A REPRODUCIBILITY

All models and data used in our work are publicly available. We additionally provide hyperparameter details in **Appendix** 2. The code will be released as a public repository upon publication.

B ADDITIONAL IMPLEMENTATION DETAILS

Moatless-Adapted is an extended version of the moatless-tools library with support for a tree structure, the ability to revert to earlier versions of the codebase, and the capability to run tests.

The standard implementation of moatless-tools is based on a finite state machine structure where a state holds information about file context and properties set in the configuration or from previous states. It can then transition to a new state when an action is executed. The request that initiates the action is created by an LLM. This follows a linear structure where one state can transition to another state. In Moatless-Adapted, this model is extended so that a state can expand by using actions to create more states. The connections between states are then represented in a tree structure with nodes.

Each state has a file context associated with it. This file context will be included in the prompt sent to an LLM. To limit the size of the prompt, files are divided into "spans," where a span could be, for example, a section of code (e.g., imports), a class, or a function. These are identified by span IDs. Thus, the LLM sees a limited part of the code at a time but can request more context by searching for or adding files and spans. The file context therefore changes over time, and a specific state of file context is linked to a specific state. In the standard implementation of moatless-tools, changes to the codebase are made linearly, and each change is saved directly to the file system. In Moatless-Adapted, however, there is a need to be able to revert to earlier states and thus return to a previous version of the codebase. To handle this, the code is stored in a git repository where each change is committed, and each state has a reference to a commit as well as the current patch of the diff from the initial commit that existed before starting. This way, one can go back to an earlier state by specifying the state ID, and the commit that was current at that time will be checked out.

The test files present in the file context are run each time the Plan state is initiated, and the test results are provided to the state. The tests are then run in Docker images built via the SWE-bench library. To use this approach in a benchmark where a larger number of instances should be able to run simultaneously, a solution is used where these images are run as pods in a Kubernetes cluster. Moatless-tools communicates with the testbed by applying patches and running commands via an API. When a new instance starts, a pod is created which is then reset at each run, applying the current patch and running tests according to the test command specified in the SWE-bench library. It's important to add here that the agent is not aware of the `PASS_TO_PASS` or `FAIL_TO_PASS` tests in the SWE-bench harness, but only knows how to run the tests. This corresponds to a real engineering environment where each project can have its own test commands.

C MCTS HYPERPARAMETERS

The Monte Carlo Tree Search (MCTS) algorithm used in this study employs several hyperparameters.

Table 2: MCTS Hyperparameters

Hyperparameter	**Description**	**Default**
c_param	UCT exploration parameter	1.41
max_expansions	Max children per node	5
max_iterations	Max MCTS iterations	100
provide_feedback	Enable feedback	True
best_first	Use best-first strategy	True
value_function_temperature	Value function temperature	0.2
max_depth	Max tree depth	20
UCT Score Calculation Parameters		
exploration_weight	UCT exploration weight	1.0
depth_weight	Depth penalty weight	0.8
depth_bonus_factor	Depth bonus factor	200.0
high_value_threshold	High-value node threshold	55.0
low_value_threshold	Low-value node threshold	50.0
very_high_value_threshold	Very high-value threshold	75.0
high_value_leaf_bonus_constant	High-value leaf bonus	20.0
high_value_bad_children_bonus_constant	High-value bad children bonus	20.0
high_value_child_penalty_constant	High-value child penalty	5.0
Action Model Parameters		
action_model_temperature	Action model temperature	0.2
Discriminator Parameters		
number_of_agents	Number of Discriminator Agents	5
number_of_round	Number of debate rounds	3
discriminator_temperature	Discriminator temperature	1.0

These hyperparameters can be adjusted to fine-tune the MCTS algorithm's performance for specific problem domains or computational constraints. The values listed here are the defaults as defined in the `TreeSearchSettings` class and the MCTS implementation.

