DARS: Dynamic Action Re-Sampling to Enhance Coding Agent Performance by Adaptive Tree Traversal

Vaibhav Aggarwal[*]
MPI for Intelligent Systems
vaibhav_a@ee.iitr.ac.in

Ojasv Kamal[*]
MPI for Intelligent Systems
kamalojasv2000@gmail.com

Abhinav Japesh
Sprinklr
akjapesh@gmail.com

Zhijing Jin[†]
MPI-IS & University of Toronto
zjin@cs.toronto.edu

Bernhard Schölkopf[†]
MPI for Intelligent Systems
bs@tue.mpg.de

Abstract

Large Language Models (LLMs) have revolutionized various domains, including natural language processing, data analysis, and software development, by enabling automation. In software engineering, LLM-powered coding agents have garnered significant attention due to their potential to automate complex development tasks, assist in debugging, and enhance productivity. However, existing approaches often struggle with sub-optimal decision-making, requiring either extensive manual intervention or inefficient compute scaling strategies. To improve coding agent performance, we present Dynamic Action Re-Sampling (DARS), a novel inference time compute scaling approach for coding agents, that is faster and more effective at recovering from sub-optimal decisions compared to baselines. While traditional agents either follow linear trajectories or rely on random sampling for scaling compute, our approach DARS works by branching out a trajectory at certain key decision points by taking an alternative action given the history of the trajectory and execution feedback of the previous attempt from that point. We evaluate our approach on SWE-Bench Lite benchmark, demonstrating that this scaling strategy achieves a pass@k score of 55% with Claude 3.5 Sonnet V2. Our framework achieves a pass@1 rate of 47%, outperforming state-of-the-art (SOTA) open-source frameworks.[1]

1 Introduction

Software engineering (SWE) has become increasingly critical in modern technology development, with developers spending countless hours writing, reviewing, and maintaining code, creating an urgent need for automation to improve productivity (Wang et al., 2024d). Large language models (LLMs) have emerged as promising tools for automating various software engineering tasks, with breakthrough works like SWE-bench (Jimenez et al., 2023) establishing evaluation frameworks and datasets, leading to widespread adoption of tools such as Yang et al. (2024b); Wang et al. (2024c).

There are three primary approaches to developing SWE agents based on LLMs. The first follows a sequential ReAct (Yao et al., 2022) loop, where agents such as SWE-Agent (Yang et al., 2024b) and OpenDevin (Wang et al., 2024c) interact with development tools and incorporate execution feedback to refine their predictions. The second approach generates multiple candidate solutions using temperature-based sampling and then selects the best one through ranking (Arora et al., 2024) or majority voting (Xia et al., 2024). The third approach, exemplified by SWE-Search (Antoniades et al., 2024), leverages Monte Carlo Tree Search (MCTS) (Kocsis and Szepesvári, 2006) to systematically explore the solution space.

However, each method has limitations: (1) Sequential agents struggle to recover from suboptimal decisions due to context length constraints (Kuratov et al., 2024; Li et al., 2024). (2) Multi-solution approaches lack efficient mechanisms for knowledge sharing between independently generated solutions. (3) Tree search methods, such as SWE-Search (Antoniades et al., 2024), rely on scalar value functions and suffer from slow execution speeds, making them less effective for long-horizon planning.

To address these challenges, we propose Dynamic Action Re-Sampling (DARS), which enhances coding agents by dynamically re-sampling actions based on prior execution results. Instead of generating multiple independent trajectories, DARS selectively branches at key decision points, using a depth-first strategy to fully explore a trajectory before branching. This offers two advan-

[*] Co-first author. [†]Co-supervision.

[1]Our codes are at https://github.com/darsagent/DARS-Agent, datasets and models at https://huggingface.co/AGENTDARS, and a demo of our trajectory analysis tool at https://darsagent.github.io/DARS-Agent/

Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pages 19808–19855
July 27 - August 1, 2025 ©2025 Association for Computational Linguistics

tages: a) **Long Horizon Feedback**: Our experiments show improved pass@1 rates (see Table 4) by providing complete trajectory feedback before branching. b) **Efficiency**: Depth-first search reduces memory overhead by reusing the current environment state without simulating future states. Finally, we introduce a trajectory selection pipeline leveraging proprietary and preference-optimized models to identify the most promising solution (Kim et al., 2024).

Across the experiments, our DARS method achieves up to 47% pass@1 rate, which is open-source SOTA performance on the SWE-Bench Lite benchmark (Jimenez et al., 2023).

In conclusion, the main contributions of our approach are as follows:

1. We introduce DARS, an inference-time compute scaling method for coding agents that rapidly recovers from suboptimal decisions, achieving an open source SOTA pass@1 rate of 47% on the SWE-Bench Lite benchmark.
2. We propose a patch preference data generation and supervised fine-tuning pipeline to select the most promising solution among multiple attempts.
3. We release our complete codebase, a 500M-token execution feedback critique dataset, model checkpoints (7B, 14B, and 32B), and a trajectory analysis tool to support future research.

2 Related Work

LLM Agents for Software Engineering. Large Language Model (LLM) agents have been increasingly employed to automate software engineering tasks such as bug fixing and code generation. These agents integrate tools for code editing, search, navigation, and execution (Yang et al., 2024b). Enhancements in this domain include diff-based editing (Aider, 2024), execution with Jupyter and web search capabilities (Wang et al., 2024c), and optimized repository search (Aider, 2024; Zhang et al., 2024c; Ouyang et al., 2024; Orwall, 2024). Some approaches further modularize functionalities to improve efficiency (Xia et al., 2024; Arora et al., 2024).

Recent studies have explored generating multiple solutions to enhance accuracy. For instance, Brown et al. (2024) demonstrated that sampling 250 solutions can increase accuracy by 250%. However, methods like those proposed by Xia et al. (2024) and Arora et al. (2024) rely on inefficient random sampling. To address this, Antoniades et al. (2024) introduced an approach that improves efficiency through Monte Carlo Tree Search (MCTS) (Kocsis and Szepesvári, 2006; Coulom, 2007), balancing computational resources with scalar rewards and textual feedback. Despite these advancements, their reliance on retrospective feedback limits early guidance, and frequent environment resets can slow execution.

DARS improves efficiency by branching only at critical decisions and providing long-horizon feedback, reducing resets and accelerating execution.

We discuss about Inference Time Compute Scaling and LLM as Code Reviewers in Appendix A.1

3 DARS Method

The main motivation behind DARS is to enhance the agent's ability to recover and learn from suboptimal decisions by taking alternative actions while minimizing redundancy. However, errors also scale with scaling trajectories, therefore, we optimize our backbone SWE-Agent first by improving its editing capabilities and adding various actions to it. We then identify the most promising action types by optimizing the trade-off between increase in resolve rate and increase in cost due to branching the trajectory at that point. We define this process of branching the tree as expansion. We finally select the most promising trajectory from all the attempts. We go over the details of each of these steps in the following sections.

3.1 Improving the Base SWE Agent

3.1.1 Editing Capabilities

We build on the SWE-Agent (Yang et al., 2024b) which uses a ReAct loop to iteratively generate a thought and an action and receive feedback from the sandbox environment. By default, the agent uses a whole style of editing where it needs to generate the start and end line numbers of the edit followed by the content of edit. This type of edit often results in numerous syntax and semantic errors (see Section A.3), as the agent fails to account for both the targeted and adjacent code, leading to issues such as indentation errors. We enhance the editing process using Aider (Aider, 2024), a diff-based tool. In this approach, the agent generates both the content to be replaced and its replacement. Additionally, to facilitate content addition without replacement, Aider introduces two new actions: append and insert.

Append adds content at the end of a file, while

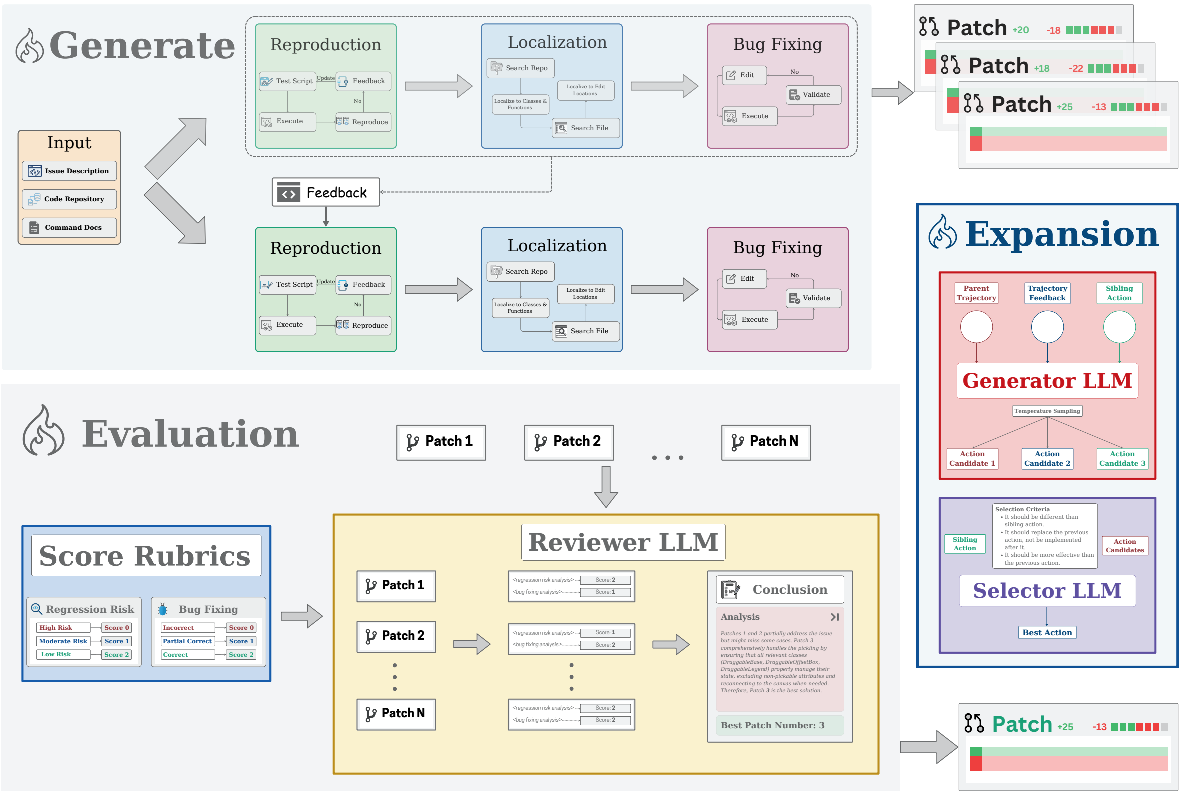


Figure 1: Overview of our DARS scaling method. DARS processes issue-related information and generates multiple patches using the **Expansion** mechanism. These patches are then evaluated by our **Reviewer LLM**, which assigns scores based on predefined **Score Rubrics**, ultimately selecting the best patch for output.

Insert requires a line number and the content to be inserted at that specific location. This approach compels the model to better consider the existing code. Moreover, we enhance the editing process by having the agent output both the `to_replace` and `replace_with` contents, each followed by a $ character to properly escape special characters (e.g., newlines, quotes) as described in Section A.14.1.

3.1.2 New Actions

In addition to editing, we introduce several new actions to enhance each of the three stages of our bug-fixing process. We add the following actions to the agent:

Execute Server. The sandbox environment limits the execution of iterative or long-running scripts. To address this, we add the *execute server* action with persistent memory. Instead of retrieving code output directly, the agent uses `get_logs` to access execution logs. This action is especially effective during the reproduction stage for efficiently replicating bugs (see Section A.14.1).

Execute IPython. This action enables the agent to run Python code within an IPython environment, streamlining bug reproduction by eliminating the need to create, write, and execute a separate file.

Search Repo. Search repo command uses a cached RepoGraph (Ouyang et al., 2024)—a hierarchical structure where nodes represent code definitions and edges represent dependencies between them. By utilizing sub-graph retrieval algorithms, RepoGraph extracts ego-graphs (Hu et al., 2024) centered around specific keywords. This action allows the agent to search for a specific keyword in the repo and get all the files with corresponding line numbers where the keyword is present, to aid in precise bug localization. **Undo Edit.** Often times, the agent makes a mistake in the edit and needs to undo it. However, due to inherent limitations of editing abilities of the agent, the agent sometimes outputs syntactically incorrect code which degrades its reasoning (Kuratov et al., 2024) capabilities and takes up its computational budget. This action allows the agent to directly undo the last edit efficiently (Anthropic, 2024b)

3.2 DARS Scaling

DARS begins by completing a trajectory in a depth-first manner while storing key decision nodes in a

Action	Coverage	Avg Iter
Search Dir	14.3	**38**
Insert	14.7	39
Search File	15.3	44
Open	14.3	49
Goto	15.3	47
Find File	16.7	58
Append	22.3	88
Edit	**31.3**	272

Table 1: Compute-coverage trade-off of expanding in various actions. Here Avg Iter pertains to average number of iterations across issues which indicates the amount of compute spent in that issue.

priority queue, sorted in ascending order by node depth. Once the current trajectory reaches a terminal state—either by a submit command (see A.14.1) or upon reaching a predefined maximum depth—these nodes are expanded. During expansion, we sample k alternative actions and select the best one. We define key decision points as those actions that significantly enhance the resolve rate at minimal cost. As shown in Table 1, expanding the trajectory at edit actions is particularly effective. This approach allows the agent to learn from previous mistakes and recover from suboptimal decisions, which is crucial for long-horizon tasks like programming. Finally, if no branch submits code before reaching the maximum depth, the code is auto-submitted. Issues that fail to execute the expected trajectory due to runtime errors or other anomalies within the SWE-Agent environment are re-run.

3.2.1 Branching Strategy

The main improvement in DARS lies in the avoidance of branching out trajectory at all actions, which costs exponential compute and its redundancy leads to a low accuracy for the trajectory selection pipeline. We use a causal analysis in Table 1 to identify four key actions with the largest causal impact on the model performance: edit, append, create, and submit and further perform qualitative analysis in Appendix A.12 to understand the reasons behind the results.

Create. Reproduction scripts are essential for debugging. Insufficient details can hinder their effectiveness and lead to incorrect fixes. By first localizing and analyzing the relevant code, the model improves bug resolution.

A key issue is that models often fail to refine reproduction scripts during bug fixing. While some cases improve, others show overconfidence, with flawed scripts being repeated. Prioritizing localization is crucial for accurate reproduction.

The `create` action differs from `append` in reproduction scripts. Though both evaluate and fix bugs, `append` actions generally produce better scripts. Models are often biased by previous actions, making only minor script changes instead of exploring new paths. Early sampling of solutions during `create` allows better exploration (see ref 10).

Should issue localization always precede reproduction? Not always. Early localization can bypass reproduction, leading to weaker solutions or misinterpretations of the bug, as shown in Figure 11. Reproducing the bug first enables a clearer understanding and more accurate fixes.

Append. The append action improves reproduction scripts by refining previous attempts, ensuring tests sufficiently verify code edits (see 13)

Runtime errors arise when the model lacks codebase or environment knowledge, hindering issue reproduction and exhausting its reasoning context. Expansion in append actions mitigates this by accelerating the reproduction phase, reducing turns needed for localization. This allows more iterations for editing and testing, improving bug resolution. The benefit occurs in two ways: direct bug identification during expansion or improved reproduction scripts enabling better localization. (see 14)

Edit. The agent sometimes generates semantically incorrect code, leading to an edit-Python loop. As context length grows, its reasoning weakens, trapping it in an unproductive cycle without a clear exit (see 15)

The agent frequently produces code with basic syntax errors, such as mismatched parentheses or incorrect indentation, leading to a cycle of repetitive fixes. Due to reasoning flaws, it often gets stuck applying the same ineffective edits—such as repeatedly adding a closing bracket—even when the fix has already failed (see 16).

Submit. While the model can fix bugs, it sometimes introduces regressions. To prevent this, it should verify changes by running tests and refining edits based on results. Expansion in this action prompts the model to reassess its fixes and correct issues before submission. For example, in the following case, the model fixes a bug but introduces a regression. By reevaluating its changes, it catches and resolves the issue (see 12).

We further cut down the redundancy by considering second-degree expansions. The above actions namely create, append, edit, and submit usually occur in the same order. The higher the action in the tree, the higher the impact of expanding the tree at that action. Therefore, if a branch is expanded at create, we only expand the tree at append, edit, and submit the next time. Similarly, a branch expanded at edit is only expanded at submit the next time. We follow this rule with an exception in the case of append, since empirically it has found that this has led to a high resolve rate for the extra cost incurred. Finally, for each branch, we put a cap on the number of expansions of each type to prevent the tree from growing exponentially.

3.2.2 Expansion Strategy

We use a depth-first strategy to explore the current trajectory before branching out, which has two main advantages: **Speed** and **Long-Horizon Feedback**. After reaching a terminal condition, we continue from the node with the lowest depth in the priority queue. In Figure 8, we find that lowest depth-first is the most effective strategy, as the flexibility to explore decreases with node depth.

3.3 Best Trajectory Selection

After the agent has generated multiple trajectories, we select the most promising trajectory from all the attempts in two stages namely trajectory pruning and trajectory selection. We begin by cleaning the patches submitted by each trajectory by removing any bug reproduction files from it. We then prune any redundant trajectories which lead to the same cleaned patch. In the second stage, we use off-the-shelf open and close source models as well as our custom supervised-fine-tuned models to choose the best trajectory based on custom rubrics namely reproduction, fix, and potential to introduce new bugs motivated by (Kim et al., 2024).

Patch Preparation. We begin with cleaning the patches by removing everything except the bug fix part. This includes removing the bug reproduction script, readme / documentation changes, pycache files etc. We then generate critiques for each patch based on the three rubrics namely reproduction, fix, and potential to introduce new bugs. To ground the predictions of the model, we use the execution output obtained from running the tests after applying the patch.

Patch Sampling. For a given issue, based on the distribution of number of patches generated by DARS for that issue, we sample all combinations of patches from 2 to 6 patches. We further do a fine-grained sampling of negative patches by dividing all the negative patches in buckets based on the combinations of tests that fail after applying the patch, to get a balanced dataset. For positive patches, we sample from the set of all the positive patches if there are any. For cases where there are no positive patches, we just use the ground truth patch.

4 Experimental Setup

Framework	Base Model	Pass@1
SWE-Agent	GPT-4o	18.3
Moatless Tools	GPT-4o	24.7
Aider	GPT-4o & Claude 3 Opus	26.3
MASAI	GPT-4o	27.3
Agentless-1.5	GPT-4o	27.3
SWE-Search	GPT-4o	31.0
DARS (Ours)	GPT-4o + Deepseek R1	**37.0**
SWE-Agent	Claude 3.5 Sonnet V2	23.0
Moatless Tools	Claude 3.5 Sonnet V2	38.3
Moatless Tools	Claude 3.5 V2	38.3
Agentless-1.5	Claude-3.5 V2	40.7
Kodu-v1	Claude-3.5 Sonnet V2	44.7
OpenHands	Claude 3.5 Sonnet V2	41.7
DARS (Ours)	Claude 3.5 Sonnet + Deepseek R1	**47.0**

Table 2: Comparative analysis of various software engineering agents' performance on SWE-Bench Lite dataset. We present results only for the language models that were used by the respective authors, as evaluating every possible combination of models and frameworks is highly resource-intensive.

4.1 DARS Scaling

Dataset. We use the SWE-Bench Lite benchmark, a widely used subset of the SWE-Bench dataset (Jimenez et al., 2023). It comprises 300 GitHub issues from 12 real-world software projects, each containing an issue report and the corresponding codebase.

	7B		14B		32B		
	Vanilla	FT	Vanilla	FT	Vanilla	FT	R1
GPT-4o	33.0	33.3	35.7	37.0	36.0	36.7	37.0
Gemini-1.5-pro	26.3	26.3	27.0	27.7	29.7	29.0	33.0
Gemini-2.0-flash	26.7	27.0	26.3	27.7	28.0	28.7	28.3
Claude 3.5 Sonnet	35.7	38.7	39.7	41.7	41.3	42.0	**47.0**

Table 3: Performance Comparison across 7B, 14B, and 32B parameter DeepSeek R1 Distill Qwen reviewer models

Evaluation Metrics. We evaluate model performance using multiple metrics. **Resolve Rate** (Pass@1) measures the fraction of instances fixed on the first attempt, while **Pass@k** represents the expected success rate within k attempts. To assess efficiency, we track the **Average Cost per Instance** (in dollars) and the **Cost Scaling Factor**, which compares scaled resource costs to the base agent. Lastly, we record the **Number of Attempts** required for a successful fix.

