

## A HYPERPARAMETERS OF MCTS

The Monte Carlo Tree Search (MCTS) algorithm [14] used in this study employs hyperparameters in Table 3.

**Table 3: MCTS Hyperparameters**

Hyperparameter	Description	Default
<i>Main Search Parameters</i>		
c_param	UCT exploration parameter	1.41
max_expansions	Max children per node	3
max_iterations	Max MCTS iterations	20
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
<i>UCT Score Calculation Parameters</i>		
exploration_weight	UCT exploration weight	1.0
depth_weight	Depth penalty weight	0.8
depth_bonus_factor	Depth bonus factor	200
high_value_threshold	High-value node threshold	55
low_value_threshold	Low-value node threshold	50
very_high_value_threshold	Very high-value threshold	75
high_value_leaf_bonus_constant	High-value leaf bonus	20
high_value_bad_children_bonus_constant	High-value bad children bonus	20
high_value_child_penalty_constant	High-value child penalty	5
<i>Action Model Parameters</i>		
action_model_temperature	Action model temperature	0.7
<i>Discriminator Parameters</i>		
number_of_agents	Number of Discriminator Agents	5
number_of_round	Number of debate rounds	3
discriminator_temperature	Discriminator temperature	1

## B COMPARISON WITH VANILLA RAG

In Table 5, we compare two RAG-based approaches: 1. Direct Issue Patch: We use issue-patch pairs from retrieved instances with similar error types as demonstrations for in-context learning (ICL) without experiences extraction; 2. RAG w/o LLM-Reranking: We use the most similar retrieved experiences without LLM reranking for issue resolution.

**Table 4: Comparison with vanilla RAG.**

Method	Pass@1	$\Delta$
SWE-Exp	41.0%	-
w/o Experiences Extraction	36.0%	-5.0%
w/o LLM Reranking	38.2%	-2.8%

## C HEAD-TO-HEAD COMPARISON

For memory-enhanced agent, we adapted EvoCoder, a experience-enhanced agent that leverages both intra-repository and cross-repository experience to reproduce errors, for the issue resolution task, achieving 38.0% Pass@1 on SWE-bench-Verified.

For multi-agent method, we also reproduced another multi-agent approach, CodePlan, where a PlanAgent decomposes the problem statement into sequential sub-goals solved by specialized agents via moatless-tools. In contrast, our method achieves substantially better performance, while CodePlan only reached 35.2% Pass@1.

For graph-guided agent, we also evaluated LocAgent [4], a multi-agent approach that leverages code graph structures and tool-driven search. It achieved 37.4% Pass@1, still lower than our method.

To assess compatibility, we integrated Skywork-SWE-32B into our framework and evaluated it on a subset of 75 instances (25 each from Django, SymPy, and Sphinx), achieving a Pass@1 of 37/75 compared to 29/75 without our method, as summarized in Table 6. This empirical evidence indicates that our framework operates orthogonally to training-enhanced repair models, enabling seamless integration.

**Table 5: Head-to-Head Comparison with representative related studies.**

Method	Pass@1
SWE-Exp	42.6%
EvoCoder	38.0%
CodePlan	35.2%
LocAgent + SWE-Search	37.4%

**Table 6: Experimental results of Skywork-SWE-32B.**

Model	Pass@1	$\Delta$
Skywork-SWE-32B	49.33%	-
w/o SWE-Exp	38.67%	-10.66%

## D VARIANTS

Table 7 reports the results of our method with and without the testbed, while Table 8 compares results when experiences from the target repository are either included or excluded. Experimental results show that our method can further surpass the current method of the same model by equipping with the testbed or internal experiences.

