OSExpert: Computer-Use Agents Learning Professional Skills via Exploration

Anonymous Authors[1]

Abstract

General-purpose computer-use agents have shown impressive performance across diverse digital environments. However, our new benchmark, **OSExpert-Eval**, indicates they remain far less helpful than human experts. Although inference-time scaling enables adaptation, these agents complete complex tasks inefficiently with degraded performance, transfer poorly to unseen UIs, and struggle with fine-grained action sequences. To solve the problem, we introduce a GUI-based depth-first search (GUI-DFS) exploration algorithm to comprehensively explore and verify an environment's unit functions. The agent then exploits compositionality between unit skills to self-construct a curriculum for composite tasks. To support fine-grained actions, we curate a database of action primitives for agents to discover during exploration; these are saved as a *skill set* once the exploration is complete. We use the learned skills to improve the agent's performance and efficiency by (1) enriching agents with ready-to-use procedural knowledge, allowing them to plan only once for long trajectories and generate accurate actions, and (2) enabling them to end inference-time scaling earlier by realizing their boundary of capabilities. Extensive experiments show that our environment-learned agent takes a meaningful step toward expert-level computer use, achieving a 17% performance gain on OSExpert-Eval and closing the efficiency gap to humans by 89%.

1. Introduction

General computer-use agents have demonstrated impressive abilities across diverse digital environments. Powered by large multimodal models, they are capable of navigating file systems, assisting with online shopping, browsing the web, and operating everyday applications by interpreting screens and executing grounded actions from natural-language instructions (Xie et al., 2024b; Wang et al., 2024; Bonatti et al., 2024b; Pan et al., 2024; Rawles et al., 2025; Agashe et al., 2024; Wang et al., 2025b; Hu et al., 2025b); Notably, Agent-S3 (Gonzalez-Pumariega et al., 2025) reports near–human-level performance on OSWorld (Xie et al., 2024b).

However, results on our new benchmark OSExpert-Eval indicate that current computer-use agents are not yet helpful enough for real-world workers operating professional software. As illustrated in Figure 1, OSExpert-Eval evaluates the competence of computer use agent along four dimensions: (1) completing complex, long-horizon tasks; (2) generalizing to unseen and creative UI designs; (3) executing fine-grained actions that require precise control; and (4) maintaining strong end-to-end performance relative to human experts who solve these tasks quickly and reliably. Across these dimensions, current agents consistently lag behind human experts: success rates decline sharply with increasing task complexity, dropping from near human-level to below 10%. Generalization to unseen environments remains weak, and fine-grained execution frequently fails. Agents also rely on high-failure trial-and-error exploration, resulting in substantial inefficiency and 5–50× higher latency than human experts.

What went wrong? Our analysis of agent–environment interaction trajectories suggests a more fundamental limitation: it stems from how we train computer-use agents in the first place. Today's foundation agents are largely trained on scaled human demonstrations across around $\sim$100 digital environments via behavior cloning and reinforcement learning (Wang et al., 2025b;a; Ye et al., 2025). However, these agents often fail to acquire environment-specific procedural knowledge, and what they do learn transfers poorly as real-world interfaces and workflows evolve. Even in familiar environments, agents frequently solve long-horizon tasks through step-by-step planning and execution rather than directly obtaining reliable procedures, which introduces substantial computational overhead and amplifies cascading errors. In unfamiliar environments, the problem is more severe: agents struggle to identify the correct UI elements to interact with, and this difficulty compounds when tasks require fine-grained actions with precise control.

[1]Anonymous Institution, Anonymous City, Anonymous Region, Anonymous Country. Correspondence to: Anonymous Author <anon.email@domain.com>.

Preliminary work. Under review by the International Conference on Machine Learning (ICML). Do not distribute.

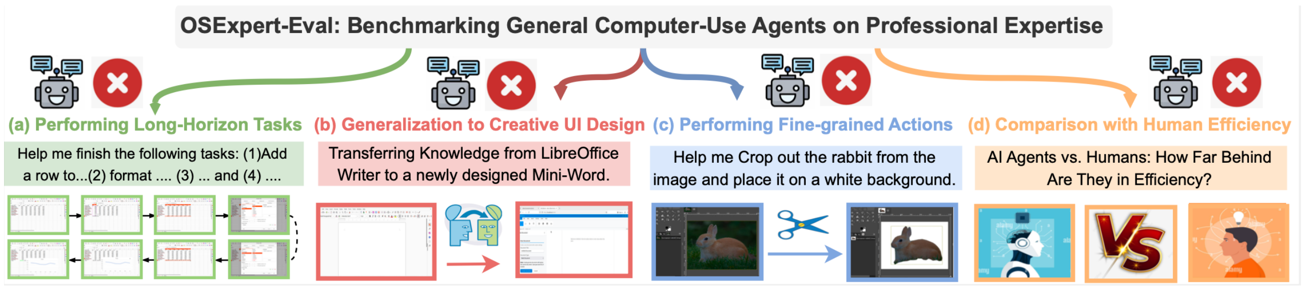

OSExpert-Eval: Benchmarking General Computer-Use Agents on Professional Expertise
(a) Performing Long-Horizon Tasks
(b) Generalization to Creative UI Design
(c) Performing Fine-grained Actions
(d) Comparison with Human Efficiency
Help me finish the following tasks: (1)Add a row to...(2) format .... (3) ... and (4) ....
Transferring Knowledge from LibreOffice Writer to a newly designed Mini-Word.
Help me Crop out the rabbit from the image and place it on a white background.
AI Agents vs. Humans: How Far Behind Are They in Efficiency?
VS

Figure 1. Our **OSExpert-Eval** shows that current computer-use agents remain far from expert-level: they struggle with long-horizon tasks, generalize poorly to unseen UI designs, lack fine-grained control over action sequences, and still fall well short of human expert efficiency.

To address this problem, we introduce OSExpert, an environment-learning paradigm designed to equip computer-use agents with professional knowledge and skills. Instead of scaling human annotated data for new digital environments, OSExpert exposes agents directly to environments, enabling them to explore, autonomously acquire and compose their own skill sets without additional human effort for each environment. We further introduce three key technical innovations that demonstrate how this paradigm enables computer-use agents to acquire environment-specific knowledge, handle fine-grained action sequences, and achieve robust and efficient performance approaching the level of human experts.

First, we introduce a **GUI-DFS** exploration algorithm to systematically discover the environment's unit functions. The exploration is carried out by three coordinated modules: (i) a *planning module* that generates exploration plans, (ii) an *action module* that executes UI actions and interacts with the environment, and (iii) a *feedback module* that evaluates the current exploration state and provides state-dependent feedback. We detail the design of these modules and their interactions in Section 3. Based on the discovered unit functions, the agent then self-proposes a learning curriculum by composing unit skills into higher-level procedures, which allows it to acquire procedural knowledge beyond the limits of common user-query distribution. The discovered unit functions and the validated composite procedures are both consolidated into the agent's skill set.

Second, to improve inference efficiency, we go beyond step-by-step imitation via reference trajectories and blind test-time scaling. Instead, we: (1) apply LoRA to fine-tune a lightweight language model that generates an entire plan in a single forward pass, reducing latency and avoiding the error accumulation of iterative planning. (2) introduce a skill-boundary check that predicts when additional test-time scaling would be unproductive; by validating failed and unsolved exploration traces, the agent can recognize low-success regimes and terminate early, avoiding the wasted latency of blind rollout-based exploration.

Third, to construct skills for accurate fine-grained manipulation and control, we provide a database of fine-grained action descriptions, action-sequence primitives, and grounding modules for constructing task-specific solutions. When the feedback module indicates a need for fine-grained control, the agent selects a promising primitive, executes it with grounding-based verification, and consolidates successful solutions into the skill set for reuse at inference time.

We deploy our learning framework across multiple digital environments and evaluate it on OSExpert-Eval. The results consistently validate our design and represent a substantial step toward expert-level performance without requiring any additional human effort. In particular, while existing computer-use agents peak at roughly 4% success on long-horizon tasks, our agent improves success rate to around 30%. Moreover, our agent transfers reliably to unseen UI environments and executes fine-grained actions with high fidelity with the obtained skill set, achieving around 21% success and improving efficiency by roughly 40% relative to the most efficient existing agent.

2. OSExpert-Eval: Benchmarking General CUAs on Professional Expertise

While recent work (Gonzalez-Pumariega et al., 2025) reports human-level results on OSWorld (Xie et al., 2024a), the benchmark itself is limited to relatively low task complexity and a narrow set of evaluation criteria that do not full reflect human-level capabilities. As a testatment, current computer-use agents still fall short of human experts in real application scenario, with slower interaction and inconsistent response quality. To quantify this gap, we introduce **OSExpert-Eval**, a new benchmark that evaluates agent expertise across four dimensions, as shown in Figure 1. OSExpert-Eval measures:

(1) Long-horizon compositional workflows, which require chaining multiple unit functions to achieve higher-level objectives beyond the short, single-function tasks in OSWorld (Xie et al., 2024a). We use the same environments, including GIMP[1] and LibreOffice, but increase only task compositionality and complexity to isolate and quantify the

[1] GIMP is an open-source image editing tool on Linux.

Table 1. Comparison of web-based agents and computer-use agent benchmarks. Most mainstream benchmarks do not explicitly evaluate composite skills, generalization to creative UI designs, fine-grained low-level interaction sequences, or latency & efficiency evaluation.

Real-World Benchmark / Evaluation Dimensions	**Long-Horizon Composite Skills**	**Creative UI Generalization**	**Fine-Grained UI Interaction**	**Latency Evalaution**
Mind2Web (Deng et al., 2023)	✓	✗	✗	✗
BEARCUBS (Song et al., 2025b)	✓	✗	✗	✗
OSWorld (Xie et al., 2024a)	✗	✗	✗	✗
WinAgentArena (Bonatti et al., 2024a)	✓	✗	✗	✗
macOSWorld (Yang et al., 2025)	✓	✗	✗	✗
OSWorld-Human (Abhyankar et al., 2025)	✗	✗	✗	✓
OSUniverse (Davydova et al., 2025)	✓	✗	✓	✗
Ours (OSExpert-Eval)	✓	✓	✓	✓

performance gap induced by OSWorld's simplified tasks.

(2) Generalization to unseen and creative user interfaces, which requires robust operation under customized layouts, icon-centric toolbars, and non-standard interaction patterns; we include the Tableau environment, a data visualization tool for data analysts that is rarely represented in the training distribution of current fundamental GUI agents and contains multiple out-of-distribution interaction patterns, and we also include our self-designed MiniWord editor, designed with creative UI layouts and vivid icons to minimize memorized interface conventions of softwares such as Microsoft Word and Google Docs.

(3) Fine-grained low-level actions, which require precise spatial control and accurate localization of interaction targets, where minor deviations can cause task failure. This stream includes tasks such as selecting target text, drag-and-drop, tracing object boundaries in images, and performing rotation or translation with specified angles or sizes.

(4) Execution efficiency, which measures the practical cost of attempting a task in terms of elapsed time, regardless of success or failure. We report human expert efficiency based on operators who are familiar with the tasks and practiced them prior to evaluation.

In total, OSExpert-Eval contains 113 evaluation tasks. Table 1 contrasts OSExpert-Eval with prior benchmarks and summarizes the key distinctions. Additional benchmark details and case examples are provided in Appendix A.

3. Environment-Learned CUAs

To equip computer-use agents with professional skills in a scalable and generalizable way, OSExpert directly exposes the agent to target digital environments and learns verifiable skills through interaction. As shown in Figure 2 and Figure 3, OSExpert improves both performance and efficiency through the following highlights: (1) a bottom-up exploration paradigm that comprehensively discovers unit-level functions for expert-level environment specific knowledge; (2) a lightweight LoRA-tuned planner that generates a complete plan in a single forward pass to avoid step-wise plan latency, together with a skill boundary check that avoids blind test-time scaling for expert efficiency; and (3) predefined action primitives and grounding modules that are verified during exploration and consolidated into the skill set for expert manipulation and control.

