%
GPT-4o	Unknown	72.5%	55.7%	52.2%
Claude-3.5-Sonnet	Unknown	74.3%	58.1%	55.0%

Due to computational constraints, our evaluation focuses on comparing localization results (at file/function/line levels) with established baselines from prior work [23]. *Baselines'* ***%FileHit, %FunHit, %LineHit*** values are from LINGMA-SWE-GPT study [23].

We conducted comprehensive evaluations of the fine-tuned *Qwen2.5-Coder-7B-Instruct*, *Qwen2.5-Coder-32B-Instruct*, and *Qwen2.5-Coder-72B-Instruct* models, benchmarking their performance against open source approaches (including *Lingma-SWE-GPT* [23]) and proprietary models (such as *GPT-4o* and *Claude 3.5 Sonnet*).

Results. The results shown in Table III demonstrate that the fine-tuned *Qwen2.5-Coder* models achieve optimal performance across all granularity levels of localization tasks. Particularly noteworthy is the 72B model, which attains localization precision of 81.8%, 68.0%, and 63.0% at the file, method, and line levels respectively - outperforming existing open-source issue resolution approaches like *Lingma-SWE-GPT*.

At comparable parameter scales, both the fine-tuned 32B and 7B models consistently surpass other open-source models across all granularities. While proprietary models (*GPT-4o* and *Claude 3.5 Sonnet*) still maintain a performance advantage, the fine-tuned 72B model has achieved comparable or even superior localization accuracy to some proprietary models,

TABLE IV: Effectiveness of *Patch Generation*

Models	*Format Compliance Rate (%)*	*Resolution Rate (% Resolved)*
Qwen2.5-Coder-Instruct 7B	13.0%	3.8%
Fine-tuned Qwen2.5-7B-Instruct	**94.6%**	**28.6%**
Qwen2.5-Coder-Instruct 32B	23.4%	9.6%
Fine-tuned Qwen2.5-32B-Instruct	**96.0%**	**38.0%**
Qwen2.5-Instruct 72B	37.0%	14.4%
Fine-tuned Qwen2.5-72B-Instruct	**96.8%**	**43.8%**
Deepseek-V3	98.0%	47.8%
Deepseek-R1	98.6%	59.0%

Since the SOTA techniques *SORFT-Qwen-7B* and *SORFT-Qwen-32B* are not open-sourced, and agent-based frameworks (e.g., *Lingma SWE-GPT*, *SWE-Agent*) cannot isolate the patch generation phase, we excluded them from baselines in RQ3. Instead, we compared the patch generation capabilities of *Qwen2.5-Coder* models before and after fine-tuning.

further validating the effectiveness of both the MCTS-REFINE framework and our fine-tuning strategy.

The fine-tuned models demonstrate the following localization capabilities across different granularities:

- ***File Localization:*** The fine-tuned models exhibit superior precision in identifying issue-relevant files, particularly within complex software repositories, with the 72B model achieving a file localization precision of 81.8%. This performance improvement is primarily attributed to the rigorous rejection sampling strategy employed by the MCTS-REFINE framework during CoT data generation.
- ***Function Localization:*** Our fine-tuned 72B model reaches 68.0% accuracy in method localization, proving its enhanced capability in comprehending code semantics and structure.
- ***Line Localization:*** For line-level localization tasks, the fine-tuned 72B model achieves a precision of 63.0%, outperforming LINGMA-SWE-GPT [23]'s 61.6%. This demonstrates the model's enhanced precision in identifying specific code modification locations, providing a reliable foundation for subsequent patch generation.

Conclusion: Our fine-tuned models achieve optimal performance across all granularity levels of localization tasks. Particularly noteworthy is the 72B model, which attains localization precision of **81.8%**, **68.0%**, and **63.0%** at the file, method, and line levels respectively - outperforming the existing open-source issue resolution approaches.

