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
Main Search Parameters		
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
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
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
Action Model Parameters		
action_model_temperature	Action model temperature	0.7
Discriminator Parameters		
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	Δ
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	Δ
Skywork-SWE-32B	**38.67%**	-
w/ SWE-Exp	**49.33%**	**+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%
SWE-Search w/ testbed	37.0%
SWE-Search w/o testbed	35.4%

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%

E RESULTS ON DIFFERENT MODELS

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

Table 9: Experimental results with different models.

Method	Model	Pass@1
Agentless	DeepSeek-V3-0324	36.6%
	GPT-4o (2024-05-13)	36.2%
SWE-Agent	DeepSeek-V3-0324	38.8%
	Claude-3.5 Sonnet	33.6%
	GPT-4o (2024-05-13)	23.0%
SWESynInfer	Claude-3.5 Sonnet	35.4%
	GPT-4o (2024-05-13)	31.8%
	Lingma SWE-GPT 72B	32.0%
SWE-Search	DeepSeek-V3-0324	35.4%
Moatless Tools	DeepSeek-V3-0324	34.6%
AutoCodeRover	GPT-4o (2024-05-13)	38.4%
CodeAct	GPT-4o (2024-05-13)	30.0%
OpenHands	DeepSeek-V3-0324	38.8%
EvoCoder	DeepSeek-V3-0324	38.0%
CodePlan	DeepSeek-V3-0324	35.2%
LocAgent + SWE-Search	DeepSeek-V3-0324	37.4%
SWE-Exp	DeepSeek-V3-0324	**41.0%**
	GPT-4o (2024-05-13)	**40.6%**

F COST ANALYSIS AND TOOLSETS

Table 10 presents the cost comparison of DeepSeek-V3-0324 between SWE-Exp (SWE-Exp) and SWE-Search. While SWE-Exp employs a more sophisticated dual-agent architecture together with retrieval, the additional overhead is modest. Specifically, the average token usage only slightly increases (203.3K vs. 189.1K), and the average USE cost remains nearly unchanged ($0.13 vs. $0.12). Although retrieval adds 37s to the pipeline, the total wall time is only marginally longer (15min 49s vs. 12min 37s). These results highlight that the performance improvements of SWE-Exp are achieved with minimal additional computational and monetary costs.

Table 10: Efficiency Metrics.

Metrics	SWE-Exp	SWE-Search
Average Token Costs	203.3K	189.1K
Average USD Costs	$0.13	$0.12
Average Wall Time	15min49s	12min37s
Average Retrieval and rerank time	37.5s	-

G IMPACT OF EXPERIENCE BANK SIZE

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

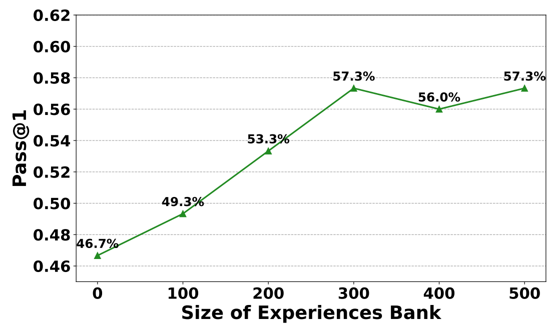

Figure 6: Impact of the number of experiences.

H PROMPT TEMPLATES

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

H.1 Instructor

Prompt 1: Instructor Prompt

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

# Workflow to guide assistants in modifying code

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

### 1. Understand the Task
    - Carefully read the <task> to determine
         exactly what is known and what still
         needs to be clarified according to the
         interaction history.
    - Focus on the cause of the <task> and
         suggested changes to the <task> that have
          been explicitly stated in the <task>.
```

```
    - Compare <task> with the code from the
        interaction history, determine what
        additional context (files, functions,
        dependencies) may be required. Request
        more information if needed.

### 2. Locate Code
    - Using your analysis, generate instructions
        to guide assistant to locate the exact
        code regions to understand or modify.
    - Once the location of the code that needs to
        be modified is determined, instruct
        assistant to modify it and provide the
        exact location.
    - Narrow down the scope of the code you need
        to look at step by step.

### 3. Modify Code
    - The generated instruction should only focus
        on the changes needed to satisfy the task
        . Do not modify unrelated code.
    - The instructions for modifying the code need
         to refer to the task and the relevant
        code retrieved, rather than being based
        on your own guesses.
    - Keep the edits minimal, correct, and
        localized.
    - If the change involves multiple locations,
        apply atomic modifications sequentially.

