

DISCLOSURE ON THE USE OF LLMs

During the final drafting stages of this paper, we consulted Large Language Models (LLMs) to improve the manuscript’s clarity and linguistic precision. The LLM served as an advanced editing tool, providing suggestions on syntax, word choice, and overall readability for the author-written text. We emphasize that this was an iterative process where the authors directed the tool and made all final decisions regarding the text. No part of the paper’s core scientific arguments, methodology, or results was generated by the LLM. The authors bear full responsibility for all content and claims presented herein.

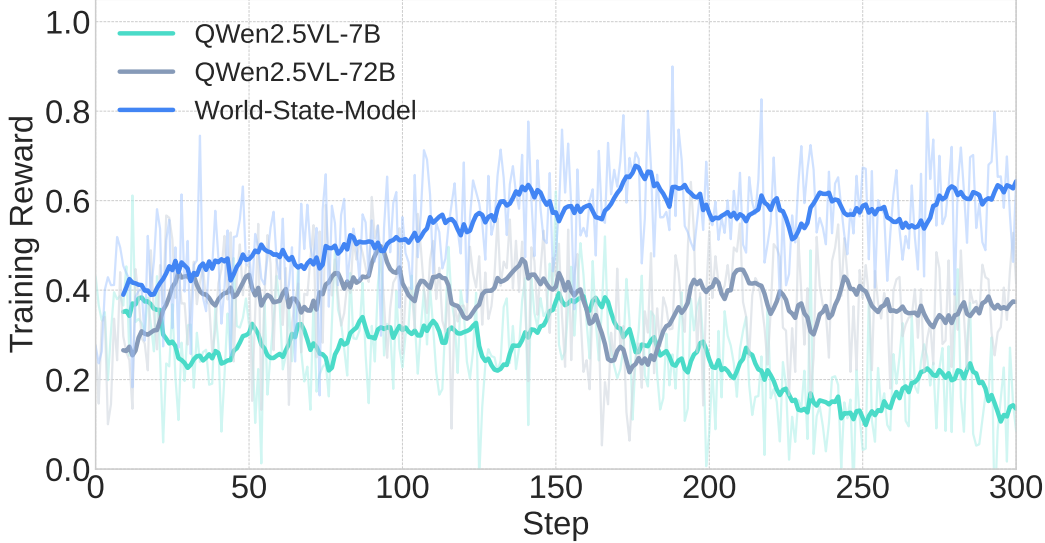


Figure 5: **Training reward with different reward signal provider.** Our World State Model provide reward signal that can achieve improved training reward compared to strong base models.

A WORLD STATE MODEL

The World State Model (WSM) is a central component of SEAgent, responsible for understanding visual state changes and evaluating the effectiveness of the agent’s actions.

A.1 MODEL ARCHITECTURE AND OPERATION

The WSM is built upon the Qwen2.5-VL-7B vision-language model. It operates in two distinct modes, each with a specific input-output structure to perform different tasks:

1. Trajectory Judgment:

Input: A sequence of screenshot images captured during an episode.

Output: Short captions for each screenshot, the reasoning process for the judgment, and a structured judgment dictionary (containing fields such as `Correctness`, `Redundant`, and `First Error Step`, as detailed in Fig. 8 of the supplementary material).

2. State Change Description:

Input: Two screenshot images, one from before and one after a single action was executed.

Output: A detailed description of the visual differences between the two images.

A.2 FINE-TUNING DATASET AND PROCESS

To equip the WSM with these capabilities, a specialized dataset was constructed for fine-tuning.

Data Construction The data construction process is as follows:

1. **Trajectory Sampling:** A Computer Using Agent (CUA), powered by UI-TARS and Gemini-2.5-Pro, was used to sample trajectories from 43 feasible tasks in Google Chrome within the OSWorld benchmark. These trajectories were saved as screenshot sequences.
2. **GPT-4o Annotation:** Using the prompts detailed in Figures 6 and 7 of the supplementary material, GPT-4o was employed to annotate the sampled trajectories, generating judgments and screenshot captions. Only samples where the judgment matched the ground truth from OSWorld evaluation protocols were retained, resulting in 860 high-quality annotated trajectories.
3. **Change Description Data:** An additional 1,000 pairs of (before action, after action) screenshots were sampled. GPT-4o was used to generate detailed descriptions of the differences, creating a 1,000-sample Change Description (CD) dataset.

Fine-Tuning Process The fine-tuning was performed using the Llama-Factory framework on 8 NVIDIA A100 (80G) GPUs for 2,000 iterations. A learning rate of 2×10^{-5} was used, and LoRA (rank=128) was employed for parameter-efficient fine-tuning. The 860 annotated trajectories serve as the core training data for teaching the model trajectory judgment, captioning, and reasoning. The 1,000-sample CD dataset acts as auxiliary data, specifically to encourage the model to focus on fine-grained visual differences, which enhances its overall state understanding. As shown in Table 1 of the main paper, incorporating CD data significantly boosts judgment performance. The two datasets were combined for training without any special re-weighting.