D. RQ3: Effectiveness of Patch Generation

Methodology. To assess the improvement of fine-tuned models in the patch generation subtask, we conducted a comparative evaluation between the fine-tuned *Qwen2.5-Coder* models (*7B-Instruct*, *32B-Instruct*, and *72B-Instruct*) and their corresponding base models. The experiment was performed on the *SWE-bench Verified* benchmark with a specific focus on ***Patch Generation*** capability. To evaluate patch generation performance, **we provided models with precise issue locations and 20 surrounding lines of contextual code**, requiring them to produce patches in strict ***search-replace format*** (where "*search*" identifies the original problematic code and "*replace*" shows the corrected version) - a prerequisite for valid diff-based integration into original software repositories.

Our two-phase assessment ***first verifies format compliance*** (ensuring patches can be properly diff-applied), then ***measures repair effectiveness through %Resolved***, thereby evaluating both syntactic correctness and actual issue-patching capability.

Results. As evidenced in Table IV, fine-tuning yields significant improvements in patch generation performance. The pre-fine-tuned *Qwen 2.5* models demonstrated poor performance in generating syntactically correct code edits, with the 72B variant producing only 37.0% properly formatted patches and achieving merely 14.4% issue resolution rate. In contrast, the fine-tuned models exhibit remarkable capabilities in generating both syntactically and structurally correct patches while substantially improving resolution rates. Notably, the fine-tuned 72B model achieves 96.8% format compliance rate and 43.8% resolution rate - significantly outperforming its untuned counterpart's 37.0% and 14.4%, respectively. Impressively, the fine-tuned 7B and 32B models surpass even the base 72B model in both edit format correctness and resolution rates. These results confirm that the MCTS-REFINE-generated CoT dataset not only enhances the models' understanding of code syntax and structure, but also effectively improves their practical patch generation capabilities.

Conclusion: The fine-tuned models exhibit remarkable capabilities in generating correct patches. Notably, the fine-tuned 72B model achieves **96.8%** format compliance rate and **43.8%** resolution rate, **comparable to the performance of *Deepseek-v3***.

V. DISCUSSION

A. Threats to Validity

Data Leakage: "Data leakage" occurs when evaluation data overlaps with a LLM's training dataset, potentially causing overfitting and biased performance metrics. Although *SWE-bench* derives from GitHub—the same platform used to train the *Qwen2.5* foundation models—recent study [35] suggests minimal leakage risk in this benchmark. For rigorous evaluation, we systematically exclude all *SWE-bench* repositories when collecting CoT data from GitHub issue-pull requests.

Reproducibility: To ensure reproducibility of LLM-generated content, we used the fixed open-source *DeepSeek-V3-0324* model for CoT data generation in MCTS-REFINE and publicly released the dataset. For *SWE-bench* evaluation of our AGENTLESS fine-tuned model, we maintained consistent parameters (*temperature*=0) and used the official *SWE-bench* Docker environment to guarantee reproducible results.

B. Clarifying for Ablation Studies

We combine *rejection sampling* and *refinement* to ensure the quantity and quality of synthesized CoT data for repository-level issue resolution tasks. We did not perform ablation studies on these components because they are integral to our framework's core functionality:

Rejection Sampling Necessity: *Rejection sampling* acts as an essential filtering mechanism within the MCTS framework to maintain high-quality reasoning paths. Without it, *refinement* alone is insufficient to effectively steer MCTS toward viable reasoning trajectories, thereby fundamentally compromising the synthesis of CoT data.

Refinement Criticality: The *refinement* mechanism plays a vital role in detecting and correcting errors during reasoning. Removing this component would significantly degrade both the quality and quantity of the generated CoT data. This is clearly supported by our empirical results: from 20,000 Issue-PR pairs, the fully implemented MCTS-REFINE produced 18,340 high-quality CoT samples for File Localization,

whereas disabling step validation resulted in a 47% reduction in output (yielding only 9,700 samples).

The absence of either mechanism would result in system failure, rendering ablation studies unfeasible. Both components are essential to guarantee the quality and volume of CoT data, thereby directly determining the overall performance of the framework.