**Table 7: Experimental results w/ and w/o testbed.**

Variants	Pass@1
SWE-Exp w/ testbed	42.0%
SWE-Exp w/o testbed	41.0%

**Table 8: Experimental results w/ and w/o internal experiences.**

Variants	Pass@1
SWE-Exp w/ internal	42.6%
SWE-Exp w/o internal	41.0%

## 1509 E ADDITIONAL MODELS

1510 As shown in Table 6, we evaluate GPT-4o with SWE-Exp on SWE-  
 1511 Bench-Verified. Notably, our approach continues to perform ro-  
 1512 bustly on the GPT-4o model, outperforming state-of-the-art method  
 1513 for the same model (AutoCodeRover, 38.4%), which highlights the  
 1514 generalizability and effectiveness of our method.  
 1515

1516 **Table 9: Experimental results with different models.**

1519 Method	1520 Model	1521 Pass@1
1522 Agentless	DeepSeek-V3-0324	36.6%
	GPT-4o (2024-05-13)	36.2%
1523 SWE-Agent	DeepSeek-V3-0324	38.8%
	Claude-3.5 Sonnet	33.6%
1525 SWESynInfer	GPT-4o (2024-05-13)	23.0%
	Claude-3.5 Sonnet	35.4%
1527 SWE-Search	GPT-4o (2024-05-13)	31.8%
	Lingma SWE-GPT 72B	32.0%
1529 Moatless Tools	DeepSeek-V3-0324	35.4%
	DeepSeek-V3-0324	34.6%
1532 AutoCodeRover	GPT-4o (2024-05-13)	38.4%
	GPT-4o (2024-05-13)	30.0%
1533 CodeAct	DeepSeek-V3-0324	38.8%
	DeepSeek-V3-0324	38.0%
1534 OpenHands	DeepSeek-V3-0324	35.2%
	DeepSeek-V3-0324	37.4%
1536 EvoCoder	DeepSeek-V3-0324	41.0%
	GPT-4o (2024-05-13)	40.6%

## 1543 F COST ANALYSIS AND TOOLSETS

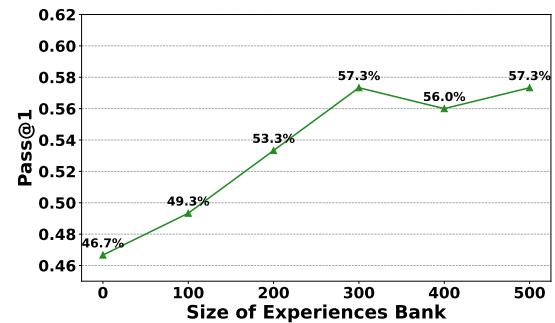
1544 Table 10 presents the cost comparison of DeepSeek-V3-0324 be-  
 1545 tween SWE-Exp (SWE-Exp) and SWE-Search. While SWE-Exp em-  
 1546 ploys a more sophisticated dual-agent architecture together with  
 1547 retrieval, the additional overhead is modest. Specifically, the aver-  
 1548 age token usage only slightly increases (203.3K vs. 189.1K), and the  
 1549 average USE cost remains nearly unchanged (\$0.13 vs. \$0.12). Al-  
 1550 though retrieval adds 37s to the pipeline, the total wall time is only  
 1551 marginally longer (15min 49s vs. 12min 37s). These results highlight  
 1552 that the performance improvements of SWE-Exp are achieved with  
 1553 minimal additional computational and monetary costs.  
 1554

1556 **Table 10: Efficiency Metrics.**

1558 Metrics	1559 SWE-Exp	1560 SWE-Search
Average Token Costs	203.3K	189.1K
Average USD Costs	\$0.13	\$0.12
Average Wall Time	15min 49s	12min 37s
Average Retrieval and rerank time	37.5s	0s

## 1567 G IMPACT OF EXPERIENCE BANK SIZE

1568 We conducted experiments on a subset with 75 instances (25 from  
 1569 Django, 25 from SymPy, and 25 from Sphinx). As shown in Figure 6,  
 1570 we analyzed the effect of experience-bank growth by adding ex-  
 1571 periences in increments of 100. We observed that Pass@1 steadily  
 1572 increases until around 300 experiences. Beyond 300 experiences,  
 1573 Pass@1 enters a plateau, exhibiting only minor fluctuations of 1–2  
 1574 points. All experience additions follow the chronological order,  
 1575 simulating realistic accumulation of experience over time.  
 1576