A.3 REWARD GENERATION FROM TRAJECTORY ANALYSIS

The trajectory judgment capability of the WSM is the core source of the reward signal for reinforcement learning. After an agent executes a full trajectory $\mathcal{H} = \{s_0, a_0, s_1, a_1, \dots, s_{\text{final}}\}$, the WSM analyzes it and outputs a structured judgment. Based on this output, actions within the trajectory are dynamically labeled as either positive actions (a_T) or failure actions (a_F):

- **Fully Successful Trajectory:** If `Correctness` is ‘True’ and there are no Redundant steps, all actions a in the trajectory are labeled as a_T .
- **Successful but Inefficient Trajectory:** If `Correctness` is ‘True’ but Redundant steps begin at step k , all actions prior to step k are labeled as a_T .
- **Failed Trajectory:** If `Correctness` is ‘False’ and the First Error Step is e , all actions prior to step e are labeled as a_T , while the erroneous action a_e is labeled as a_F .

These dynamically labeled a_T and a_F actions constitute the reward signals for the RL pipeline. During training, the actor predicts an action a_t based on the history $\{a_0, s_0, \dots, s_t\}$ and uses these labels to calculate rewards.

B CURRICULUM GENERATOR

The Curriculum Generator is designed to dynamically produce tasks of increasing difficulty and diversity, guiding the agent through a systematic exploration of the software’s capabilities.

B.1 TASK GENERATION MECHANISM

The workflow of the Curriculum Generator is detailed in the pseudocode in our supplementary material. Its core idea is to leverage the WSM’s analysis of completed tasks to generate new ones. The process, illustrated by the “add a rectangle” example from Figure 5, involves three main steps:

1. **Analysis and Feedback:** The agent successfully completes an initial task, “add a rectangle.” The WSM analyzes the execution trajectory and extracts two key pieces of information: a task evaluation (`Exam`) and a list of observed state changes (`CD_list`).
`CD_list`: {“add a rectangle”: [“The Edit bar is expanded...”, “The cursor has changed into a cross...”, “A blue box appears on the screen with side bars showing

properties such as fill, line, color, width, transparency, and corner style..."], ...}
 Exam: [{"task": "add a rectangle", "status": "success"}], ...]

2. **Knowledge Integration and Task Generation:** The `CD_list` and `Exam` are fed into the Curriculum Generator. It distills new knowledge, such as "properties of a rectangle," and integrates it into its internal `Software guidebook`. Based on this new knowledge, it generates more challenging tasks like "Add a green rectangle" or "Add a red rectangle with 50% transparency," which are then added to the task buffer.
3. **Iterative Learning:** In the next RL phase, the agent samples from this updated, more challenging task buffer. The continuously enriched `Software guidebook` acts as the system's long-term memory, driving the Curriculum Generator to propose increasingly sophisticated and unexplored tasks in subsequent rounds, thereby guiding the agent toward mastery.

C DETAILS OF CURRICULUM GENERATOR.

C.1 EXEMPLAR CASE DURING TASK EVOLUTION.

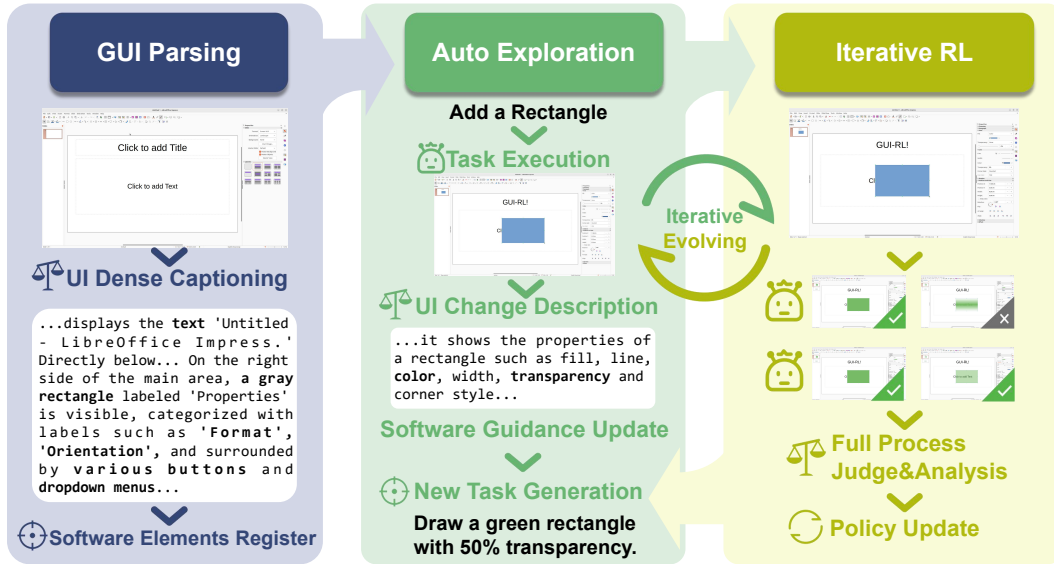


Figure 6: **SEAgent autonomous exploration pipeline.** The agent (policy model) and World State Model iteratively generate new task and perform RL to become a specialist in novel software.