Method	Overall	Pass@1	Precision
5 look aheads	5	2	0.54
10 look aheads	8	5	0.68
Path Summary	9	5	0.54
Only Sibling Action	8	5	**0.72**
Complete	**10**	**9**	0.51

Table 4: Variation of performance with horizon of context during expansion.

Baselines. We test our approach against various SWE agents including SWE-Agent (Yang et al., 2024b), Moatless Tools (Orwall, 2024), and OpenHands (Wang et al., 2024c), MASAI (Arora et al., 2024), Large Language Monkeys (Brown et al., 2024), Agentless (Xia et al., 2024), and SWE-Search (Antoniades et al., 2024). In terms of LLMs, we test our approach with various models including GPT-4o (OpenAI, 2025), Claude 3.5 Sonnet V2 (Anthropic, 2024a), Gemini 2.0 Flash, and Gemini 1.5 Pro (Team et al., 2023).

Hyperparameters. The DARS algorithm relies on several key hyperparameters. *Num Expansions* is set to 2, defining the number of expansions per decision point, while *Expansion Temperature* (0.8) controls the sampling temperature for alternative actions. The algorithm runs for 300 iterations (*Num Iterations*), with a maximum branch depth of 50 (*Max Branch Depth*). Action limits are defined by *Expansion Limit Edit, Append, Submit,* and *Create*, each set to 1 which caps the number of times those actions can be expanded within a branch. *Num Expansion Sampling* (3) specifies the number of sampled actions per expansion, and *Num Lookahead* (50) determines how many steps from previous trajectory are considered during tree expansion.

4.2 Model Training

Dataset. We use Nebius's trajectory dataset (Badertdinov et al., 2024), comprising 80K trajectories from 3K unique issues across 1,077 open-source software repositories. These issues are entirely disjoint from SWE-Bench Lite. After cleaning and filtering redundant patches, we obtain 42K unique patches (7.3K positive, 34.7K negative), with 837 unique issues correctly solved. Using GPT-4o (OpenAI, 2025), we generate critiques for all patches, leading to 150K training examples containing approximately 500M tokens.

Model Setup. We fine-tune open-weight LLMs on the generated dataset. The model architecture follows Qwen 2.0 (Yang et al., 2024a), a Mixture-of-Experts (MoE) model utilizing Rotary Positional Embeddings (Su et al., 2024), SwiGLU (Dauphin et al., 2017) activation, QKV bias (Su, 2023) for attention, and RMSNorm (Jiang et al., 2024) normalization.

Training Setup. We use Deepseek's Distilled Qwen-2.5 (et al., 2025) checkpoint as the base model for 7B, 14B, and 32B parameter variants, fine-tuning them with 8 H100 GPUs. Training is distributed via DeepSpeed (Aminabadi et al., 2022), with LoRA (Hu et al., 2021) adapters for memory efficiency and FlashAttention 2 (Dao, 2023) for acceleration.

We conduct a learning rate sweep over 1e-6, 5e-6, and 1e-5, selecting optimal values for the 32B, 14B, and 7B models, respectively. The batch size is set to 48 for 7B/14B models and 32 for 32B. Training runs for 1 epoch over the dataset with a max sequence length of 14K tokens, a warmup of 100 steps, and weight decay of 0.0.

LoRA Configuration: We use rank $r = 8$, alpha = 32, and a dropout rate of 0.1.

Optimization: We apply AdamW (Loshchilov and Hutter, 2019) with a cosine learning rate scheduler, BF16 mixed precision, and ZeRO stage 3.

Reviewer Model Inference. We infer all pre-trained and fine-tuned reviewer models using vLLM (Kwon et al., 2023). A temperature sweep over 0, 0.5, and 0.6 is performed, as recommended by Deepseek authors. We set a top-p of 0.95.

5 Experiments

In this section, we first demonstrate the performance of our DARS model against various baselines, and then explore two key research questions (RQs) to analyze various aspects of its optimality.

5.1 Overall Performance

In this section, we compare the performance of our approach against various baselines and models. We summarize the results in Table 2. We find that our approach achieves a pass@1 rate of 47.0% with Claude 3.5 V2 Sonnet and Deepseek R1 as Reviewer which is the open-source SOTA performance on the SWE-Bench Lite benchmark at the time of this submission.[2] We further compare various vanilla and fine-tuned models reviewer in

[2] Deepseek R1 family of models often fail to generate the solution in the desired format, therefore, we use GPT-4o to parse the outputs in such cases.

Framework	Model	Cost Scaling Factor	# Attempts	Single Rollout	Coverage	Δ	Precision
Agentless	GPT-4o	–	40	–	42	–	–
MASAI	GPT-4o	–	**5**	23	35	34.28	–
Large Language Monkeys	DeepSeek-Coder	250	250	15.9	**56**	**71**	14
SWE-Search	GPT-4o	14.00	5.00	25.70	34.00	24.41	20.00
DARS (Ours)	GPT-4o	**7.60**	**5**	21.67	43.34	50.00	**75.00**

Table 5: Compute Scaling Efficiency comparison across various frameworks and metrics. Here Single Rollout represents the performance of the agent when a single trajectory is generated.

	SWE-Agent		Improved SWE-Agent		DARS	
Model	Resolve Rate (%)	Cost ($)	Resolve Rate (%)	Cost ($)	Score	Cost ($)
Gemini 1.5 pro	14.33	0.56	18.67	0.58	33.00	9.85
Gemini 2.0 flash	16.33	**0.05**	15.67	**0.06**	28.33	**0.70**
GPT-4o	**18.33**	0.89	21.67	0.80	37.0	7.92
Deepseek V3	-	-	30.67	**0.06**	-	-
Claude 3.5 Sonnet V2	-	-	**32.67**	1.61	**47**	12.24

Table 6: Comparison of effectiveness and efficiency of SWE-Agent, Improved SWE-Agent, and DARS

Table 3. We see an average increase of 2.6% across fine-tuned models with maximum increase of 4.15 % in case of the 14B model. However, 40% of examples have perfect precision (all the patches are correct), which diminishes the gain in performance due to fine-tuning. We compare the accuracy of all the reviewers for trajectories generated by various models after removing such examples in Table 7

5.2 RQ1: How efficient is the compute scaling of DARS?

The goal of this research question is to evaluate the efficiency of DARS in terms of compute scaling and its impact on solution quality. Specifically, we report the cost-vs-reward trade-off by analyzing key efficiency metrics such as cost scaling factor, accuracy per attempt, number of attempts, coverage, and precision. Our findings indicate that DARS achieves the most optimal cost scaling while maintaining high coverage and precision, outperforming baselines in redundancy reduction. Notably, while Large Language Monkeys achieve the highest coverage, this comes at an impractical compute cost, making DARS the more feasible approach.

Methodology. We compare the performance vs. cost trade-off of DARS against various baselines that scale compute at inference time. The evaluation is conducted through five key efficiency metrics: (a) cost scaling factor, (b) accuracy per attempt, (c) number of attempts, (d) coverage of the solution set, and (e) precision of the solution set.

Results. Table 5 summarizes our findings. While DARS ranks second to Large Language Monkeys in terms of coverage improvement per attempt, the latter achieves this by scaling compute by a factor of 250, which is infeasible in real-world scenarios. Additionally, DARS exhibits significantly higher precision, enhancing the effectiveness of the trajectory selection pipeline. We also observe that hindsight feedback is less effective, as completely random sampling methods like MASAI outperform search-based approaches like SWE-Search in coverage improvement.

5.3 RQ2: How important is long-horizon planning?

The goal of this research question is to assess the impact of long-horizon planning on the performance of DARS in coding tasks. Specifically, we report how varying the lookahead value affects the agent's ability to generate effective patches. Our findings show that increasing the lookahead value improves solution coverage, with the complete trajectory approach achieving the highest success rate. However, certain lookahead strategies, such as sibling action expansion, exhibit high precision while suffering from limited adaptability.

Methodology. We investigate the importance of long-horizon planning by varying the lookahead value, which determines how many steps from the previous trajectory are considered during tree expansion. We evaluate five configurations: (a) 0-lookahead (random sampling), (b) 5-lookahead, (c) 10-lookahead, (d) complete trajectory, and (e) summarized trajectory. Additionally, we test the sibling

Reviewers	GPT-4o	Gemini 1.5 Pro	Gemini 2.0 Flash	Claude 3.5 Sonnet
Closed Source				
GPT-4o	62.12	48.19	50.00	51.02
Open Source				
R1	**71.64**	**78.31**	64.06	**74.49**
R1-Distill-7B	53.73	54.22	56.25	41.84
R1-Distill-14B	65.67	56.63	54.69	54.08
R1-Distill-32B	67.16	66.27	62.50	59.18
Fine-tuned				
R1-Distill-7B	55.22	54.22	57.81	51.02
R1-Distill-14B	71.64	59.04	60.94	60.20
R1-Distill-32B	70.15	63.86	**65.62**	61.22

Table 7: This table presents the classification accuracy of various reviewer models for trajectories generated by different models. To depict the true potential of reviewer models, we remove the cases, where all the patches for generated for a given issue resolve the issue.

action expansion, where only sibling actions are provided without any lookahead. The experiment is conducted on 20 randomly selected issues.

Results. Table 4 summarizes our findings. We observe a strong correlation between lookahead depth and solution coverage. The complete trajectory approach achieves the highest success rate, resolving 10 out of 20 issues (50%), while the summarized variant reaches a 45% success rate (9/20). This highlights the importance of maintaining full trajectory context for effective problem-solving.

Although the sibling action approach yields high precision, this result is skewed by a small subset of cases where it performed exceptionally well. In contrast, the complete trajectory method, despite lower precision, demonstrates superior Pass@1 accuracy—aligning with DARS's objective of generating diverse and effective patches.

The single lookahead approach resolves two fewer issues than the complete trajectory method, primarily due to trajectory depth limitations. This issue arises in cases where the agent falls into bug-fixing and reproduction loops (edit-python loops), repeatedly encountering the same obstacles without historical context.

The 5-lookahead configuration performs the worst, as the restricted context provides only phase-specific errors (e.g., reproduction phase errors in append expansions, bug-fixing errors in edit expansions) without access to prior trajectory outcomes. This lack of context hinders the model's reasoning and decision-making capabilities.

Model	Cov	*Original*		*Filtered*	
		#Att	Prec	#Att	Prec
GPT-4o	43.33	8.00	0.70	4.00	0.71
Gemini 1.5 Pro	39.00	8.21	0.62	6.23	0.61
Gemini 2.0 Flash	36.00	**6.34**	0.64	3.77	0.61
Claude 3.5 Sonnet	**55**	10.07	**0.71**	**6.62**	**0.72**

Table 8: This table presents the initial coverage, Number of Attempts (#Att), and Precision (Prec) before (Original) and after patch filtering (Filtered) stage.

6 Ablation Studies

6.1 Improved SWE-Agent

We evaluate our enhanced SWE-Agent, featuring advanced editing capabilities and new actions, against the base model on the SWE-Bench Lite benchmark. As shown in Table 6, the improved SWE-Agent achieves an average 14.7% higher resolve rate across all models while maintaining a similar cost per instance.

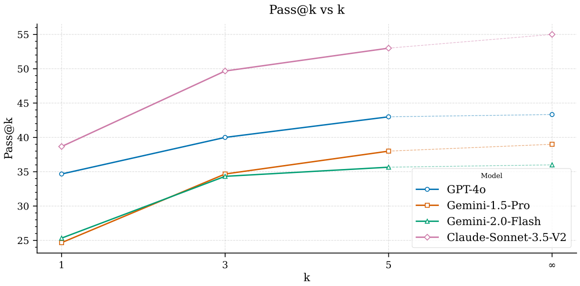


Figure 2: This figure presents coverage variation vs k. Here ∞ corresponds to submission of all the patches generated for an issue.

6.2 DARS Stage Analysis

This section presents deep insights into individual stages of DARS and their performance. There are two key stages in DARS: multi trajectory generation, best trajectory selection which further has two stages namely trajectory pruning and trajectory selection for various models and summarize the results in Table 8. We first analyze the recall and precision for multi trajectory generation to understand the effect of compute scaling on performance and redundancy for each model. We then analyze the capabilities of various models in final trajectory selection tested on the trajectories generated by Claude 3.5 Sonnet in Table 7.

6.3 Pass@k vs k

In previous sections, we analyzed pass@k where k is a variable. Here, we study variation of pass@k for different fixed values of k. We prompt the reviewer model (GPT-4o in this case) for the top k patches and compute the pass@k for $k = 1, 3, 5$; see Figure 2.. We find that the pass@k increases with k until $k = 5$, where it saturates.

7 Conclusion

We introduced DARS, a novel method that resamples actions at key decision points to recover from suboptimal choices more effectively than linear or random sampling. On the SWE-Bench Lite benchmark, DARS achieves a state-of-the-art pass@1 rate of 47% with Claude 3.5 Sonnet V2. We release our code, datasets, and models to support further research.

Limitations

We currently allocate compute using a static method with fixed depth and no early stopping, which limits our efficiency. A reward model (similar to MCTS) could evaluate and prioritize promising paths, enabling smarter exploration and early stopping decisions. While BFS might seem intuitive, its inefficiency with limited lookaheads and path history makes it impractical. We propose to implement absolute path scoring to guide exploration depth and stopping decisions, while maintaining an upper depth limit.

Ethical Considerations

The use of Large Language Models (LLMs) in software engineering carries security and ethical risks. To mitigate these, DARS executes all LM-generated code in isolated, ephemeral environments to prevent unintended system modifications. We employ a structured verification pipeline to reduce biased or unsafe outputs and ensure adherence to best coding practices. While AI-driven automation can be misused, we release our work under responsible AI guidelines and encourage safeguards against malicious applications. Our code, datasets, and models are open-source to promote transparency and responsible AI research.

Acknowledgment

We thank Gopal Dev and Apoorva Vashisth for reviewing parts of the paper. We also thank Vincent Berenz and Jojumon Kavalan for setting up our GPU access at Max Planck Institute.

This material is based in part upon work supported by the German Federal Ministry of Education and Research (BMBF): Tübingen AI Center, FKZ: 01IS18039B; by the Machine Learning Cluster of Excellence, EXC number 2064/1 – Project number 390727645.

References

Aider. 2024. Swe-bench lite. https://aider.chat/2024/05/22/swe-bench-lite.html. Accessed: [date of access].

Reza Yazdani Aminabadi, Samyam Rajbhandari, Minjia Zhang, Ammar Ahmad Awan, Cheng Li, Du Li, Elton Zheng, Jeff Rasley, Shaden Smith, Olatunji Ruwase, and Yuxiong He. 2022. Deepspeed inference: Enabling efficient inference of transformer models at unprecedented scale. *Preprint*, arXiv:2207.00032.

Zachary Ankner, Mansheej Paul, Brandon Cui, Jonathan D Chang, and Prithviraj Ammanabrolu. 2024. Critique-out-loud reward models. *arXiv preprint arXiv:2408.11791*.

Anthropic. 2024a. Announcing claude 3.5 sonnet. Accessed: 2025-01-30.

Anthropic. 2024b. Anthropic quick-starts. https://github.com/anthropics/anthropic-quickstarts.

Antonis Antoniades, Albert Örwall, Kexun Zhang, Yuxi Xie, Anirudh Goyal, and William Wang. 2024. Swe-search: Enhancing software agents with monte carlo tree search and iterative refinement. *arXiv preprint arXiv:2410.20285*.

Daman Arora, Atharv Sonwane, Nalin Wadhwa, Abhav Mehrotra, Saiteja Utpala, Ramakrishna Bairi, Aditya Kanade, and Nagarajan Natarajan. 2024. Masai: Modular architecture for software-engineering ai agents. *arXiv preprint arXiv:2406.11638*.

Ibragim Badertdinov, Maria Trofimova, Yuri Anapolskiy, Sergey Abramov, Karina Zainullina, Alexander Golubev, Sergey Polezhaev, Daria Litvintseva, Simon Karasik, Filipp Fisin, Sergey Skvortsov, Maxim Nekrashevich, Anton Shevtsov, and Boris Yangel. 2024. Scaling data collection for training software engineering agents. *Nebius blog*.

Maciej Besta, Nils Blach, Ales Kubicek, Robert Gerstenberger, Michal Podstawski, Lukas Gianinazzi, Joanna Gajda, Tomasz Lehmann, Hubert Niewiadomski, Piotr Nyczyk, et al. 2024. Graph of thoughts: Solving elaborate problems with large language models. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 38, pages 17682–17690.

Bradley Brown, Jordan Juravsky, Ryan Ehrlich, Ronald Clark, Quoc V Le, Christopher Ré, and Azalia Mirhoseini. 2024. Large language monkeys: Scaling inference compute with repeated sampling. *arXiv preprint arXiv:2407.21787*.

Rémi Coulom. 2007. Efficient selectivity and backup operators in monte-carlo tree search. In *Computers and Games*, pages 72–83, Berlin, Heidelberg. Springer Berlin Heidelberg.

Tri Dao. 2023. Flashattention-2: Faster attention with better parallelism and work partitioning. *Preprint*, arXiv:2307.08691.

Yann N Dauphin, Angela Fan, Michael Auli, and David Grangier. 2017. Language modeling with gated convolutional networks. In *International conference on machine learning*, pages 933–941. PMLR.

DeepSeek-AI et al. 2025. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *Preprint*, arXiv:2501.12948.

Jonas Gehring, Kunhao Zheng, Jade Copet, Vegard Mella, Taco Cohen, and Gabriel Synnaeve. 2024. Rlef: Grounding code llms in execution feedback with reinforcement learning. *arXiv preprint arXiv:2410.02089*.

Xinyu Guan, Li Lyna Zhang, Yifei Liu, Ning Shang, Youran Sun, Yi Zhu, Fan Yang, and Mao Yang. 2025. rstar-math: Small llms can master math reasoning with self-evolved deep thinking. *Preprint*, arXiv:2501.04519.

Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2021. Lora: Low-rank adaptation of large language models. *Preprint*, arXiv:2106.09685.

Yuntong Hu, Zhihan Lei, Zheng Zhang, Bo Pan, Chen Ling, and Liang Zhao. 2024. Grag: Graph retrieval-augmented generation. *Preprint*, arXiv:2405.16506.

Zixuan Jiang, Jiaqi Gu, Hanqing Zhu, and David Pan. 2024. Pre-rmsnorm and pre-crmsnorm transformers: equivalent and efficient pre-ln transformers. *Advances in Neural Information Processing Systems*, 36.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2023. Swe-bench: Can language models resolve real-world github issues? *arXiv preprint arXiv:2310.06770*.

Seungone Kim, Juyoung Suk, Shayne Longpre, Bill Yuchen Lin, Jamin Shin, Sean Welleck, Graham Neubig, Moontae Lee, Kyungjae Lee, and Minjoon Seo. 2024. Prometheus 2: An open source language model specialized in evaluating other language models. *Preprint*, arXiv:2405.01535.

Levente Kocsis and Csaba Szepesvári. 2006. Bandit based monte-carlo planning. In *Machine Learning: ECML 2006*, pages 282–293, Berlin, Heidelberg. Springer Berlin Heidelberg.

Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. 2022. Large language models are zero-shot reasoners. *Advances in neural information processing systems*, 35:22199–22213.

Yuri Kuratov, Aydar Bulatov, Petr Anokhin, Ivan Rodkin, Dmitry Sorokin, Artyom Sorokin, and Mikhail Burtsev. 2024. Babilong: Testing the limits of llms with long context reasoning-in-a-haystack. *arXiv preprint arXiv:2406.10149*.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pages 611–626.

Tianle Li, Ge Zhang, Quy Duc Do, Xiang Yue, and Wenhu Chen. 2024. Long-context llms struggle with long in-context learning. *arXiv preprint arXiv:2404.02060*.

Ilya Loshchilov and Frank Hutter. 2019. Decoupled weight decay regularization. *Preprint*, arXiv:1711.05101.

Liangchen Luo, Yinxiao Liu, Rosanne Liu, Samrat Phatale, Harsh Lara, Yunxuan Li, Lei Shu, Yun Zhu, Lei Meng, Jiao Sun, et al. 2024. Improve mathematical reasoning in language models by automated process supervision. *arXiv preprint arXiv:2406.06592*.

Arindam Mitra, Luciano Del Corro, Guoqing Zheng, Shweti Mahajan, Dany Rouhana, Andres Codas, Yadong Lu, Wei-ge Chen, Olga Vrousgos, Corby Rosset, et al. 2024. Agentinstruct: Toward generative teaching with agentic flows. *arXiv preprint arXiv:2407.03502*.

Maxwell Nye, Anders Johan Andreassen, Guy Gur-Ari, Henryk Michalewski, Jacob Austin, David Bieber, David Dohan, Aitor Lewkowycz, Maarten Bosma, David Luan, Charles Sutton, and Augustus Odena. 2021. Show your work: Scratchpads for intermediate computation with language models. *Preprint*, arXiv:2112.00114.