3.1. Bottom-Up Self-Exploration for Expert Knowledge

As shown in Algorithm 1, our core design is a GUI-DFS procedure that comprehensively discovers unit-level functionalities of a digital environment first, and later organizes and consolidates them into a skill set for robust and efficient inference. As illustrated in Figure 2, this approach reduces reliance on inference-time scaling and enables exploration without additional human effort, such as curating queries from tutorials or providing environment-specific use cases.

Initialization: Before exploration, the agent initializes three coordinated modules: a planning module, an action module, and a feedback module. As shown in Figure 2, the planning module forms lightweight assumptions about the application's data types and prepares downloaded or self generated template inputs. The action module then performs shallow interactions and UI analysis to infer the hierarchical structure of visible interface elements on the initial screen. Based on this structure, the agent enumerates top layer UI elements and selects an initial subset as targets to start the DFS based exploration. Each selected target is pushed onto the DFS stack as an exploration state node containing the current screenshot, the plan and action sequence to reach the state, and the next step plan conditioned on the state.

Exploration with GUI-DFS: As shown in Figure 2, the algorithm follows a DFS stack discipline and iteratively pops the top exploration state node in a last in first out manner.

To recover the node state, the environment is restarted and the stored action sequence is replayed. The action module then predicts and executes actions to satisfy the node's next step plan. After execution, the feedback module evaluates the new state with respect to the exploration history and classifies it as an intermediate state, a terminal state, or an error state due to an invalid plan or action. For intermediate states, the planning module proposes follow up plans, updates the action sequence to reach each new state, and pushes the resulting state nodes onto the stack. For terminal states, the validated plan and action sequence are condensed into a unit function skill and added to the skill set with a short description of when and how to use it. For error states, the feedback module provides targeted critiques of the plan or action, and the agent revises and retries for a bounded number of attempts. If failures persist, the error case is summarized and recorded in the skill set. Fine-grained action handling from error states is described in Section 3.3.

Organize and Extend the Skill Set: As shown in Figure 3 (left), the exploration stage produces a set of unit function skills by backtracking to each initial UI entry and systematically exploring its available functions. Each successful skill stores the planning sequence and action sequence required to complete the corresponding unit operation, while failed attempts are recorded as exploration failures. We further extend the skill set by prompting the agent to propose a curriculum of composite tasks that leverage natural compositions of unit functions, and we add the resulting composite skills to the unit skill set. To organize the skills, the agent summarizes what each skill can do, and we train a small language model to map each summary to its corresponding planning sequence, improving efficiency as discussed in Section 3.2. Skills associated with error states can be improved by pushing the corresponding nodes back onto the stack for additional exploration with stronger agents, or by manual curation from human experts. In our experiments, we run exploration only once and use no human curation.

Algorithm 1 GUI-DFS Algorithm

Definitions. Digital environment E; Environment State S_i; Exploration Plans Π; Action Sequence α; Exploration State Node $n \triangleq (\Pi, \alpha)$; Exploration Outcome $T \in \{\texttt{Continue}, \texttt{Final}, \texttt{Error}\}$.
Inputs. Planner Module P; Action Module A; Feedback Module F; Max Retries R.
Output. Unit Skill Set $\mathcal{K}$ for Environment E.

```
1:  $\mathcal{Q} \leftarrow \emptyset$, $\mathcal{K} \leftarrow \emptyset$, $(S_0) \leftarrow \texttt{Reset}(E)$
2:  $\mathcal{Q} \leftarrow \texttt{PushAll}(\mathcal{Q},\ P(E, S_0))$
3:  while $\mathcal{Q} \neq \emptyset$ do
4:      $(\Pi, \alpha) \leftarrow \texttt{Pop}(\mathcal{Q})$; $(S_0) \leftarrow \texttt{Reset}(E)$
5:      $S_i \leftarrow \texttt{Execute}(E, \alpha)$; $r \leftarrow 0$
6:      while $r < R$ do
7:          $\alpha' \leftarrow A(\Pi, \alpha, S_i)$; $S_{i+1} \leftarrow \texttt{Execute}(E, S_i, \alpha')$
8:          $(T, feedback) \leftarrow F(S_i, S_{i+1}, \Pi, \alpha')$
9:          if $T = \texttt{Continue}$ then
10:             $\mathcal{K} \leftarrow \mathcal{K} \cup \{(\Pi,\ \alpha \oplus \alpha')\}$
11:             for each $\Pi^{new} \in P(E, S_{i+1}, \Pi, \alpha \oplus \alpha')$ do
12:                 $\mathcal{Q} \leftarrow \texttt{Push}(\mathcal{Q},\ (\Pi \oplus \Pi^{new},\ (\alpha \oplus \alpha')))$
13:             end for
14:             break
15:         else if $T = \texttt{Final}$ then
16:             $\mathcal{K} \leftarrow \mathcal{K} \cup \{(\Pi,\ \alpha \oplus \alpha')\}$
17:             break
18:         else
19:             $\mathcal{Q} \leftarrow \texttt{Push}(\mathcal{Q},\ (\Pi \oplus feedback,\ \alpha \oplus \alpha'))$
20:             $r \leftarrow r + 1$
21:             break
22:         end if
23:     end while
24:     if $r \geq R$ then
25:         $\mathcal{K} \leftarrow \mathcal{K} \cup \{(\Pi,\ \alpha)\}$
26:     end if
27: end while
28: return $\mathcal{K}$
```

3.2. Fast Planner and Skill Check for Expert Efficiency

We adopt two approaches to reduce latency when using the skill set obtained in Section 3.1, each addressing a distinct bottleneck in existing computer use agents. First, agents often require many more interaction steps than humans and incur high overhead from step wise planning without procedural knowledge. Second, agents spend substantial time on inference time scaling for difficult tasks that still end in failure. Accordingly, we use a fast planner to generate a complete plan in a single forward pass, reducing step wise planning and extra interactions, and we apply a skill boundary check to early stop on tasks deemed infeasible by the skill set, avoiding wasted inference time exploration.

Single Pass Fast Planner: Prior work (Abhyankar et al., 2025) shows that current agents take about 1.4× to 2.7× more interaction steps than humans, and step-wise planning accounts for nearly 50% of total latency on OSWorld (Xie et al., 2024a). Our learned skill set provides procedural knowledge that support direct plan induction and reduce mid execution hesitations. We use the small language model trained on task planning pairs in Section 3.1 to generate a full plan in a single forward pass. The planner is LoRA tuned, and we store only the fine tuned weights for each digital environment, enabling low overhead deployment. Note that the fast planner reduces planning overhead but does not remove per step perception and action selection: the action module still inspects the screen at each step to produce the next action. When execution fails, the system falls back to the general planning stack, where relevant skills are injected into the agent context for test time scaling.

Skill Boundary Check for Early Stopping: As shown in

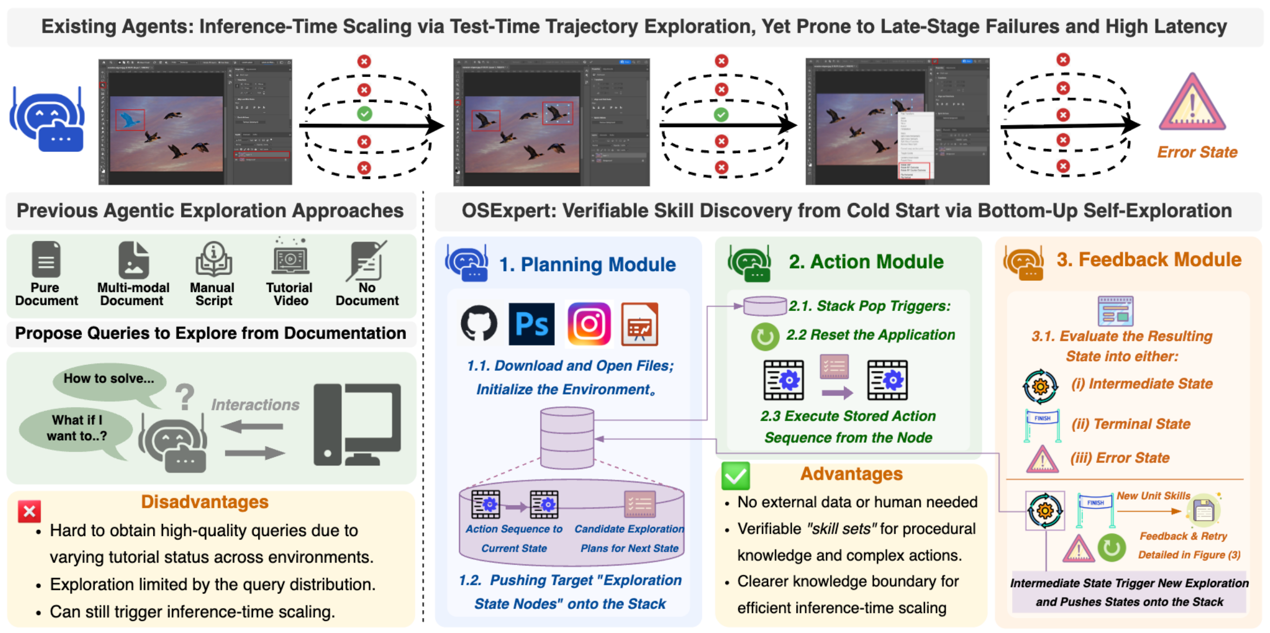

Existing Agents: Inference-Time Scaling via Test-Time Trajectory Exploration, Yet Prone to Late-Stage Failures and High Latency
Error State
Previous Agentic Exploration Approaches
Pure Document
Multi-modal Document
Manual Script
Tutorial Video
No Document
Propose Queries to Explore from Documentation
How to solve...
What if I want to..?
Interactions
Disadvantages
Hard to obtain high-quality queries due to varying tutorial status across environments.
Exploration limited by the query distribution.
Can still trigger inference-time scaling.
OSExpert: Verifiable Skill Discovery from Cold Start via Bottom-Up Self-Exploration
1. Planning Module
1.1. Download and Open Files; Initialize the Environment.
Action Sequence to Current State
Candidate Exploration Plans for Next State
1.2. Pushing Target "Exploration State Nodes" onto the Stack
2. Action Module
2.1. Stack Pop Triggers:
2.2 Reset the Application
2.3 Execute Stored Action Sequence from the Node
Advantages
No external data or human needed
Verifiable "skill sets" for procedural knowledge and complex actions.
Clearer knowledge boundary for efficient inference-time scaling
3. Feedback Module
3.1. Evaluate the Resulting State into either:
(i) Intermediate State
(ii) Terminal State
(iii) Error State
FINISH
New Unit Skills
Feedback & Retry Detailed in Figure (3)
Intermediate State Trigger New Exploration and Pushes States onto the Stack

Figure 2. **Up:** Current general-purpose computer-use agents rely on inference-time scaling, yet remain prone to failures and high latency. **Left:** Prior approaches explore digital environments using human-curated queries or tutorial-derived queries, which are often unavailable or difficult to obtain for arbitrary environments. **Right:** Our framework does not require external data or human effort for exploration queries and more comprehensively discover the unit functions of the digital environment, and benefits both performance and efficiency. We introduce how we handle the fine-grained actions during the exploration and how we organize the learned skill set in Figure 3.

Figure 2 (top), current computer use agents are typically unaware of their skill boundaries. When test time scaling has remaining iterations, they often continue attempting infeasible solutions, which leads to failure with substantial latency. Our learned skill set provides an explicit notion of capability. In particular, failed entries record unit functions that the agent repeatedly attempted during exploration but could not complete within bounded retries, suggesting low likelihood of success even with additional trials. Therefore, if an input query maps to a skill marked as failed, the agent stops early and reports an error, avoiding prolonged futile attempts during inference.