We provide an exemplar case of our task evolution pipeline in Fig. 6, demonstrated using LibreOffice Impress. Initially, the World State Model parses a screenshot of the Impress interface into detailed captions describing the layout and individual buttons. The Task Generator then produces an initial task set, $\mathcal{I}_0 = \{I_0^{(1)}, I_0^{(2)}, \dots\}$, and summarizes the initial software guidance memory U_0 . The initial agent executes tasks in \mathcal{I}_0 , such as "Add a Rectangle," while the World State Model evaluates these actions, providing judgments and detailed descriptions of resulting changes. As shown in the Auto-Exploration stage, this includes generating captions for newly appeared property panels and assessing execution success. The Task Generator incorporates feedback on execution success and newly revealed properties (e.g., transparency) to evolve new tasks, such as "Draw a green rectangle with 50% transparency." This process iteratively improves through reinforcement learning, enabling continuous task evolution and agent self-improvement.

C.2 COMPARATIVE ANALYSIS OF INSTRUCTION GENERATION STRATEGIES.

To validate the effectiveness of our Curriculum Generator, we conducted a comparative analysis against state-of-the-art instruction generation methods, namely those from NNetNav (Murty et al., 2025) and WebRL (Qi et al., 2024).

Experimental Setup We adapted the official code and prompts from these prior works from web environments to general software applications. To ensure a fair comparison of the curriculum quality, for each strategy, we employed two leading LLMs: the open-source Qwen2.5-72B (Bai et al., 2025) and the proprietary Gemini-2.5-Pro (Google DeepMind, 2025). The tasks generated by each strategy were used to train an RL agent (using GRPO only), with reward signals uniformly provided by our fine-tuned WSM. The evaluation was performed on two applications: VSCode from OSWorld (a standard software) and Celestia from ScienceBoard (?) (a more challenging, out-of-domain scientific application). The primary metric was the task success rate.

Table 5: Success rate (%) comparison of different task generation strategies on two software applications.

Task Generation Strategy	LLM	VSCode	Celestia
WebRL	Qwen2.5-72B	27.5	0.00
WebRL	Gemini2.5-Pro-thinking	36.2	3.03
NNetNav	Qwen2.5-72B	34.6	0.00
NNetNav	Gemini2.5-Pro-thinking	43.6	5.05
Curriculum Generator (Ours)	Qwen2.5-72B	37.7	9.09
Curriculum Generator (Ours)	Gemini2.5-Pro-thinking	42.3	12.12

Results and Discussion The results are presented in Table 5. As shown, the reverse instruction generation strategy from NNetNav (Murty et al., 2025) is highly effective on the in-domain application (VSCode), demonstrating high data generation efficiency by producing successful trajectories. However, a critical trade-off was observed: this approach tends to generate many similar tasks, limiting its ability to explore the full breadth of the software’s functionalities. This limitation becomes more pronounced when the task generator is unfamiliar with the target software, as seen in the OOD Celestia environment.

In contrast, our guidebook-based method, while having a lower initial data generation efficiency, excels at systematic exploration. It builds structured knowledge of the software from scratch, making it more robust for tackling novel applications. This is evidenced by its superior performance on the more challenging Celestia software.

We conclude that these two strategies are complementary. Reverse instruction generation can efficiently exploit known functionalities, while our guidebook-based method can systematically explore new ones and help the task generator build a more comprehensive understanding of the target software. A hybrid approach combining both strategies is a promising direction for future work.

D EXPERIMENTS ON ANDROIDWORLD

Table 6: Success Rate on AndroidWorld (Rawles et al., 2024)

Model	AndroidWorld_SR
Qwen2.5-VL-7B	8.0
Qwen2.5-VL-7B+SEAgent	19.5
UI-TARS-7B-SFT	33.0
UI-TARS-7B-SFT+SEAgent	38.0

To evaluate SEAgent’s application to other format of GUI, we conduct new experiments on the AndroidWorld (Rawles et al., 2024) benchmark, which focuses on mobile GUIs. We apply our SEAgent pipeline to two distinct backbone models. As shown in the table below, our method yields substantial performance improvements for both, demonstrating that its self-evolving approach is effective across different model architectures and GUI formats. Specifically, SEAgent improves the success rate of Qwen2.5-VL by +11.5% and UI-TARS by +5.0%. This result strongly indicate the effectiveness of our pipeline also generalize to other form of GUI.

E SENSITIVITY ANALYSIS ON KEY HYPERPARAMETERS

We conducted a sensitivity analysis on key hyperparameters to evaluate their impact on the SEAgent pipeline. For model sampling, we set the temperature $t = 0$ for better reproducibility. We analyze two specific parameters: the number of generated tasks and the number of change descriptions. The results are presented in Table 7 and discussed below.

Table 7: Sensitivity analysis for key hyperparameters in the SEAgent pipeline, evaluated on VSCode. The metric is Success Rate (%).

# Tasks Generated	VScode SR	# Change Descriptions	VScode SR
30	31.88	30	33.33
50	36.23	50	37.68
100	37.68	100	37.68
200	37.68	200	34.78

Number of Generated Tasks This parameter controls the breadth of exploration in each learning cycle. As shown in our analysis, performance improves as more diverse tasks are generated, eventually plateauing around 100 tasks.

Number of Change Descriptions This parameter controls how much new information the generator receives to update its "software guidebook." We found a clear trade-off: A sufficient number of descriptions (50–100) is essential for the generator to learn about new UI functionalities and create meaningful, unexplored tasks. However, providing too many descriptions (e.g., 200) creates an overly long context for the LLM, which degrades the quality of task generation and hurts final performance.

F ABLATION ON THE LOSS BALANCE FACTOR.