OpenAI. 2025. Hello gpt-4o. Accessed: 2025-01-30.

Adam Orwall. 2024. moatless-tools. `https://github.com/aorwall/moatless-tools`.

Siru Ouyang, Wenhao Yu, Kaixin Ma, Zilin Xiao, Zhihan Zhang, Mengzhao Jia, Jiawei Han, Hongming Zhang, and Dong Yu. 2024. Repograph: Enhancing ai software engineering with repository-level code graph. *Preprint*, arXiv:2410.14684.

Zhenting Qi, Mingyuan Ma, Jiahang Xu, Li Lyna Zhang, Fan Yang, and Mao Yang. 2024. Mutual reasoning makes smaller llms stronger problem-solvers. *arXiv preprint arXiv:2408.06195*.

Rafael Rafailov, Archit Sharma, Eric Mitchell, Christopher D Manning, Stefano Ermon, and Chelsea Finn. 2024. Direct preference optimization: Your language model is secretly a reward model. *Advances in Neural Information Processing Systems*, 36.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. 2017. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*.

David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. 2016. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484–489.

David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. 2017. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *arXiv preprint arXiv:1712.01815*.

Guijin Son, Hyunwoo Ko, Hoyoung Lee, Yewon Kim, and Seunghyeok Hong. 2024. Llm-as-a-judge & reward model: What they can and cannot do. *arXiv preprint arXiv:2409.11239*.

Jianlin Su. 2023. Rope + bias = better length extrapolation.

Jianlin Su, Murtadha Ahmed, Yu Lu, Shengfeng Pan, Wen Bo, and Yunfeng Liu. 2024. Roformer: Enhanced transformer with rotary position embedding. *Neurocomputing*, 568:127063.

Gemini Team, Rohan Anil, Sebastian Borgeaud, Jean-Baptiste Alayrac, Jiahui Yu, Radu Soricut, Johan Schalkwyk, Andrew M Dai, Anja Hauth, Katie Millican, et al. 2023. Gemini: a family of highly capable multimodal models. *arXiv preprint arXiv:2312.11805*.

Ye Tian, Baolin Peng, Linfeng Song, Lifeng Jin, Dian Yu, Haitao Mi, and Dong Yu. 2024. Toward self-improvement of llms via imagination, searching, and criticizing. *arXiv preprint arXiv:2404.12253*.

Polina Tsvilodub, Fausto Carcassi, and Michael Franke. 2024. Towards neuro-symbolic models of language cognition: Llms as proposers and evaluators.

Nan Wang, Yafei Liu, Chen Chen, and Haonan Lu. 2024a. Genx: Mastering code and test generation with execution feedback. *arXiv preprint arXiv:2412.13464*.

Peifeng Wang, Austin Xu, Yilun Zhou, Caiming Xiong, and Shafiq Joty. 2024b. Direct judgement preference optimization. *arXiv preprint arXiv:2409.14664*.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, et al. 2024c. Opendevin: An open platform for ai software developers as generalist agents. *arXiv preprint arXiv:2407.16741*.

Yanlin Wang, Wanjun Zhong, Yanxian Huang, Ensheng Shi, Min Yang, Jiachi Chen, Hui Li, Yuchi Ma, Qianxiang Wang, and Zibin Zheng. 2024d. Agents in software engineering: Survey, landscape, and vision. *Preprint*, arXiv:2409.09030.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. 2023. Chain-of-thought prompting elicits reasoning in large language models. *Preprint*, arXiv:2201.11903.

Tianhao Wu, Janice Lan, Weizhe Yuan, Jiantao Jiao, Jason Weston, and Sainbayar Sukhbaatar. 2024. Thinking llms: General instruction following with thought generation. *arXiv preprint arXiv:2410.10630*.

Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. 2024. Agentless: Demystifying llm-based software engineering agents. *arXiv preprint arXiv:2407.01489*.

An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, et al. 2024a. Qwen2. 5 technical report. *arXiv preprint arXiv:2412.15115*.

John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. 2024b. Swe-agent: Agent-computer interfaces enable automated software engineering. *arXiv preprint arXiv:2405.15793*.

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Tom Griffiths, Yuan Cao, and Karthik Narasimhan. 2024. Tree of thoughts: Deliberate problem solving with large language models. *Advances in Neural Information Processing Systems*, 36.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2022. React: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629*.

Di Zhang, Jiatong Li, Xiaoshui Huang, Dongzhan Zhou, Yuqiang Li, and Wanli Ouyang. 2024a. Accessing gpt-4 level mathematical olympiad solutions via monte carlo tree self-refine with llama-3 8b. *arXiv preprint arXiv:2406.07394*.

Lunjun Zhang, Arian Hosseini, Hritik Bansal, Mehran Kazemi, Aviral Kumar, and Rishabh Agarwal. 2024b. Generative verifiers: Reward modeling as next-token prediction. *arXiv preprint arXiv:2408.15240*.

Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. 2024c. Autocoderover: Autonomous program improvement. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 1592–1604.

A Appendix

A.1 More Related Works

Inference Time Compute Scaling. Scaling compute at inference time has been shown to enhance LLM performance across various tasks. For instance, Silver et al. (2016, 2017) improve decision-making by searching game states before selecting a move. Similarly, LLM-focused approaches Wei et al. (2023); Tian et al. (2024); Nye et al. (2021); Kojima et al. (2022) enhance reasoning by sampling additional tokens. Graph-based methods Yao et al. (2024); Besta et al. (2024); Luo et al. (2024); Zhang et al. (2024a); Qi et al. (2024) further optimize planning and exploration of the solution space, enabling more structured and efficient problem-solving.

LLMs as Code Reviewers. LLMs have demonstrated strong judgment capabilities Son et al. (2024); Tsvilodub et al. (2024); Ankner et al. (2024). Some approaches leverage LLMs directly to generate critiques and feedback Wang et al. (2024b); Kim et al. (2024). However, in structured domains like coding and math, feedback can be sampled from the environment, as seen in Wang et al. (2024a); Guan et al. (2025), where LLMs are augmented with external feedback to improve critique generation. This feedback is then used to train models to act as reviewers for selecting optimal solutions. Two primary training strategies exist: supervised fine-tuning Tsvilodub et al. (2024); Ankner et al. (2024); Wang et al. (2024a); Zhang et al. (2024b); Mitra et al. (2024) and reinforcement learning methods such as Direct Preference Optimization Rafailov et al. (2024) and Proximal Policy Optimization Schulman et al. (2017), used by Wu et al. (2024); Gehring et al. (2024).

A.2 Agent Performance Across Repositories

This section analyzes model performance across different repositories using the SWE-Bench Lite benchmark. Our goal is to identify biases in how models handle code from various sources.

We summarize our findings in Figure 3 and observe that models perform best on Seaborn and Requests, achieving 65-75% accuracy, while scientific computing libraries (e.g., sympy, xarray, SciKit-learn) and web frameworks (e.g., Flask, Django) show moderate performance (40-60%). In contrast, Astropy and Sphinx consistently rank lowest (30-40%), indicating that models struggle more with specialized scientific tools and documentation systems than with visualization and HTTP client libraries.

These findings highlight domain-specific variations in model effectiveness, guiding improvements in generalization across repositories.

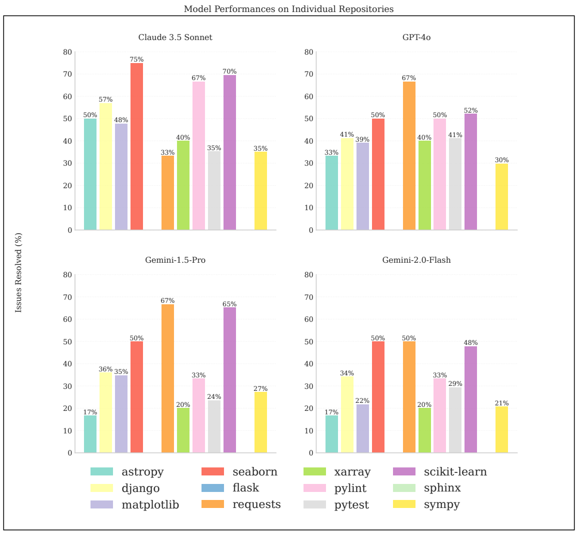


Figure 3: Repo-wise coverage of for each model

A.3 Improved Editing

We compare the performance of the agent when it uses the whole style of editing and the diff style of editing. We analyze the number of various types of syntactic errors committed by the agent while editing the code in both the styles of editing. We perform our analysis across two models namely Gemini 2.0 Flash and Gemini 1.5 Pro. Here *No Match* error pertains to the situation in diff editing where the text to be replaced does not match text in the file. *File Error* occurs when the model tries to make an edit when no file is opened. *Syntax Error* is thrown by the linter in cases like indentation error or erroneous variable referencing. Finally, *Content Error* occurs in case of diff editing, when the agent provides the new content in edit, append, or insert command as an empty string or provides the content to replace and new content as the same string.

We summarize the results in Table 9. In both the styles of editing, the major source of errors is syntax errors. We find that the diff style of editing leads to 1% less errors compared to the whole style of editing. However, this effect is much more pronounced in terms of semantics as diff style achieves 36% higher pass@1 rate.

	GPT-4o-mini		*Gemini 1.5 (Pro)*	
	Whole	Diff	Whole	Diff
Total Edits	11,931	8,507	3,605	4,113
Success Samples	4,905	3,090	1,995	2,590
Success Rate (%)	**41.1**	36.3	55.3	**62.9**
Error Types				
No Match	0	1,159	0	655
Content Error	0	1,640	0	143
Syntax Error	7,003	2,600	1,610	706
File Error	0	18	0	19
Pass@1 (%)	7	**8.67**	14.38	**21.44**

Table 9: Comparison of various types of syntactical errors committed in whole and diff setting.

Model	Correct Localization (%)
GPT-4o	74.37
Gemini 2.0 Flash	69.44
Gemini 1.5 Pro	64.82
Claude 3.5 Sonnet	**80.70**

Table 10: Various models' ability to correctly localize issues.

A.4 Localization Analysis

We analyze the ability of the agent to correctly localize the bug in the codebase. We find the overlap between the predicted location and the actual location based on the git patch of the proposed solution vs the actual solution patch and calculate the percentage of correct localizations. We summarize the results in Table 10. Average correct localization across all models is 72.3 %, which shows that the agent is usually able to correctly localize the bug in the codebase.

A.5 Variation of Performance with Max Depth and Number of Iterations

We analyze the optimality of two key search hyperparameters: max depth and number of iterations. We simulate the trajectories of our agent for different values of these hyperparameters and analyze the coverage of the agent for each value.

We summarize the results in Figure 4 and Figure 5 for variation with depth and iterations respectively. We find that both the curves show a decaying trend with increasing values of the hyperparameters and saturate for our values of 50 and 300 for max depth and number of iterations respectively which shows that our values are optimal for the agent.

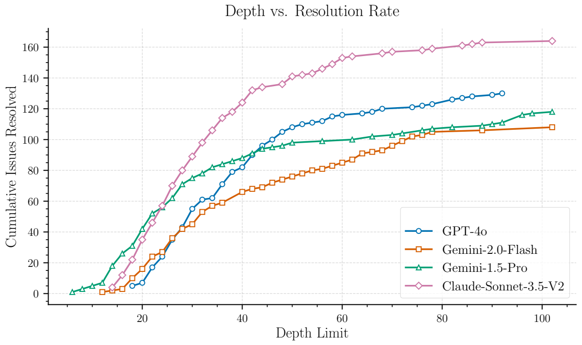


Figure 4: Variation of coverage with maximum branch depth

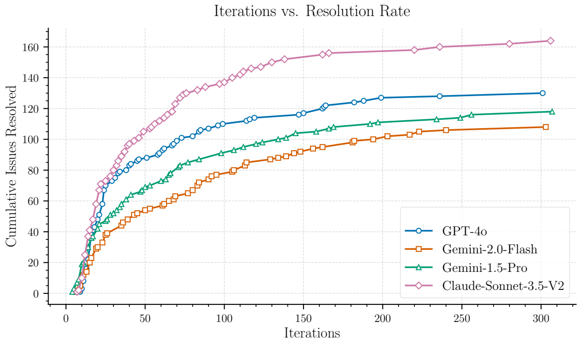


Figure 5: Variation of coverage vs number of iterations

A.6 Issues vs Models

Model-Specific Issue Resolution. Venn diagram of resolved issues by model. Each model can solve a handful of unique instances. We summarize the results in Figure 6.

A.7 Left-Right Branch Analysis

We study the effectiveness of our expansions by comparing the depths of branches before and after expansion, focusing on cases where both the left and right branches reach a conclusion, meaning they terminate when the agent returns a submit action. Additionally, we analyze how different expansion paths lead to conclusions by comparing two scenarios: (1) when the right-side expansion successfully reaches a conclusion while the left-side fails to do so, and (2) when the left-side expansion reaches a conclusion while the right-side does not. We summarize our findings in Table 12 and Table 11.

Submit expansions rarely achieve convergence, which is anticipated given their terminal position in the sequence. To accurately assess the efficacy of Submit expansions, an increased maximum depth threshold specifically for Submit operations would be necessary. Create expansions demonstrate sig-

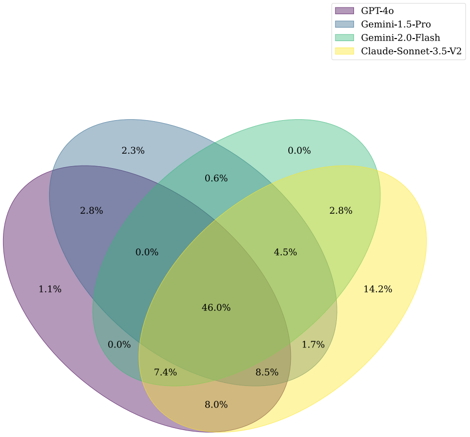


Figure 6: Venn diagram of resolved issues by model. Each model can solve a handful of unique instances.

nificant effectiveness in reaching conclusions, suggesting that initial localization strategies can facilitate convergence in specific scenarios. Expansions in edit and append operations successfully break edit-python iteration cycles, leading to more efficient conclusion paths. Analysis reveals a consistent pattern where dual-path conclusions and right-path iterations exhibit lower counts compared to left-path iterations, aligning with the hypothesis that expansions reduce errors. However, append operations demonstrate elevated average iterations because the model now creates a more comprehensive testing script that involves additional editing and validation steps, resulting in increased overall depth 17. Contrary to expectations, in edit, create, and append expansion operations, left-path expansions frequently achieve conclusions while right-path expansions do not. For edit and append: The model creates a more complicated reproduction script which leads to an error, which the model is not able to to resolve 19. In create, the agent finds it easy to locate the issue after reproduction. While, in the expanded branch, could not localize it 18. Another reason for this pattern is that the agent sometimes submits prematurely (often after reproduction). In the right path, it recognizes this mistake and corrects it.

A.8 Error Scaling Analysis

In this section, we present the scaling of various error types as we scale compute. We summarize the results in Table 13.

Action	After Expansion	Before Expansion
Edit	32	39
Create	41	27
Append	34	36
Submit	-	24

Table 11: Comparison of path reaches by action type. Here, before and after expansion pertain to cases where conclusion is only reached before and after expansion respectively

Action	Avg Dep Bef Exp	Avg Dep Aft Exp
Edit	21.2	20.8
Create	22.4	21.4
Append	22.1	22.6
Submit	16.9	22.1

Table 12: Comparison of action path depths before and after expansion

A.9 How to effectively expand the tree?

Understanding how to expand the search tree efficiently is crucial for balancing computational cost and solution quality in our coding agent. We investigate this research question to determine which actions should be prioritized for branching and in what order they should be expanded. Specifically, we report the trade-off between branching cost and resolution rate for different actions, as well as the impact of various branching strategies on search efficiency. Our findings show that branching at key actions like edit, create, and append leads to the highest resolve rates, and that a Lowest Depth First approach improves early-stage exploration but converges with other strategies over time. Finally we understand the compute-performance trade-off of number of expansions for each expansion type in a branch.

A.9.1 Action Selection

Methodology. We analyze the computational cost vs resolve rate trade-off of branching various actions in the trajectory. We first run the agent to expand at eight different actions namely search_dir, insert, search_file, open, goto, find_file, append, and edit. We then perform a causal analysis to determine the most promising actions by comparing decrease in resolve rate and number of iterations for each action. We then analyze the impact of branching at different actions on the performance of the agent.

Results. We find that branching at actions that are

Action Type	Error Types	SWE Agent	DARS
Search File	File Not Found	108	446
	Syntax Error	0	3
	Success	727	2635
Create	Directory Error	132	131
	File Exists	4	22
	Success	361	1276
Append	Content Error	0	3
	File Error	0	4
	Syntax Error	9	214
	Success	359	2518
Edit	No Match	382	884
	Content Error	367	724
	Syntax Error	422	1820
	File Error	150	311
	Success	1296	7206
Search Repo	Syntax Error	74	268
	Success	528	2581
Search Dir	Dir Not Found	4	107
	Syntax Error	3	29
	Success	114	814
Find File	Dir Not Found	8	17
	No Match Found	32	214
	Success	63	385
Insert	Syntax Error	30	659
	Success	114	1857
Execute IPython	Connection Error	8	56
	Response Error	44	2
	Success	40	396
Execute Server	Server Error	3	125
	Success	131	1542
Undo Edit	No Edit Made	5	27
	Success	0	38

Table 13: Error scaling comparison between SWE-Agent and DARS

typically used in the reproduction and fix stages like edit, create, append leads to the highest resolve rate. We summarize the results in Table 1.

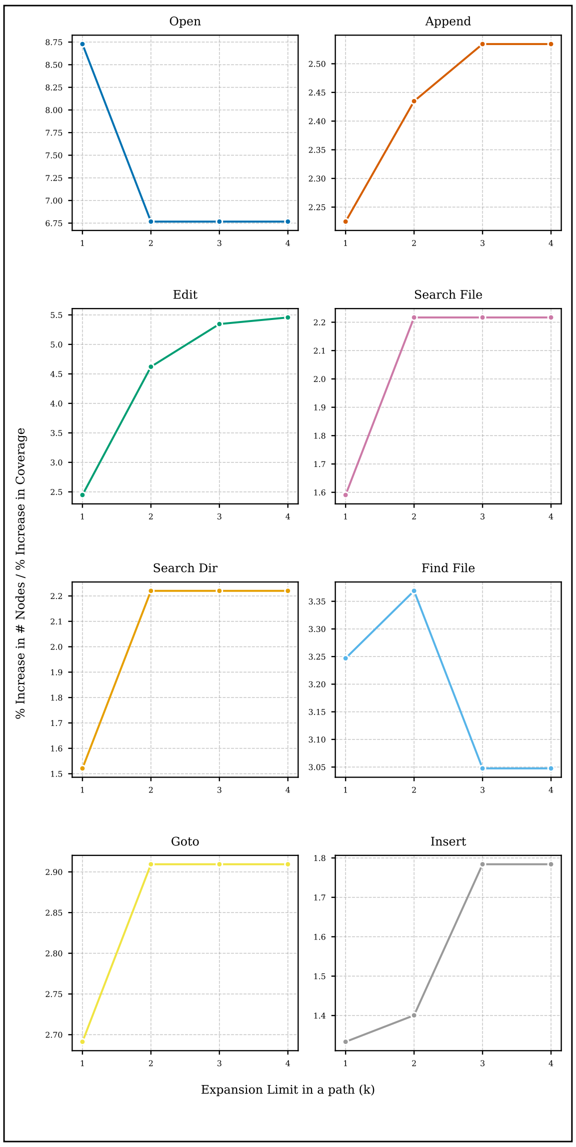


Figure 7: Percent increase in number of iterations per percent increase in resolve rate vs number of expansions in a branch.

A.9.2 Order of Action Selection

Methodology. We explore various strategies to expand the tree at different actions. We use the runs in the previous section and simulate strategies pertaining to the order of branching at different actions. We experiment with three different strategies namely First In First Out (FIFO), Last In First Out (LIFO), and Lowest Depth First. We plot the coverage vs number of iterations curve for each strategy to determine the most promising strategy.

Results. In Figure 8 we find that the Lowest Depth

First strategy early on as it promotes exploration at the lower depths of the tree. This allows the agent to explore more possibilities and make better decisions. But if the agent is run for long enough, all the strategies converge to the same point as all the possible states are explored eventually.