3.3. Skill Construction for Expert Fine-grained Control

We observe that many GUI manipulation tasks require precise visual grounding and accurate low level action generation. However, training data for such fine-grained behaviors remains scarce, causing current computer use agents to struggle with operations such as selecting an exact text span, dragging with pixel level precision, rotating objects by a specified angle, or tracing object boundaries in images. Without dedicated handling, the skill set constructed in Section 3.1 can accumulate many failure states.

To address this issue, we identify recurring fine-grained action primitives and organize them into a database with concise descriptions. As indicated in Figure 3 (right), we use this dataset to support skill construction during exploration. Specifically, when a fine-grained action is required, the feedback module typically triggers an error state and, based on the current exploration context, searches the database for suitable action primitives that may resolve the situation. For the GIMP scissor select tool usage example shown in figure, the matched primitive specifies calling an external segmentation tool, extracting boundary coordinates, and clicking a sequence of points along the object contour. We add the procedure to the skill set only if it succeeds under verification, together with a brief condition describing when to invoke the primitive. This process requires minimal human effort beyond defining common fine-grained patterns and providing feasible solutions to the agent. It substantially reduces annotation cost compared to approaches that rely on large scale manual labeling and supervised training for fine-grained control (Hu et al., 2025a).

4. Experiments

4.1. Experimental Settings

Environment and Benchmark We evaluate current computer-use agents and our environment learning framework on OSExpert-Eval, which consists of six interactive GUI environments. These include LibreOffice Writer, LibreOffice Calc, LibreOffice Impress, GIMP, web-based Tableau Public, and MiniWord, a lightweight text-editing environment developed in-house. Among these environments detailed in Section 2 and Appendix A, Tableau and Mini-

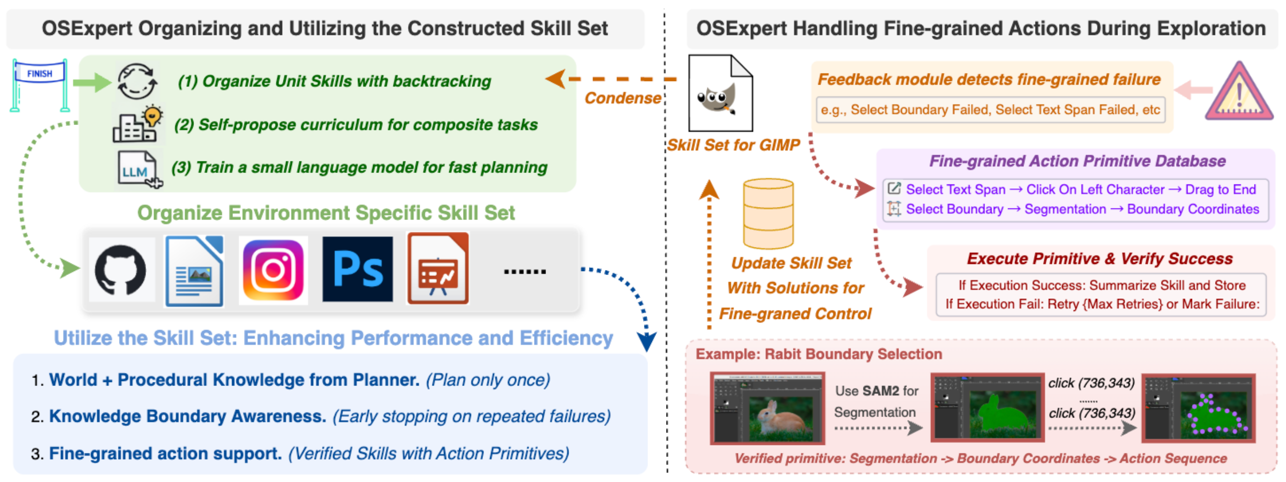

OSExpert Organizing and Utilizing the Constructed Skill Set
FINISH
(1) Organize Unit Skills with backtracking
(2) Self-propose curriculum for composite tasks
(3) Train a small language model for fast planning
LLM
Organize Environment Specific Skill Set
Utilize the Skill Set: Enhancing Performance and Efficiency
1. World + Procedural Knowledge from Planner. (Plan only once)
2. Knowledge Boundary Awareness. (Early stopping on repeated failures)
3. Fine-grained action support. (Verified Skills with Action Primitives)
Condense
OSExpert Handling Fine-grained Actions During Exploration
Feedback module detects fine-grained failure
e.g., Select Boundary Failed, Select Text Span Failed, etc
Skill Set for GIMP
Fine-grained Action Primitive Database
Select Text Span → Click On Left Character → Drag to End
Select Boundary → Segmentation → Boundary Coordinates
Update Skill Set With Solutions for Fine-graned Control
Execute Primitive & Verify Success
If Execution Success: Summarize Skill and Store
If Execution Fail: Retry {Max Retries} or Mark Failure:
Example: Rabit Boundary Selection
Use SAM2 for Segmentation
click (736,343)
click (736,343)
Verified primitive: Segmentation -> Boundary Coordinates -> Action Sequence

Figure 3. **Left:** How our framework organizes and utilize the self-constructed skill set for robust and efficient inference. The unit functions are obtained from the terminal states as shown in Figure 2. **Right:** How we handled potential fine-grained actions during exploration stage. The fine-grained action handling is usually triggered by an error state in the exploration, as shown in Figure 2. The selected primitive fine-grained action template will be added to the skill set for solving future queries if verified helpful.

WORD serve as unseen UIs that are not commonly included in existing computer-use benchmarks.

Model Configuration We use different model configurations for exploration and inference. During exploration, we use GPT-5 as the high-level planner and feedback module, and UI-TARS-1.5-7B (Qin et al., 2025) as the action module. We set the maximum retry budget to $R = 4$. We further include auxiliary grounding tools for fine-grained perception and verification, using Qwen-3-VL (Bai et al., 2025) for grounding verification and SAM2 (Ravi et al., 2024) for segmentation. After environment learning, we switch to inference using the constructed skill set and fast planning cache. We use Qwen-3-VL-8B as the base inference agent and Qwen-3-4B as the fast planner. Following Section 3.1 and Section 3.2, we first execute the plan generated by the fast planner for an input query, and only fall back to test-time scaling with relevant skills injected into the model context if execution fails. All experiments use the automatically constructed skill set without manual correction. For baselines, we include representative computer-use agents covering (i) specialized systems (e.g., OpenCUA and Computer-Use-Preview), (ii) general-purpose multimodal LLMs (e.g., Qwen-3-VL-8B), and (iii) agentic frameworks (e.g., the Agent-S series and CoAct). These methods typically rely on test-time scaling to adapt to new environments.

Evaluation Protocol To ensure fair comparison across methods, we enforce a unified interaction budget for all agents. Most OSExpert-Eval tasks require fewer than 10 interaction steps for a human expert, and the longest-horizon task takes 14 steps; we therefore cap each episode at 30 interaction steps for all methods. We always set base model temperature to 1.0, and we disallow Best-of-N sampling and evaluate each method with a single run per task. For OpenAI models, we use the official API, and we serve open-source models via vLLM (Kwon et al., 2023). We reporting average task success rate (SR) and average task completion time (Seconds) for one single run over the benchmarks. As shown in Tables 2 and 3, we report results separately for each environment.

4.2. Main Results

Limited Performance and Efficiency with Current CUAs Table 2 and Table 3 reveal a large gap between current computer use agents and human experts on the complex tasks introduced by our OSExpert-Eval. Agent capability is strongly bounded by task complexity and by the environments and interfaces observed during training. On unseen UIs and fine-grained action execution, none of the agents exhibits meaningful generalization, with success rates remaining in the 0% to 10% range. In these settings, agents often misinterpret UI elements, become confused by novel screens, and either loop through incorrect actions or rely on trial and error without discovering the correct procedure. Although most agents perform well on OSWorld, where evaluation often emphasizes unit function tests and relatively simple tasks, their success rates drop sharply on long horizon workflows due to cascading errors. Trajectory analysis further suggests two primary sources of inefficiency. First, high per step latency: agents typically generate a new plan from the current observation at each step and then produce a single action, and this one step planning is repeated after every UI change. Second, reliance on test time scaling: agents persist through repeated failures on difficult steps and often terminate only when the interaction budget is exhausted. Overall, compared to human experts, current agents exhibit less sta-

Table 2. Performance comparison on **OSExpert-Eval** using **Pass@1 success rate (↑)**. With a maximum of 30 interaction steps per task, current computer-use agents struggle on long-horizon tasks and fail to generalize to unseen UIs or execute fine-grained actions reliably.

General CUAs	**Agent Type**	**Long-Horizon Composite Skills**		**Unseen UI Generalization**		**Fine-Grained Action Execution**	
		GIMP	**LibreOffice**	**Tableau**	**MiniWord**	**GIMP**	**LibreOffice**
OpenCUA-7B (Wang et al., 2025b)	Specialized Model	0.00	0.00	0.00	0.00	0.07	0.05
Computer-Use-Preview (OpenAI, 2025)	Specialized Model	0.00	0.04	0.00	0.00	0.00	0.00
Qwen-3-VL-8B (Bai et al., 2025)	General Model	0.00	0.00	0.00	0.00	0.07	0.10
Agent-S2.5 w/ o3 (Agashe et al., 2025)	Agentic Framework	0.00	0.00	0.00	0.00	0.00	0.00
CoAct-1 (Song et al., 2025a)	Agentic Framework	0.00	0.04	0.05	0.03	0.00	0.05
Agent-S3 w/ GPT-5 (Gonzalez-Pumariega et al., 2025)	Agentic Framework	0.00	0.04	0.00	0.00	0.00	0.10
Ours (Environment-Learned CUA)	Agentic Framework	**0.33**	**0.29**	**0.25**	**0.17**	**0.28**	**0.26**

Table 3. Efficiency comparison on **OSExpert-Eval**, measured by **average execution time (seconds, ↓)**. We allow a maximum of 30 interaction steps per task, as previous general computer-use agents adopt a test-time scaling paradigm. While human experts typically complete these tasks within minutes, current agents often lack a reliable stopping criterion and spend excessive time on unnecessary or unproductive exploration, resulting in substantially longer execution times.

General CUAs	**Agent Type**	**Long-Horizon Composite Skills**		**Unseen UI Generalization**		**Fine-Grained Action Execution**	
		GIMP	**LibreOffice**	**Tableau**	**MiniWord**	**GIMP**	**LibreOffice**
OpenCUA-7B (Wang et al., 2025b)	Specialized Model	169	198	214	235	156	172
Computer-Use-Preview (OpenAI, 2025)	Specialized Model	236	241	378	296	257	283
Qwen-3-VL-8B (Bai et al., 2025)	General Model	140	135	132	153	86	136
Agent-S2.5 w/ o3 (Agashe et al., 2025)	Agentic Framework	834	850	513	487	549	462
CoAct-1 (Song et al., 2025a)	Agentic Framework	1315	1264	974	864	916	783
Agent-S3 w/ GPT-5 (Gonzalez-Pumariega et al., 2025)	Agentic Framework	1011	1245	659	628	710	647
Human Expert	Human Agent	**18**	**25**	**16**	**48**	**22**	**11**
Ours (Environment-Learned CUA)	Agentic Framework	71	66	65	87	63	84

ble performance and require about 5× to 50× longer elapsed time on OSExpert-Eval tasks. While the exact ratio depends on the iteration cap, stochasticity, and task difficulty, the latency gap remains substantial, and the performance gap persists even with additional interaction budget.