In Sec.3.2, we use γ to balance the ratio of two loss item: adversarial imitation that learn from error and GRPO that learn to achieve success. We ablate the choice of γ in Tab.8, according to which we set $\gamma = 0.2$ in main experiments.

γ	0.0	0.1	0.2	0.3	0.5	0.8
Success Rate (%)	34.8	36.2	37.7	31.9	26.1	23.1

Table 8: VScode Success Rate on OSWorld (Xie et al., 2024) under different loss balance factor γ values.

G REWARD FUNCTION FOR DIFFERENT ACTIONS.

Action Type	Description	Distance-based Reward
click, left_single, right_single, hover	Click or hover on a location	Normalized L1 distance between predicted and ground-truth coordinates
left_double, double_click	Double click on a region	Normalized L1 distance between clicked coordinates
drag, select	Drag from start box to end box	Intersection over Union (IoU) between predicted and ground-truth boxes
type	Type textual input	Character-level BLEU score between predicted and ground-truth text
hotkey	Press multiple keys at once	Character-level BLEU score between predicted and ground-truth key combinations
press	Press a single key	Character-level BLEU score between predicted and ground-truth key
scroll	Scroll in a certain direction	Character-level BLEU score between predicted and ground-truth direction
move_mouse	Move mouse to a specific location	Normalized L1 distance between predicted and ground-truth coordinates
highlight	Highlight a rectangular UI region	IoU between predicted and ground-truth region
copy, paste	Clipboard operations	BLEU score between copied/pasted content
wait	Explicit wait command	Fixed reward + 1
finished, finish.task	Finish current task/trajectory	Fixed reward + 1

Table 9: Reward computation for each action type in GUI agent

	Phase0	Phase1	Phase2	Phase3
VSCode	112/39	282/83	161/34	98/55
GIMP	104/51	309/90	183/50	95/52
Impress	102/44	290/92	185/61	87/51
VLC	85/29	114/41	160/48	53/27
Writer	123/62	278/101	201/69	101/43

Table 10: Number of episode (Success/Failure) across four phases for different software tools during self-evolution. Each episode contains 8.8 multi-turn conversions in average.

H DATA STATISTICS DURING ITERATIVE REINFORCEMENT LEARNING.

I DETAILED PROMPT TEMPLATES.

For evaluation on AgentRewardBench (Lù et al., 2025), we use their official template for final state screenshot only testing and modified prompt in Fig.7 for entire process (or sampled middle screenshots) testing.

For evaluation on OSWorld Sampled trajectories, we use prompt in Fig.8 to prompt GPT-4o to provide step level judges, the sampled judges on Chrome in OSWorld (Xie et al., 2024) serves as training data of GUI-Judge. This template is also used in training GUI-Judge and at inference time in autonomous exploration stage.

For navigator, we use prompt template in Fig.9, which takes previous software usage manual and the performance of actor agent evaluated by judge (Empty if in initial phase.) as well as detailed exploration caption as input and output the updated usage manual as well as new task for agent to execute.

J SELF DOCUMENTED USAGE MANUAL ON DIFFERENT SOFTWARE DURING EXPLORATION.

In Fig.10 Fig.12, Fig.11, Fig.13, we demonstrate the self-documented usage manuals of the navigator (Qwen2.5-72B (Yang et al., 2024)) in the exploration and learning system introduced in Sec.3.1.

K BROADER IMPACTS

Potential positive societal impacts: SEAgent introduces a self-evolving paradigm for Computer Use Agents (CUAs), enabling them to autonomously learn and adapt to previously unseen software without human supervision. This significantly reduces the need for extensive manual data annotation and domain-specific customization, allowing intelligent agents to assist users across a wide range of applications—including productivity tools, multimedia editing, and educational software. By automating repetitive tasks and providing guidance in complex software environments, SEAgent holds promise for improving accessibility, enhancing digital literacy, and reducing cognitive workload in both professional and everyday settings.

Potential negative societal impacts: The capability of SEAgent to autonomously explore and operate complex software also introduces risks of misuse. Malicious actors might repurpose SEAgent for unauthorized software automation, such as automating account creation, spamming interfaces, or conducting surveillance via GUI interactions. In addition, as the agent learns from its own experience, there exists a risk that the agent may inadvertently inherit or amplify software-specific biases, potentially leading to unfair or inappropriate behaviors in sensitive applications (e.g., finance, legal automation). Mitigation strategies include controlled release of models, behavior filters during deployment, and incorporating safeguards in the World State Model to detect and prevent unintended or adversarial behavior.