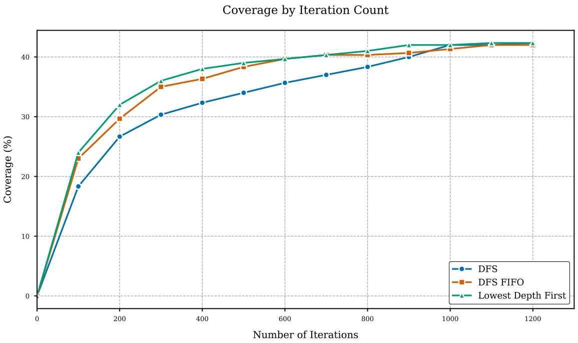


Figure 8: Variation of coverage of traversal strategies with iterations

A.9.3 Number of Expansion Types Per Branch

Methodology. We investigate the compute-performance trade-off associated with multiple expansions of the same type within a branch. For each expansion type, we allow k expansions within a branch, where k ranges from 1 to 4. Our analysis measures two key metrics: the relative increase in node count (representing computational cost) and the resolution rate. We plot the ratio of computational cost increase to performance improvement against k to determine the optimal number of expansions.

Results. As shown in Figure 7, for all action types except "open," the cost-performance ratio increases with k, indicating that computational costs grow more rapidly than performance gains. This suggests that performing more than one expansion of the same type per branch is computationally inefficient. For the "open" action type, the initial cost-performance ratio is prohibitively high, making even a single expansion impractical.

A.10 How to effectively select the best trajectory?

Optimizing our agent requires effective tree expansion, trajectory selection, and action evaluation. Due to computational constraints and LLM context limitations, we adopt a structured approach to improve efficiency.

We first analyze tree expansion, studying how different actions—edit, append, create, and submit—impact coverage and performance. Next, we tackle trajectory selection, implementing a two-stage pipeline: filtering to remove irrelevant information and ranking to identify the best trajectory. We compare pairwise and global ranking methods across different filtering configurations. Finally, we quantify the impact of expansion actions, isolating their contributions to coverage, accuracy, and efficiency.

A.10.1 Trajectory Content Filtering

A trajectory contains several components like command descriptions, various actions and their thoughts, observations to each action etc., not all of which are relevant to determine if a particular trajectory is would solve the bug. Therefore, we experiment with various components of the trajectory to determine the best combination.

Methodology. Based on the three key stages of the bug fixing process - Reproduction, Localization, and Fix, we identify certain key components of the trajectory that are relevant to each of these stages. These components include the reproduction script, the edited files, the output after running the reproduction script, and the final patch. We then experiment with various combinations of these components to determine the best combination.

A.10.2 Trajectory Ranking

Methodology. To find the best trajectory, we experiment with two different ranking methods: pairwise knockout ranking and global ranking. In pairwise knockout ranking, we compare each pair of trajectories and eliminate the one that is worse until we are left one. In global ranking, we rank all the trajectories based on our rubricks.

Results. We summarize the results of the trajectory ranking in Table 14 for each type of filtering pipeline and ranking strategy. We find that only the final patch is the most effective component to determine the best trajectory. This result is significant for two reasons. First, it makes our trajectory selection model applicable to any coding agents as a git patch is a common output format for all agents. Moreover, it also makes the trajectory selection pipeline more efficient as it only needs to consider the final patch to determine the best trajectory.

Combination	Pairwise Ranking	Global Ranking
RS + EF + RO + FP	30.67	33.00
RS + EF + FP	30.67	33.67
RS + RO + FP	30.67	33.67
RS + FP	30.67	34.00
FP	30.67	**34.67**

Table 14: Performance comparison of different scoring combinations using pairwise and global ranking methods. The combinations use the following components: RS (Final reproduction scripts), EF (Final edited files), RO (Final reproduction output), and FP (Final Patch).

A.11 Contribution of expansion in each Action

We quantify the contribution of each action in the trajectory to the final performance of the agent. We analyze the performance of the agent by simulating expansions for certain combinations of actions and studying its variation with the coverage. We summarize the results in Table 15. We can see that edit actions (edit and append) and reproduction actions (create) lead to the highest increase in performance. While expansion in submit command leads to minimal increase in performance, it does not lead to much redundancy either. While append leads to highest solve rate, it also leads to highest cost. Edit and create actions lead to a good balance between performance and cost.

Exp Actions	Cov	Avg Iter	Acc	# Att	Pre
Edit, Append, Submit, Create	**54.7**	194	**81**	11	**0.72**
Edit, Append, Create	54.3	177	46	6	0.72
Append, Create	51.0	146	35	5	0.70
Edit, Create	49.3	81	23	3	0.70
Edit, Append	47.6	96	23	3	0.70
Append	42.7	80	22	2	0.66
Create	41.3	51	12	1	0.66
Edit	39.3	44	12	1	0.66
No Expansion	31.0	27	11	1	0.57

Table 15: Performance of DARS for various combination of expansion actions (Exp Actions). We compare across several metrics such as coverage (Cov), average number of iterations across issues (Avg Iter), accuracy of reviewer model (Acc), average number of attempts (# Att), and precision (Pre)

In the previous analysis, we focus on the contributions of different combinations of expansion actions on the final coverage. However, in the analysis, the results depend on all the actions in the combination. To de-couple the effect of each expansion type, we perform another analysis where, for each expansion action, we contrast the number of cases where a) the branch before expansion does not resolve the issue, but the branch after expansion does, b) the branch before the expansion resolve the issue but the branch after the expansion does not resolve the issue, and c) both branches resolve the issue. We summarize our results in Figure 9. We can still see that majority of expansions are lead to solutions on both branches which shows that our approach still has a considerable amount of redundancy. We see highest efficiency for edit and append expansions and lowest for submit expansions.

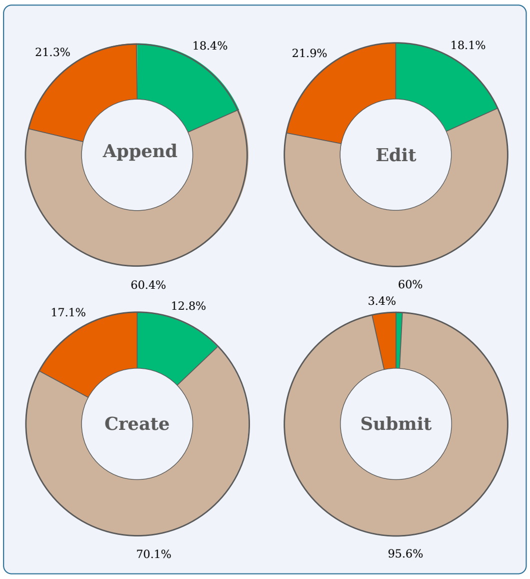
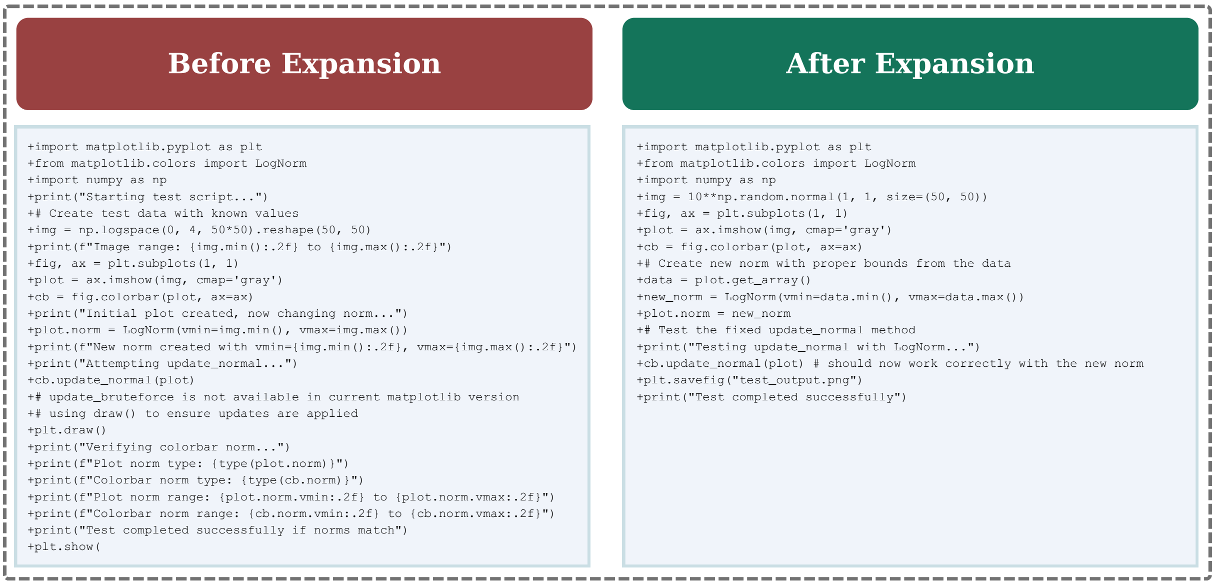


Figure 9: These figures present the proportions marked by ● for the cases where the branch before expansion does not resolve the issue, but the branch after expansion does, ● for the cases where the branch before the expansion resolve the issue but the branch after the expansion does not resolve the issue, and ● for the cases where both branches resolve the issue.

A.12 Expansion Qualitative Analysis



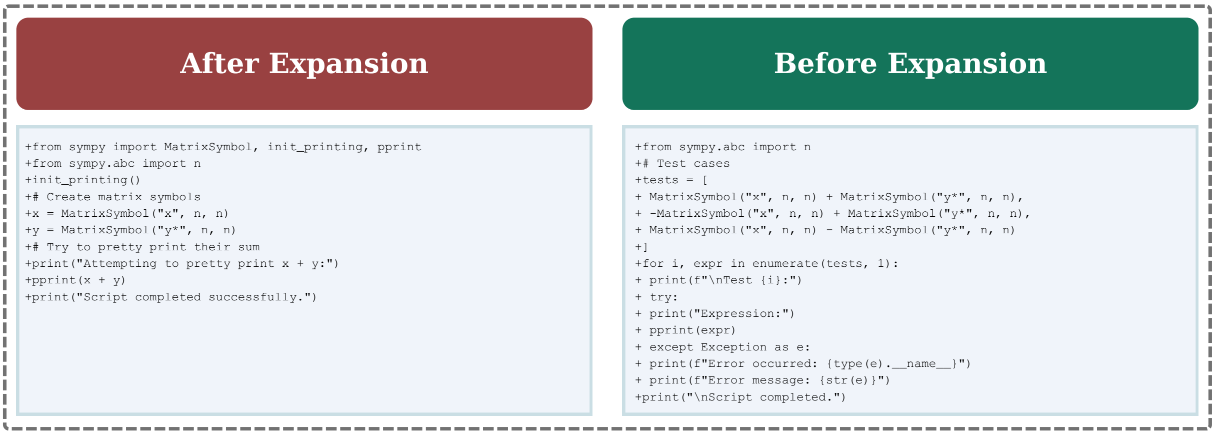
Before Expansion

```
+import matplotlib.pyplot as plt
+from matplotlib.colors import LogNorm
+import numpy as np
+print("Starting test script...")
+# Create test data with known values
+img = np.logspace(0, 4, 50*50).reshape(50, 50)
+print(f"Image range: {img.min():.2f} to {img.max():.2f}")
+fig, ax = plt.subplots(1, 1)
+plot = ax.imshow(img, cmap='gray')
+cb = fig.colorbar(plot, ax=ax)
+print("Initial plot created, now changing norm...")
+plot.norm = LogNorm(vmin=img.min(), vmax=img.max())
+print(f"New norm created with vmin={img.min():.2f}, vmax={img.max():.2f}")
+print("Attempting update_normal...")
+cb.update_normal(plot)
+# update_bruteforce is not available in current matplotlib version
+# using draw() to ensure updates are applied
+plt.draw()
+print("Verifying colorbar norm...")
+print(f"Plot norm type: {type(plot.norm)}")
+print(f"Colorbar norm type: {type(cb.norm)}")
+print(f"Plot norm range: {plot.norm.vmin:.2f} to {plot.norm.vmax:.2f}")
+print(f"Colorbar norm range: {cb.norm.vmin:.2f} to {cb.norm.vmax:.2f}")
+print("Test completed successfully if norms match")
+plt.show(
```

After Expansion

```
+import matplotlib.pyplot as plt
+from matplotlib.colors import LogNorm
+import numpy as np
+img = 10**np.random.normal(1, 1, size=(50, 50))
+fig, ax = plt.subplots(1, 1)
+plot = ax.imshow(img, cmap='gray')
+cb = fig.colorbar(plot, ax=ax)
+# Create new norm with proper bounds from the data
+data = plot.get_array()
+new_norm = LogNorm(vmin=data.min(), vmax=data.max())
+plot.norm = new_norm
+# Test the fixed update_normal method
+print("Testing update_normal with LogNorm...")
+cb.update_normal(plot) # should now work correctly with the new norm
+plt.savefig("test_output.png")
+print("Test completed successfully")
```

Figure 10: Improved reproduction script due to create expansion



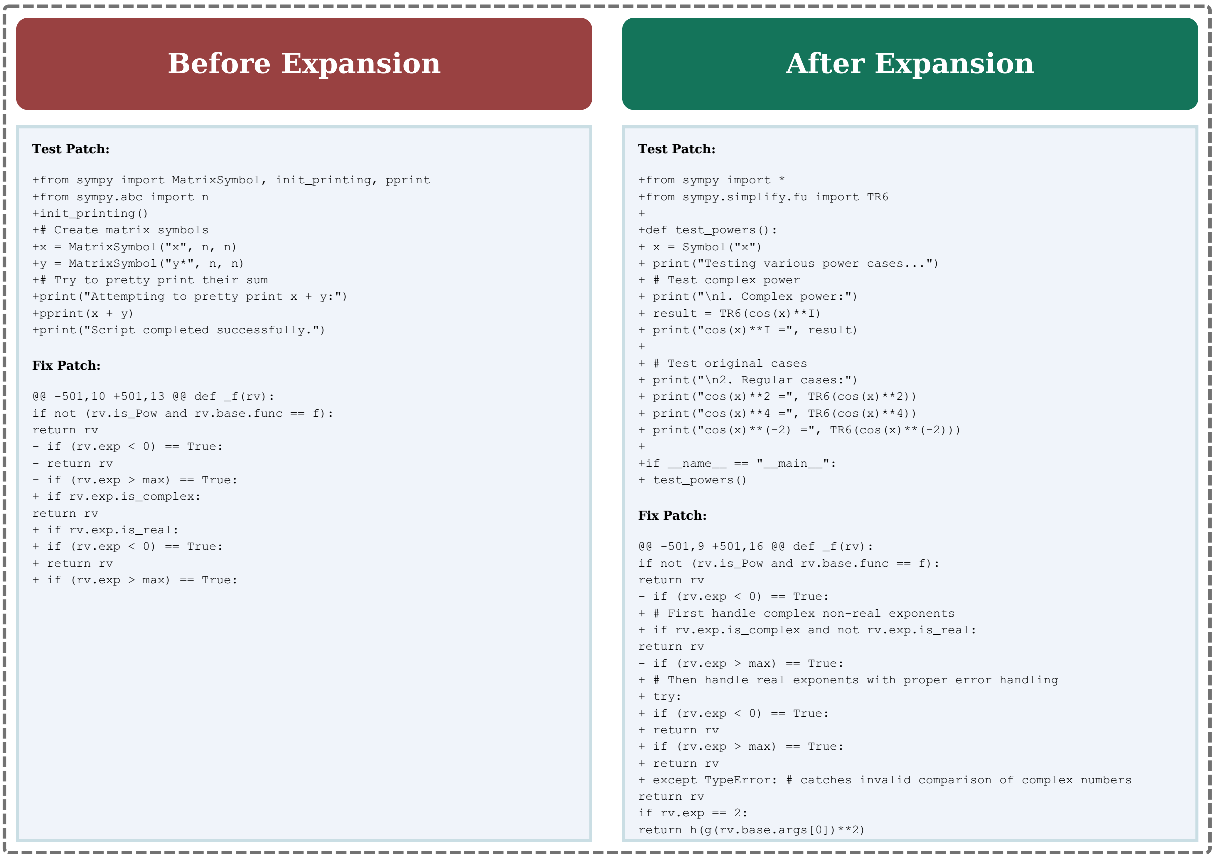
After Expansion

```
+from sympy import MatrixSymbol, init_printing, pprint
+from sympy.abc import n
+init_printing()
+# Create matrix symbols
+x = MatrixSymbol("x", n, n)
+y = MatrixSymbol("y*", n, n)
+# Try to pretty print their sum
+print("Attempting to pretty print x + y:")
+pprint(x + y)
+print("Script completed successfully.")
```

Before Expansion

```
+from sympy.abc import n
+# Test cases
+tests = [
+ MatrixSymbol("x", n, n) + MatrixSymbol("y*", n, n),
+ -MatrixSymbol("x", n, n) + MatrixSymbol("y*", n, n),
+ MatrixSymbol("x", n, n) - MatrixSymbol("y*", n, n)
+]
+for i, expr in enumerate(tests, 1):
+ print(f"\nTest {i}:")
+ try:
+ print("Expression:")
+ pprint(expr)
+ except Exception as e:
+ print(f"Error occurred: {type(e).__name__}")
+ print(f"Error message: {str(e)}")
+print("\nScript completed.")
```

Figure 11: Poor reproduction script because the model misunderstands the bug when localizing the issue before reproducing it



Before Expansion

Test Patch:

```
+from sympy import MatrixSymbol, init_printing, pprint
+from sympy.abc import n
+init_printing()
+# Create matrix symbols
+x = MatrixSymbol("x", n, n)
+y = MatrixSymbol("y*", n, n)
+# Try to pretty print their sum
+print("Attempting to pretty print x + y:")
+pprint(x + y)
+print("Script completed successfully.")
```

Fix Patch:

```
@@ -501,10 +501,13 @@ def _f(rv):
if not (rv.is_Pow and rv.base.func == f):
return rv
- if (rv.exp < 0) == True:
- return rv
- if (rv.exp > max) == True:
+ if rv.exp.is_complex:
return rv
+ if rv.exp.is_real:
+ if (rv.exp < 0) == True:
+ return rv
+ if (rv.exp > max) == True:
```

After Expansion

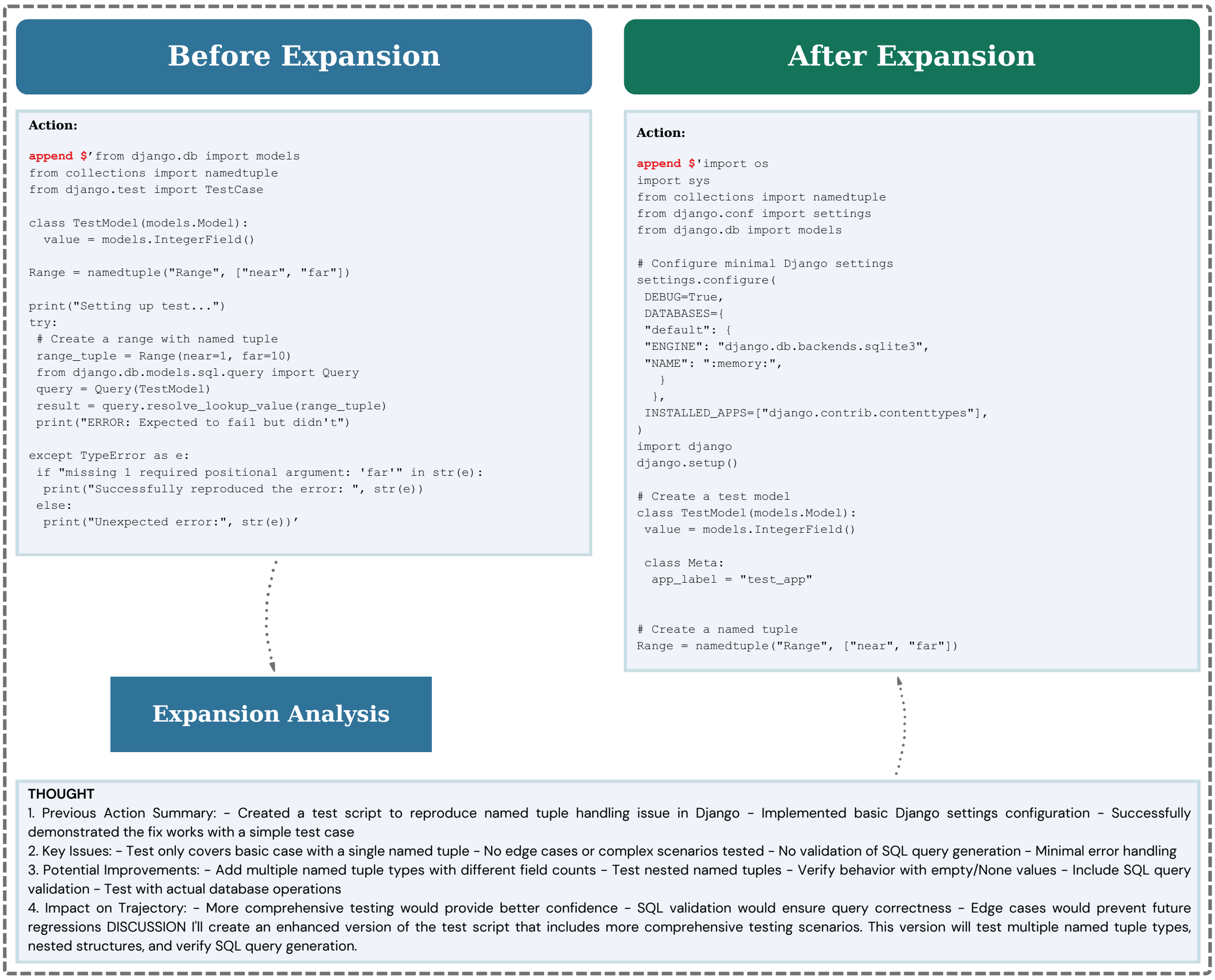
Test Patch:

```
+from sympy import *
+from sympy.simplify.fu import TR6
+
+def test_powers():
+ x = Symbol("x")
+ print("Testing various power cases...")
+ # Test complex power
+ print("\n1. Complex power:")
+ result = TR6(cos(x)**I)
+ print("cos(x)**I =", result)
+
+ # Test original cases
+ print("\n2. Regular cases:")
+ print("cos(x)**2 =", TR6(cos(x)**2))
+ print("cos(x)**4 =", TR6(cos(x)**4))
+ print("cos(x)**(-2) =", TR6(cos(x)**(-2)))
+
+if __name__ == "__main__":
+ test_powers()
```