Performance and Efficiency Gain with OSExpert Table 2 and Table 3 show that our environment learning framework yields substantial gains on OSExpert Eval. Starting from near zero performance for current computer use agents, OSExpert increases success rates to roughly 20% to 30% using only automated exploration, without additional human effort on curating exploration failures. Models equipped with the learned skill set also execute tasks more efficiently, reducing overall completion time and closing a large portion of the efficiency gap to human experts. These results highlight the potential of our environment learning framework for constructing next generation computer use agents.

4.3. Qualitative Analysis

Gains from Environment Specific Knowledge We find that the gains on OSExpert-Eval subsets targeting long horizon compositional workflows and unseen UI generalization primarily come from environment specific procedural knowledge stored in the skill set. On these tasks, existing computer use agents often rely on step wise planning without procedural knowledge, which makes them vulnerable to early mistakes that propagate across later steps. Such errors can trigger misguided exploration and poor recovery behavior, and agents may fail to return to a valid earlier state even after detecting the mistake. In contrast, our agent executes verified plans and action templates from the skill set, so each step is more likely to be correct and the probability of entering unrecoverable states is reduced. These results underscore the importance of environment specific procedural knowledge for robust multi step execution.

Gains from Skills for Fine-Grained Actions As shown in Table 2, We find that the gains on OSExpert-Eval subsets targeting fine-grained action execution mainly come from the skills constructed in Section 3.3. While current computer use agents are not trained to reliably perform these precise manipulations, our exploration procedure allows the agent to invoke action primitives from the database and succeed during environment learning. The resulting condensed skills then guide the agent to reproduce these behaviors at inference time. These results suggest an alternative to continuously scaling manual annotation or relying on complex end to end training for fine-grained control tasks such as tracing object boundaries, and point to a practical path toward human expert level manipulation.

Efficiency Gains from Fast Planner Table 3 shows substantial efficiency variation across computer use agents. For agentic frameworks such as Agent S and CoAct, although they perform well on OSWorld style tasks, they are typically slower due to step wise planning and separate calls to planning and action models. In contrast, single model agents also plan step by step, but often couple planning with action prediction, avoiding repeated screen perception. Our fast planner is therefore most effective for agentic frameworks, as it largely removes per step planning overhead, while providing smaller gains for single model agents as they has already been quite efficient.

Efficiency Gains from Skill Boundary Check Our skill boundary check improves efficiency by early stopping when a required skill is marked as failed during exploration. Unlike standard test time scaling, which may repeatedly retry without a capability check and incur long tail latency, our agent uses the skill set to detect low likelihood steps. If a required unit function repeatedly failed under similar conditions, the agent stops with an error signal instead of continuing retries. This primarily reduces latency on failure cases and has limited effect when tasks succeed.

5. Relatd Work

General Computer Use Agents With recent advances in LLMs and MLLMs, agents for digital environments have progressed from text-only interactive worlds (Côté et al., 2018; Hausknecht et al., 2019) and simulated benchmarks (Liu et al., 2018; Yao et al., 2022) to operating in realistic web interfaces (Zhou et al., 2023; Koh et al., 2024) and real-world desktop and mobile applications (Bonatti et al., 2024a; Rawles et al., 2024; Xie et al., 2024a; Davydova et al., 2025). Trained on human demonstrations, either carefully annotated datasets or in-the-wild online video sources, these agents learn to perceive and interpret the screen, generate natural-language plans and reasoning, and ground them into executable action sequences (Wang et al., 2025b; Song et al., 2025a; Wang et al., 2025a; Agashe et al., 2025). While their performance on daily, simple tasks is approaching human levels (Gonzalez-Pumariega et al., 2025), their efficiency often still lags behind and remains insufficient for practical real-world deployment (Abhyankar et al., 2025), and their ability to generalize to complex professional workflows remains limited. OSWorld-Human (Abhyankar et al., 2025) evaluates the efficiency of computer-use agents on OSWorld (Xie et al., 2024a), showing that agents can incur substantial latency, particularly in the planning stage. OSUniverse (Davydova et al., 2025) stratifies tasks by complexity and reports that current agents struggle to solve the hardest tier. Recently, Anthropic advocated “Do not build agents, build skills instead” (Zhang & Murag, 2025), suggesting that scaling training data for base models may not be the optimal way to construct general computer use agents. In this work, we introduce an alternative: automatically constructing a skill set by deploying computer use agents in the target digital environment.

Self-Evolving GUI Agents The vast and constantly changing nature of GUI environments makes exhaustive human annotation infeasible, self-evolving GUI agents (Fang et al., 2025; Zhang et al., 2025; Li et al., 2025; Wang et al., 2025c; Sun et al., 2025) therefore aim to autonomously acquire application-specific knowledge from interaction (Gao et al., 2025). WebEvolver coevolves a world model for better self-training and look-ahead planning (Fang et al., 2025), UI-Evol retraces interaction traces and critiques/refines externalized knowledge to better align with execution (Zhang et al., 2025), and Mobile-Agent-E distills experience into reusable *Tips* and executable *Shortcuts* for complex mobile tasks (Wang et al., 2025c). They leverage exploration differently: Most works like AppAgent v2 build UI knowledge and uses retrieval-augmented execution (Li et al., 2025), while SEAgent continually updates skills from experience via reinforcement learning (Sun et al., 2025). In contrast, our environment-learned agents learn comprehensively from unit functions rather than user-query-driven exploration, and introduce procedural skill caching for efficient planning and fine-grained tool creation toward expert-level performance.

6. Conclusion

In this work, we propose a paradigm for developing more capable and general computer use agents by directly deploying them in digital environments and enabling them to autonomously construct skill sets through bottom up exploration. During this process, agents learn skills for executing fine-grained actions that require precise control, and their efficiency is further improved through a fast planner derived from the learned skills and through explicit awareness of their skill boundaries. Together, our algorithm and design choices enable agents to acquire professional level competencies, bringing general computer use agents a significant step closer to human expert level performance.

By enabling agents to discover, compose, and reuse skills across environments, our approach also addresses a central bottleneck in building more general intelligent systems. Current agents are often tied to specific tasks, interfaces, or training distributions, which limits their applicability in open world scenarios. Paradigms like OSExpert, which emphasize environment driven skill discovery and compositional procedural knowledge, offer a promising way forward. While we focus on computer use agents, these principles extend naturally to software engineering agents, embodied robots, and vision language action models deployed on robots to assist humans. We hope OSExpert serves not only as a practical system but also as a step toward adaptive, next generation self improving agents and, ultimately, more general purpose intelligence.

Impact Statement

This paper aims to advance machine learning by improving the capability, efficiency, and robustness of computer use agents through environment driven skill discovery. If deployed responsibly, such agents may increase productivity in professional software workflows by assisting users with complex, multi step tasks. As with other automation systems, risks include overreliance, unexpected failures in novel interfaces, and misuse in settings where errors are costly. Our approach mitigates some of these risks by constructing a verifiable skill set through bottom up exploration, organizing successful procedures into reusable skills and recording persistent failures as explicit boundaries. We also use a fast planner derived from the learned skills to reduce step wise planning overhead while retaining per step perception and action execution. For fine-grained GUI operations that require precise visual grounding, we leverage a database of action primitives and add a procedure to the skill set only when it succeeds under verification, reducing reliance on uncontrolled trial and error. Overall, we do not identify ethical concerns beyond those commonly associated with deploying machine learning based automation and decision support systems, but we emphasize the importance of human oversight and cautious use in high stakes environments.

Limitations

While OSExpert demonstrates substantial improvements in both performance and efficiency over existing computer use agents, several limitations remain.

Exploration cost and scalability. Our environment learning framework relies on explicit interaction with the target digital environment, including repeated environment resets, action replays, and verification. Although GUI-DFS is designed to yield verified skills early, exploration can still be time and compute intensive for large applications with deeply nested menus, high branching factors, or highly combinatorial interfaces. Scaling OSExpert to environments with hundreds of tools, dynamic UI layouts, or frequent version changes remains an open challenge.

Dependence on base model capabilities. The quality of the learned skill set depends on the underlying planner, action module, and feedback module used during exploration. Stronger models may discover more reliable skills and recover from difficult states more effectively, while weaker models may produce incomplete or noisy skill sets. Although OSExpert avoids environment specific human demonstrations, it does not eliminate reliance on strong foundation models.

Incomplete coverage and brittle failure modes. Under finite budgets, GUI-DFS does not guarantee exhaustive coverage of all valid interaction paths. Some rare or deeply nested functionalities may remain undiscovered, and certain interactions may still fail due to visual ambiguity, occlusions, non deterministic UI behavior, or unexpected pop ups. While failed exploration traces are recorded and used for early stopping, they may reflect limitations of the exploration policy rather than true infeasibility.

Manual design of fine-grained primitives. Our fine-grained action handling requires identifying recurring manipulation patterns and providing corresponding action primitives and grounding tools. Although this effort is substantially smaller than large scale annotation or end to end supervised training, it still introduces manual design and domain knowledge. Automating the discovery, verification, and refinement of such primitives is an important direction for future work.

Statistical stability and budget accounting. Our main results are reported from a single run per task, and we do not fully characterize variance across random seeds, model stochasticity, or UI nondeterminism. In addition, OSExpert introduces an explicit exploration phase whose cost depends on the environment, retry budget, and underlying models. While we report inference time efficiency, a more complete accounting of exploration cost and a controlled compute budget comparison across methods would further strengthen the empirical analysis.

Evaluation scope. OSExpert Eval focuses on professional desktop applications and GUI based workflows. While it captures challenges that are under represented in existing benchmarks, it does not cover all interaction settings, such as mobile interfaces, purely web native automation, or continuous real time environments. Extending environment learning and skill construction to these settings is left for future work.

Overall, these limitations reflect trade offs inherent in environment driven learning. We view OSExpert as a step toward more adaptive and self improving computer use agents, and we believe addressing these challenges will further advance the field.

References

Abhyankar, R., Qi, Q., and Zhang, Y. Osworld-human: Benchmarking the efficiency of computer-use agents, 2025.

Agashe, S., Han, J., Gan, S., Yang, J., Li, A., and Wang, X. E. Agent s: An open agentic framework that uses computers like a human, 2024. URL https://arxiv.org/abs/2410.08164.

Agashe, S., Wong, K., Tu, V., Yang, J., Li, A., and Wang, X. E. Agent s2: A compositional generalist-specialist framework for computer use agents. *arXiv preprint*, 2025. URL https://arxiv.org/abs/2504.00906.

Bai, S., Cai, Y., Chen, R., Chen, K., Chen, X., Cheng, Z., Deng, L., Ding, W., Gao, C., Ge, C., Ge, W., Guo, Z., Huang, Q., Huang, J., Huang, F., Hui, B., Jiang, S., Li, Z., Li, M., Li, M., Li, K., Lin, Z., Lin, J., Liu, X., Liu, J., Liu, C., Liu, Y., Liu, D., Liu, S., Lu, D., Luo, R., Lv, C., Men, R., Meng, L., Ren, X., Ren, X., Song, S., Sun, Y., Tang, J., Tu, J., Wan, J., Wang, P., Wang, P., Wang, Q., Wang, Y., Xie, T., Xu, Y., Xu, H., Xu, J., Yang, Z., Yang, M., Yang, J., Yang, A., Yu, B., Zhang, F., Zhang, H., Zhang, X., Zheng, B., Zhong, H., Zhou, J., Zhou, F., Zhou, J., Zhu, Y., and Zhu, K. Qwen3-vl technical report, 2025. URL https://arxiv.org/abs/2511.21631.

Bonatti, R., Zhao, D., Bonacci, F., Dupont, D., Abdali, S., Li, Y., Lu, Y., Wagle, J., Koishida, K., Bucker, A., Jang, L., and Hui, Z. Windows agent arena: Evaluating multi-modal os agents at scale, 2024a.