Algorithm 1 SEAgent Specialized Self-Evolution Training Loop

```

1: Input: Initial policy  $\pi_0$ , World State Model  $\mathcal{M}_{\text{state}}$ , Curriculum Generator  $\mathcal{M}_{\text{task}}$ , Initial GUI
   state  $S_0$ 
2: 1. Task Initialization
3:  $\mathcal{C}_0 \leftarrow \text{CaptionGUI}(S_0)$   $\triangleright$  Parse initial GUI layout (menu bar, buttons, etc.)
4:  $\mathcal{I}_0, U_0 \leftarrow \mathcal{M}_{\text{task}}(\emptyset, \emptyset, \emptyset, \mathcal{C}_0)$   $\triangleright$  Generate basic initial tasks and usage guide
5: for  $p = 0$  to  $P - 1$  do  $\triangleright$  2. Self-Evolution Phase Loop
6:   2.1 Autonomous Exploration
7:    $\mathcal{D}_{\text{traj}} \leftarrow \emptyset$ 
8:   for all  $I \in \mathcal{I}_p$  do
9:      $\tau \leftarrow \text{ExecuteInstruction}(\pi_p, I)$   $\triangleright$  Actor executes task in the virtual environment
10:    2.2 Effect Evaluation
11:     $\mathcal{J}_I, \mathcal{C}_I \leftarrow \mathcal{M}_{\text{state}}(\tau)$   $\triangleright$  Step-level trajectory judgment and new state captions
12:     $\mathcal{D}_{\text{traj}} \leftarrow \mathcal{D}_{\text{traj}} \cup \{(\tau, \mathcal{J}_I, \mathcal{C}_I)\}$   $\triangleright \mathcal{J}_I$ : a sequence of per-step feedback labels ( $a_T$  or  $a_F$ )
13:   end for
14:   2.3 Policy Update (RFT)
15:   Split  $\mathcal{D}_{\text{traj}}$  into:
16:      $\mathcal{D}_{\text{pos}}$ : steps labeled as positive  $a_T$ 
17:      $\mathcal{D}_{\text{neg}}$ : steps labeled as negative  $a_F$ 
18:   Compute GRPO loss on  $\mathcal{D}_{\text{pos}}$ :
19:      $r(a, a_T) = \mathbb{I}[\text{type}(a) = \text{type}(a_T)] + r_{\text{dist}}(a, a_T)$ 
20:   Compute Adversarial Imitation loss on  $\mathcal{D}_{\text{neg}}$ :
21:      $\mathcal{L}_{\text{AI}} = -\log \frac{\pi_{\theta}(a|s, I)}{\pi_{\text{ref}}(a_F|s, I)}$ 
22:   Total loss:  $\mathcal{L}_{\text{total}} = \mathcal{L}_{\text{GRPO}} + \gamma \mathcal{L}_{\text{AI}}$ 
23:    $\pi_{p+1} \leftarrow \text{Update}(\pi_p, \mathcal{L}_{\text{total}})$ 
24:   2.4 Task Update
25:    $\mathcal{I}_{p+1}, U_{p+1} \leftarrow \mathcal{M}_{\text{task}}(U_p, \mathcal{I}_p, \{\mathcal{J}_I\}, \{\mathcal{C}_I\})$   $\triangleright$  Generate more complex tasks based on new
   software knowledge and performance feedback
26: end for
27: Output: Specialized agent policy  $\pi_P$  after  $P$  stages of self-evolution

```

Web Step Level Judge Prompt Template

You are a Language Model specialized in judging the performance of web agents in web-navigation tasks. For a certain website, you are given the goal of a navigation task, the current URL of the webpage, the actions taken by the agent, and the thought process of the agent. **Additionally, you will have access to the sequence of key frame screenshots** Your task is to answer several questions about the agent's performance in the task.

You should carefully look at the sequential screenshot images in order to decide whether its successfully finish the task or failed halfway.

Question 1: Was the sequence of actions successful in achieving the goal?

Choices: <success>Successful</success>, <success>Unsuccessful</success>

Question 2: Did the agent perform unnecessary actions that could lead to unintended side effects?

Choices: <side>Yes</side>, <side>No</side>

Question 3: Did the agent perform the task optimally, by only performing necessary actions and avoiding unnecessary ones?

Choices:

<optimal>1. Complete Failure</optimal>

<optimal>2. Suboptimal</optimal>

<optimal>3. Somewhat Optimal</optimal>

<optimal>4. Completely Optimal</optimal>

Question 4: Did the agent loop through a sequence of actions that did not make progress towards the goal?

Choices: <loop>Yes</loop>, <loop>No</loop>

Provide your reasoning for each question.

Your answer **must** follow this exact format:

<reasoning>your reasoning here</reasoning>

<success>answer</success>

<side>answer</side>

<optimal>answer</optimal>

<loop>answer</loop>

Figure 7: **Prompt Template of GUI-Judge for web agent trajectories evaluations** with history screenshots as input, its difference with default prompt of AgentRewardBench (Lù et al., 2025) is highlighted in bold.

L SEAGENT SELF-EVOLUTION ALGORITHM

Algorithm 1 presents the core self-evolution training loop of SEAgent in a specialized software environment. The procedure is divided into four major stages:

(1) **Task Initialization.** Given the initial GUI state of a target software application, the World State Model performs dense captioning to extract structural semantics (e.g., menu bar, buttons), which is used by the Curriculum Generator to create an initial set of executable tasks and an editable software guidebook.

(2) **Autonomous Exploration and Effect Evaluation.** The agent explores each task via its current policy. The World State Model then performs step-level trajectory analysis, assigning each action a feedback label—either correct (a_T) or incorrect (a_F)—and generating GUI state change captions. This produces rich supervision signals for both policy learning and downstream task generation.

(3) **Policy Update via Reinforcement Fine-Tuning.** Based on the labeled execution data, positive and negative action steps are separated. We apply Group Relative Policy Optimization (GRPO) to reinforce correct actions, and Adversarial Imitation (AI) to suppress failure-prone behaviors. The updated policy is used for the next exploration round.

OSWorld Step Level Judge Prompt Template

I am evaluating the performance of a UI agent. The images provided are sequential keyframes that represent the full execution trajectory of the agent when attempting to follow a command. These keyframes correspond to the instruction: [INSTRUCTION].