D ABILITY OF MCTS TO ESCAPE UNPRODUCTIVE LOOPS VS. BASELINE

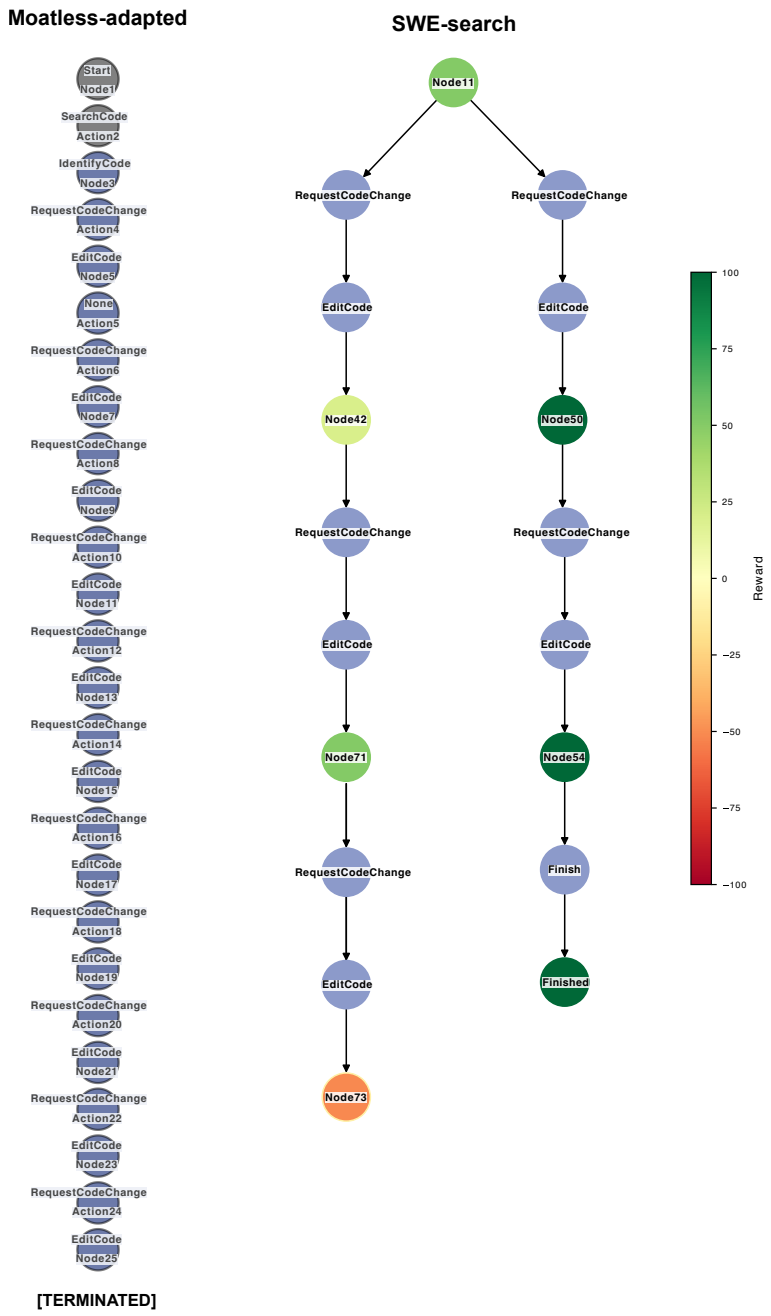

Figure 6: **Avoiding Repetitive Actions, django__django__10914.** We found that the base agent can often get stuck performing repetitive actions that do not bring it closer to solving the issue, and which commonly lead to unresolvable dead-ends. In this example, the base agent was stuck implementing wrong tests which continuously returned errors. In contrast, when this happens in SWE-Search, the Value Agent recognizes this, terminating these trajectories quickly, as happens in Node 73 (orange).

E Model Instance Resolution Uniqueness

To understand the complementary strengths of different models in resolving software issues, we analyzed how unique their resolved issue subsets where. Figure 7 illustrates the resolution patterns for each model across five of the codebases in SWE-bench-lite.