C. Computational Overhead

A critical factor in CoT data synthesis is the associated API cost. In this study, the *DeepSeek-v3* model was employed during the synthesis phase. For online access, *DeepSeek-v3* offers highly competitive pricing: \$0.14 per million input tokens and \$0.6 per million output tokens [4]. Generating a CoT sequence for each issue-PR pair requires an average of 70 interactions with the *DeepSeek-v3* API, totaling approximately 3 minutes in duration. Each interaction typically consumes 3.2k input tokens and produces 827 output tokens. As a result, the total API cost for synthesizing CoT data per issue-PR pair is approximately \$0.07.

With ongoing advancements in LLMs, these expenses are anticipated to decrease further. Additionally, local deployment of the *DeepSeek-v3* model can substantially lower operational costs, making it an efficient and scalable solution for large-scale CoT data synthesis.

VI. Related Works

Training-Data Synthesis for LLMs. To address the inefficiency of manual annotation, researchers have increasingly utilized large language models (LLMs) for scalable and high-quality data synthesis, capitalizing on their ability to efficiently generate large-scale datasets [36]–[38]. Recent work has focused on automating CoT data generation for issue-resolution tasks, aiming to enhance open-source models' reasoning via instruction tuning or reinforcement learning [23], [25]–[28]. SWE-Gym [27] synthesized OpenHands Agent trajectories by prompting *GPT-4o* and *Claude-3.5-Sonnet* on open-source repositories, filtering outputs via unit tests before fine-tuning *Qwen*. Similarly, *Lingma-SWE GPT* [23] generated trajectories with *GPT-4o* and filtered them by similarity, while SWE-Fixer [28] and SORFT [25] used single-turn Q&A prompting (*GPT-4o* and *Claude-3.5-Sonnet*, respectively) to produce CoT data for edit generation and subtasks, applying overlap-based filtering. However, these approaches rely entirely on closed-source LLMs and suffer from critical limitations: their CoT synthesis lacks validation of intermediate reasoning steps and employs permissive rejection sampling (see discussions in Section II). To overcome these limitations, we propose MCTS-Refine, a novel framework that replaces simplistic single-turn Q&A prompting with a systematic, stepwise generation process, incorporating ground truth-aligned rejection sampling to rigorously validate each reasoning step and ensure robust, reliable reasoning-path synthesis.

LLMs for Repository Issue Resolution. LLMs pretrained on massive code corpora have demonstrated exceptional capabilities in code-related tasks, from generation to repair [39], [40], with recent frameworks like *SWE-Bench* [11] extending these capabilities to repository-level problem solving through two dominant paradigms: **(1)** ***Agent-Based systems*** like SWE-Agent [24], where LLMs autonomously interact with development environments via specialized interfaces (e.g., Agent-Computer Interfaces) to perform editing, navigation, and testing; and **(2)** ***Pipeline-Based Approaches:*** such as RAG-SWE [11] and Agentless [30], which employ structured workflows combining retrieval, context aggregation, and targeted editing. Agent-based systems enable broader applications but demand advanced LLMs and multi-turn computation, while pipeline methods boost efficiency through focused, staged task execution. In this paper, we integrate our fine-tuned model into the Agentless 1.0 [30] framework, which is compatible with our CoT data, enabling end-to-end evaluation of MCTS-Refine's *Rejection Sampling* and *Refinement* methodology.

VII. Conclusion

We proposed MCTS-Refine, a novel framework that enhances open-source LLMs' issue resolution capabilities through high-quality CoT data generation. By combining MCTS with a reflection mechanism and rigorous subtask validation, our approach achieves state-of-the-art performance on *SWE-bench*. The released resources - including the MCoT dataset and fine-tuned models - provide valuable tools for advancing AI-powered software engineering.

Acknowledgment

This work was supported by the National Key Research and Development Program of China (Grant No. 2024YFF0908000), the Tencent Rhino-Bird Basic Research Program (Project No. JR2024TEG004), and the China Postdoctoral Science Foundation (Grant No. 2024M750375).

References

[1] Fan A, Gokkaya B, Harman M, et al. Large language models for software engineering: Survey and open problems[C]//2023 IEEE/ACM International Conference on Software Engineering: Future of Software Engineering (ICSE-FoSE). IEEE, 2023: 31-53

[2] Hou X, Zhao Y, Liu Y, et al. Large language models for software engineering: A systematic literature review[J]. ACM Transactions on Software Engineering and Methodology, 2024, 33(8): 1-79.