Please thoroughly analyze the sequence to assess the following aspects:

1. Correctness — Did the agent successfully complete the task as instructed?
2. Redundant Steps — Identify any unnecessary or repeated actions that do not contribute to the goal.
3. Optimization — Did the agent follow an efficient plan with a minimal number of steps?
5. First Error Step — If the execution is incorrect or sub-optimal, determine the index of the first 5. keyframe where a mistake occurred.
6. Error Analysis — Provide a brief explanation of the mistake at that step.
7. Correct Action Suggestion — Explain what the agent should have done instead at the point of error.

Important Instructions:

The agent may have made progress toward the goal, but unless the task is fully and correctly completed, you must set 'Correctness' to False.

Be cautious in determining success. Missing confirmation screens, skipped inputs, or wrong UI elements clicked all count as errors.

Carefully examine all UI changes, button interactions, text entries, and any visual feedback in the screenshots.

Clearly indicate which exact steps are redundant (starting from 1).

Once you finish the analysis, return your evaluation in the following dictionary format. Include your step-by-step reasoning above the result.

```
<thinking>step by step reasoning.</thinking>
res_dict = {
    "Correctness": True or False,
    "Redundant": [step numbers],
    "Optimized": True or False,
    "First_Error_Step": step number or None,
    "Error_Type": "brief description of the mistake",
    "Correct_Action": "what should have been done instead"
}
```

Figure 8: **Prompt Template of GUI-Judge for OSWorld (Xie et al., 2024) trajectories**, which prompts judge model to provide step level reward signal.

(4) **Task Update.** The Curriculum Generator leverages feedback signals (\mathcal{J}) and GUI state transitions (\mathcal{C}) to propose more diverse and challenging tasks, thereby expanding the task frontier in a curriculum fashion.

This process repeats over multiple curriculum phases, ultimately yielding a specialized agent policy capable of mastering complex operations in the given software environment.

Task Buffer Update Prompt Template

You are now a teacher training a Computer Use Agent (CUA). This CUA is exposed to a new software environment and undergoes multiple rounds of iterative training. Your task is to issue new tasks for the agent to explore and train on, based on the feedback from the agent's actions. You are also responsible for summarizing a software usage manual to help the agent remember knowledge about the software.

The agent has provided the following feedback on its operations within the software:

```
{json.dumps(action_decription_list)}
```

Here is the software usage document you summarized in the previous round: {document}

Here is the agent's performance on the task you provided in the previous round:

```
{json.dumps(exam)}
```

You are also access to the previous given tasks with the screenshot caption after agent's execution. You can also use these captions and results to evaluate the agent's capability and generate new task and update document accordingly given the caption of the new screen and the corresponding intruction with judged evaluation: {json.dumps(prev_states)}

Please:

- Analyze the agent's performance.
- Integrate new knowledge from the feedback.
- Update the usage manual accordingly.
- Design a new set of tasks (with increased difficulty) (30 or more) that reinforce the concepts the agent struggled with in the last round.
- Each task **must be concise and specific**, targeting a concrete atomic action, based on the document and agent's observations, such as:
 - "Create a file named main.py."
 - "Open Terminal card."
- Each task must be executable from software initial state with no file open, e.g. you should not generate task like save xxx.txt if xxx.txt doesn't exist or created.
- if task is in sequential order with reliance, you should output a seq list like [subtask1, subtask2, ...], if there is no reliance, output [task].
- Decompose and target previous errors in a more focused way.

Output your reasoning and analysis process first. Then output the updated usage document and task list in the following JSON format within a SINGLE JSON DICT easier for me to parse:

```
{
  "software_document_new": "...",
  "exam_new": [[subtask1, subtask2, ...], [task]...]
```

Figure 9: **Prompt Template for task buffer update**, which generates new tasks in a curriculum manner and update software documents. The new tasks are used for actor to perform next phase of RL.

Visual Studio Code Usage Manual (v2)

1. Overview

Visual Studio Code (VS Code) is a source code editor. Its interface consists of a Menu Bar, Activity Bar (primary sidebar with icons), Editor Group (where files are opened), Panel (for terminal, output, etc.), and Status Bar.

2. Menu Bar

Located at the top of the application window.

2.1. File Menu

- **New Text File (Ctrl+N)**: Creates a new, untitled, unsaved text file (e.g., "Untitled-1").
- **New File... (Alt+Ctrl+N)**: Opens a dialog (often within VS Code, not an OS dialog) to specify a name and path for a new file, then creates it. Requires clicking a "Create File" button in the dialog.
- **New Window (Ctrl+Shift+N)**: Opens an entirely new VS Code window.
- **Open File... (Ctrl+O)**: Opens a system dialog to browse and select an existing file to open in the editor.
- **Open Folder... (Ctrl+K Ctrl+O)**: Opens a system dialog to browse and select a folder to open as the current workspace.
- **Save (Ctrl+S)**: Saves the currently active file. If the file is untitled, it behaves like "Save As..."
- **Save As... (Ctrl+Shift+S)**: Opens a system dialog to save the currently active file with a specific name and location.
- **Auto Save**: Toggles the auto save feature. When checked, files are saved automatically based on configured settings (e.g., after a delay).
- **Preferences**:
 - **Settings (Ctrl+,)**: Opens the Settings UI tab, allowing modification of user and workspace settings.
 - **Color Theme**: Opens a command palette dropdown to select and apply a different UI color theme.
 - **Keyboard Shortcuts**: Opens a UI tab displaying all keyboard shortcuts with a search bar to find specific shortcuts.
- **Close Editor (Ctrl+W)**: Closes the currently active file tab in the editor group.
- **Close Folder**: If a folder is open as a workspace, this closes it, returning to a "NO FOLDER OPENED" state in the Explorer.
- **Exit (Ctrl+Q)**: Closes the VS Code application.