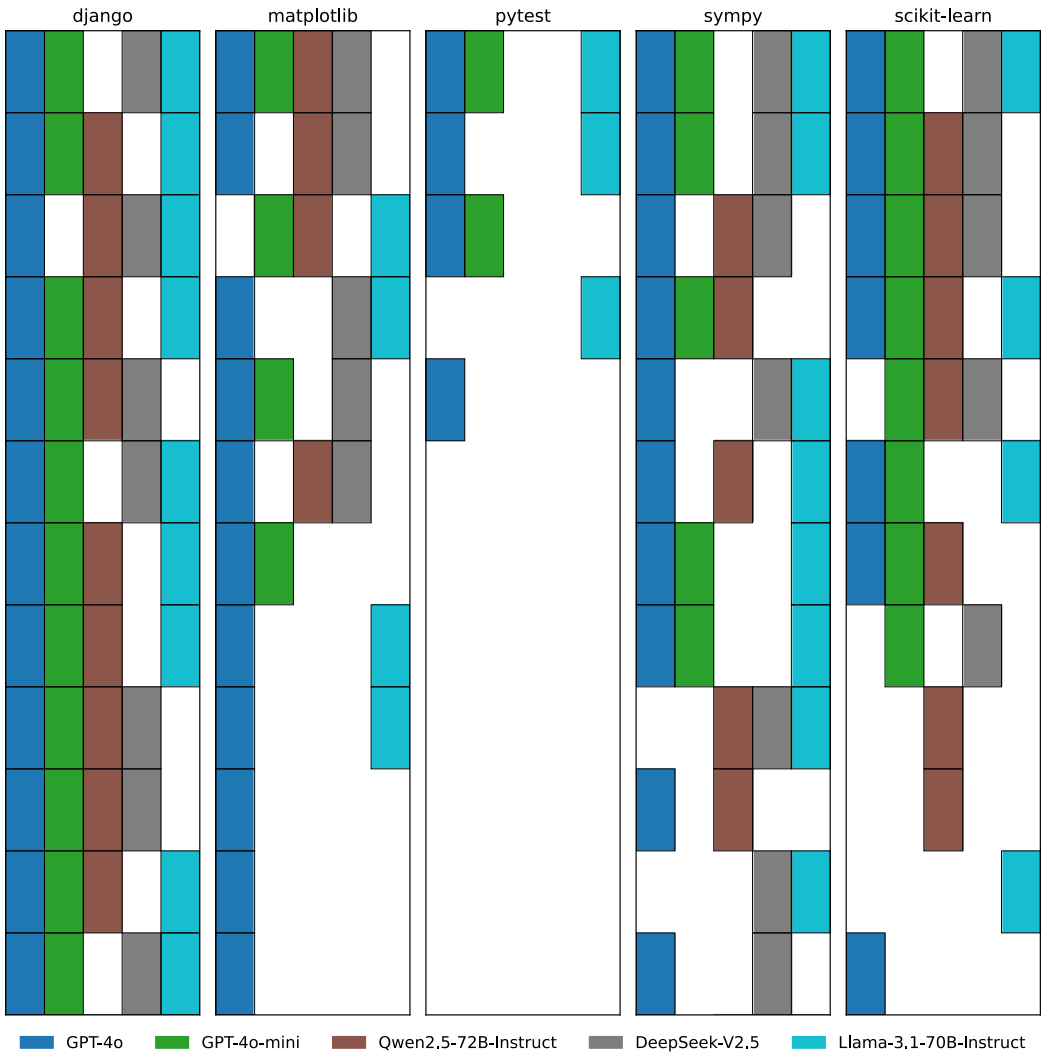

Figure 7: **Unique Issue Resolution Patterns Across Models and Libraries.** Each column represents a different Python repository, and each row within a column represents a specific issue. Colored blocks indicate successful resolution by the corresponding model (see legend). White spaces denote unresolved issues. This visualization highlights the diverse problem-solving capabilities of different models across various software domains, demonstrating that no single model dominates across all issues and libraries.