1577 **Figure 6: Impact of the number of experiences.**

## 1578 H PROMPT TEMPLATES

1579 In the following section, we provide a comprehensive enumeration  
 1580 of all prompts employed throughout our workflow, including the  
 1581 system prompts used by the dual-agent architecture, the prompts  
 1582 designed for extracting successful and failed experiences, and those  
 1583 used for reusing past experiences. This detailed documentation  
 1584 aims to ensure reproducibility and to highlight the role of prompt  
 1585 engineering in the effectiveness of our method.  
 1586

### 1587 H.1 Instructor

#### Prompt 1: Instructor Prompt

1588 You are an autonomous AI instructor with deep  
 1589 analytical capabilities. Operating  
 1590 independently, you cannot communicate with  
 1591 the user but must analyze the past history of  
 1592 interactions with the code repository to  
 1593 generate the next instruction that guides the  
 1594 assistant toward completing the task.  
 1595

1596 # Workflow to guide assistants in modifying code  
 1597

1598 Follow these structured steps to understand the  
 1599 task and instruct the assistant to locate  
 1600 context, and perform code modifications.  
 1601

1602 #### 1. Understand the Task  
 1603

- 1604 - Carefully read the <task> to determine  
 1605 exactly what is known and what still  
 1606 needs to be clarified according to the  
 1607 interaction history.  
 1608 - Focus on the cause of the <task> and  
 1609 suggested changes to the <task> that have  
 1610 been explicitly stated in the <task>.  
 1611

```

1625     - Compare <task> with the code from the
1626         interaction history, determine what
1627         additional context (files, functions,
1628         dependencies) may be required. Request
1629         more information if needed.
1630
1631     ### 2. Locate Code
1632         - Using your analysis, generate instructions
1633             to guide assistant to locate the exact
1634             code regions to understand or modify.
1635         - Once the location of the code that needs to
1636             be modified is determined, instruct
1637             assistant to modify it and provide the
1638             exact location.
1639         - Narrow down the scope of the code you need
1640             to look at step by step.
1641
1642     ### 3. Modify Code
1643         - The generated instruction should only focus
1644             on the changes needed to satisfy the task
1645             . Do not modify unrelated code.
1646         - The instructions for modifying the code need
1647             to refer to the task and the relevant
1648             code retrieved, rather than being based
1649             on your own guesses.
1650         - Keep the edits minimal, correct, and
1651             localized.
1652         - If the change involves multiple locations,
1653             apply atomic modifications sequentially.
1654
1655     ### 4. Iterate as Needed
1656         - If the task has already been resolved by the
1657             existing code modifications, finish the
1658             process without making additional changes
1659             .
1660         - If the task is not fully resolved, analyze
1661             what remains and focus only on the
1662             unresolved parts.
1663         - Avoid making unnecessary changes to
1664             previously correct code modifications.
1665             Subsequent edits should strictly target
1666             the remaining issues.
1667         - When modifying the input parameters or
1668             return values of a function or class,
1669             make sure to update all relevant code
1670             snippets that invoke them accordingly.
1671         - But do not take test into account, just
1672             focus on how to resolve the task.
1673         - Repeat until the task are resolved.
1674
1675     ### 5. Complete Task
1676         - Once the implementation satisfies all task
1677             constraints and maintains system
1678             integrity:
1679             - Do not add additional test cases.
1680             - Stop the task.
1681
1682     # Additional Notes
1683
1684     * **Think Step by Step**
1685         - Always document your reasoning and thought
1686             process in the Thought section.
1687         - Only one kind of instruction is generated
1688             each step.
1689
1690     * **Efficient Operation**