2.2. View Menu

- **Command Palette... (Ctrl+Shift+P / F1)**: Opens the command palette, a searchable list of all available commands.
- **Explorer (Ctrl+Shift+E)**: Toggles the visibility of the Explorer panel in the sidebar.

- **Search (Ctrl+Shift+F)**: Toggles the visibility of the Search panel in the sidebar.
- **Source Control (Ctrl+Shift+G)**: Toggles the visibility of the Source Control (Git) panel in the sidebar.
- **Run (Ctrl+Shift+Q)**: Toggles the visibility of the Run and Debug panel in the sidebar.
- **Extensions (Ctrl+Shift+X)**: Toggles the visibility of the Extensions panel in the sidebar.
- **Terminal (Ctrl+`)**: Toggles the visibility of the integrated Terminal panel (usually at the bottom).
- **Word Wrap (Alt+Z)**: Toggles word wrapping for the text in the active editor.
- **Appearance**: Contains sub-menu for controlling visibility of UI elements like Activity Bar, Status Bar, etc.
- **Editor Layout**: Contains sub-menu for splitting editor, changing layout.

2.3. Go Menu

- **Go to File... (Ctrl+P)**: Opens a quick search palette to find and open files within the current workspace by name.

2.4. Run Menu

- **Start Debugging (F5)**: Initiates a debugging session. If no debug configuration exists, it may prompt to "Select debugger" or to create a "Launch.json" file.

2.5. Terminal Menu

- **New Terminal (Ctrl+Shift+T)**: Opens a new terminal instance in the integrated Terminal panel.
- **Split Terminal (Ctrl+Shift+T)**: Splits the currently active terminal in the Terminal panel into two.

2.6. Help Menu

- **About**: Displays a dialog with information about the VS Code version, commit, environment, etc.

3. Activity Bar

Vertical bar of icons on the far left, used to switch between different views in the sidebar.

- **Explorer icon (File Explorer)**: Shows the file and folder structure of the open workspace. If no folder is open, displays "NO FOLDER OPENED" and buttons to "Open Folder" or "Clone Repository". Right-clicking the "EXPLORER" title bar shows a context menu for panel settings.
- **Search icon (Magnifying Glass)**: Opens the Search panel to search for text across files in the workspace.
- **Source Control icon (Branching)**: Opens the Source Control panel for Git operations. If no folder is open or no repository is detected, it provides options like "Open Folder" or "Initiate Repository".
- **Run and Debug icon (Bug with Play)**: Opens the Run and Debug panel to configure and manage debugging sessions, set breakpoints, and control execution flow.

- **Extensions icon (Blocks)**: Opens the Extensions panel to browse, search, install, and manage VS Code extensions from the Marketplace. Features a "Search Extensions in Marketplace" input field.

4. Integrated Terminal Panel

- Accessed via View > Terminal or Terminal > New Terminal or Ctrl+`.
 - Appears typically at the bottom of the window.
 - Contains tabs: PROBLEMS, OUTPUT, DEBUG CONSOLE, TERMINAL, PORTS.
 - The **TERMINAL** tab provides a command-line interface (CLI).
 - Displays a prompt (e.g., user@hostname: ~).
 - Allows typing and executing shell commands.
 - Has a dropdown menu to select the default shell (e.g., bash, sh, zsh).
 - Includes icons for creating a new terminal (VS), splitting the terminal, and killing (closing) the current terminal instance.

5. Command Palette

- Opened via Ctrl+Shift+P / F1 or View > Command Palette.
- A text input field appears at the top of the editor area.
- Type to search for and execute various VS Code commands (e.g., preferences: editor: theme, change language mode).

6. Settings UI

- Accessed via File > Preferences > Settings or Ctrl+,.
- Opens in a new tab.
- Features a search bar to find specific settings.
- Settings are grouped into categories (User, Workspace) and sections (Commonly Used, Text Editor, Files, etc.).
- Provides GUI controls (dropdowns, input fields, checkboxes) to modify settings.
- An icon (document with X) in the top-right of the Settings tab allows opening the settings.json file for direct text-based configuration.

7. Editor Area

- Main area where the contents are displayed and edited.
- Supports multiple tabs for open files (e.g., "Untitled-1", "open1.txt").
- New, empty file may show placeholder text like "Select a language...".
- The left margin (Gutter) is used to set and remove breakpoints (displayed by clicking).

8. Dialogs & Pop-ups

- **"Save As" / "Create File" Dialogs**: These dialogs sometimes integrate into VS Code, sometimes OS-native (require typing a filename into an input field and then clicking a confirmation button (e.g., "Save", "Create File").