### 4. Iterate as Needed
    - If the task has already been resolved by the
         existing code modifications, finish the
        process without making additional changes
        .
    - If the task is not fully resolved, analyze
        what remains and focus only on the
        unresolved parts.
    - Avoid making unnecessary changes to
        previously correct code modifications.
        Subsequent edits should strictly target
        the remaining issues.
    - When modifying the input parameters or
        return values of a function or class,
        make sure to update all relevant code
        snippets that invoke them accordingly.
    - But do not take test into account, just
        focus on how to resolve the task.
    - Repeat until the task are resolved.

### 5. Complete Task
    - Once the implementation satisfies all task
        constraints and maintains system
        integrity:
      - Do not add additional test cases.
      - Stop the task.

# Additional Notes

 * **Think Step by Step**
   - Always document your reasoning and thought
       process in the Thought section.
   - Only one kind of instruction is generated
       each step.

 * **Efficient Operation**
```

```
   - Use previous observations to inform your next
        actions.
   - Avoid instructing assistant to execute
       similar actions as before.
   - Focus on minimal viable steps: Prioritize
       actions that maximize progress with
       minimal code exploration or modification.

 * **Never Guess**
   - Do not guess line numbers or code content.
   - All code environment information must come
       from the real environment feedback.

# Instructor Output Format
For each input, you must output a JSON object with
      exactly three fields:
    1. thoughts: A natural language description
        that summarizes the current code
        environment, previous steps taken, and
        relevant contextual reasoning.
    2. instructions:
        - One specific and actionable objective
            for the assistant to complete next.
            This should be phrased as a goal
            rather than an implementation detail,
             guiding what should be achieved
            based on the current context.
        - Instruction related to modifying the
            code must strictly refer to the task
            at the beginning, and you shouldn't
            guess how to modify.
        - Do not include any instructions related
            to test cases.
        - The more detailed the better.
    3. context:
        - If the next step involves retrieving
            additional context according to the
            previous observations, ensure the
            context includes the following
            specific details from the code
            environment (as applicable):
            -- Exact file path or vague file
                pattern(e.g., **/dictionary/*.py)
            -- Exact Class names from environment
                feedback
            -- Exact Function names from
                environment feedback
            -- Exact Code block identifiers from
                environment feedback (e.g.,
                method headers, class
                declarations)
            -- Exact Corresponding line ranges
                from environment feedback (
                start_line and end_line)
            -- The span ids of the code you hope
                to view
        - If the code environment is uncertain or
            specific classes and functions cannot
             be retrieved multiple times,
            -- Only output a natural language
                query describing the
                functionality of the code that
                needs to be retrieved, without
                exact file, class, function, or
                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 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**
```

```
   - Limit code changes to files in the code you
       can see.
   - If you need to examine more code, use
       ViewCode to see it.

 * **Task Completion**
   - Finish the task only when the task is fully
       resolved.
   - Do not suggest code reviews or additional
       changes beyond the scope.

# Additional Notes

 * **Think Step by Step**
   - Always document your reasoning and thought
       process in the Thought section.
   - Build upon previous steps without unnecessary
        repetition.

 * **Never Guess**
   - Do not guess line numbers or code content.
       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.

## 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:
    - `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>",
        ...
        ],
```

```
    "modification": [
        "<one key reflection>",
        ...
        ]
}
```

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:
```

```
- A `problem_statement` describing the current
     issue
- A past trajectory with:
  - `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. Adapt the old experience to the current issue
     and produce a new applicable experience.
3. Identify the most likely entry point in the
     codebase - based on the problem statement -
     that is critical to resolving the current
     issue.

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

Your output must strictly follow the JSON format
     below:
{
    "new_experience": "<the new experience>"
}
```

H.7.2 *Reuse Modification Experience Prompt.*

Prompt 9: Reuser – Reuse Modification Experience Prompt

```
You are a strategic assistant helping an agent
     improve its next-step instruction in a
     debugging task.

You are given:
- A `problem_statement`: a natural language
     description of the current software problem
- A `current_code_exploration_history`: The recent
      exploration steps taken to understand or
     debug the current codebase. This may include
     what has been examined, eliminated, or
     hypothesized so far.
- An `instruction`: the next step the agent is
     expected to take
- A list of `experiences`: each offering past
     insights about how to better approach the
     corresponding issue.

Your task is to:
1. Analyze how the current `instruction` relates
     to the given `issue` and `
     current_code_exploration_history`
2. Identify useful, transferable, generalized
     insights from the past experiences of **
     modification** type
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>"
}
```