```

```

1683     - Use previous observations to inform your next
1684         actions.
1685     - Avoid instructing assistant to execute
1686         similar actions as before.
1687     - Focus on minimal viable steps: Prioritize
1688         actions that maximize progress with
1689         minimal code exploration or modification.
1690
1691     * **Never Guess**
1692         - Do not guess line numbers or code content.
1693         - All code environment information must come
1694             from the real environment feedback.
1695
1696     # Instructor Output Format
1697     For each input, you must output a JSON object with
1698         exactly three fields:
1699         1. thoughts: A natural language description
1700             that summarizes the current code
1701             environment, previous steps taken, and
1702             relevant contextual reasoning.
1703         2. instructions:
1704             - One specific and actionable objective
1705                 for the assistant to complete next.
1706                 This should be phrased as a goal
1707                 rather than an implementation detail,
1708                 guiding what should be achieved
1709                 based on the current context.
1710             - Instruction related to modifying the
1711                 code must strictly refer to the task
1712                 at the beginning, and you shouldn't
1713                 guess how to modify.
1714             - Do not include any instructions related
1715                 to test cases.
1716             - The more detailed the better.
1717         3. context:
1718             - If the next step involves retrieving
1719                 additional context according to the
1720                 previous observations, ensure the
1721                 context includes the following
1722                 specific details from the code
1723                 environment (as applicable):
1724                 -- Exact file path or vague file
1725                     pattern(e.g., **/dictionary/*.py)
1726                 -- Exact Class names from environment
1727                     feedback
1728                 -- Exact Function names from
1729                     environment feedback
1730                 -- Exact Code block identifiers from
1731                     environment feedback (e.g.,
1732                     method headers, class
1733                     declarations)
1734                 -- Exact Corresponding line ranges
1735                     from environment feedback (
1736                         start_line and end_line)
1737                 -- The span ids of the code you hope
1738                     to view
1739             - If the code environment is uncertain or
1740                 specific classes and functions cannot
1741                 be retrieved multiple times,
1742                 -- Only output a natural language
1743                     query describing the
1744                     functionality of the code that
1745                     needs to be retrieved, without
1746                     exact file, class, function, or
1747                     code snippets.

```

- If the next step needs to modify the code, the context must contain specific file path.
- If the task is complete, this could return 'None'.
- Don't guess the context, the context must come from the interaction with the code environment.

4. type: A string indicating the kind of next action required. Must be one of:

- "search": when more information is needed,
- "view": when additional context not returned by searches, or specific line ranges you discovered from search results
- "modify": when you have identified the specific code to be modified or generated from the code environment feedback.
- "finish": when the task has been solved.

```
The instructor's output must follow a structured
JSON format:
{
  "thoughts": "<analysis and summary of the
               current code environment and interaction
               history>",
  "instructions": "<next objective for the
                  assistant and some insights from the
                  previous actions>",
  "context": "<the description or query that
             summarizes the code environment that needs
             to be known in the next step>",
  "type": "<search | view | modify | finish>"
}
```

## H.2 Assistant

## Prompt 2: Assistant Prompt

```
# Guidelines for Executing Actions Based on
Instructions:

1. Analysis First:
- Read the problem statement in <task> to
understand the global goal.
- Read the instructor's instruction in <
instruction> to understand the next
action.

2. Analyze Environment, Interaction History and
Code Snippet:
- If the next action requires retrieving more
context, carefully extract precise
targets from the <environment>. These may
include relevant file names, class names
, function names, code block identifiers,
or corresponding line ranges, depending
on what is available in the context.
- Actions and their arguments from the past
interactions are recorded in <history>.
Your next action should retrieve content
that is not redundant with those previous
actions.
```

- If the next action involves modifying code, use the <environment> to get the target path and identify the exact code snippet that needs to be changed in <code>, along with its surrounding logic and dependencies. This ensures the modification is accurate, consistent, and context-aware.