Bonatti, R., Zhao, D., Bonacci, F., Dupont, D., Abdali, S., Li, Y., Lu, Y., Wagle, J., Koishida, K., Bucker, A., Jang, L., and Hui, Z. Windows agent arena: Evaluating multi-modal os agents at scale, 2024b. URL https://arxiv.org/abs/2409.08264.

Côté, M.-A., Kádár, Á., Yuan, X., Kybartas, B., Barnes, T., Fine, E., Moore, J., Tao, R. Y., Hausknecht, M., Asri, L. E., Adada, M., Tay, W., and Trischler, A. Textworld: A learning environment for text-based games, 2018.

Davydova, M., Jeffries, D., Barker, P., Márquez Flores, A., and Ryan, S. Osuniverse: Benchmark for multimodal gui-navigation ai agents, 2025.

Deng, X., Gu, Y., Zheng, B., Chen, S., Stevens, S., Wang, B., Sun, H., and Su, Y. Mind2web: Towards a generalist agent for the web, 2023.

Fang, T., Zhang, H., Zhang, Z., Ma, K., Yu, W., Mi, H., and Yu, D. Webevolver: Enhancing web agent self-improvement with coevolving world model. *arXiv preprint arXiv:2504.21024*, 2025.

Gao, H.-a., Geng, J., Hua, W., Hu, M., Juan, X., Liu, H., Liu, S., Qiu, J., Qi, X., Wu, Y., et al. A survey of self-evolving agents: On path to artificial super intelligence. *arXiv preprint arXiv:2507.21046*, 2025.

Gonzalez-Pumariega, G., Tu, V., Lee, C.-L., Yang, J., Li, A., and Wang, X. E. The unreasonable effectiveness of scaling agents for computer use, 2025. URL https://arxiv.org/abs/2510.02250.

Hausknecht, M., Ammanabrolu, P., Côté, M.-A., and Yuan, X. Interactive fiction games: A colossal adventure, 2019.

Hu, S., Lin, K. Q., and Shou, M. Z. Showui-π: Flow-based generative models as gui dexterous hands, 2025a. URL https://arxiv.org/abs/2512.24965.

Hu, X., Xiong, T., Yi, B., Wei, Z., Xiao, R., Chen, Y., Ye, J., Tao, M., Zhou, X., Zhao, Z., Li, Y., Xu, S., Wang, S., Xu, X., Qiao, S., Wang, Z., Kuang, K., Zeng, T., Wang, L., Li, J., Jiang, Y. E., Zhou, W., Wang, G., Yin, K., Zhao, Z., Yang, H., Wu, F., Zhang, S., and Wu, F. Os agents: A survey on mllm-based agents for general computing devices use, 2025b. URL https://arxiv.org/abs/2508.04482.

Koh, J. Y., Lo, R., Jang, L., Duvvur, V., Lim, M., Huang, P.-Y., Neubig, G., Zhou, S., Salakhutdinov, R., and Fried, D. VisualWebArena: Evaluating multimodal agents on realistic visual web tasks. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 881–905, Bangkok, Thailand, August 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.acl-long.50. URL https://aclanthology.org/2024.acl-long.50/.

Kwon, W., Li, Z., Zhuang, S., Sheng, Y., Zheng, L., Yu, C. H., Gonzalez, J. E., Zhang, H., and Stoica, I. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*, 2023.

Li, Y., Zhang, C., Jiang, W., Yang, W., Fu, B., Cheng, P., Chen, X., Chen, L., and Wei, Y. Appagent v2: Advanced agent for flexible mobile interactions, 2025. URL https://arxiv.org/abs/2408.11824.

Liu, E. Z., Guu, K., Pasupat, P., Shi, T., and Liang, P. Reinforcement learning on web interfaces using workflow-guided exploration. In *International Conference on Learning Representations (ICLR)*, 2018.

OpenAI. computer-use-preview model (openai api documentation). Technical documentation, 2025. URL https://platform.openai.com/docs/models/computer-use-preview. Accessed: 2026-01-08.

Pan, Y., Kong, D., Zhou, S., Cui, C., Leng, Y., Jiang, B., Liu, H., Shang, Y., Zhou, S., Wu, T., and Wu, Z. Webcanvas: Benchmarking web agents in online environments, 2024. URL https://arxiv.org/abs/2406.12373.

Qin, Y., Ye, Y., Fang, J., Wang, H., Liang, S., Tian, S., Zhang, J., Li, J., Li, Y., Huang, S., Zhong, W., Li, K., Yang, J., Miao, Y., Lin, W., Liu, L., Jiang, X., Ma, Q., Li, J., Xiao, X., Cai, K., Li, C., Zheng, Y., Jin, C., Li, C., Zhou, X., Wang, M., Chen, H., Li, Z., Yang, H., Liu, H., Lin, F., Peng, T., Liu, X., and Shi, G. Ui-tars: Pioneering automated gui interaction with native agents, 2025. URL https://arxiv.org/abs/2501.12326.

Ravi, N., Gabeur, V., Hu, Y.-T., Hu, R., Ryali, C., Ma, T., Khedr, H., Rädle, R., Rolland, C., Gustafson, L., Mintun, E., Pan, J., Alwala, K. V., Carion, N., Wu, C.-Y., Girshick, R., Dollár, P., and Feichtenhofer, C. Sam 2: Segment anything in images and videos, 2024. URL https://arxiv.org/abs/2408.00714.

Rawles, C., Clinckemaillie, S., Chang, Y., Waltz, J., Lau, G., Fair, M., Li, A., Bishop, W., Li, W., Campbell-Ajala, F., Toyama, D., Berry, R., Tyamagundlu, D., Lillicrap, T., and Riva, O. Androidworld: A dynamic benchmarking environment for autonomous agents, 2024.

Rawles, C., Clinckemaillie, S., Chang, Y., Waltz, J., Lau, G., Fair, M., Li, A., Bishop, W., Li, W., Campbell-Ajala, F., Toyama, D., Berry, R., Tyamagundlu, D., Lillicrap, T., and Riva, O. Androidworld: A dynamic benchmarking environment for autonomous agents, 2025. URL https://arxiv.org/abs/2405.14573.

Song, L., Dai, Y., Prabhu, V., Zhang, J., Shi, T., Li, L., Li, J., Savarese, S., Chen, Z., Zhao, J., Xu, R., and Xiong, C. Coact-1: Computer-using agents with coding as actions. *arXiv preprint*, 2025a. doi: 10.48550/arXiv.2508.03923. URL https://arxiv.org/abs/2508.03923.

Song, Y., Thai, K., Pham, C. M., Chang, Y., Nadaf, M., and Iyyer, M. Bearcubs: A benchmark for computer-using web agents, 2025b.

Sun, Z., Liu, Z., Zang, Y., Cao, Y., Dong, X., Wu, T., Lin, D., and Wang, J. Seagent: Self-evolving computer use agent with autonomous learning from experience. *arXiv preprint arXiv:2508.04700*, 2025.

Wang, H., Zou, H., Song, H., Feng, J., Fang, J., Lu, J., Liu, L., Luo, Q., Liang, S., Huang, S., et al. Ui-tars-2 technical report: Advancing gui agent with multi-turn reinforcement learning, 2025a.

Wang, X., Wang, B., Lu, D., Yang, J., Xie, T., Wang, J., Deng, J., Guo, X., Xu, Y., Wu, C. H., et al. Opencua: Open foundations for computer-use agents, 2025b.

Wang, Z., Cui, Y., Zhong, L., Zhang, Z., Yin, D., Lin, B. Y., and Shang, J. Officebench: Benchmarking language agents across multiple applications for office automation, 2024. URL https://arxiv.org/abs/2407.19056.

Wang, Z., Xu, H., Wang, J., Zhang, X., Yan, M., Zhang, J., Huang, F., and Ji, H. Mobile-agent-e: Self-evolving mobile assistant for complex tasks, 2025c. URL https://arxiv.org/abs/2501.11733.

Xie, T., Zhang, D., Chen, J., Li, X., Zhao, S., Cao, R., Hua, T. J., Cheng, Z., Shin, D., Lei, F., Liu, Y., Xu, Y., Zhou, S., Savarese, S., Xiong, C., Zhong, V., and Yu, T. Osworld: Benchmarking multimodal agents for open-ended tasks in real computer environments, 2024a.

Xie, T., Zhang, D., Chen, J., Li, X., Zhao, S., Cao, R., Hua, T. J., Cheng, Z., Shin, D., Lei, F., Liu, Y., Xu, Y., Zhou, S., Savarese, S., Xiong, C., Zhong, V., and Yu, T. Osworld: Benchmarking multimodal agents for open-ended tasks in real computer environments, 2024b. URL https://arxiv.org/abs/2404.07972.

Yang, P., Ci, H., and Shou, M. Z. macosworld: A multilingual interactive benchmark for gui agents, 2025. URL https://arxiv.org/abs/2506.04135.

Yao, S., Chen, H., Yang, J., and Narasimhan, K. Webshop: Towards scalable real-world web interaction with grounded language agents. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2022.

Ye, J., Zhang, X., Xu, H., Liu, H., Wang, J., Zhu, Z., Zheng, Z., Gao, F., Cao, J., Lu, Z., Liao, J., Zheng, Q., Huang, F., Zhou, J., and Yan, M. Mobile-agent-v3: Fundamental agents for gui automation, 2025. URL https://arxiv.org/abs/2508.15144.

Zhang, B. and Murag, M. Don't build agents, build skills instead. YouTube video (AI Engineer), December 2025. URL https://www.youtube.com/watch?v=CEvIs9y1uog. Talk by Anthropic.

Zhang, Z., Liu, X., Zhang, X., Wang, J., Chen, G., and Lu, Y. Ui-evol: Automatic knowledge evolving for computer use agents, 2025. URL https://arxiv.org/abs/2505.21964.

Zhou, S., Xu, F. F., Zhu, H., Zhou, X., Lo, R., Sridhar, A., Cheng, X., Ou, T., Bisk, Y., Fried, D., Alon, U., and Neubig, G. Webarena: A realistic web environment for building autonomous agents, 2023.

A. OSExpert-Eval Benchmark Details

OSExpert-Eval is a curated benchmark designed to evaluate computer-use agents on professional-level challenges that go beyond routine GUI interaction. The benchmark consists of 113 tasks spanning three major categories: *Long-Horizon Compositional Workflows*, *Unseen UI Generalization*, and *Fine-Grained Action Execution*. The overall task composition and environment breakdown are summarized in Figure 4.

Specifically, **Long Horizon contains 30 tasks**, including 24 Office tasks and 6 GIMP tasks, emphasizing multi-step workflows that require composing multiple unit functions in a correct and robust order. **Unseen UI contains 50 tasks**, including 20 Tableau tasks and 30 MiniWord tasks, targeting novel layouts and interaction patterns that are uncommon in current agents' training distributions. **Fine-Grained contains 33 tasks**, including 14 GIMP tasks and 19 Office tasks, requiring precise low-level control such as accurate text selection, object manipulation, and spatial alignment.

Below, we present representative examples from each category, including the task instructions and their corresponding ground-truth outcomes (Figures 5, 6, and 7).

The dataset was manually constructed by the paper's coauthors, who each spent over two hours becoming familiar with the core functionalities of the target software. After designing the tasks, they executed and recorded their own task completion trajectories to establish reference performance. We will release the full dataset and evaluation code upon acceptance.