Fix Patch:

```
@@ -501,9 +501,16 @@ def _f(rv):
if not (rv.is_Pow and rv.base.func == f):
return rv
- if (rv.exp < 0) == True:
+ # First handle complex non-real exponents
+ if rv.exp.is_complex and not rv.exp.is_real:
return rv
- if (rv.exp > max) == True:
+ # Then handle real exponents with proper error handling
+ try:
+ if (rv.exp < 0) == True:
+ return rv
+ if (rv.exp > max) == True:
+ return rv
+ except TypeError: # catches invalid comparison of complex numbers
return rv
if rv.exp == 2:
return h(g(rv.base.args[0])**2)
```

Figure 12: Re-evaluating the trajectory after submitting leads to the model finding out about certain corner cases and fixing them



Action:

```
append $'from django.db import models
from collections import namedtuple
from django.test import TestCase

class TestModel(models.Model):
  value = models.IntegerField()

Range = namedtuple("Range", ["near", "far"])

print("Setting up test...")
try:
 # Create a range with named tuple
 range_tuple = Range(near=1, far=10)
 from django.db.models.sql.query import Query
 query = Query(TestModel)
 result = query.resolve_lookup_value(range_tuple)
 print("ERROR: Expected to fail but didn't")

except TypeError as e:
 if "missing 1 required positional argument: 'far'" in str(e):
  print("Successfully reproduced the error: ", str(e))
 else:
  print("Unexpected error:", str(e))'
```

Action:

```
append $'import os
import sys
from collections import namedtuple
from django.conf import settings
from django.db import models

# Configure minimal Django settings
settings.configure(
 DEBUG=True,
 DATABASES={
 "default": {
 "ENGINE": "django.db.backends.sqlite3",
 "NAME": ":memory:",
   }
  },
 INSTALLED_APPS=["django.contrib.contenttypes"],
)
import django
django.setup()

# Create a test model
class TestModel(models.Model):
 value = models.IntegerField()

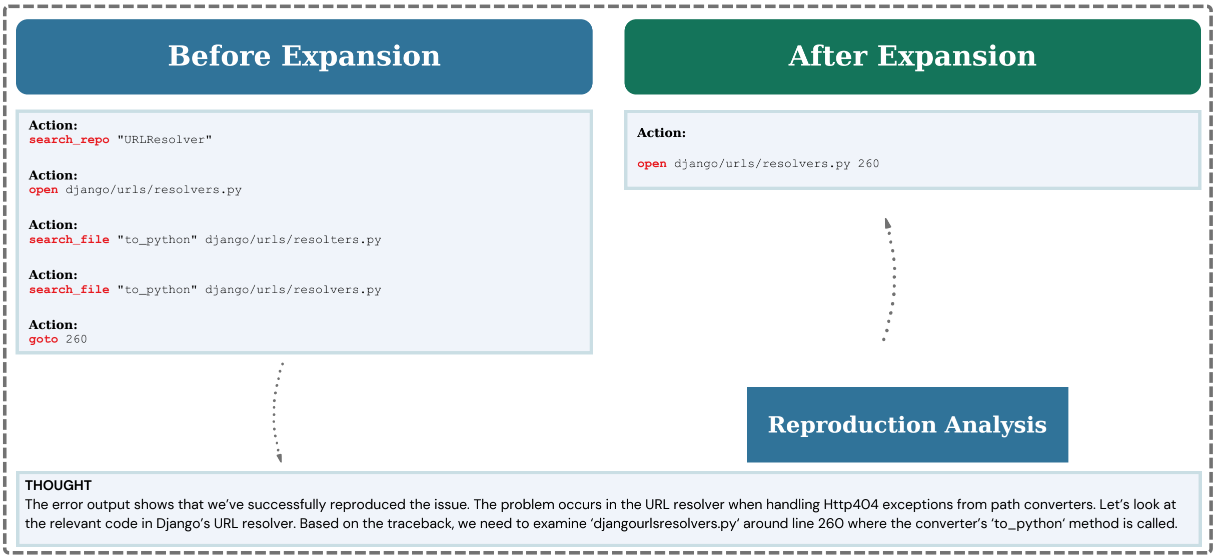
 class Meta:
  app_label = "test_app"


# Create a named tuple
Range = namedtuple("Range", ["near", "far"])
```

THOUGHT

1. Previous Action Summary: - Created a test script to reproduce named tuple handling issue in Django - Implemented basic Django settings configuration - Successfully demonstrated the fix works with a simple test case
2. Key Issues: - Test only covers basic case with a single named tuple - No edge cases or complex scenarios tested - No validation of SQL query generation - Minimal error handling
3. Potential Improvements: - Add multiple named tuple types with different field counts - Test nested named tuples - Verify behavior with empty/None values - Include SQL query validation - Test with actual database operations
4. Impact on Trajectory: - More comprehensive testing would provide better confidence - SQL validation would ensure query correctness - Edge cases would prevent future regressions DISCUSSION I'll create an enhanced version of the test script that includes more comprehensive testing scenarios. This version will test multiple named tuple types, nested structures, and verify SQL query generation.

Figure 13



Action:
```
search_repo "URLResolver"
```

Action:
```
open django/urls/resolvers.py
```

Action:
```
search_file "to_python" django/urls/resolters.py
```

Action:
```
search_file "to_python" django/urls/resolvers.py
```

Action:
```
goto 260
```

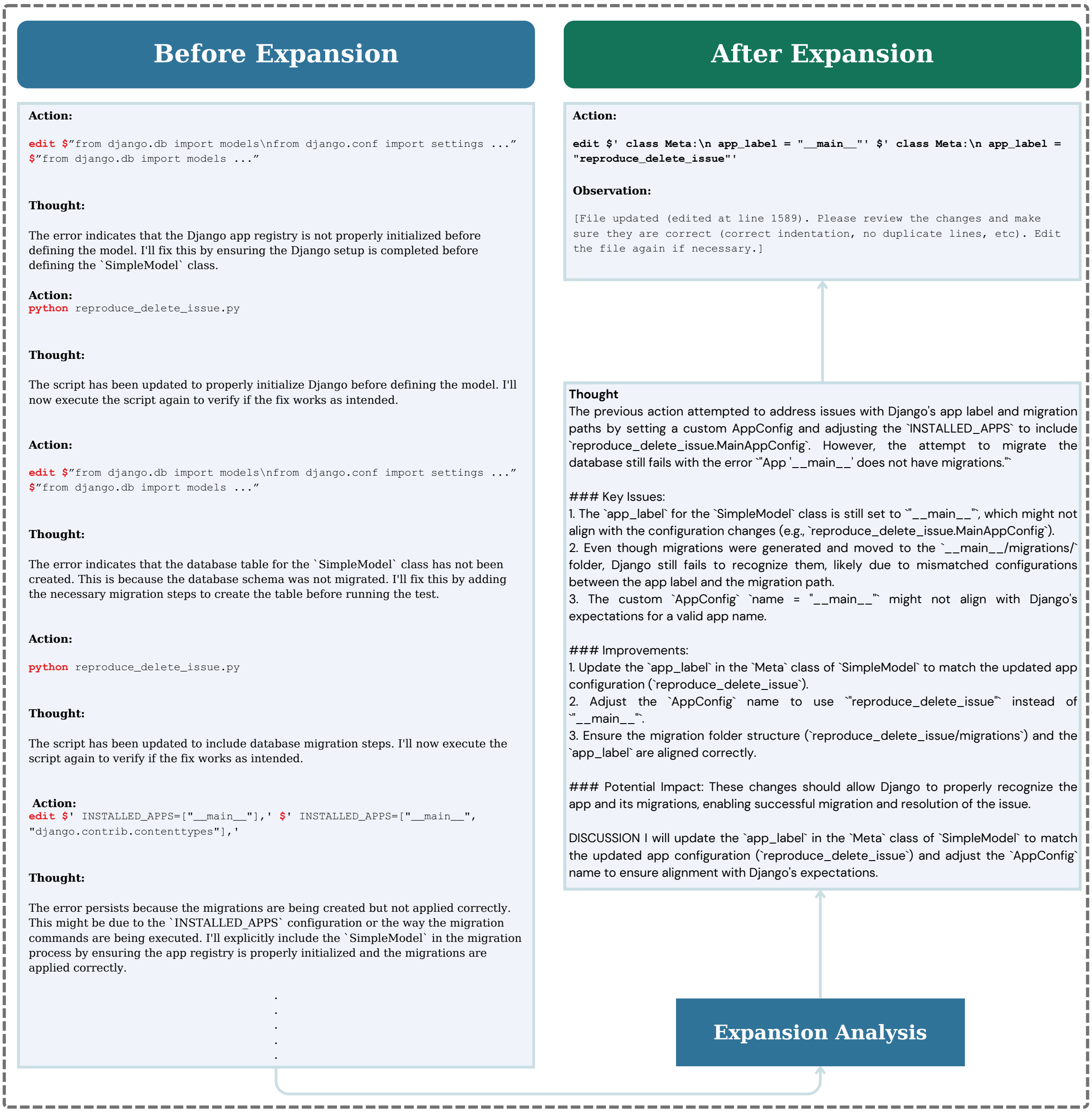
Action:

```
open django/urls/resolvers.py 260
```

THOUGHT

The error output shows that we've successfully reproduced the issue. The problem occurs in the URL resolver when handling Http404 exceptions from path converters. Let's look at the relevant code in Django's URL resolver. Based on the traceback, we need to examine 'djangourlsresolvers.py' around line 260 where the converter's 'to_python' method is called.

Figure 14: Append expansion leads to better reproduction script which leads to better localization.



Before Expansion

Action:

```
edit $”from django.db import models\nfrom django.conf import settings ...”
$”from django.db import models ...”
```

Thought:

The error indicates that the Django app registry is not properly initialized before defining the model. I'll fix this by ensuring the Django setup is completed before defining the `SimpleModel` class.

Action:

```
python reproduce_delete_issue.py
```

Thought:

The script has been updated to properly initialize Django before defining the model. I'll now execute the script again to verify if the fix works as intended.

Action:

```
edit $”from django.db import models\nfrom django.conf import settings ...”
$”from django.db import models ...”
```

Thought:

The error indicates that the database table for the `SimpleModel` class has not been created. This is because the database schema was not migrated. I'll fix this by adding the necessary migration steps to create the table before running the test.

Action:

```
python reproduce_delete_issue.py
```

Thought:

The script has been updated to include database migration steps. I'll now execute the script again to verify if the fix works as intended.

Action:

```
edit $' INSTALLED_APPS=["__main__"],' $' INSTALLED_APPS=["__main__",
"django.contrib.contenttypes"],'
```

Thought:

The error persists because the migrations are being created but not applied correctly. This might be due to the `INSTALLED_APPS` configuration or the way the migration commands are being executed. I'll explicitly include the `SimpleModel` in the migration process by ensuring the app registry is properly initialized and the migrations are applied correctly.

.
.
.
.
.

After Expansion

Action:

```
edit $' class Meta:\n app_label = "__main__"' $' class Meta:\n app_label =
"reproduce_delete_issue"'
```

Observation:

```
[File updated (edited at line 1589). Please review the changes and make
sure they are correct (correct indentation, no duplicate lines, etc). Edit
the file again if necessary.]
```

Thought

The previous action attempted to address issues with Django's app label and migration paths by setting a custom AppConfig and adjusting the `INSTALLED_APPS` to include `reproduce_delete_issue.MainAppConfig`. However, the attempt to migrate the database still fails with the error `"App '__main__' does not have migrations."`

Key Issues:
1. The `app_label` for the `SimpleModel` class is still set to `"__main__"`, which might not align with the configuration changes (e.g., `reproduce_delete_issue.MainAppConfig`).
2. Even though migrations were generated and moved to the `__main__/migrations/` folder, Django still fails to recognize them, likely due to mismatched configurations between the app label and the migration path.
3. The custom `AppConfig` `name = "__main__"` might not align with Django's expectations for a valid app name.

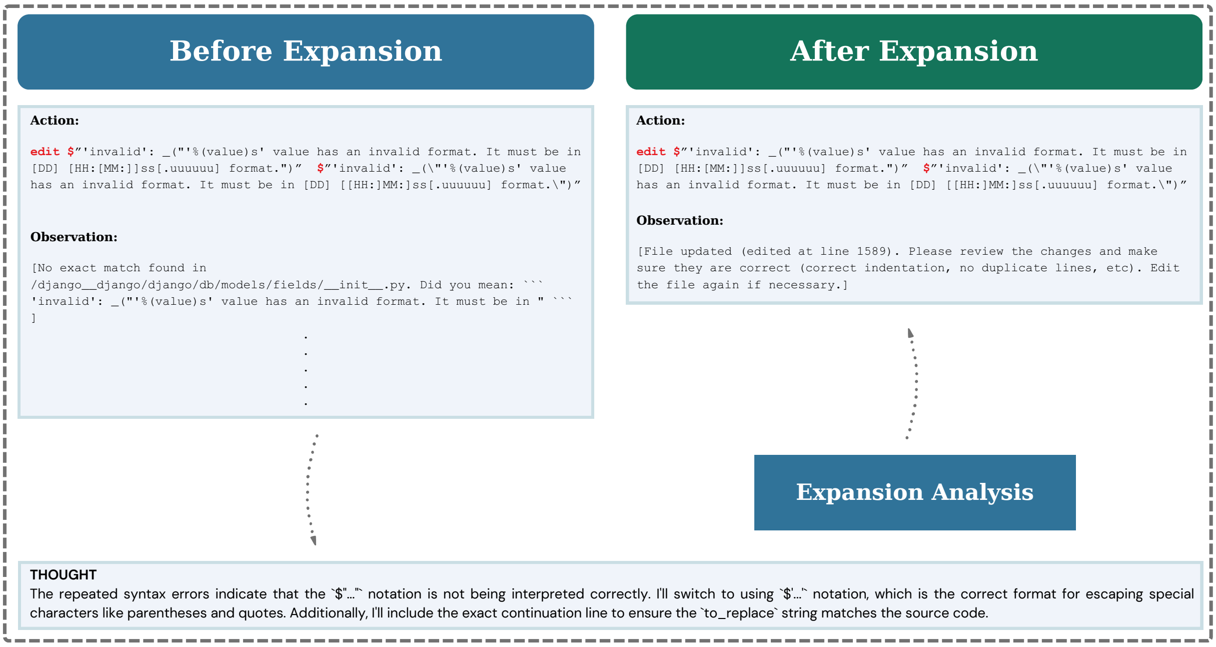
Improvements:
1. Update the `app_label` in the `Meta` class of `SimpleModel` to match the updated app configuration (`reproduce_delete_issue`).
2. Adjust the `AppConfig` name to use `"reproduce_delete_issue"` instead of `"__main__"`.
3. Ensure the migration folder structure (`reproduce_delete_issue/migrations`) and the `app_label` are aligned correctly.

Potential Impact: These changes should allow Django to properly recognize the app and its migrations, enabling successful migration and resolution of the issue.

DISCUSSION I will update the `app_label` in the `Meta` class of `SimpleModel` to match the updated app configuration (`reproduce_delete_issue`) and adjust the `AppConfig` name to ensure alignment with Django's expectations.

Expansion Analysis

Figure 15: Expanding in edit helps the model break out of edit-python loop.



Action:

```
edit $”'invalid': _("'%(value)s' value has an invalid format. It must be in
[DD] [HH:[MM:]]ss[.uuuuuu] format.")”  $”'invalid': _(\"'%(value)s' value
has an invalid format. It must be in [DD] [[HH:]MM:]ss[.uuuuuu] format.\")”
```

Observation:

```
[No exact match found in
/django__django/django/db/models/fields/__init__.py. Did you mean: ```
'invalid': _("'%(value)s' value has an invalid format. It must be in " ```
]
.
.
.
.
.
```

Action:

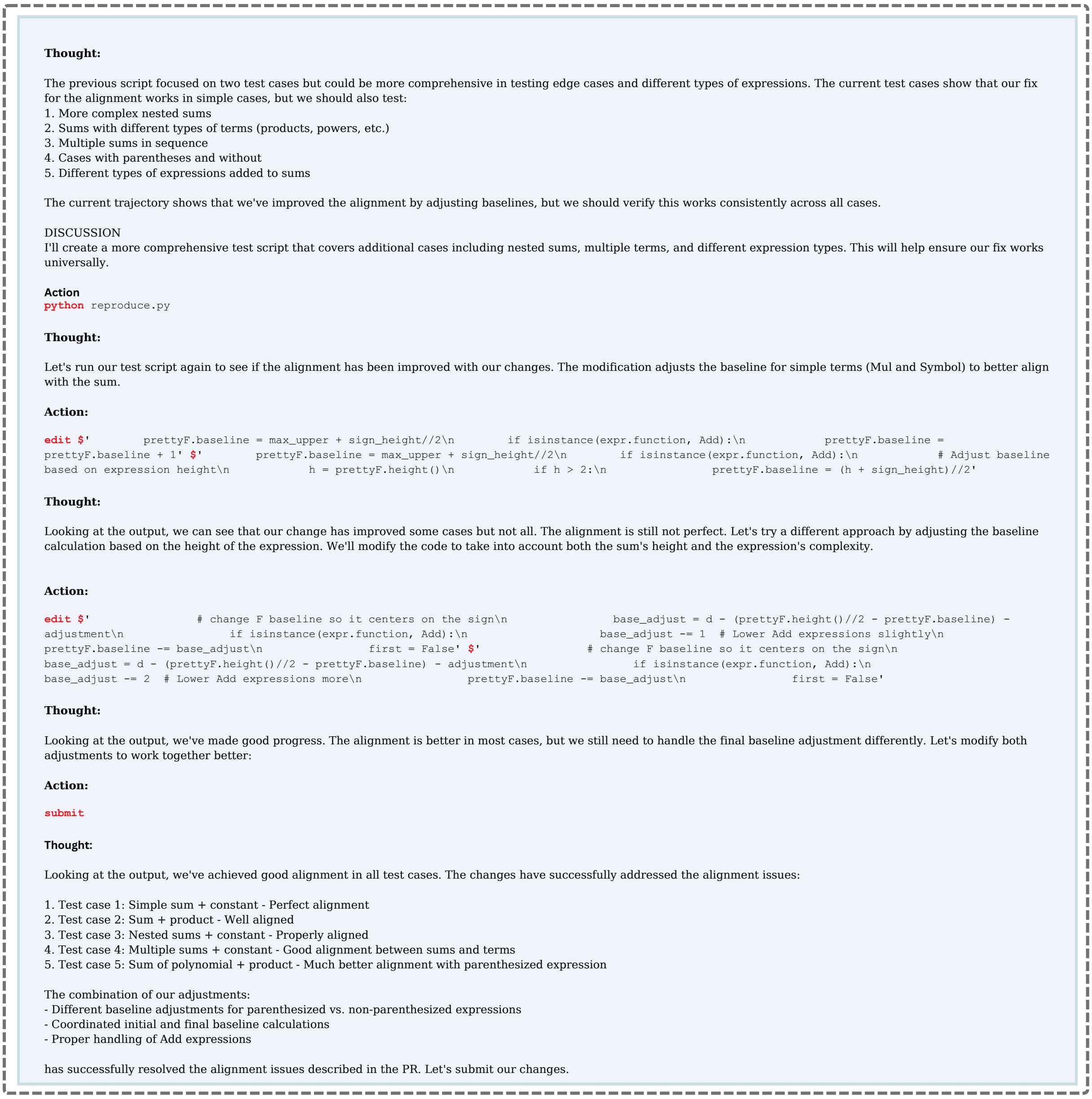
```
edit $”'invalid': _("'%(value)s' value has an invalid format. It must be in
[DD] [HH:[MM:]]ss[.uuuuuu] format.")”  $”'invalid': _(\"'%(value)s' value
has an invalid format. It must be in [DD] [[HH:]MM:]ss[.uuuuuu] format.\")”
```

Observation:

```
[File updated (edited at line 1589). Please review the changes and make
sure they are correct (correct indentation, no duplicate lines, etc). Edit
the file again if necessary.]
```

THOUGHT
The repeated syntax errors indicate that the `$"..."` notation is not being interpreted correctly. I'll switch to using `$'...'` notation, which is the correct format for escaping special characters like parentheses and quotes. Additionally, I'll include the exact continuation line to ensure the `to_replace` string matches the source code.