F ABILITY OF VALUE FUNCTION TO DISCERN SUCCESSFUL TRAJECTORIES

Before implementing SWE-Search, we conducted a general study across many models to evaluate the models' ability to differentiate states which led to resolved vs. unresolved issues. Figure 8 shows the results of this study. We found that in general, models assigned higher rewards to states which eventually led to resolved issues. Of particular interest was the Deepseek model, which seemed to identify critical errors in trajectories effectively. This was also observed in the final agent (see Fig. 5a).

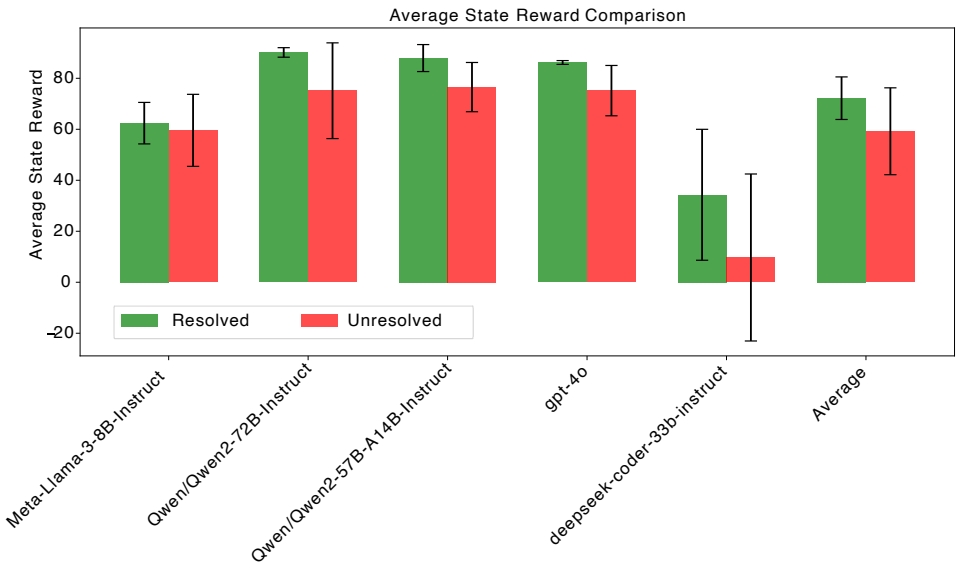

Figure 8: **Average State Reward Comparison Across Models.** This graph compares the average state rewards assigned by different language models for resolved (green) and unresolved (red) issues. Error bars indicate standard deviation. Most models consistently assign higher rewards to states leading to resolved issues, with the exception of the. The 'Average' column represents the mean across all models, demonstrating a clear distinction between resolved and unresolved states.

G VALUE FUNCTION PROMPTS

Value Function Search Prompt

Your task is to evaluate a search action executed by an AI agent, considering the search parameters, the resulting file context, and the identified code from the search results. Your evaluation will focus on whether the search action was well-constructed, whether the resulting file context is relevant and useful for solving the problem at hand, and whether the identified code is appropriate and helpful.

You will be provided with four inputs:

- **Problem Statement**: This will be provided within the `<problem_statement>` XML tag and contains the initial message or problem description the coding agent is trying to solve.
- **The Search Request**: This will be provided within the `<search_request>` XML tag and contains the search parameters used by the agent to define the search.
- **The Search Result**: The content retrieved based on the search parameters provided within a `<search_results>` XML tag.
- **Identified Code**: The specific code identified from the search results, provided within the `<identified_code>` XML tag.

Search request parameters:

- **File Pattern** (file_pattern): Glob patterns (e.g., **/*.py) to filter search results to specific files or directories.
- **Query** (query): A natural language query for semantic search.
- **Code Snippet** (code_snippet): Specific code snippets for exact matching.
- **Class Names** (class_names): Specific class names to include in the search.
- **Function Names** (function_names): Specific function names to include in the search.