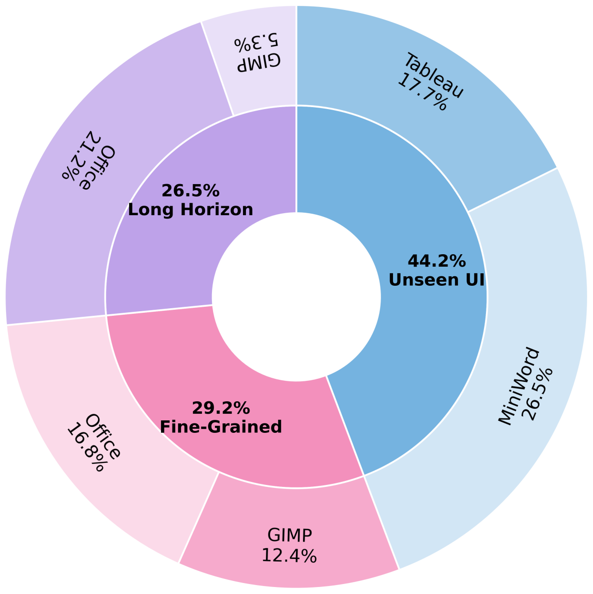


Figure 4. Composition of our evaluation tasks (113 total). The inner ring shows the three high-level categories (Unseen UI, Fine-Grained, and Long Horizon), while the outer ring breaks each category down by environment (Tableau, MiniWord, Office, and GIMP); slice sizes are proportional to the number of tasks. Here, Office includes LibreOffice Writer, LibreOffice Impress, and LibreOffice Calc.

B. Algorithm Design: GUI-BFS or GUI-DFS

We discuss the trade offs between the GUI-DFS procedure used in Algorithm 1 and a GUI-BFS 2 variant that follows the same exploration interface but replaces the stack with a queue. Both strategies are sound in the sense that, given sufficient budget and the same planner, action module, and feedback module, they can eventually enumerate comparable exploration targets and yield a unit function skill set. The practical differences arise from the constraints of computer use scenarios, where each node expansion requires an environment restart, action replay, and verification, and where interaction budgets, latency, and memory are limited.

Time to first useful skills. A key advantage of GUI-DFS is fast skill yield. DFS commits to one branch of the interface hierarchy and quickly reaches terminal states where a unit function can be verified and condensed into a reusable skill. This matters in computer use settings because skills are valuable as soon as they are discovered. Even if exploration is terminated early due to time limits or a stopping rule, DFS typically returns a nontrivial set of completed procedures and action templates that can already improve downstream performance. In contrast, GUI-BFS prioritizes breadth and spends early budget expanding many shallow nodes. This can delay reaching terminal states, resulting in fewer finalized skills under the same exploration time budget.

Memory and frontier growth. GUI-BFS maintains a wide frontier of intermediate nodes, which can grow rapidly with the branching factor of GUI menus, toolbars, and dialogs. In practice, this implies higher memory usage for storing exploration nodes and their associated plans and action prefixes. GUI-DFS keeps only a single active path plus a smaller set of deferred siblings, making its peak memory substantially lower in typical GUI trees. This difference is especially pronounced in professional software where top level menus expose many options and each option can further branch into dialogs and subpanels.

Budget allocation and restart overhead. Unlike classical graph search where expanding a node is cheap, GUI exploration expands nodes through environment restarts and action replays. Under BFS, the algorithm frequently switches among many shallow nodes, which increases repeated restart and replay overhead while the agent is still far from terminal outcomes. DFS amortizes restart overhead along a deeper trajectory and is more likely to spend compute on completing a full procedure once it enters a relevant subworkflow. As a result, DFS often yields better skill acquisition per unit of interaction budget when restarts are costly.

Coverage and systematic exploration. GUI-BFS has a natural advantage when the goal is uniform coverage of shallow UI elements. By exploring level by level, BFS can provide more balanced coverage of top level functions and reduce the chance that the algorithm spends excessive time in one deep subworkflow while neglecting other major menus. This can be beneficial when the environment contains many independent unit functions and the exploration budget is sufficiently large to reach terminal states across the frontier.

Error recovery and local traps. GUI-DFS can be more sensitive to deep local traps. If a branch repeatedly triggers difficult fine-grained interactions or brittle sequences, DFS may spend a disproportionate share of budget on retries before backtracking, depending on the retry rule and feedback quality. BFS can be more robust in this case because it interleaves progress across branches and can still collect skills from easier parts of the interface while hard branches remain unresolved. In our implementation, bounded retries and failure recording mitigate this risk for DFS by forcing backtracking and preventing unbounded time spent in a single branch.

Summary. For computer use agents, we generally prefer GUI-DFS, as used in Algorithm 1, because it produces verified unit skills quickly, supports useful partial results under early stopping, and has lower peak memory due to a smaller frontier. GUI-BFS is a reasonable alternative when the primary objective is broad coverage of shallow UI functions and sufficient budget is available, but it can incur higher memory and restart overhead and may delay the discovery of terminal state skills.

Algorithm 2 GUI-BFS Algorithm

Definitions. Digital environment E; Environment State S_i; Exploration Plans Π; Action Sequence α; Exploration State Node $n \triangleq (\Pi, \alpha)$; Exploration Outcome $T \in \{\texttt{Continue}, \texttt{Final}, \texttt{Error}\}$.
Inputs. Planner Module P; Action Module A; Feedback Module F; Max Retries R.
Output. Unit Skill Set $\mathcal{K}$ for Environment E.

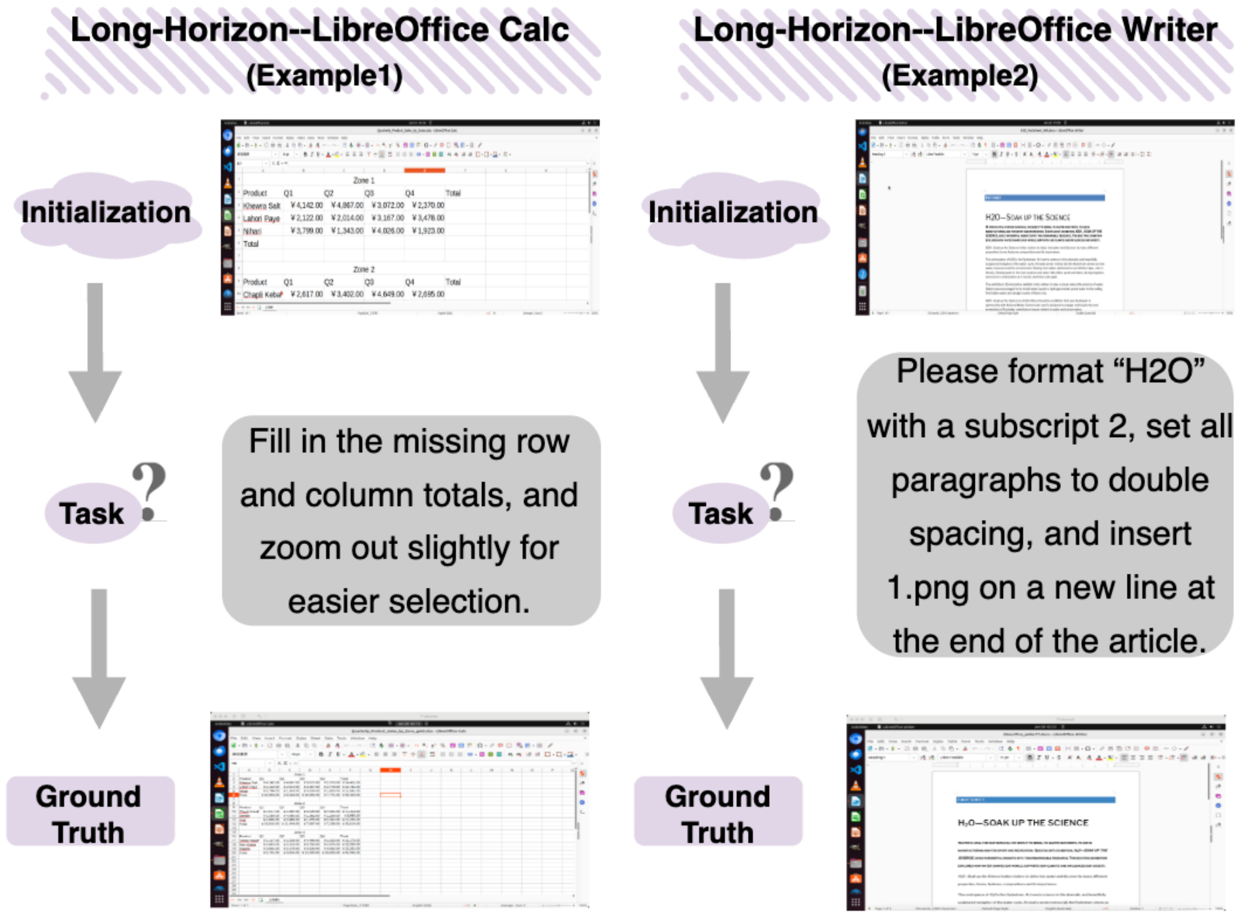
```
1:  $\mathcal{Q} \leftarrow \emptyset$, $\mathcal{K} \leftarrow \emptyset$, $(S_0) \leftarrow \texttt{Reset}(E)$
2:  $\mathcal{Q} \leftarrow \texttt{EnqueueAll}(\mathcal{Q},\ P(E, S_0))$
3:  while $\mathcal{Q} \neq \emptyset$ do
4:      $(\Pi, \alpha) \leftarrow \texttt{Dequeue}(\mathcal{Q})$; $(S_0) \leftarrow \texttt{Reset}(E)$
5:      $S_i \leftarrow \texttt{Execute}(E, \alpha)$; $r \leftarrow 0$
6:      while $r < R$ do
7:          $\alpha' \leftarrow A(\Pi, \alpha, S_i)$; $S_{i+1} \leftarrow \texttt{Execute}(E, S_i, \alpha')$
8:          $(T, feedback) \leftarrow F(S_i, S_{i+1}, \Pi, \alpha')$
9:          if $T = \texttt{Continue}$ then
10:             $\mathcal{K} \leftarrow \mathcal{K} \cup \{(\Pi,\ \alpha \oplus \alpha')\}$
11:             for each $\Pi^{new} \in P(E, S_{i+1}, \Pi, \alpha \oplus \alpha')$ do
12:                 $\mathcal{Q} \leftarrow \texttt{Enqueue}(\mathcal{Q},\ (\Pi \oplus \Pi^{new},\ (\alpha \oplus \alpha')))$
13:             end for
14:             break
15:         else if $T = \texttt{Final}$ then
16:             $\mathcal{K} \leftarrow \mathcal{K} \cup \{(\Pi,\ \alpha \oplus \alpha')\}$
17:             break
18:         else
19:             $\mathcal{Q} \leftarrow \texttt{Enqueue}(\mathcal{Q},\ (\Pi \oplus feedback,\ \alpha \oplus \alpha'))$
20:             $r \leftarrow r + 1$
21:             break
22:         end if
23:     end while
24:     if $r \geq R$ then
25:         $\mathcal{K} \leftarrow \mathcal{K} \cup \{(\Pi,\ \alpha)\}$
26:     end if
27: end while
28: return $\mathcal{K}$
```

C. Discussion and Future Work

Despite the effectiveness of GUI-DFS and environment-driven skill learning, our study reveals several open challenges and opportunities that point to promising future directions.

Exploration and Inference-Time Scaling. Exploration can be viewed as a form of scaling that shifts computation from inference time to a one-time environment learning phase. While inference-time scaling repeatedly explores alternative trajectories at test time, self-exploration amortizes this cost by converting experience into reusable procedural knowledge. However, both forms of scaling face similar challenges, including diminishing returns, long-tail failure cases, and difficulty identifying when additional attempts are unlikely to succeed. Our skill boundary check provides an initial step toward principled early stopping, but a deeper understanding of how to allocate exploration budgets and how to trade off exploration versus inference-time scaling remains an important research direction.

Structuring Complex and Nested Interfaces. Modern GUIs often expose deeply nested functionalities through menus, dialogs, toolbars, and mode-dependent panels. Identifying functional boundaries and hierarchical structure within such interfaces remains challenging, especially when visual grouping does not align with functional composition. More explicit representations of interface structure, potentially informed by accessibility trees, UI metadata, or design conventions, could significantly improve exploration efficiency and skill organization.