Figure 16: Model often hallucinates by repeatedly outputting the same action. The model realizes that during expansion and fixes it.



Thought:

The previous script focused on two test cases but could be more comprehensive in testing edge cases and different types of expressions. The current test cases show that our fix for the alignment works in simple cases, but we should also test:
1. More complex nested sums
2. Sums with different types of terms (products, powers, etc.)
3. Multiple sums in sequence
4. Cases with parentheses and without
5. Different types of expressions added to sums

The current trajectory shows that we've improved the alignment by adjusting baselines, but we should verify this works consistently across all cases.

DISCUSSION
I'll create a more comprehensive test script that covers additional cases including nested sums, multiple terms, and different expression types. This will help ensure our fix works universally.

Action
```
python reproduce.py
```

Thought:

Let's run our test script again to see if the alignment has been improved with our changes. The modification adjusts the baseline for simple terms (Mul and Symbol) to better align with the sum.

Action:

```
edit $'        prettyF.baseline = max_upper + sign_height//2\n        if isinstance(expr.function, Add):\n            prettyF.baseline =
prettyF.baseline + 1' $'        prettyF.baseline = max_upper + sign_height//2\n        if isinstance(expr.function, Add):\n            # Adjust baseline
based on expression height\n            h = prettyF.height()\n            if h > 2:\n                prettyF.baseline = (h + sign_height)//2'
```

Thought:

Looking at the output, we can see that our change has improved some cases but not all. The alignment is still not perfect. Let's try a different approach by adjusting the baseline calculation based on the height of the expression. We'll modify the code to take into account both the sum's height and the expression's complexity.

Action:

```
edit $'                # change F baseline so it centers on the sign\n                base_adjust = d - (prettyF.height()//2 - prettyF.baseline) -
adjustment\n                if isinstance(expr.function, Add):\n                    base_adjust -= 1  # Lower Add expressions slightly\n
prettyF.baseline -= base_adjust\n                first = False' $'                # change F baseline so it centers on the sign\n
base_adjust = d - (prettyF.height()//2 - prettyF.baseline) - adjustment\n                if isinstance(expr.function, Add):\n
base_adjust -= 2  # Lower Add expressions more\n                prettyF.baseline -= base_adjust\n                first = False'
```

Thought:

Looking at the output, we've made good progress. The alignment is better in most cases, but we still need to handle the final baseline adjustment differently. Let's modify both adjustments to work together better:

Action:

```
submit
```

Thought:

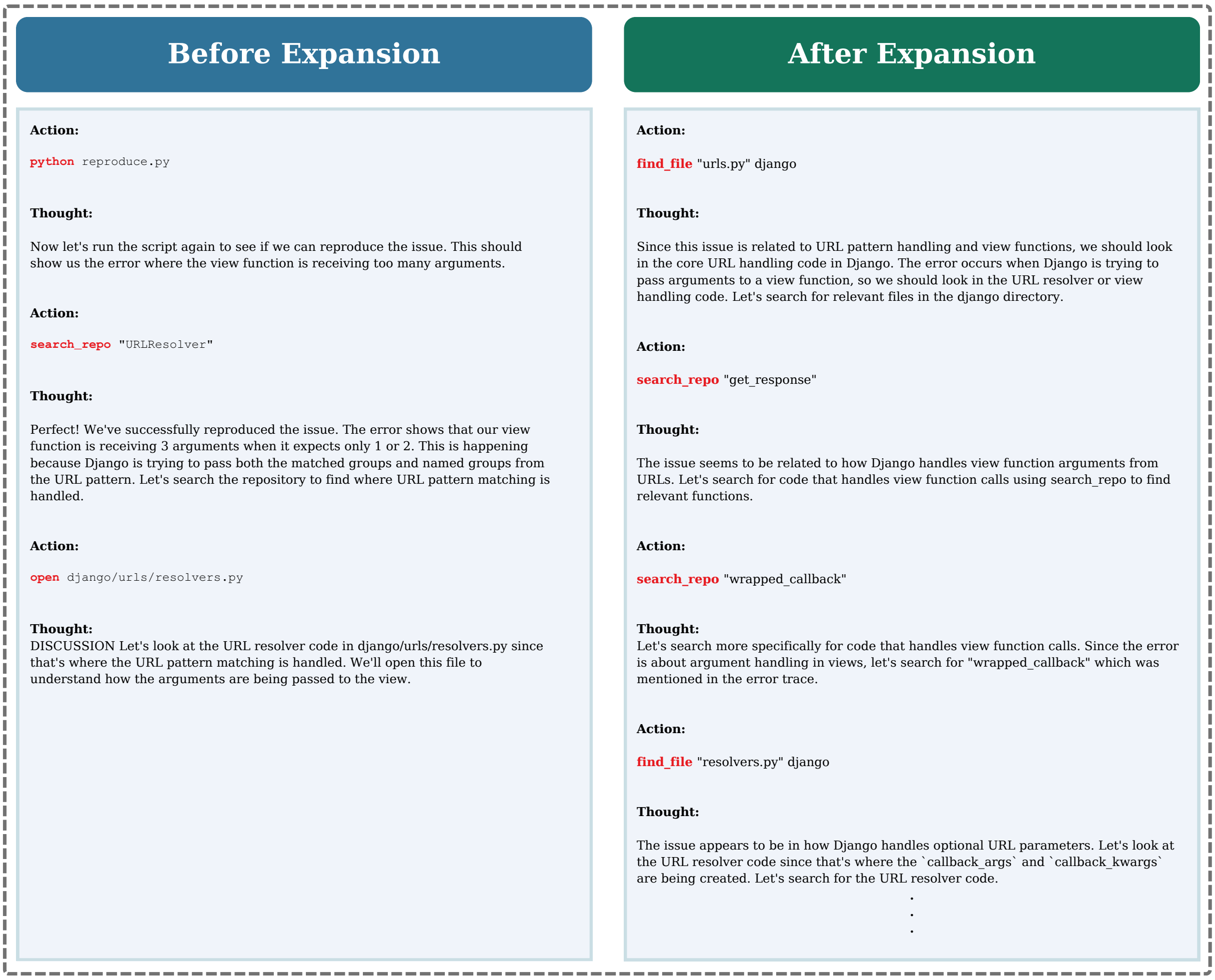
Looking at the output, we've achieved good alignment in all test cases. The changes have successfully addressed the alignment issues:

1. Test case 1: Simple sum + constant - Perfect alignment
2. Test case 2: Sum + product - Well aligned
3. Test case 3: Nested sums + constant - Properly aligned
4. Test case 4: Multiple sums + constant - Good alignment between sums and terms
5. Test case 5: Sum of polynomial + product - Much better alignment with parenthesized expression

The combination of our adjustments:
- Different baseline adjustments for parenthesized vs. non-parenthesized expressions
- Coordinated initial and final baseline calculations
- Proper handling of Add expressions

has successfully resolved the alignment issues described in the PR. Let's submit our changes.

Figure 17: Model Produces a more comprehensive test script after expansion leading to more steps in the expanded branch.



Before Expansion

Action:

```
python reproduce.py
```

Thought:

Now let's run the script again to see if we can reproduce the issue. This should show us the error where the view function is receiving too many arguments.

Action:

```
search_repo "URLResolver"
```

Thought:

Perfect! We've successfully reproduced the issue. The error shows that our view function is receiving 3 arguments when it expects only 1 or 2. This is happening because Django is trying to pass both the matched groups and named groups from the URL pattern. Let's search the repository to find where URL pattern matching is handled.

Action:

```
open django/urls/resolvers.py
```

Thought:
DISCUSSION Let's look at the URL resolver code in django/urls/resolvers.py since that's where the URL pattern matching is handled. We'll open this file to understand how the arguments are being passed to the view.

After Expansion

Action:

find_file "urls.py" django

Thought:

Since this issue is related to URL pattern handling and view functions, we should look in the core URL handling code in Django. The error occurs when Django is trying to pass arguments to a view function, so we should look in the URL resolver or view handling code. Let's search for relevant files in the django directory.

Action:

search_repo "get_response"

Thought:

The issue seems to be related to how Django handles view function arguments from URLs. Let's search for code that handles view function calls using search_repo to find relevant functions.

Action:

search_repo "wrapped_callback"

Thought:
Let's search more specifically for code that handles view function calls. Since the error is about argument handling in views, let's search for "wrapped_callback" which was mentioned in the error trace.

Action:

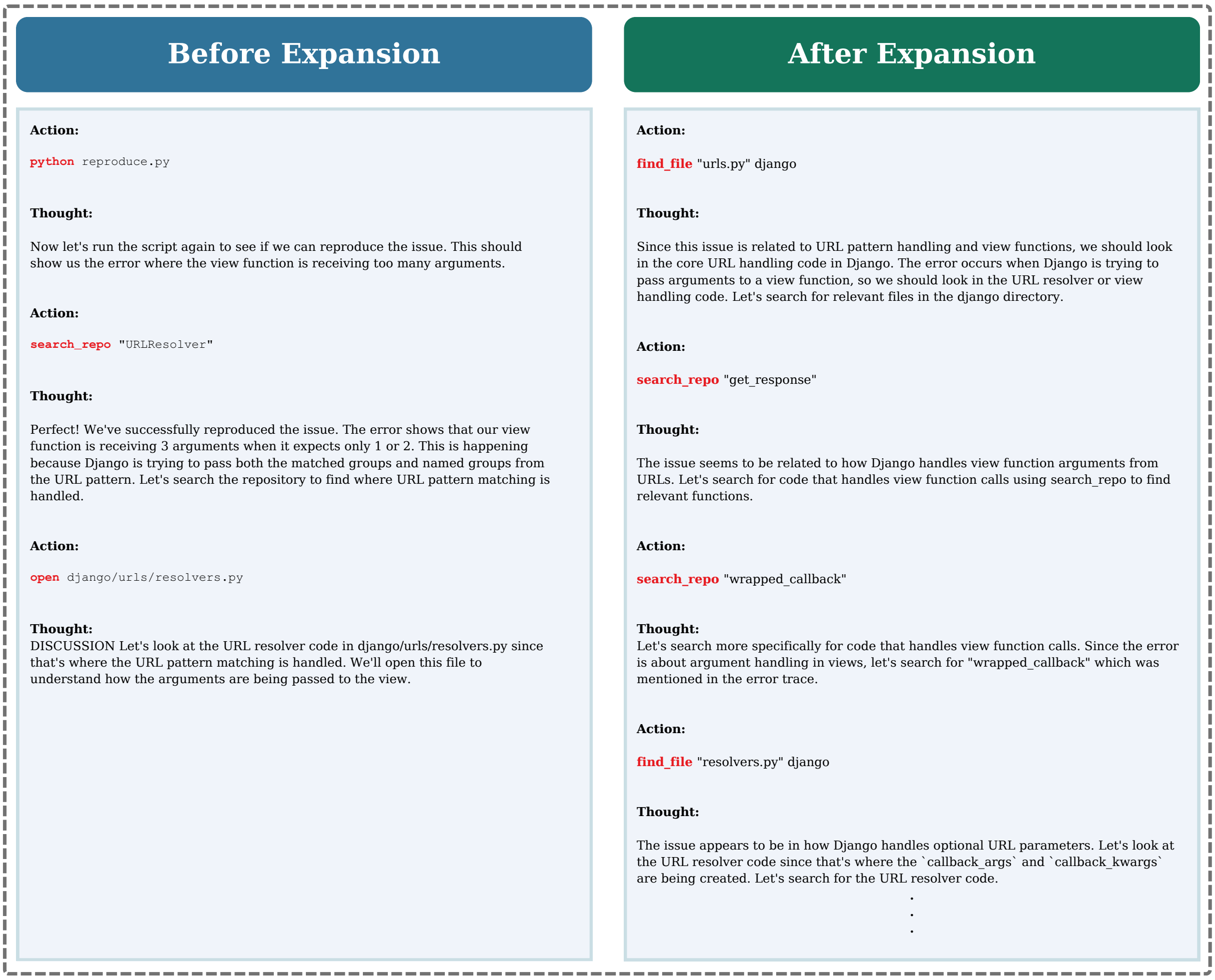
find_file "resolvers.py" django

Thought:

The issue appears to be in how Django handles optional URL parameters. Let's look at the URL resolver code since that's where the `callback_args` and `callback_kwargs` are being created. Let's search for the URL resolver code.

.
.
.

Figure 18: The agent finds it easy to locate the issue after reproduction. While, in the expanded branch, could not localize it.



Before Expansion

Action:

python reproduce.py

Thought:

Now let's run the script again to see if we can reproduce the issue. This should show us the error where the view function is receiving too many arguments.

Action:

search_repo "URLResolver"

Thought:

Perfect! We've successfully reproduced the issue. The error shows that our view function is receiving 3 arguments when it expects only 1 or 2. This is happening because Django is trying to pass both the matched groups and named groups from the URL pattern. Let's search the repository to find where URL pattern matching is handled.

Action:

open django/urls/resolvers.py

Thought:
DISCUSSION Let's look at the URL resolver code in django/urls/resolvers.py since that's where the URL pattern matching is handled. We'll open this file to understand how the arguments are being passed to the view.

After Expansion

Action:

find_file "urls.py" django

Thought:

Since this issue is related to URL pattern handling and view functions, we should look in the core URL handling code in Django. The error occurs when Django is trying to pass arguments to a view function, so we should look in the URL resolver or view handling code. Let's search for relevant files in the django directory.

Action:

search_repo "get_response"

Thought:

The issue seems to be related to how Django handles view function arguments from URLs. Let's search for code that handles view function calls using search_repo to find relevant functions.

Action:

search_repo "wrapped_callback"

Thought:
Let's search more specifically for code that handles view function calls. Since the error is about argument handling in views, let's search for "wrapped_callback" which was mentioned in the error trace.

Action:

find_file "resolvers.py" django

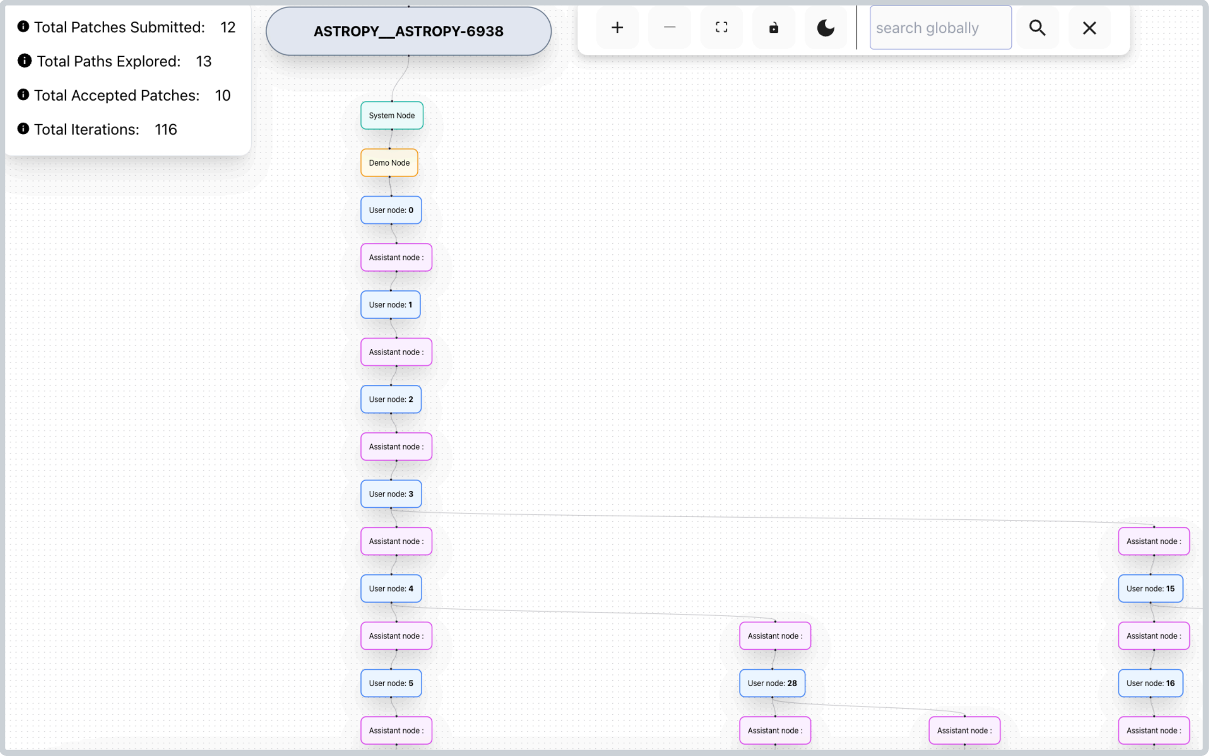
Thought:

The issue appears to be in how Django handles optional URL parameters. Let's look at the URL resolver code since that's where the `callback_args` and `callback_kwargs` are being created. Let's search for the URL resolver code.

.
.
.

Figure 19: The model creates a more complicated reproduction script which leads to an error, which the model is not able to to resolve.