[3] Wang J, Huang Y, Chen C, et al. Software testing with large language models: Survey, landscape, and vision[J]. IEEE Transactions on Software Engineering, 2024.

[4] Liu A, Feng B, Xue B, et al. Deepseek-v3 technical report[J]. arXiv preprint arXiv:2412.19437, 2024.

[5] Guo D, Zhu Q, Yang D, et al. DeepSeek-Coder: When the Large Language Model Meets Programming–The Rise of Code Intelligence[J]. arXiv preprint arXiv:2401.14196, 2024.

[6] Roziere B, Gehring J, Gloeckle F, et al. Code llama: Open foundation models for code[J]. arXiv preprint arXiv:2308.12950, 2023.

[7] Cordeiro J, Noei S, Zou Y. An Empirical Study on the Code Refactoring Capability of Large Language Models[J]. arXiv preprint arXiv:2411.02320, 2024.

[8] OpenAI. 2024. Introducing GPT-4o. https://openai.com/index/hello-gpt-4o/

[9] Anthropic. 2024. Introducing Claude 3.5 Sonnet. https://www.anthropic.com/news/claude-3-5-sonnet

[10] Grattafiori A, Dubey A, Jauhri A, et al. The llama 3 herd of models[J]. arXiv preprint arXiv:2407.21783, 2024.

[11] Jimenez C E, Yang J, Wettig A, et al. Swe-bench: Can language models resolve real-world github issues?[J]. arXiv preprint arXiv:2310.06770, 2023.

[12] Yang G, Zhou Y, Chen X, et al. Chain-of-thought in neural code generation: From and for lightweight language models[J]. IEEE Transactions on Software Engineering, 2024.