Long-Horizon--LibreOffice Calc
(Example1)
Long-Horizon--LibreOffice Writer
(Example2)
Initialization
Initialization
Task
Fill in the missing row and column totals, and zoom out slightly for easier selection.
Task
Please format "H2O" with a subscript 2, set all paragraphs to double spacing, and insert 1.png on a new line at the end of the article.
Ground Truth
Ground Truth
H2O—SOAK UP THE SCIENCE

Zone 1

Product	Q1	Q2	Q3	Q4	Total
Khewra Salt	¥4,142.00	¥4,867.00	¥3,072.00	¥2,370.00	
Lahori Paye	¥2,122.00	¥2,014.00	¥3,167.00	¥3,478.00	
Nihari	¥3,799.00	¥1,343.00	¥4,026.00	¥1,923.00	
Total					

Zone 2

Product	Q1	Q2	Q3	Q4	Total
Chapli Kebab	¥2,617.00	¥3,402.00	¥4,649.00	¥2,695.00	

Figure 5. Two examples from the *Long-Horizon Compositional Workflows* category in OSExpert-Eval. The left example shows a task in LibreOffice Calc, where the agent must complete a spreadsheet by filling missing row and column totals and adjusting the zoom for reliable interaction, illustrating a multi-step workflow that combines computation with interface control. The right example shows a task in LibreOffice Writer, where the agent must apply document-wide formatting, set double line spacing, and insert an image at the end of the article, requiring correct ordering and composition of multiple unit functions. For each example, we show the initial environment state, the task instruction, and the corresponding ground-truth outcome.

Compositionality and Skill Abstraction. Many professional workflows arise from nontrivial compositions of atomic UI operations. While our curriculum-based extension allows agents to construct composite skills from unit functions, the combinatorial space of possible compositions grows rapidly with environment complexity. Future work could explore more principled abstractions for skill composition, such as hierarchical skill graphs or program induction methods that reason over reusable subprocedures rather than enumerating combinations.

Interaction Beyond Visible UI Elements. Some interactive elements are not directly observable from static screenshots, including keyboard shortcuts, hidden context menus, gesture-based interactions, and state-dependent behaviors. These latent affordances pose challenges for purely visual exploration. Incorporating multimodal signals such as system logs, accessibility APIs, or limited instrumentation could help agents reason about actions that are visually implicit but functionally critical.

Skill Representation and Knowledge Transfer. In OSExpert, skills are stored as verified plans and action templates tied to specific environments. A natural extension is to study how such procedural knowledge can be represented more abstractly and transferred across related applications, interface versions, or task domains. Learning environment-agnostic representations of skills, while preserving their verifiability and grounding, would move computer-use agents closer to generalizable competence.

Human–Agent Collaboration in Exploration. While our framework emphasizes fully autonomous exploration, human input remains a valuable source of guidance, especially for rare failure cases or ambiguous UI behaviors. Future systems could support mixed-initiative exploration, where agents propose candidate skills and humans provide lightweight feedback, correction, or validation. Such collaboration could further improve skill quality while maintaining low annotation cost.

Beyond GUI-Based Agents. Although this work focuses on desktop GUI environments, the underlying principles of environment-driven skill discovery, verifiable procedural knowledge, and boundary-aware execution extend naturally to broader settings. These include vision–language–action models operating in embodied environments, robotics systems interacting with physical tools, and software engineering agents navigating code editors and development workflows. Exploring these extensions may help unify environment learning across digital and physical domains.

Overall, we view OSExpert as a step toward shifting agent design from reactive inference-time scaling toward structured, environment-grounded learning. Addressing these challenges may enable more adaptive, efficient, and trustworthy agents across a wide range of interactive settings.

D. Prompts Used During Exploration

This section documents the prompts used by different modules during environment exploration. We separate prompts by functional role, including file preparation, planning, and feedback. The planning prompts guide the agent in proposing exploration targets and interaction strategies, while the feedback prompts are responsible for classifying state transitions, verifying terminal outcomes, and diagnosing failures. All prompts are designed to be lightweight, modular, and reusable across environments, and are included to support transparency and reproducibility of the exploration process.

File Preparation Planner Prompt

Your job is to decide whether this application needs a specific example file or document to be ready before deeper exploration can proceed.

Based ONLY on what you can see in the screenshot and the exploration goal, you must answer:

- **1)** Does the agent need to open or create a concrete file/document/data object to meaningfully explore this application?

- **2)** If yes, what is the simplest, most representative file or object type that should be used?

- **3)** Should the agent: - create a new simple example file (“self-create-file”), - load a human-provided example file (“use-provided-file”), or - proceed without preparing a file because there is already sufficient content visible (“no-file-needed”)?

Exploration Goal: {*instruction*}

You MUST output a JSON object with the following fields:

{{ “file-decision”: “one of ‘self-create-file’, ‘use-provided-file’, or ‘no-file-needed’”,

“justification-decision”: “Short explanation of why this choice is appropriate given the visible UI”,

“suggested-file-type”: “If a file is needed, what is the simplest representative type (e.g., ‘short text document’, ‘2-column CSV with dummy data’, ‘small image file’); otherwise use ‘none’”,

“justification-alternatives”: “Very brief explanation of why the other options are less suitable in this specific UI state”

}}

Guidelines:

- If the UI clearly shows an empty document area with tools for editing, and no specific file content is required, it is often acceptable to choose “no-file-needed”.

- If the UI suggests typical “File/Open” or “File/New” workflows and there is no existing content, strongly consider “self-create-file” with a minimal example (e.g., one short paragraph, a tiny table).

- Use “use-provided-file” when the UI or goal explicitly suggests working with a specific sample dataset or external document (e.g., “open the provided report”, “analyze the given CSV”).

- Always keep your justifications short and grounded in what is visible.

Provide your response in JSON format only.

Planning Prompt: Initialize Exploration Targets

You are an INITIAL EXPLORATION PLANNER analyzing a completely new and unknown software application called {*app-name*}.

CRITICAL INSTRUCTIONS:

1. You may use your general knowledge about software interfaces, but your analysis must be grounded in what is visible in the screenshot.

2. Do not hallucinate UI elements or behaviors that are not supported by visible evidence in the screenshot.

3. Treat this as if you are seeing this specific software interface for the very first time.

4. Focus ONLY on the app-name application window. Ignore operating system elements like the top OS bar, clock, calendar, system notifications, or window controls (minimize/maximize/close buttons). Only analyze the application's own UI elements.

Exploration Goal: {*instruction*}

Current Step (Stage 1 - initial exploration): {*step-idx*}
Recent Initial-Exploration History: {*history-text if history-text else 'No initial exploration history yet'*}

Analyze the {*app-name*} application window by carefully examining the screenshot.

MAIN OBJECTIVE:
(Stage 1 - Initial Screen Understanding):

- Infer, based on the screenshot, what this software is probably used for.

- Infer what kinds of files or objects it most likely operates on (e.g., text documents, tabular data, images, project files, etc.).

- Propose a SMALL set of short exploratory actions that an action agent could perform to quickly test these hypotheses, with a focus on: * what kinds of data or files the application processes, and * how the main menus / side bars might be organized hierarchically.

You MUST output a JSON object with the following fields:
{{

"application-hypothesis": {{

"likely-purpose": "Short description of what this software is probably used for, grounded in the screenshot",

"possible-file-types": ["List of likely file or object types this software can open/edit/create (e.g., 'text documents', 'spreadsheets', 'presentations')"]

}},

"initial-probing-actions": [

{{

"description": "Short natural language description of what to try to better understand the software (e.g., 'open the Edit menu and inspect available options')",

"instruction": "Concrete instruction for the action agent to execute in order to perform this probing action",

"target-area": "Optional: name or description of the UI area where this action should be applied (e.g., 'top menu bar', 'left toolbar')",

"expected-signal": "What this probing action should reveal about EITHER (a) what data/file types the application works with OR (b) how the main menus / side bars are organized hierarchically"
}}
],

"is-terminal": false
}}

Guidelines:

- The probing actions should be SHORT and focused, suitable for a single or small sequence of clicks/keystrokes.

- Prefer actions that reveal high-level information such as available file operations, document types, or major modes of the application.

- Use the recent initial-exploration history to avoid repeating actions that have already been tried.

- You may set "is-terminal" to true when you believe the initial exploration goal has been sufficiently achieved for this stage.

Provide your response in JSON format only.

Planning Prompt: Initialize Shallow UI Interaction

You are a high-level exploration planner analyzing a completely new and unknown software application called {*app-name*}.

CRITICAL INSTRUCTIONS:

1. This is a BRAND NEW software that you have NO prior knowledge about. Do NOT use any knowledge about existing software interfaces (like GIMP, Photoshop, or any other applications).

2. You must base your analysis EXCLUSIVELY on what you observe in the screenshot. Do not make assumptions based on prior knowledge.

3. Treat this as if you are seeing this software interface for the very first time.

4. Focus ONLY on the {*app-name*} application window. Ignore operating system elements like the top OS bar, clock, calendar, system notifications, or window controls (minimize/maximize/close buttons). Only analyze the application's own UI elements.

Exploration Goal: {*instruction*}

Current Step: {*step-idx*}

Recent Exploration History: {*history-text if history-text else 'No history yet'*}

Previous Stage 1 Summary(if available):

- Application hypothesis:
json.dumps(application-hypothesis) if application-hypothesis is not None else 'N/A'

- Menu hierarchy summary:
menu-hierarchy-summary if menu-hierarchy-summary else 'N/A'

Analyze the {*app-name*} application window by carefully examining the screenshot.

STEP 1: Identify major UI areas by observing the screenshot layout. Name each area based on what you actually see - their location (top, left, right, bottom, center) and visual appearance (bar, panel, canvas, dock, toolbox, etc.). The area names should describe their location and content based purely on visual observation from the screenshot.

STEP 2: Using BOTH the screenshot and the Stage 1 menu hierarchy summary (if provided), identify which interactive elements are:

- highest-level menus/controls that should each be the ROOT of a dedicated exploration (to go into "exploration-plan"), and

- lower-hierarchy elements (submenus, toolbar buttons, nested controls) that will naturally be covered when exploring those root items (to go into "lower-hierarchy-elements").

When the Stage 1 hierarchy suggests that a region is context-dependent or subordinate (for example, a toolbar row whose contents change based on the active menu or mode), you MUST represent that region as lower-hierarchy under its controlling root(s), and NOT as its own root item in "exploration-plan".

STEP 3: For areas or elements that appear static, decorative, or unlikely to support meaningful interaction (e.g., logos, static backgrounds, non-clickable labels), list them separately in "low-priority-areas" with a brief explanation.

CRITICAL CONNECTION:

The keys in "exploration-plan" MUST exactly match the "area-name" values from "screen-areas-summary". Each area you identify in screen-areas-summary should have a corresponding key in exploration-plan with its list of interactive elements. This ensures the hierarchical organization is consistent.

Output format (JSON only):
{{

"screen-areas-summary": [

{{

"area-name": "Name this area based on what you observe in the screenshot (location + visual appearance, e.g., 'Top horizontal bar with text menus', 'Left vertical panel with icon buttons', 'Center canvas area showing image', 'Right side panel with tabs')",

"description": "Brief natural language description of this area and its main purpose based on visual observation"

}}
],

"exploration-plan":

{{
"[Area Name 1 - must exactly match area-name from screen-areas-summary]": [

{{

"exploration-space": "Identifier for a HIGHEST-LEVEL menu/control or major functional region that should be explored as a root (e.g., a top menu like 'File', a primary sidebar section, a major mode switch button)",

"instruction": "ONE short, concrete next-step instruction for what the agent should do RIGHT NOW with this element (e.g., 'Click the File menu', 'Click the Insert tab'). Do NOT describe long multi-step procedures here."
}}

],

"[Area Name 2 - must exactly match area-name from screen-areas-summary]": [

{{

"exploration-space": "Identifier for another highest-level menu/control or functional region",

"instruction": "ONE short, concrete next-step instruction for what the agent should do with this element based on the current observation."