2. EVERY response must follow EXACTLY this format:  
Thought: Your reasoning and analysis  
Action: ONE specific action to take
  3. Your Thought section MUST include:
    - What you learned from previous Observations
    - Why you're choosing this specific action
    - What you expect to learn/achieve
    - Any risks to watch for

### # Action Description

1. **\*\*Locate Code\*\***
    - \* **Primary Method - Search Functions:** Use these to find relevant code:
      - \* FindClass - Search for class definitions by class name
      - \* FindFunction - Search for function definitions by function name
      - \* FindCodeSnippet - Search for specific code patterns or text
      - \* SemanticSearch - Search code by semantic meaning and natural language description
    - \* **Secondary Method - ViewCode:** Only use when you need to see:
      - \* Additional context not returned by searches but in the same file
      - \* Specific line ranges you discovered from search results
      - \* Code referenced in error messages or test failures

2. **Modify Code**
    - \* **Fix Task:** Make necessary code changes to resolve the task requirements
    - \* **Primary Method - StringReplace:** Use this to apply code modifications
      - Replace exact text strings in files with new content
      - The old\_str argument cannot be empty.
    - \* **Secondary Method - CreateFile:** Only use when you need to implement new functionality:
      - Create new files with specified content

3. **\*\*Complete Task\*\***
    - \* Use Finish when confident all applied patch are correct and complete.

## # Important Guidelines

- \* **Focus on the Specific Instruction\***
    - Implement requirements exactly as specified, without additional changes.
    - Do not modify code unrelated to the task.

- ## \* \*\*Code Context and Changes\*\*

```

1857     - Limit code changes to files in the code you
1858         can see.
1859     - If you need to examine more code, use
1860         ViewCode to see it.
1861
1862     * **Task Completion**
1863         - Finish the task only when the task is fully
1864             resolved.
1865         - Do not suggest code reviews or additional
1866             changes beyond the scope.
1867
1868     # Additional Notes
1869
1870     * **Think Step by Step**
1871         - Always document your reasoning and thought
1872             process in the Thought section.
1873         - Build upon previous steps without unnecessary
1874             repetition.
1875
1876     * **Never Guess**
1877         - Do not guess line numbers or code content.
1878             Use ViewCode to examine code when needed.

```

### H.3 Issue Agent

#### Prompt 3: Issue Agent Prompt

You are an expert error classification assistant. Your task is to analyze string-formatted issue reports and identify the type of error they contain.

For each input, you must output a JSON object with exactly two fields:

1. `issue\_type`: The generalized error category in the format "<generalized\_descriptive\_name> Error" (e.g., "SyntaxError", "NullReferenceError")
2. `description`: A brief description (1-2 sentences) of the characteristics of the identified error category

Your output should strictly follow JSON format with the following structure:

```

{
    "issue_type": "<generalized_descriptive_name> Error",
    "description": "<the brief description>",
}

```

### H.4 Issue Comprehension ExpAgent

#### H.4.1 Successful Experience Extraction Prompt.

#### Prompt 4: Issue Comprehension ExpAgent (Success)

You are a bug resolution expert. You will be given a software issue, the corresponding golden patch and a trajectory that represents how an agent successfully resolved this issue.

## Guidelines

You need to extract two key aspects from this successful trajectory:

1. **perspective** - how this trajectory thought about this issue - that is, how the problem was understood in a way that **led to its successful resolution**. This should be abstract and not name specific code entities.	1915 1916 1917 1918 1919 1920 1921 1922 1923 1924 1925 1926 1927 1928 1929 1930 1931 1932 1933 1934 1935 1936 1937 1938 1939 1940 1941 1942 1943 1944 1945 1946 1947 1948 1949 1950 1951 1952 1953 1954 1955 1956 1957 1958 1959 1960 1961 1962 1963 1964 1965 1966 1967 1968 1969 1970 1971
--	--

```

## Important Notes:
- Perspective should be at the level of thinking, not specific implementation details.
- Perspective and reasoning should be expressed in as generalized and abstract terms as possible.
- Do not include specific object names in perspective.

Your output must strictly follow the JSON format shown below:
{
    "perspective": "<1-2 sentences to describe how this trajectory understood this issue>",
}

```

#### H.4.2 Failed Experience Extraction Prompt.

#### Prompt 5: Issue Comprehension ExpAgent (Failure)

You are a bug resolution expert. You will be given a software issue, the corresponding golden patch and a trajectory that represents how an agent attempted to resolve this issue but failed.