Evaluation Criteria:

Search Parameters:

- Are they appropriately defined to focus the search on relevant files or code?
- Do they align well with the problem statement?

Resulting File Context:

- Does it contain relevant and useful information for solving the problem?
- Are there missing or irrelevant results indicating a need to refine the search?

Identified Code Review (most crucial):

- Is the identified code directly related to the problem?
- Does it provide the necessary functionality to address the issue?

Overall Relevance:

- Does the combination of search parameters, file context, and identified code effectively address the problem?
- Could there be a better approach or improvements?

Reward Scale and Guidelines:

Assign a single integer value between -100 and 100 based on how well the search action, resulting file context, and identified code addressed the task at hand. Use the following scale:

100:

- Search Parameters: Precisely match the problem needs; no irrelevant or missing elements.
- Identified Code: Completely and accurately solves the problem with no issues.

75 to 99:

- Search Parameters: Well-defined and mostly relevant; minor improvements possible.
- Identified Code: Effectively addresses the problem with minor issues that are easily fixable.

0 to 74:

- Search Parameters: Partially relevant; noticeable inaccuracies or omissions.
- Identified Code: Partially solves the problem but has significant gaps or errors.

-1 to -49:

- Search Parameters: Misaligned with the problem; poorly defined.
- Identified Code: Fails to address the problem effectively; may cause confusion.

-50 to -100:

- Search Parameters: Irrelevant or incorrect; hinders problem-solving.
- Identified Code: Unrelated to the problem; provides no useful information.

Output Format:

Please ensure your output strictly adheres to the following structure:
`<Explanation>` [A brief explanation of the evaluation in max one paragraph.]
`<Reward>` [A single integer reward value between -100 and 100]

Value Function Plan Prompt

Your role is to evaluate the executed action of the search tree that our AI agents are traversing, to help us determine the best trajectory to solve a programming issue. The agent is responsible for identifying and modifying the correct file(s) in response to the problem statement.

Input Data Format:

- **Problem Statement**: This will be provided within the `<problem_statement>` XML tag and contains the initial message or problem description the coding agent is trying to solve.
- **File Context**: The relevant code context will be provided within the `<file_context>` XML tag and pertains to the state the agent is operating on.
- **History**: The sequence of state transitions and actions taken prior to the current state will be contained within the `<history>` XML tag. This will include information on the parts of the codebase that were changed, the resulting diff, test results, and any reasoning or planned steps.
- **Executed Action**: The last executed action of the coding agent will be provided within the `<executed_action>` XML tag, this includes the proposed changes and the resulting diff of the change.
- **Full Git Diff**: The full Git diff up to the current state will be provided within the `<full_git_diff>` XML tag. This shows all changes made from the initial state to the current one and should be considered in your evaluation to ensure the modifications align with the overall solution.
- **Test Results**: The results of any test cases run on the modified code will be provided within the `<test_results>` XML tag. This will include information about passed, failed, or skipped tests, which should be carefully evaluated to confirm the correctness of the changes.

Evaluation Criteria:

Code Correctness: Evaluate whether the implemented code correctly addresses the problem. This includes verifying that the correct lines or sections of code have been identified and modified appropriately. Ensure that the changes are both syntactically and logically correct, and that the diffs accurately represent the intended modifications without introducing unrelated changes. Assess whether the modifications effectively solve the problem without introducing new issues or inefficiencies.

Mistakes in Editing Code: Identify any errors made during the code editing process. This involves checking for unintended deletions, incorrect modifications, or syntax errors introduced through the changes. Ensure that the Git diffs maintain integrity by only including the intended modifications and no accidental alterations to unrelated parts of the codebase.

Testing: Assess the proposed changes against existing test cases. Determine if the changes pass all relevant tests and evaluate whether any test failures could have been reasonably foreseen and avoided by the agent. Consider whether the agent anticipated potential test outcomes and addressed them proactively in the solution.