Figure 10: **Automatically generated usage manual during self exploration on VScode.**

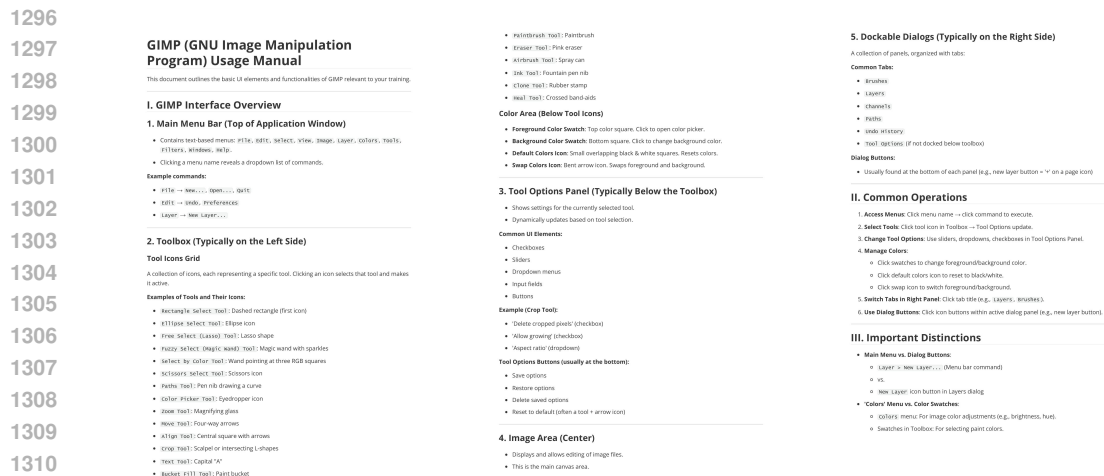


Figure 11: **Automatically generated usage manual during self exploration on GIMP.**

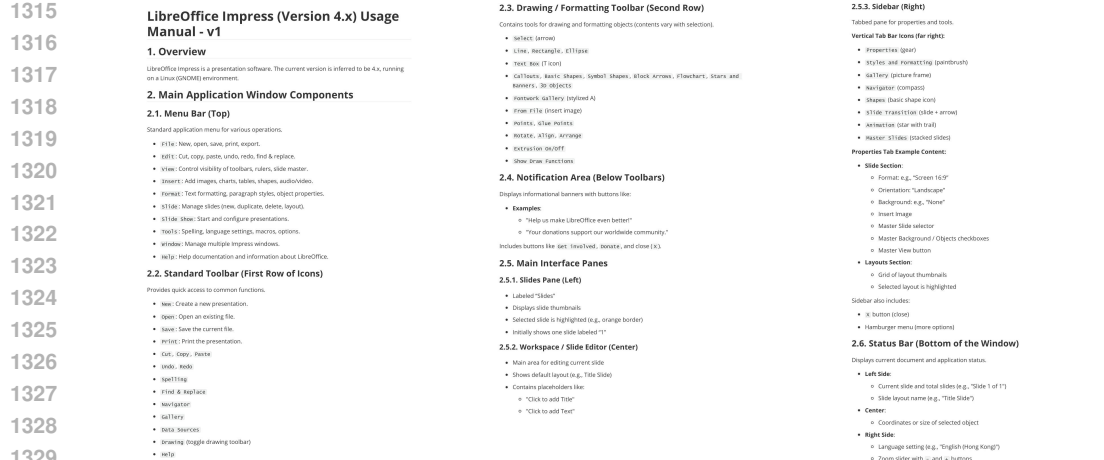


Figure 12: **Automatically generated usage manual during self exploration on LibreOffice_Impress.**

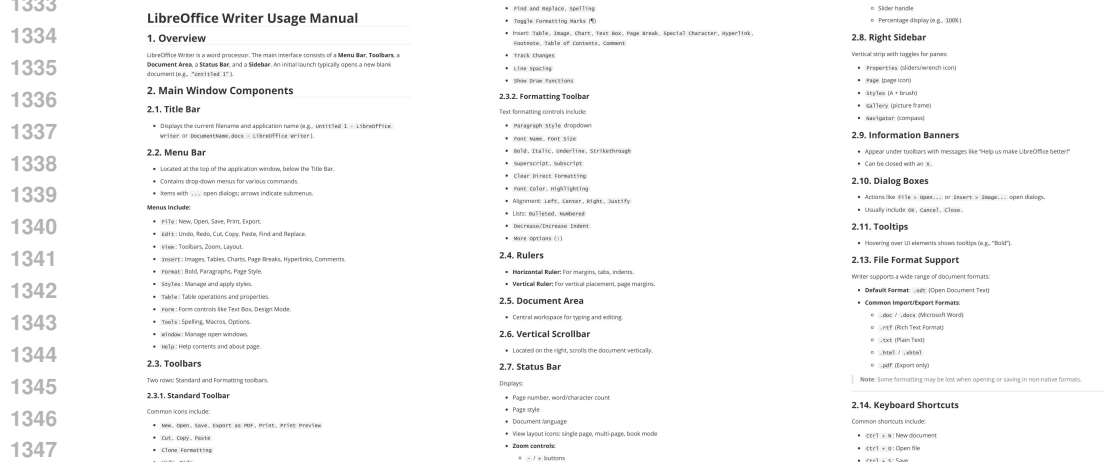


Figure 13: **Automatically generated usage manual during self exploration** on LibreOffice_Writer.