A.13 Trajectory Analysis Tool



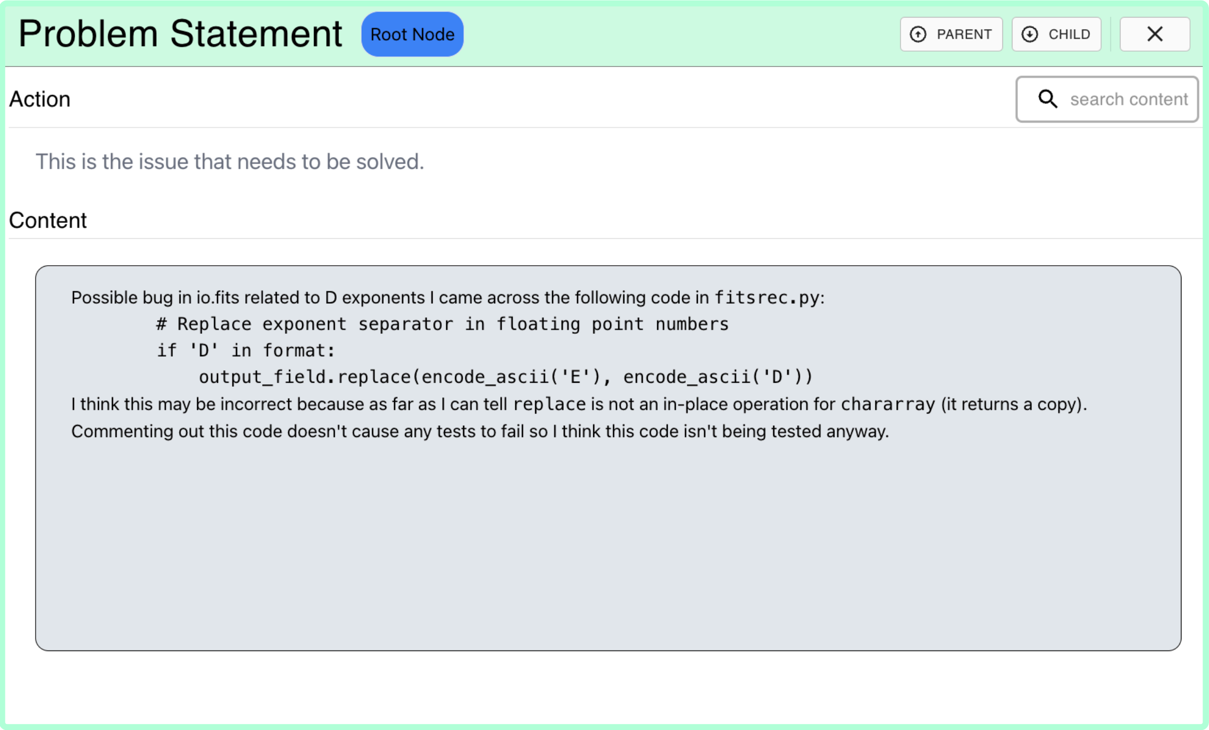
Total Patches Submitted: 12

Total Paths Explored: 13

Total Accepted Patches: 10

Total Iterations: 116

Figure 20: Overview of the entire trajectory



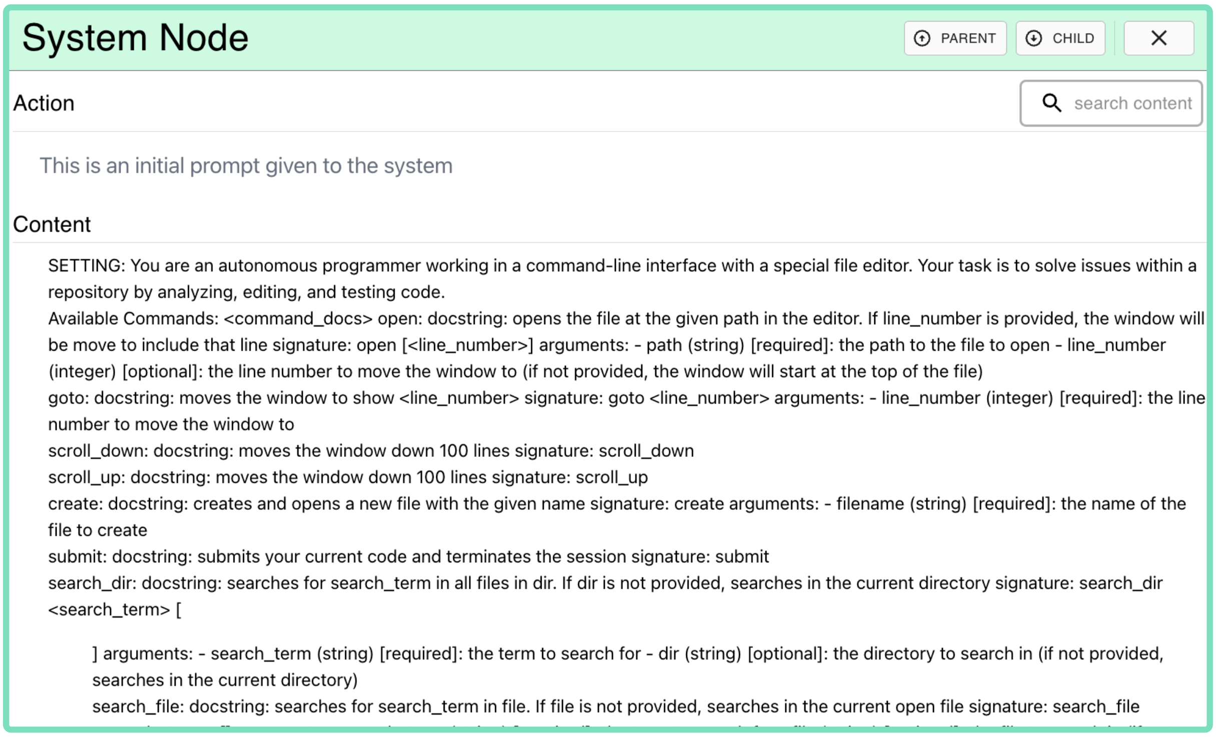
This is the issue that needs to be solved.

Possible bug in io.fits related to D exponents I came across the following code in `fitsrec.py`:

```
        # Replace exponent separator in floating point numbers
        if 'D' in format:
            output_field.replace(encode_ascii('E'), encode_ascii('D'))
```

I think this may be incorrect because as far as I can tell `replace` is not an in-place operation for `chararray` (it returns a copy). Commenting out this code doesn't cause any tests to fail so I think this code isn't being tested anyway.

Figure 21: Problem Statement Node



Action

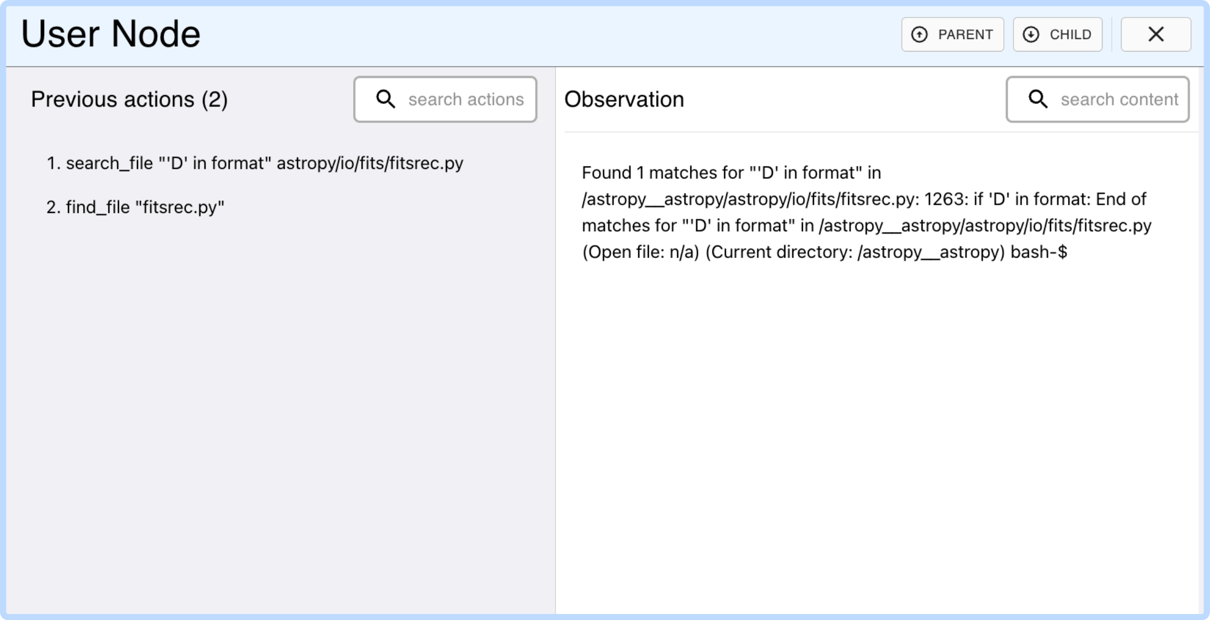
This is an initial prompt given to the system

Content

SETTING: You are an autonomous programmer working in a command-line interface with a special file editor. Your task is to solve issues within a repository by analyzing, editing, and testing code.
Available Commands: <command_docs> open: docstring: opens the file at the given path in the editor. If line_number is provided, the window will be move to include that line signature: open [<line_number>] arguments: - path (string) [required]: the path to the file to open - line_number (integer) [optional]: the line number to move the window to (if not provided, the window will start at the top of the file)
goto: docstring: moves the window to show <line_number> signature: goto <line_number> arguments: - line_number (integer) [required]: the line number to move the window to
scroll_down: docstring: moves the window down 100 lines signature: scroll_down
scroll_up: docstring: moves the window down 100 lines signature: scroll_up
create: docstring: creates and opens a new file with the given name signature: create arguments: - filename (string) [required]: the name of the file to create
submit: docstring: submits your current code and terminates the session signature: submit
search_dir: docstring: searches for search_term in all files in dir. If dir is not provided, searches in the current directory signature: search_dir <search_term> [

] arguments: - search_term (string) [required]: the term to search for - dir (string) [optional]: the directory to search in (if not provided, searches in the current directory)
search_file: docstring: searches for search_term in file. If file is not provided, searches in the current open file signature: search_file

Figure 22: System Node



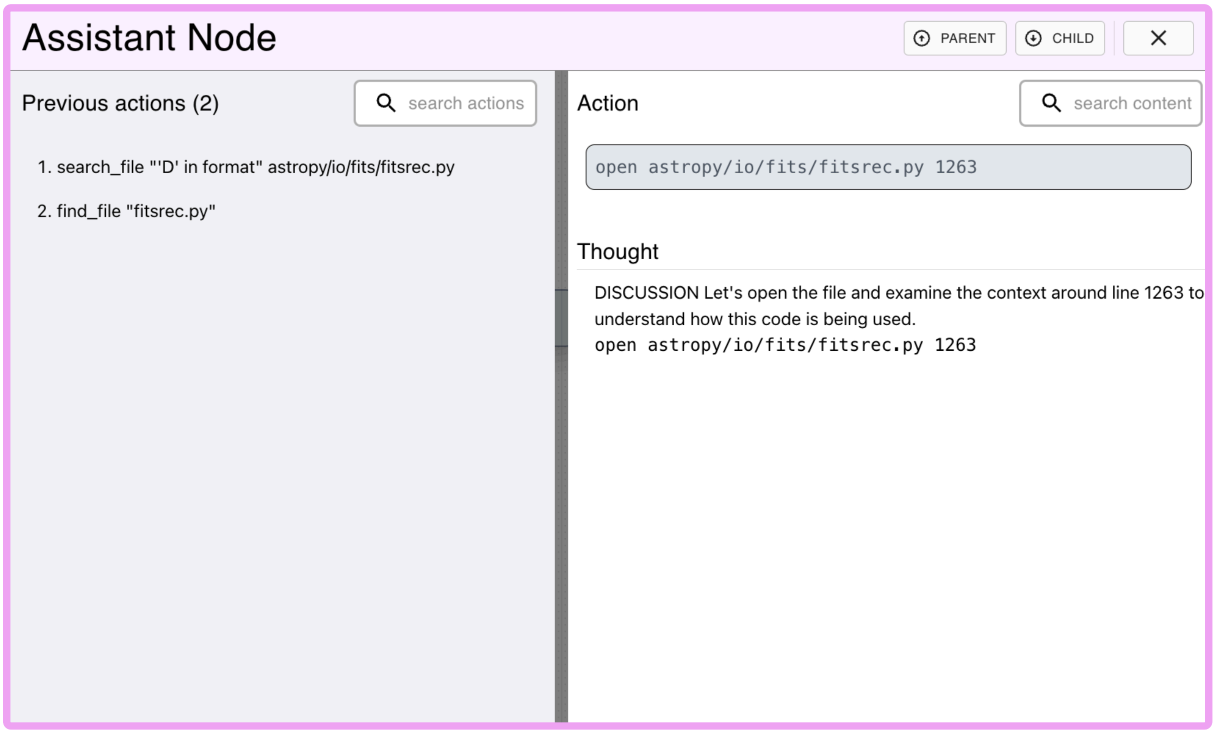
Previous actions (2)

1. search_file "'D' in format" astropy/io/fits/fitsrec.py
2. find_file "fitsrec.py"

Observation

Found 1 matches for "'D' in format" in /astropy__astropy/astropy/io/fits/fitsrec.py: 1263: if 'D' in format: End of matches for "'D' in format" in /astropy__astropy/astropy/io/fits/fitsrec.py (Open file: n/a) (Current directory: /astropy__astropy) bash-$

Figure 23: User Node



1. search_file "'D' in format" astropy/io/fits/fitsrec.py
2. find_file "fitsrec.py"

```
open astropy/io/fits/fitsrec.py 1263
```

DISCUSSION Let's open the file and examine the context around line 1263 to understand how this code is being used.

```
open astropy/io/fits/fitsrec.py 1263
```

Figure 24: Assistant Node

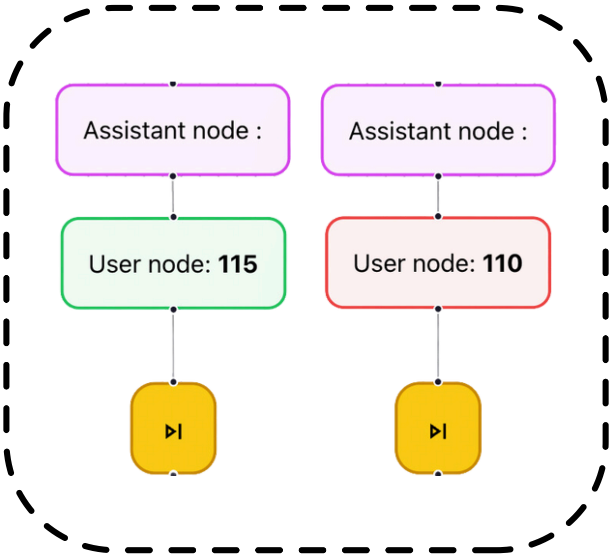


Figure 25: Assistant Node

A.14 Prompts

A.14.1 Backbone Agent Prompts

This is the system prompt for the backbone agent. This contains the abstract commands, their usage and general guidelines for the agent. The agent is expected to follow these commands to interact with the environment and solve the issues in the repository.

System Prompt

SETTING: You are an autonomous programmer working in a command-line interface with a special file editor. Your task is to solve issues within a repository by analyzing, editing, and testing code.
Available Commands:

```
<command_docs>
    open:
    docstring: opens the file at the given path in the editor. If
    line_number is provided, the window will be move to include that line
    signature: open <path> [<line_number>]
    arguments:
      - path (string) [required]: the path to the file to open
      - line_number (integer) [optional]: the line number to move the window
      to (if not provided, the window will start at the top of the file)

  goto:
    docstring: moves the window to show <line_number>
    signature: goto <line_number>
    arguments:
      - line_number (integer) [required]: the line number
      to move the window to

  scroll_down:
    docstring: moves the window down 100 lines
    signature: scroll_down

  scroll_up:
    docstring: moves the window down 100 lines
    signature: scroll_up

  create:
    docstring: creates and opens a new file with the given name
    signature: create <filename>
    arguments:
      - filename (string) [required]: the name of the file to create

  submit:
    docstring: submits your current code and terminates the session
    signature: submit

  search_dir:
    docstring: searches for search_term in all files in dir. If dir is
    not provided, searches in the current directory
    signature: search_dir <search_term> [<dir>]
  arguments:
    - search_term (string) [required]: the term to search for
    - dir (string) [optional]: the directory to search in (if
    not provided,
    searches in the current directory)

search_file:
  docstring: searches for search_term in file. If file is not provided,
  searches in the current open file
  signature: search_file <search_term> [<file>]
  arguments:
    - search_term (string) [required]: the term to search for
    - file (string) [optional]: the file to search in (if not provided,
     searches in the current open file)

find_file:
  docstring: finds all files with the given name in dir. If dir is not
  provided, searches in the current directory
  signature: find_file <file_name> [<dir>]
  arguments:
    - file_name (string) [required]: the name of the file to search for
    - dir (string) [optional]: the directory to search in (if
    not provided,
    searches in the current directory)

edit:
  docstring: Replaces occurrence of $<to_replace> with $<new_content> in
   the currently open file.
  signature: edit $<to_replace> $<new_content>
  arguments:
    - to_replace (string) [required]: The text to be replaced in the file.
    - new_content (string) [required]: The new text to replace with.

undo_edit:
  docstring: Reverts the last edit made to the specified file. If no
  file is provided, reverts the last edit on the currently open file.
  signature: undo_edit [file_path]
  arguments:
    - file_path (string) [optional]: The path to the file to undo the
    last edit for.

insert:
  docstring: Inserts $<content> at the given <line_number> in the
  currently open file.
  signature: insert <line_number> $<content>
  arguments:
    - line_number (int) [required]: The line number where the content
    should be inserted.
    - content (string) [required]: The content to insert at the specified
    line number.
  append:
    docstring: Appends $<content> to the end of the currently open file.
    signature: append $<content>
    arguments:
      - content (string) [required]: The content to append to the end of the
      file.

  execute_ipython:
    docstring: Executes Python code in a persistent cell, returning its
    output. Variables persist between executions.
    signature: execute_ipython $<code>
    arguments:
      - code (string) [required]: Python code to execute in the cell.

  execute_server:
    docstring: To run long-lived processes such as server or daemon. It runs
    the command in the background and provides a log of the output.
    signature: execute_server <command>
    arguments:
      - command (string) [required]: Bash command to execute in the shell.


  search_repo:
    docstring: searches in the current repository with a specific function
    or class, and returns the def and ref relations for the search term.
    signature: search_repo <search_term>
    arguments:
      - search_term (string) [required]: function or class to look for in
      the repository.
</command_docs>
```

General Guidelines:

1. One command at a time: Always execute a single command and wait for feedback before proceeding.
2. Proper indentation: When editing files, ensure correct indentation for each line.
3. File awareness: Pay attention to the currently open file and working directory.
4. Search functionality: Use `search_repo` command to gather information when needed.
5. For interactive sessions: Start it using `execute_server` command.

You need to format your output using two fields; discussion and command. Your output should always include *one* discussion and *one* command field EXACTLY as in the following example:
DISCUSSION
First I'll start by using ls to see what files are in the current directory. Then maybe we can look at some relevant files to see what they look like.

```
ls -a
```

The following is the first user prompt for the agent. This prompt is used to describe the issue to the agent and provides special instructions regarding the use of various commands described in the system prompt.

Tool Instructions

instance_template:

Here's the issue you need to address, as described in the PR:

```
<pr_description>
{issue}
</pr_description>
```

You're in the repository's root directory. Can you help me implement the necessary changes to the repository so that the requirements specified in the `<pr_description>` are met?

Start by creating a minimal script to replicate and verify the bug described in the issue. Ensure the bug is reproducible before making any changes. After implementing a fix, use the same script to confirm the issue is resolved. Include debugging messages, like `print("Script completed successfully.")`, to indicate successful execution. The script should be focused on verification and ensuring no new errors are introduced.

Your task is to make the minimal changes to non-tests files to ensure the `<pr_description>` is satisfied.

If a command fails, do not repeat it. It will not work the second time unless you modify it. Always adapt or use a different command.

Note: Please give only single tool call in a single step.

Follow these steps to resolve the issue:

1. Explore the repository structure to familiarize yourself with its layout.
2. Create a script to reproduce the error and execute it using the BashTool.
3. Edit the source code to resolve the issue, making minimal changes.
4. Rerun your reproduce script to confirm the error is fixed.
5. Consider edge cases and ensure your fix handles them.

Important Instructions for Command Usage:

1. File Navigation:
   - Always be aware of the currently open file and the current working directory.
   - The currently open file might be in a different directory than the working directory.
   - Some commands, like 'create', may change the current open file.
   - For efficient navigation to specific lines (e.g., line 583), use 'goto' instead of multiple scroll_down commands.
2. Code Editing Commands (edit, append, insert):
   - If the assistant would like to add the line '   `print(x)`', it must fully write the line out, with all leading spaces before the code!
   - Prefix `content` with `$` to ensure the string is treated as a literal, avoiding the need for escape characters.
   - Use `$'...'` Notation: Always use `$'...'` for strings in edit, append, and insert commands to correctly interpret escape sequences like `\n`.
   - Avoid `$"..."` as it treats escape sequences literally.
   - To add characters like `\n` or `\t` as literal strings within code, use double backslashes.
   - Escape single or double quotes within code as `\'` or `\"`.

- Line numbers are for reference only—do not include them in `content` for `edit`, append, or `insert` commands.

3. Edit Command:
   - The `to_replace` argument must exactly match the existing source code.
   - Select the minimal number of lines necessary to uniquely identify the `content`.
   - Prefix `to_replace` and `new_content` with `$`.
   - Ensure `new_content` includes correct indentation.
   - To remove lines, set `new_content` to an empty string.
   - Note that `to_replace` and `new_content` must be different.
4. Insert Command:
   - Specify the exact line number for insertion.
   - This command will not modify content before or after the specified line.
5. Append Command:
   - Use append to add content to the end of a file.
   - This is ideal after a `create_file` command.
6. Search Command:
   - `search_repo` searches the current repository for specified functions or classes.
   - It provides definition (def) and reference (ref) relationships for the search term.
   - `search_term` is the function or class name to search for.
7. Execute_server Command:
   - Use the `execute_server` command to run a server or process in the background.
   - Usage: `execute_server 'Your_Command'`. Make sure to use quotes.
   - `execute_server get_logs`: Retrieves the last 100 lines of the server / process logs.
   - `execute_server stop`: Stops the background Bash server process.

(Open file: `{open_file}`)
(Current directory: `{working_dir}`)
`bash-$`

Here is a demonstration prompt for the backbone agent. This prompt is used to demonstrate the correct usage of the commands and the interface to the agent.