History and Action Evaluation: Review the agent's previous state transitions and actions

to determine if the current action contributes positively to solving the problem. Pay special attention to detect if the agent is engaging in repetitive actions without making meaningful progress. Evaluate whether the last executed action is appropriate and logical given the current progress and history of actions.

Reward Scale and Guidelines:

The reward value must be based on how confident you are that the agent's solution is the most optimal one possible with no unresolved issues or pending tasks. The scale ranges from -100 to 100, where:

100: You are fully confident that the proposed solution is the most optimal possible, has

been thoroughly tested, and requires no further changes.

75-99: The approach is likely the best one possible, but there are minor issues or opportu-

nities for optimization. All major functionality is correct, but some small improvements or additional testing may be needed. There might be some edge cases that are not covered.

0-74: The solution has been partially implemented or is incomplete or there are likely alter-

native approaches that might be better, i.e., this is likely not the most optimal approach. The core problem might be addressed, but there are significant issues with tests, logical flow, or side effects that need attention. There are likely alternative approaches that are much better.

0: The solution is not yet functional or is missing key elements. The agent's assertion that

the task is finished is incorrect, and substantial work is still required to fully resolve the issue. Modifying the wrong code, unintentionally removing or altering existing code, introducing syntax errors, or producing incorrect diffs fall into this range.

-1 to -49: The proposed solution introduces new issues or regresses existing functionality,

but some elements of the solution show potential or may be salvageable. Repetitive actions without progress fall into this range.

-50 to -100: The solution is entirely incorrect, causing significant new problems, or fails

to address the original issue entirely. Immediate and comprehensive changes are necessary. Persistent repetitive actions without progress should be heavily penalized.

Output Format:

Please ensure your output strictly adheres to the following structure:
`<Explanation>` [Your brief explanation of the evaluation in max one paragraph.]
`<Reward>` [A single integer reward value between -100 and 100]

Value Function Request More Context Prompt

Your role is to evaluate the executed action of the search tree that our AI agents are traversing, specifically for the RequestMoreContext action. This action is used when the agent requests to see code that is not in the current context, potentially revealing an understanding that relevant code is wholly or partially not visible, and enabling the agent to uncover important missing information.

Evaluation Criteria:

- **Relevance**: Are the requested files and code spans likely to be relevant to the problem at hand?
- **Necessity**: Is the additional context truly needed, or is the agent unnecessarily expanding the scope?
- **Specificity**: Has the agent been specific in its request, or is it asking for overly broad sections of code?
- **Contextual Understanding**: Does the request demonstrate a good understanding of the codebase structure and the problem domain?
- **Efficiency**: Is the agent making targeted requests, or is it asking for too much unnecessary information?
- **Progress**: Does this request seem likely to move the problem-solving process forward?

Input Data Format:

- **Problem Statement**: Provided within the `<problem_statement>` XML tag, containing the initial problem description.
- **File Context**: The current code context within the `<file_context>` XML tag.
- **History**: Previous state transitions and actions within the `<history>` XML tag.
- **Executed Action**: The RequestMoreContext action details within the `<executed_action>` XML tag, including the files and code spans requested.

Reward Scale and Guidelines: Assign a single integer value between -100 and 100 based on how well the RequestMoreContext action addresses the task at hand:

100: Perfect request that is highly likely to provide crucial missing information.

75-99: Good request with minor improvements possible in specificity or relevance.

0-74: Partially relevant request, but with noticeable inaccuracies or potential for better targeting.

-1 to -49: Poor request that is likely to provide mostly irrelevant information or expand the scope unnecessarily.

-50 to -100: Very poor request that is entirely irrelevant or demonstrates a fundamental

misunderstanding of the problem or codebase structure.

Output Format: Please ensure your output strictly adheres to the following structure:

`<Explanation>` [Your explanation of the evaluation in max two paragraphs.]
`<Reward>` [A single integer reward value between -100 and 100]

Value Function Edit Prompt

Your role is to evaluate the executed action of the search tree that our AI agents are traversing, with the goal of ensuring that a complete and verified solution is in place. The agent believes that it has finished solving the programming issue.