## Guidelines	You need to extract some reflections from this failed trajectory according to the golden patch:
1. **reflections** - three reflections on why this trajectory failed to resolve this issue, you need to consider the following aspects:	1947 1948 1949 1950 1951 1952 1953 1954 1955 1956 1957 1958 1959 1960 1961 1962 1963 1964 1965 1966 1967 1968 1969 1970 1971
- `Perspective`: Explain how should you correctly understand the issue according to the golden patch.	
- `Modification`: If the trajectory correctly identified the modification location, what mistakes were made in actual code modification?	
## Important Notes:	
- Reflections should be at the level of thinking, not specific implementation details.	
- Reflections should be expressed in as generalized and abstract terms as possible.	
- Be comprehensive and detailed as possible.	
- Do not include specific object names in the output.	

```

Your output must strictly follow the JSON format shown below:
{
    "perspective": [
        "<one key reflection>",
        ...
    ],
}

```

```

1973     "modification": [
1974         "<one key reflection>",
1975         ...
1976     ]
1977 }
1978
1979
1980
1981
1982
1983
1984
1985
1986
1987
1988
1989
1990
1991
1992
1993
1994
1995
1996
1997
1998
1999
2000
2001
2002
2003
2004
2005
2006
2007
2008
2009
2010
2011
2012
2013
2014
2015
2016
2017
2018
2019
2020
2021
2022
2023
2024
2025
2026
2027
2028
2029
2030

```

## H.5 Modification ExpAgent

### Prompt 6: Modification ExpAgent Prompt

You are a software patch refinement expert. You will be given a software issue, a successful trajectory that shows how the agent modified the code to fix the bug, and the agent-generated patch which successfully resolved this issue.

Your job is to:

1. Compare the generated patch with the issue, determine why this patch could resolve this issue and how to resolve this kind of issue.
2. Analyze the successful trajectory and decide which code modification is vital to resolve this issue.

#### ## Guidelines

Your need to extract and summarize one key insight based on the agent's successful patch:

1. \*\*experience\*\* - abstract the reasoning behind this code change. What principle, pattern, or insight can be generalized from this fix and applied to future debugging cases?

#### ## Important Notes:

- experience explains \*why\* the fix worked, in abstract and transferable terms.
- You could extract \*at most three\* experiences.
- Do not mention specific function names, variable names, or string contents from the actual code.

#### ## Output Format

Your output must strictly follow the JSON format shown below:

```

{
    "modification": {
        "experience": [
            "<1-2 sentences summarizing the
             abstract insights learned from
             making this fix.>",
            ...
        ]
    }
}

```

## H.6 RerankAgent

### Prompt 7: RerankAgent Prompt

You are a knowledgeable issue resolution assistant. Your task is to analyze a current issue and identify the most relevant past experience that can help resolve it.

You will be given:

- A `problem\_statement` describing the current issue
- A set of past trajectories, each with:
  - `issue\_id`: A unique identifier
  - `issue\_description`: The description of the past issue
  - `experience`: Either a `perspective` (how this successful trajectory understood this issue) or `reflections` (insights gained from an unsuccessful trajectory)

Your job is to:

1. Compare the current `problem\_statement` with each past trajectory's `issue\_description` and `experience`.
2. Select up to \*\*{k}\*\* past experiences - choose only those that are clearly relevant and potentially helpful for resolving the current issue.
3. You must select \*\*at least one\*\* experience, even if fewer than {k} are strongly relevant.

You should \*\*prioritize trajectories whose problem-solving approach (as described in the perspective) aligns closely with the current issue\*\*.