[13] Li J, Li G, Li Y, et al. Structured chain-of-thought prompting for code generation[J]. ACM Transactions on Software Engineering and Methodology, 2025, 34(2): 1-23.
[14] Qu Y, Zhang T, Garg N, et al. Recursive introspection: Teaching language model agents how to self-improve[J]. Advances in Neural Information Processing Systems, 2024, 37: 55249-55285.
[15] Tang Z, Li Z, Xiao Z, et al. RealCritic: Towards Effectiveness-Driven Evaluation of Language Model Critiques[J]. arXiv preprint arXiv:2501.14492, 2025.
[16] Zhang S, Dong L, Li X, et al. Instruction tuning for large language models: A survey[J]. arXiv preprint arXiv:2308.10792, 2023.
[17] Guo D, Yang D, Zhang H, et al. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning[J]. arXiv preprint arXiv:2501.12948, 2025.
[18] Zhang D, Zhoubian S, Hu Z, et al. Rest-mcts*: Llm self-training via process reward guided tree search[J]. Advances in Neural Information Processing Systems, 2024, 37: 64735-64772.
[19] Chen Q, Qin L, Liu J, et al. Towards reasoning era: A survey of long chain-of-thought for reasoning large language models[J]. arXiv preprint arXiv:2503.09567, 2025.
[20] Guan X, Zhang L L, Liu Y, et al. rStar-Math: Small LLMs Can Master Math Reasoning with Self-Evolved Deep Thinking[J]. arXiv preprint arXiv:2501.04519, 2025.
[21] Chen Q, Qin L, Liu J, et al. Towards reasoning era: A survey of long chain-of-thought for reasoning large language models. arXiv preprint arXiv:2503.09567, 2025.
[22] Hui B, Yang J, Cui Z, et al. Qwen2. 5-coder technical report[J]. arXiv preprint arXiv:2409.12186, 2024.
[23] Ma Y, Cao R, Cao Y, et al. Lingma SWE-GPT: An open development-process-centric language model for automated software improvement. The ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2025), 2025.
[24] Yang J, Jimenez C E, Wettig A, et al. Swe-agent: Agent-computer interfaces enable automated software engineering[J]. Advances in Neural Information Processing Systems, 2024, 37: 50528-50652.
[25] Ma Z, Peng C, Gao P, et al. Sorft: Issue resolving with subtask-oriented reinforced fine-tuning. arXiv preprint arXiv:2502.20127, 2025.
[26] Wei Y, Duchenne O, Copet J, et al. Swe-rl: Advancing llm reasoning via reinforcement learning on open software evolution. arXiv preprint arXiv:2502.18449, 2025.
[27] Pan, J., Wang, X., Neubig, G., Jaitly, N., Ji, H., Suhr, A., & Zhang, Y. Training Software Engineering Agents and Verifiers with SWE-Gym. arXiv preprint arXiv:2412.21139.
[28] Xie, C., Li, B., Gao, C., Du, H., Lam, W., Zou, D., & Chen, K. SWE-Fixer: Training Open-Source LLMs for Effective and Efficient GitHub Issue Resolution. arXiv preprint arXiv:2501.05040.
[29] Li Q, Xia W, Du K, et al. RethinkMCTS: Refining Erroneous Thoughts in Monte Carlo Tree Search for Code Generation[J]. arXiv preprint arXiv:2409.09584, 2024.
[30] Xia, C. S., Deng, Y., Dunn, S., & Zhang, L. (2024). Agentless: Demystifying llm-based software engineering agents. arXiv preprint arXiv:2407.01489.
[31] Zhang D, Zhoubian S, Hu Z, et al. Rest-mcts: Llm self-training via process reward guided tree search[J]. Advances in Neural Information Processing Systems, 2024, 37: 64735-64772.
[32] Yang J, Jimenez C E, Wettig A, et al. Swe-agent: Agent-computer interfaces enable automated software engineering. Advances in Neural Information Processing Systems, 2024, 37: 50528-50652.
[33] fWang X, Li B, Song Y, et al. Openhands: An open platform for ai software developers as generalist agents.//The Thirteenth International Conference on Learning Representations. 2024.
[34] Ma Z, An S, Lin Z, et al. Repository Structure-Aware Training Makes SLMs Better Issue Resolver. arXiv preprint arXiv:2412.19031, 2024.
[35] Chang J, Zhou X, Wang L, et al. Bridging Bug Localization and Issue Fixing: A Hierarchical Localization Framework Leveraging Large Language Models[J]. arXiv preprint arXiv:2502.15292, 2025.
[36] Toshniwal S, Du W, Moshkov I, et al. Openmathinstruct-2: Accelerating ai for math with massive open-source instruction data[J]. arXiv preprint arXiv:2410.01560, 2024.
[37] Maini P, Seto S, Bai H, et al. Rephrasing the web: A recipe for compute and data-efficient language modeling[J]. arXiv preprint arXiv:2401.16380, 2024.
[38] Wang K, Zhu J, Ren M, et al. A survey on data synthesis and augmentation for large language models[J]. arXiv preprint arXiv:2410.12896, 2024.
[39] Jiang N, Liu K, Lutellier T, et al. Impact of code language models on automated program repair[C]//2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE). IEEE, 2023: 1430-1442.
[40] Schäfer M, Nadi S, Eghbali A, et al. An empirical evaluation of using large language models for automated unit test generation[J]. IEEE Transactions on Software Engineering, 2023, 50(1): 85-105.
[41] Zheng Y, Zhang R, Zhang J, et al. Llamafactory: Unified efficient fine-tuning of 100+ language models[J]. arXiv preprint arXiv:2403.13372, 2024.
[42] Hu E J, Shen Y, Wallis P, et al. Lora: Low-rank adaptation of large language models[J]. ICLR, 2022, 1(2): 3.
[43] Sun X, Ji Y, Ma B, et al. A comparative study between full-parameter and lora-based fine-tuning on chinese instruction data for instruction following large language model[J]. arXiv preprint arXiv:2304.08109, 2023.
[44] Antoniades A, Örwall A, Zhang K, et al. Swe-search: Enhancing software agents with monte carlo tree search and iterative refinement[J]. arXiv preprint arXiv:2410.20285, 2024.
[45] Lin J, Guo Y, Han Y, et al. SE-Agent: Self-Evolution Trajectory Optimization in Multi-Step Reasoning with LLM-Based Agents[J]. arXiv preprint arXiv:2508.02085, 2025.