Evaluation Criteria

Solution Correctness and Quality: Verify that the proposed changes logically address the problem statement. Ensure the changes fit contextually within the existing codebase without introducing new issues. Confirm syntactic correctness and that there are no syntax errors or typos. Assess whether the solution represents an overall improvement and is the most optimal approach possible.

Accuracy of Code Modifications: Check that the agent correctly identified the appropriate code spans to modify. Ensure the changes made are accurate and do not include unintended modifications. Look for any alterations to unrelated parts of the code that could introduce new problems.

Testing and Test Results Analysis:

- **Importance of Test Updates:** It is crucial that the agent updated existing tests or added new tests to verify the solution. Failure to do so should be heavily penalized. The agent should ensure that code changes are validated by appropriate tests to confirm correctness and prevent regressions.
- **Assess Test Coverage:** Evaluate whether the agent has adequately tested the solution, including adding new tests for new functionality or changes. Verify that the tests cover relevant cases and edge conditions.
- **Penalization for Lack of Testing:** When calculating the reward, heavily penalize the agent if they failed to update or add necessary tests to verify the solution.

Consideration of Alternative Approaches: Always assess whether there could be a better alternative approach to the problem. Mention any potential alternative solutions in your explanation if they are applicable.

Identification and Explanation of Mistakes: If the agent made incorrect actions, identify exactly where and why the mistakes occurred. Explain the impact of any syntax errors, incorrect code modifications, or unintended changes.

Assessment of Agent's Completion Assertion: Verify if the agent's assertion that the task is finished is accurate. Determine if substantial work is still required to fully resolve the issue and address this in your evaluation.

Input Data Format:

- **Problem Statement**: This will be provided within the `<problem_statement>` XML tag and contains the initial message or problem description the coding agent is trying to solve.
- **File Context**: The relevant code context will be provided within the `<file_context>` XML tag and pertains to the state the agent is operating on.
- **History**: The sequence of state transitions and actions taken prior to the current state will be contained within the `<history>` XML tag. This will include information on the parts of the codebase that were changed, the resulting diff, test results, and any reasoning or planned steps.
- **Reasoning for Completion**: The reasoning provided by the agent for why the task is finished will be provided within the `<reasoning_for_completion>` XML tag. This includes the agent's explanation of why no further changes or actions are necessary.
- **Full Git Diff**: The full Git diff up to the current state will be provided within the `<full_git_diff>` XML tag. This shows all changes made from the initial state to the current one and should be considered in your evaluation to ensure the modifications align with the overall solution.
- **Test Results**: The results of any test cases run on the modified code will be provided within the `<test_results>` XML tag. This will include information about passed, failed, or skipped tests, which should be carefully evaluated to confirm the correctness of the changes.

Reward Scale and Guidelines:

The reward value must be based on how confident you are that the agent's solution is the most optimal one possible with no unresolved issues or pending tasks. It is important that the agent updated or added new tests to verify the solution; failure to do so should be heavily penalized. The scale ranges from -100 to 100, where:

100: You are fully confident that the proposed solution is the most optimal possible, has been thoroughly tested (including updated or new tests), and requires no further changes.

75-99: The approach is likely the best one possible, but there are minor issues or opportunities for optimization. All major functionality is correct, but some small improvements or additional testing may be needed. There might be some edge cases that are not covered.

0-74: The solution has been partially implemented or is incomplete, or there are likely alternative approaches that might be better. The core problem might be addressed, but there are significant issues with tests (especially if the agent did not update or add new tests), logical flow, or side effects that need attention.

0: The solution is not yet functional or is missing key elements. The agent's assertion that

the task is finished is incorrect, and substantial work is still required to fully resolve the issue.

-1 to -49: The proposed solution introduces new issues or regresses existing functionality,

but some elements show potential or may be salvageable. Modifying the wrong code, unintentionally removing or altering existing code, introducing syntax errors, producing incorrect diffs, or failing to update or add necessary tests fall into this range.