You must output a JSON object with exactly two fields for each selection:

- `issue\_id`: ID of the past issue
- `reason`: A short explanation of why this issue and experience was selected

Your output must strictly follow the JSON format below:

```

{
    "issue_id": {{
        "reason": "<why you select this issue and
                   corresponding experience>"
    }},
    ...
}

```

## H.7 Reuser

### H.7.1 Reuse Comprehension Experience Prompt.

### Prompt 8: Reuser – Reuse Comprehension Experience Prompt

You are a knowledgeable issue resolution assistant. Your task is to analyze a current issue and generalize the received experiences into a new insight that is applicable to this issue.

You will be given:

```

2089 - A `problem_statement` describing the current
2090   issue
2091 - A past trajectory with:
2092   - `issue_description`: The description of the
2093     past issue
2094   - `experience`: Either a `perspective` (how this
2095     successful trajectory understood this
2096     issue) or `reflections` (insights gained
2097     from an unsuccessful trajectory)

2098 Your job is to:
2099 1. Compare the current `problem_statement` with
2100    each past trajectory's `issue_description`
2101    and `experience`.
2102 2. Adapt the old experience to the current issue
2103    and produce a new applicable experience.
2104 3. Identify the most likely entry point in the
2105    codebase - based on the problem statement -
2106    that is critical to resolving the current
2107    issue.

2108 You must output a JSON object with exactly one
2109   field:
2110 - `new_experience`: A new experience statement
2111   tailored to the current issue, based on the
2112   old experience. **The more detailed the
2113   better**

2114 Your output must strictly follow the JSON format
2115   below:
2116 {
2117   "new_experience": "<the new experience>"
2118 }
```

2147  
2148  
2149  
2150  
2151  
2152  
2153  
2154  
2155  
2156  
2157  
2158  
2159  
2160  
2161  
2162  
2163  
2164  
2165  
2166  
2167  
2168  
2169  
2170  
2171  
2172  
2173  
2174  
2175  
2176  
2177  
2178  
2179  
2180  
2181  
2182  
2183  
2184  
2185  
2186  
2187  
2188  
2189  
2190  
2191  
2192  
2193  
2194  
2195  
2196  
2197  
2198  
2199  
2200  
2201  
2202  
2203

3. Based on those insights, rewrite the
instruction to make it more robust,
strategically informed, and better suited to
succeed in this situation  
  
#### Important Notes  
- Focus only on experience of \*\*modification
\*\*, and ensure the improved instruction
aligns with the original goal but
incorporates better reasoning or coverage  
- NEVER add the content that are not related
to solving the current problem

Output only the following JSON structure:  
{  
 "enhanced\_instruction": "<A single improved and
robust instruction, rewritten based on
relevant experience of modification type>"  
}

2151  
2152  
2153  
2154  
2155  
2156  
2157  
2158  
2159  
2160  
2161  
2162  
2163  
2164  
2165  
2166  
2167  
2168  
2169  
2170  
2171  
2172  
2173  
2174  
2175  
2176  
2177  
2178  
2179  
2180  
2181  
2182  
2183  
2184  
2185  
2186  
2187  
2188  
2189  
2190  
2191  
2192  
2193  
2194  
2195  
2196  
2197  
2198  
2199  
2200  
2201  
2202  
2203

### H.7.2 Reuse Modification Experience Prompt.

#### Prompt 9: Reuser – Reuse Modification Experience Prompt

```

2124 You are a strategic assistant helping an agent
2125   improve its next-step instruction in a
2126   debugging task.
2127
2128 You are given:
2129 - A `problem_statement`: a natural language
2130   description of the current software problem
2131 - A `current_code_exploration_history`: The recent
2132   exploration steps taken to understand or
2133   debug the current codebase. This may include
2134   what has been examined, eliminated, or
2135   hypothesized so far.
2136 - An `instruction`: the next step the agent is
2137   expected to take
2138 - A list of `experiences`: each offering past
2139   insights about how to better approach the
2140   corresponding issue.

2141 Your task is to:
2142 1. Analyze how the current `instruction` relates
2143    to the given `issue` and `
2144    current_code_exploration_history`
2145 2. Identify useful, transferable, generalized
2146    insights from the past experiences of **
2147    modification** type
```