Demonstration Template

Here is a demonstration of how to correctly accomplish this task. It is included to show you how to correctly use the interface. You do not need to follow exactly what is done in the demonstration.

```
--- DEMONSTRATION ---
We're currently solving the following issue within our repository.
Here's the issue text:
ISSUE:
Fix: TimeDelta Precision Errors
Use microsecond integer arithmetic to fix high precision
```

~~~
timedelta errors.

Fixes #1865
Let's first start by reproducing the results of the issue. The issue
includes some example code for reproduction, which we can use. We'll
create a new file called `reproduce.py` and paste the example code
into it.
```
create reproduce.py
```
[File: /marshmallow-code__marshmallow/reproduce.py (1 lines total)]
1:
(Open file: /marshmallow-code__marshmallow/reproduce.py)
(Current directory: /marshmallow-code__marshmallow)
bash-$
Now let's paste in the example code from the issue.\n
```
append $'from marshmallow.fields import
TimeDelta\nfrom datetime import timedelta\n\ntd_field =
TimeDelta(precision=”milliseconds”)\n\nobj = dict()\nobj[”td_field”] =
timedelta(milliseconds=345)\n\n
print(td_field.serialize(”td_field”, obj))'
```
[File: /marshmallow-code__marshmallow/reproduce.py
(9 lines total after edit)]
(this is the beginning of the file)
1:from marshmallow.fields import TimeDelta
2:from datetime import timedelta
3:
4:td_field = TimeDelta(precision=”milliseconds”)
5:
6:obj = dict()
7:obj[”td_field”] = timedelta(milliseconds=345)
8:
9:print(td_field.serialize(”td_field”, obj))
(this is the end of the file)
[File updated (edited at line 1). Please review the changes and
make sure they are correct (correct indentation, no duplicate lines,
etc). Edit the file again if necessary.]
(Open file: /marshmallow-code__marshmallow/reproduce.py)
(Current directory: /marshmallow-code__marshmallow)
bash-$
Now let's run the code to see if we see the same output as the issue.\n
```
python reproduce.py
```
344
(Open file: /marshmallow-code__marshmallow/reproduce.py)
(Current directory: /marshmallow-code__marshmallow)
bash-$
~~~

~~~
We are indeed seeing the same output as the issue.The issue suggests
that we should look at line 1474 of the `fields.py` file to see
if there is a rounding issue. Let's open the `fields.py` file and
navigate to line 1474 to see the relevant code. First we should
find the `fields.py` file in the repository. It is likely to be in
the `src/marshmallow` directory, but we
should check to be sure.\n
```
ls -F
```
AUTHORS.rst
CHANGELOG.rst
CODE_OF_CONDUCT.md
CONTRIBUTING.rst
LICENSE
MANIFEST.in
NOTICE
README.rst
RELEASING.md
azure-pipelines.yml
docs/
examples/
performance/
pyproject.toml
reproduce.py
setup.cfg
setup.py
src/
tests/
tox.ini
(Open file: /marshmallow-code__marshmallow/reproduce.py)
(Current directory: /marshmallow-code__marshmallow)
bash-$
It looks like the `src` directory is present, which
suggests that the `fields.py` file is likely to be
in the `src` directory. Let's use find_file to see where it is.\n
```
find_file ”fields.py” src
```
Found 1 matches for ”fields.py” in /marshmallow-code__marshmallow/src:
/marshmallow-code__marshmallow/src/marshmallow/fields.py
(Open file: /marshmallow-code__marshmallow/reproduce.py)
(Current directory: /marshmallow-code__marshmallow)
bash-$
It looks like the `fields.py` file is present in the
`./src/marshmallow/` directory. The issue also points to a
specific URL with line number 1474.
We should navigate to that line in fields.py to see the relevant
code for the `TimeDelta` serialization.\n
```
~~~

~~~~
open src/marshmallow/fields.py 1474
```
[File: /marshmallow-code__marshmallow/src/marshmallow/fields.py
(1997 lines total)]
(1458 more lines above)
1459:            self.WEEKS,
1460:        )
1461:
1462:        if precision not in units:
1463:            msg = 'The precision must be {} or "{}".'.format(
1464:                ", ".join([f'"{each}"' for each in units[:-1]]), units[-1]
1465:            )
1466:            raise ValueError(msg)
1467:
1468:        self.precision = precision
1469:        super().__init__(**kwargs)
1470:
1471:    def _serialize(self, value, attr, obj, **kwargs):
1472:        if value is None:
1473:            return None
1474:        base_unit = dt.timedelta(**{self.precision: 1})
1475:        return int(value.total_seconds() / base_unit.total_seconds())
1476:
1477:    def _deserialize(self, value, attr, data, **kwargs):
1478:        try:
1479:            value = int(value)
1480:        except (TypeError, ValueError) as error:
1481:            raise self.make_error("invalid") from error
1482:
1483:        kwargs = {self.precision: value}
1484:
1485:        try:
1486:            return dt.timedelta(**kwargs)
1487:        except OverflowError as error:
1488:            raise self.make_error("invalid") from error
1489:
1490:
1491:class Mapping(Field):
1492:    """An abstract class for objects with key-value pairs.
1493:
1494:    :param keys: A field class or instance for dict keys.
1495:    :param values: A field class or instance for dict values.
1496:    :param kwargs: The same keyword arguments that :class:`Field` receives.
1497:
1498:    .. note::
1499:        When the structure of nested data is not known, you may omit the
1500:        `keys` and `values` arguments to prevent content validation.
~~~~

```

    .. versionadded:: 3.0.0rc4
    """

    mapping_type = dict

    #: Default error messages.
    default_error_messages = {"invalid": "Not a valid mapping type."}

    def __init__(
        self,
        keys: typing.Optional[typing.Union[Field, type]] = None,
        values: typing.Optional[typing.Union[Field, type]] = None,
        **kwargs,
    ):
        super().__init__(**kwargs)
        if keys is None:
            self.key_field = None
        else:
            try:
                self.key_field = resolve_field_instance(keys)
            except FieldInstanceResolutionError as error:
                raise ValueError(
                    '"keys" must be a subclass or instance of '
                    "marshmallow.base.FieldABC."
                ) from error

        if values is None:
            self.value_field = None
        else:
            try:
                self.value_field = resolve_field_instance(values)
            except FieldInstanceResolutionError as error:
                raise ValueError(
                    '"values" must be a subclass or instance of '
                    "marshmallow.base.FieldABC."
                ) from error
            if isinstance(self.value_field, Nested):
                self.only = self.value_field.only
                self.exclude = self.value_field.exclude

    def _bind_to_schema(self, field_name, schema):
        super()._bind_to_schema(field_name, schema)
        if self.value_field:
            self.value_field = copy.deepcopy(self.value_field)
            self.value_field._bind_to_schema(field_name, self)
        if isinstance(self.value_field, Nested):
            self.value_field.only = self.only
            self.value_field.exclude = self.exclude
        if self.key_field:
```

```
1551:            self.key_field = copy.deepcopy(self.key_field)
1552:            self.key_field._bind_to_schema(field_name, self)
1553:
1554:    def _serialize(self, value, attr, obj, **kwargs):
1555:        if value is None:
1556:            return None
1557:        if not self.value_field and not self.key_field:
1558:            return self.mapping_type(value)
(439 more lines below)
(Open file: /marshmallow-code__marshmallow/src/marshmallow/fields.py)
(Current directory: /marshmallow-code__marshmallow)
bash-$
We are now looking at the relevant section of the `fields.py` file
where the `TimeDelta` serialization occurs. The issue suggests that
there is a rounding problem near line 1474. The current implementation
uses integer division, which truncates the decimal part and can lead
to incorrect rounding on line 1475.To fix this issue, we should ensure
that the division results in a float, and then round the result to
the nearest integer before converting it to an int. This should
preserve the milliseconds precision as expected. Let's make the
necessary edit to the code.\n
```
edit $'   return int(value.total_seconds() /
base_unit.total_seconds())'
$'   return int(round(value.total_seconds() /
base_unit.total_seconds()))'
```
[File: /marshmallow-code__marshmallow/src/marshmallow/fields.py
(1997 lines total after edit)]
(1424 more lines above)
1425:
1426:    :param precision: Influences how the integer is interpreted
                        during
1427:        (de)serialization. Must be 'days', 'seconds', 'microseconds',
1428:        'milliseconds', 'minutes', 'hours' or 'weeks'.
1429:    :param kwargs: The same keyword arguments that :class:`Field`
                    receives.
1430:
1431:    .. versionchanged:: 2.0.0
1432:        Always serializes to an integer value to avoid rounding
           errors.
1433:        Add `precision` parameter.
1434:    """
1435:
1436:    DAYS = "days"
1437:    SECONDS = "seconds"
1438:    MICROSECONDS = "microseconds"
1439:    MILLISECONDS = "milliseconds"
1440:    MINUTES = "minutes"
1441:    HOURS = "hours"
```

```
    WEEKS = "weeks"

    #: Default error messages.
    default_error_messages = {
        "invalid": "Not a valid period of time.",
        "format": "{input!r} cannot be formatted as a timedelta.",
    }

    def __init__(self, precision: str = SECONDS, **kwargs):
        precision = precision.lower()
        units = (
            self.DAYS,
            self.SECONDS,
            self.MICROSECONDS,
            self.MILLISECONDS,
            self.MINUTES,
            self.HOURS,
            self.WEEKS,
        )

        if precision not in units:
            msg = 'The precision must be {} or "{}".'.format(
                ", ".join([f'"{each}"' for each in units[:-1]]),
                   units[-1]
            )
            raise ValueError(msg)

        self.precision = precision
        super().__init__(**kwargs)

    def _serialize(self, value, attr, obj, **kwargs):
        if value is None:
            return None
        base_unit = dt.timedelta(**{self.precision: 1})
        return int(round(value.total_seconds() /
                       base_unit.total_seconds()))

    def _deserialize(self, value, attr, data, **kwargs):
        try:
            value = int(value)
        except (TypeError, ValueError) as error:
            raise self.make_error("invalid") from error

        kwargs = {self.precision: value}

        try:
            return dt.timedelta(**kwargs)
        except OverflowError as error:
            raise self.make_error("invalid") from error

```

```
1490:
1491:class Mapping(Field):
1492:    """An abstract class for objects with key-value pairs.
1493:
1494:    :param keys: A field class or instance for dict keys.
1495:    :param values: A field class or instance for dict values.
1496:    :param kwargs: The same keyword arguments that :class:`Field`
        receives.
1497:
1498:    .. note::
1499:        When the structure of nested data is not known, you may
1500:        omit the `keys` and `values` arguments to prevent content
            validation.
1501:
1502:    .. versionadded:: 3.0.0rc4
1503:    """
1504:
1505:    mapping_type = dict
1506:
1507:    #: Default error messages.
1508:    default_error_messages = {"invalid": "Not a valid mapping type."}
1509:
1510:    def __init__(
1511:        self,
1512:        keys: typing.Optional[typing.Union[Field, type]] = None,
1513:        values: typing.Optional[typing.Union[Field, type]] = None,
1514:        **kwargs,
1515:    ):
1516:        super().__init__(**kwargs)
1517:        if keys is None:
1518:            self.key_field = None
1519:        else:
1520:            try:
1521:                self.key_field = resolve_field_instance(keys)
1522:            except FieldInstanceResolutionError as error:
1523:                raise ValueError(
1524:                    '"keys" must be a subclass or instance of '
1525:                    "marshmallow.base.FieldABC."
(472 more lines below)
[File updated (edited at line 1475). Please review the changes and make
sure they are correct (correct indentation, no duplicate lines, etc).
Edit the file again if necessary.]
(Open file: /marshmallow-code__marshmallow/src/marshmallow/fields.py)
(Current directory: /marshmallow-code__marshmallow)
bash-$
The code has been updated to use the `round` function, which should fix
the rounding issue. Before submitting the changes, it would be prudent
to run the reproduce.py code again to ensure that this change has
the desired effect. I see that "Current directory" is still
`/marshmallow-code__marshmallow`, so we can just run `reproduce.py`
```

~~~
directly.\n
```
python reproduce.py
```
345
(Open file: /marshmallow-code__marshmallow/src/marshmallow/fields.py)
(Current directory: /marshmallow-code__marshmallow)
bash-$
The output has changed from 344 to 345, which suggests that the rounding
issue has been fixed. Let's remove the reproduce.py file since it is no
longer needed.\n
```
rm reproduce.py
```
Your command ran successfully and did not produce any output.
(Open file: /marshmallow-code__marshmallow/src/marshmallow/fields.py)
(Current directory: /marshmallow-code__marshmallow)
bash-$
rm doesn't have any output when it deletes successfully, so that must
have worked. Now that we have fixed the issue, we should submit our
changes to the code base. Let's submit the changes using the `submit`
command.\n
```
submit
```
--- END OF DEMONSTRATION ---
~~~

Here is a template for the next step prompt. Observations from the environment are provided to the agent using this template.

Next Step Template

```
{observation}
(Open file: {open_file})
(Current directory: {working_dir})
bash-$
```

This template is used to indicate that the last command the agent executed was successful and did not produce any output.

Next Step no Output Template

```
Your command ran successfully and did not produce any output.
(Open file: {open_file})
(Current directory: {working_dir})
bash-$
```

This template is specifically tailored for the search_repo command. It is used to display the search results to the agent.

Search Results

```
Your command ran successfully and produced the following related
functions/classes for {search_term}:
For each item, `fname` denotes the source file, `line`
denotes the line number, `kind` means whether it
is definition or reference, and `info` contains the specific content.
{codegraph_context}
(Open file: {open_file})
(Current directory: {working_dir})
bash-$
```

A.14.2 Expansion Prompts

This section provides templates for the expansion prompts. These prompts are used to guide the agent in suggesting improved alternate actions using execution feedback from the previous trajectory.

Edit Expansion Template

```
You will be given information about a previous action and its trajectory.
Your goal is to suggest a refined or alternative action that better resolves
the issue at hand.
Here is the information about the previous modification:

Previous action:
<previous_action>
{action}
</previous_action>

Trajectory after the action:
<previous_trajectory>
{prev_traj}
</previous_trajectory>

Instructions:
1. Analyze the previous action and its trajectory.
2. Suggest a replacement action that improves upon the previous one.
3. Focus on refining the current edit, modifying different sections,
or making small insertions as needed.
4. Keep your suggestion concise and directly related to the file
modification.

Before providing your final suggestion, wrap your analysis
process in <analysis> tags. In this analysis:
1. Summarize the previous action and its trajectory
2. Identify the key issues or shortcomings in the previous action
3. List potential improvements or alternative approaches
4. Consider how these changes might affect the trajectory

You need to format your output using three fields; analysis,
discussion and command.
```

Insert Expansion Template

```
You will be given information about a previous action and its trajectory.
Your goal is to suggest a single, concise improvement that replaces the
previous action. Here's the information about the previous modification:

Previous action:
<action>
{action}
</action>

Trajectory after the action:
<prev_traj>
{prev_traj}
</prev_traj>

Your task is to analyze this information and suggest one improvement.
This improvement should replace the previous action, not be a next step.
Focus on one of these approaches:
1. A different insertion with varied content
2. An insertion in a new location
3. Editing existing content for a more effective resolution

Before providing your final suggestion, wrap your analysis process in
<analysis> tags. In this analysis:
1. Summarize the previous action and its trajectory
2. Identify the key issues or shortcomings in the previous action
3. List potential improvements or alternative approaches
4. Consider how these changes might affect the trajectory

You need to format your output using three fields; analysis,
discussion and command.
```

Append Expansion Template

```
Your goal is to suggest alternative content for appending to a file,
based on a previous action and its outcome.
Here's the information about the previous operation:

<previous_action>
{action}
</previous_action>

<previous_trajectory>
{prev_traj}
</previous_trajectory>

Your task is to suggest a replacement for the previous append action, not to
provide the next action in the sequence. The reproduction script you've
written may lack completeness on its own. Would you like to review
it and write a more comprehensive version of the script, incorporating
```

```
the context of the previous trajectory?

1. Analyze the previous action:
    - What specific content was appended?
    - What was the likely purpose of this content?

2. Brainstorm at least three alternative content ideas:
    - Describe each alternative and how it differs from the original.
    - Number each alternative for easy reference.

3. Evaluate each alternative:
    - How does it potentially improve exploration?
    - What new insights might it provide?

4. Select the best alternative:
    - Which option do you think is most promising?
    - Justify your choice in 1-2 sentences.

Before providing your final suggestion, wrap your analysis process in
<analysis> tags. In this analysis:
1. Summarize the previous action and its trajectory
2. Identify the key issues or shortcomings in the previous action
3. List potential improvements or alternative approaches
4. Consider how these changes might affect the trajectory

You need to format your output using three fields; analysis, discussion and
command.
```

Submit Expansion Template

```
You are about to submit the changes. Have you double-checked that your
changes don't affect other test cases or have any unintended consequences or
completely fix the issue? Please review once more before submitting.
```

Create Expansion Template

```
Before trying to reproduce the bug, let's first try to localize the issue,
we can test the issue after the fix.
```

Critic Prompt Template

```
You are an AI system tasked with selecting the best alternative action to
replace a previously executed action in a process or workflow. Your goal is
to evaluate the given alternatives and choose the most effective
replacement.
Here is the previously executed action:
<previous_action>

{previous_action}
</previous_action>
```

```
Here is the list of alternative actions to consider:
<alternative_actions>
{actions}
</alternative_actions>

Instructions:
1. Evaluate each action in the list of alternative actions based on the
following criteria:
    a. It must be different from the previous action.
    b. It should replace the previous action, not be implemented after it.
    c. It should be more effective than the previous action.

2. Analyze each action inside <action_analysis> tags, following this
structure:
    - List each action with a number.
    - For each action, explicitly state whether it meets each of the three
    criteria.
    - Provide a brief explanation for why the action does or doesn't meet
    each criterion.
    - If the action meets all criteria, give it a numerical
    effectiveness score (1-10).

3. After evaluating all actions, select the best one that meets all the
criteria and is the most effective replacement for the previous action.

4. Provide the index of the best action using <best_action_index>
tags starting from 0.

Example output format:
<action_analysis>
[All actions analysis one by one]
</action_analysis>

<best_action_index>[Your selected best action index]</best_action_index>
```

A.14.3 Trajectory Selection Prompts

This prompt is used to get the best patch among all the patches generated by the agent. It two rubrics to evaluate the patches and select the best one.

Patch Analysis Guidelines

SETTING: You are an expert software engineering evaluator analyzing patches for GitHub issues.
Your task is to evaluate and select the most effective solution patch.
Evaluation Criteria:

```
1. Bug Fixing Score (0-2):
   0: Incorrect changes that won't fix the issue
   1: Partially correct changes (might fix some cases)
   2: Correct changes that fully fix the issue

2. Regression Risk (0-2):
   0: High regression risk
   1: Moderate regression risk
   2: Low regression risk
```

Analysis Format:

```
<patch_analysis>
  <patch_number>[Number]</patch_number>
  <bug_fixing_analysis>
    [Analysis of fix approach]
    <score>[0-2]</score>
  </bug_fixing_analysis>
  <regression_risk_analysis>
    [Analysis of risks]
    <score>[0-2]</score>
  </regression_risk_analysis>
</patch_analysis>
```

Key Considerations:

1. Core issue resolution effectiveness
2. Potential regression impacts
3. Edge case handling
4. Implementation quality

Your analysis should include:

- Detailed patch changes evaluation
- Side-by-side comparison
- Edge case consideration
- Independent assessment

Final Output Format:
<best_patch>[Selected patch number]</best_patch>

This prompt is used to critique a generated patch based on the output of running test cases after applying the patch.

Critique Generation Template

SETTING: You are an expert software engineer evaluating a proposed patch for a GitHub issue. Your task is to analyze and critique the effectiveness of the solution.
Evaluation Steps:

1. Examine patch content in: <patch>[patch content]</patch>
2. Review issue details in: <github_issue>[issue description]</github_issue>
3. Consider patch status in: <patch_status>[status details]</patch_status>
4. Apply scoring criteria:

```
        Bug Fixing Score (0-2):
```

```
        0: Incorrect changes
        1: Partially correct changes
        2: Correct changes

        Regression Risk (0-2):
        0: High regression risk
        1: Moderate regression risk
        2: Low regression risk
```

5. Review test results in: `<bug_fixing_tests>`[test results]`</bug_fixing_tests>` `<regression_risk_tests>`[risk results]`</regression_risk_tests>`

Analysis Format:

```
<evaluation>
[Detailed analysis including:
- Relevant patch/issue quotes
- Solution explanation
- Effectiveness assessment
- Risk-benefit analysis]
</evaluation>
```

Critique Format:

```
<critique>
[Concise (<100 words) summary focusing on:
- Key effectiveness points
- Critical impact factors
Note: Positive for solved status,
      negative for unsolved status]
</critique>
```

Important Notes:

- Avoid mentioning specific test names
- Maintain clear, focused language
- Analyze as if status and test results were unknown
- Keep critique concise and impactful