-50 to -100: The solution is entirely incorrect, causing significant new problems or failing

to address the original issue entirely. Immediate and comprehensive changes are necessary. Persistent repetitive actions without progress, or failure to update or add tests when necessary, should be heavily penalized.

Output Format: Please ensure your output strictly adheres to the following structure:

`<Explanation>` [Your explanation of the evaluation in max two paragraphs.]
`<Reward>` [A single integer reward value between -100 and 100]

H MOATLESS TOOLS STATE RIGIDITY

The Moatless-tools version (v0.0.2) enforces a rigid transition structure where agents must follow a specific sequence (search → identify → plan → edit). The implementation of this state transition system can be found here: [2].

H.1 STATE TRANSITION SYSTEM

The transition system is configured through a function that accepts three optional parameters:

- `max_tokens_in_edit_prompt`: Controls the token limit for edit operations
- `global_params`: Defines parameters applicable across all states
- `state_params`: Specifies state-specific parameters

H.2 STATE FLOW

The system defines a directed graph of states with specific transition rules:

1. **Search Phase** (`SearchCode`):
   - Initial state for code operations
   - Can transition to `IdentifyCode` upon successful search
   - Can move directly to `PlanToCode` when complete
2. **Identification Phase** (`IdentifyCode`):
   - Processes search results
   - Can return to `SearchCode` if needed
   - Progresses to `DecideRelevance` when finished
3. **Decision Phase** (`DecideRelevance`):
   - Evaluates identified information
   - Can trigger new searches

[2] https://github.com/aorwall/moatless-tools/blob/8ec5d5193b6dce88ec6273c7ec31f9ea3a0bba6f/moatless/transitions.py#L184

- Transitions to planning when ready, excluding message field

This rigid structure ensures that tools are accessed in a predictable sequence, preventing conflicts while maintaining system integrity. Additional transitions defined in `CODE_TRANSITIONS` complete the state machine's behavior set.

I Cost Analysis

Table 3 presents the API costs for Moatless-Adapted and SWE-Search across different models. Search-based exploration of multiple solutions results in higher computational costs.

Model	Moatless-Adapted	SWE-Search
GPT-4o	$40.86	$576.00
GPT-4o-mini	$9.90	$52.34
Qwen-2.5-72b-Instruct*	$8.50	$42.50
DeepseekCoderV2.5	$3.66	$18.37
Llama-3.1-70b-Instruct*	$9.00	$45.00

*Estimated costs based on comparable API pricing

Table 3: Cost comparison (USD) between Moatless-Adapted and SWE-Search

J Compute-Matching Analysis

Table 4 compares SWE-Search against compute-matched baselines. SWE-Search Pass@5 uses the 5 generated answers in 1 run, while for Moatless-Adapted uses the 5 generated solutions across 5 runs. We avoid doing the comparison on GPT-4o to avoid exorbitant API costs.

Model	SWE-Search		Moatless-Adapted
	Pass@1	Pass@5	Pass@5
GPT-4o	31.0	34.0	-
GPT-4o-mini	17.0	22.3	17.0
Qwen-2.5-72b-Instruct	24.7	25.7	22.3
DeepseekCoderV2.5	21.0	23.3	22.0
Llama-3.1-70b-Instruct	17.7	22.3	21.7

Table 4: Performance comparison (%) between SWE-Search and compute-matched baselines

K Interactive Demo

To help visualize the search process and provide transparency into our method, we provide an interactive demo at `http://74.241.196.91`. The demo presents a tree visualization where each node represents a state/action pair in the search process. Clicking on a node reveals detailed information including:

- Complete LLM interactions and tool calls
- State-specific value function outputs and reasoning
- Context information used for decision-making
- File changes and test results where applicable
- Test creation/execution and their outputs

This interface allows readers to explore how the search algorithm navigates through different states, makes decisions, and evaluates potential solutions. The visualization particularly highlights how state-specific value functions guide the exploration process and how the discriminator compares candidate solutions.