}}

]

}},

"lower-hierarchy-elements": [{{

"area-name": "Area this lower-hierarchy group belongs to (must match some area-name in screen-areas-summary)", "covered-by": "Name of the higher-level exploration-space or menu under which these elements will naturally be explored",

"reason-not-explored-independently": "Short explanation of why these elements do not need their own top-level exploration items (e.g., 'will be covered when exploring the File menu and its submenus')"

}}],

"low-priority-areas": [
{{ "area-name": "Area or region that appears low-priority or mostly static",

"reason-low-priority": "Why this area is unlikely to require focused exploration (e.g., decorative logo, static background, non-interactive status bar)"

}}] }}

IMPORTANT:

The area names you use in exploration-plan keys MUST be exactly the same as the area-name values you create in screen-areas-summary. Analyze the screenshot to determine what areas exist, then use those same names consistently in both places.

CRITICAL GUIDELINES for exploration-space:

1. If the element has visible text/label: Use that text exactly as it appears (e.g., "File", "Edit", "Opacity", "New Layer")

2. If the element has NO visible text/label: You MUST provide BOTH:

- Shape description: What the icon/button looks like (e.g., "dotted rectangle", "brush icon", "eyedropper shape", "circular button with gradient"')

- Position description: Precise location within its area (e.g., “first row second column”, “top row leftmost”, “second row third column”, “bottom section left side”) - Example: “the button in the left toolbox, first row second column, shaped like a dotted rectangle with dashed border”

- Example: “the icon in the right dock, top section, third from left, shaped like a brush/paintbrush”

- Example: “the slider in the left panel, middle section, labeled ‘Opacity’ or appearing as a horizontal bar”

Important guidelines:

- **OBSERVATION-BASED ANALYSIS:** Describe only what you can actually see in the screenshot. Do not infer functionality based on names or icons that might remind you of other software.

- **AREA IDENTIFICATION:** Analyze the screenshot to identify distinct UI areas. Name them based on:

* Their location in the window (top, left, right, bottom, center)

* Their visual appearance (horizontal bar, vertical panel, canvas area, dock panel, toolbox)

* Their content (text menus, icon buttons, image display, etc.)

* Example naming patterns: “Top horizontal bar with text labels”, “Left vertical panel with icons”, “Center canvas area”, “Right side panel with tabs”

* DO NOT use generic names like ”Top Menu Bar” unless you actually see a menu bar at the top. Base names on what you observe in the screenshot.

- **AREA NAME CONSISTENCY:**

The area names you create in screen-areas-summary MUST be used as keys in exploration-plan. Each area-name in screen-areas-summary should have a corresponding key in exploration-plan with the exact same name. This creates the hierarchical connection between the summary and the plan.

- **HIERARCHICAL ORGANIZATION:**

Organize exploration-plan by the areas you identified. The structure should be: {{“Area Name 1”: [items], “Area Name 2”: [items], ...}} where Area Names exactly match the area-name values from screen-areas-summary. The items listed in exploration-plan should be restricted to the HIGHEST-LEVEL menus/controls and major functional regions.

- **COMPREHENSIVE COVERAGE:**

Ensure that ALL interactive elements you can observe are accounted for EITHER as:

* top-level items in exploration-plan (for the highest-level menus/controls that should be explored directly), OR

* grouped entries in lower-hierarchy-elements (for buttons/controls that will be naturally explored when interacting with a higher-level item), OR

* entries in low-priority-areas (for areas that appear mostly static or low-value, with an explanation).

- **NO OVERLAP / MUTUAL EXCLUSIVITY:**

Do NOT place the same concrete interactive element or area in more than one of these structures. Every element/area must belong to exactly ONE of: exploration-plan, lower-hierarchy-elements, or low-priority-areas. Use the Stage 1 menu hierarchy summary to decide whether a region is best treated as a root, a lower-hierarchy group, or low priority.

- **POSITION DETAILS:**

For elements without text labels, ALWAYS include:

* Row/column position (e.g., “first row”, “second column”)

* Relative position within the area (e.g., “top section”, “middle”, “bottom left”)

* Visual shape/icon description

* This helps the grounding model locate elements accurately

CRITICAL GUIDELINES FOR instruction:

- Each instruction should be a **single, short next-step action** the agent can take immediately based on the current observation (e.g., "Click 'File"', "Open the Tools menu", "Click the zoom dropdown at the bottom right").

- Avoid long multi-step or comprehensive phrases like "fully explore", "systematically inspect every option", or "explore all submenus". Those broader behaviors will be handled by higher-level controllers, not in this instruction field.

- The instruction should:

* Specify exactly ONE main interaction (e.g., click, open, hover, type a short string, toggle a control).

* Be grounded in visible UI elements and the Stage 1 hierarchy (e.g., refer to the exact label or clear visual description).

- Instructions must be concrete, concise, and observation-based, focused on the immediate next action rather than an entire exploration sequence.

- Base exploration on what hasn't been explored yet according to the history

- Focus on the application window only, ignore OS-level UI

- **REMEMBER:** You have NO prior knowledge about this software. Base everything on the screenshot observation. The area names should come from your analysis of the screenshot, not from prior knowledge or assumptions.

Provide your response in JSON format only.

Feedback Prompt: State Classification

You are a feedback model for GUI exploration. Please follow the instructions below to classify the exploration step.

The area the agent is actively exploring is: {*node-path*}

The recent instruction for the exploration step is: {*instruction*}

The historical exploration action sequences (stepwise) in this DFS step are: {history-actions-text}

The stop reason for this step is: {*history-summary*}

You are given TWO screenshots:

1) The original screenshot BEFORE these actions.

2) The new screenshot AFTER these actions.

First, carefully read the following FOUR SITUATIONS that describe typical exploration outcomes.

Your job is to decide which ONE situation best matches the current case.

Situation 1 (non-terminal but reasonable):

- The instruction is completed and the actions are overall reasonable.

- However, this step does NOT correspond to a final unit function; more exploration is still needed from the current screen.

- **Typical cases:**

* The agent clicks a top-level menu (e.g., "File") and a menu opens, but no specific sub-function (like "Save As") has been fully explored yet.

* The agent opens a side panel or toolbar that reveals many options, but does not yet test any particular option in detail.

- **Potential Harder Cases:**

* While the screen shows no change, the click is the correct one, but only the sidebars to open by this action are already opened (that explains why nothing happens on the screen, but the click is still correct and should be counted as a reasonable action.)

Situation 2 (terminal and reasonable):

- The instruction is completed and the actions are overall reasonable.

- This step DOES correspond to a completed unit function whose behavior can be summarized from the AFTER screenshot.

- **Typical cases:**

* The agent successfully applies a formatting change (e.g., makes selected text bold) and the visual change is clearly visible.

* The agent completes a dialog workflow (e.g., opens “Save As”, chooses a name, and confirms), and the result is visible on screen.

* Usually, the button or menu that is clicked, its name shall be reasonable correspond to the screen change that you would be able to observe.

Situation 3 (reasonable instruction, problematic actions):

- The exploration instruction is reasonable for this UI.

- However, the concrete actions are NOT reasonable (e.g., wrong target, missing key steps, or obviously ineffective).

- This is often paired with a stop reason like “max steps exceeded” or no meaningful visual change.

- Typical harder cases:

* The instruction says “Select the target text and apply bold to it”, but the agent failed to select the target text. Or it has no idea how to come up with a good way to perform this fine-grained action.

Situation 4 (problematic instruction, roughly reasonable actions align with the instruction):

- The action sequence itself is roughly reasonable for what the agent seems to be trying to do.

- However, the EXPLORATION INSTRUCTION is poorly posed for the current screen and the current target (underspecified, missing prerequisites, or misleading).

- Typical harder cases:

* The instruction says“ test/click the bold function”, but there is no text selected and the agent just clicks the Bold button once.

* The instruction says “test/click the crop tool”, but no region is selected or adjusted after clicking the crop icon.

YOUR TASK:

1) Choose the most appropriate situation among 1, 2, 3, and 4.

2) Provide a brief natural-language reasoning string explaining why this situation fits best.

Output STRICTLY the following JSON:

{{ “situation”: 1 or 2 or 3 or 4, “reasoning”: “brief explanation of why you chose this situation” }}

Feedback Prompt: Terminal State Verification

You are a feedback model for GUI exploration. This may be ANY application (known or unknown).

Your job: analyze screenshots and ground feedback to VISUAL OBSERVATIONS ONLY.

CONTEXT:

Exploring: {*node-path*}

Instruction: {instruction}

Actions: {history-actions-text}

Stop reason: {history-summary}

Classification: {reasoning}

Screenshots: BEFORE (state before actions) and AFTER (state after actions)

SITUATION 1: Non-terminal but reasonable - more exploration needed from current state.

YOUR TASK: Design NEXT exploration items that CONTINUE from this step.

REQUIREMENTS:

1. **GROUND TO VISUALS (Priority 1):**

- Examine BEFORE/AFTER screenshots carefully

- Base ALL suggestions on what you SEE in images

- Identify what changed: new menus, panels, dialogs, buttons

- DO NOT fabricate features based on assumptions about this software type

- Use general UI patterns (buttons, menus, dialogs), not software-specific knowledge

2. **Describe Elements Visually:**

- If unclear labels: use position + visual description

- Examples: “second button from left with bold ‘B’ icon”, “third menu item in dropdown”, “circular blue button at bottom-right”

- DO NOT invent names - describe what you actually see

3. **Sequential vs. Combined:**

- SEPARATE items: Independent buttons/options → one item per element, include ALL visible ones

- COMBINED item: Multi-step workflow (e.g., dialog with fields+button) → one item with all steps

4. **Continuation Logic:**

- Only suggest elements VISIBLE in AFTER screenshot

- Only suggest elements that appeared/became accessible from this step

- Menu opened → explore visible menu items

- Panel appeared → explore visible controls in it

- Dialog opened → explore visible options in it

5. **DO NOT:**

- List all clickable elements

- Suggest elements not visible in screenshots

- Assume features that ”should” exist

- Jump to unrelated areas

OUTPUT (JSON only):

{{

“next-exploration-items”: [

{{

“area”: “VISIBLE area description (e.g., ‘dropdown menu at top’, ‘toolbar second row’)”,

“exploration-space”: “Element description you can SEE (e.g., ‘third button with scissors icon’, ‘Save As menu item’)”,

“instruction”: “Clear instruction based on VISIBLE elements”

}}

]

}}

Requirements:

- Non-empty list

- Each item describes VISUALLY CONFIRMED elements in AFTER screenshot

- Grounded to what changed from this step

- No fabrication

Feedback Prompt: Error Attribution (Planning)

You are a feedback model for GUI exploration with deep knowledge of various desktop applications and their functionalities.

CONTEXT:

CONTEXT:

Exploring: {*node-path*}

Instruction: {instruction}

Actions: {history-actions-text}

Stop reason: {history-summary}

Classification: {reasoning}

Screenshots: BEFORE (state before actions) and AFTER (state after actions)

SITUATION 2: Terminal and Reasonable Exploration

The instruction was successfully completed. This represents a COMPLETE unit function that should be documented.

YOUR TASK:

Create a HIGH-QUALITY function summary that accurately captures what was accomplished.

CRITICAL REQUIREMENTS:

1. **Use Your Prior Knowledge:**

Apply your understanding of application features and typical user workflows to interpret what happened.

2. **Trajectory-Based Summary:**

- Analyze the COMPLETE action trajectory (buttons clicked, menus accessed, etc.)

- Identify the EXACT functionality that was exercised

- Note the visual changes in the AFTER screenshot

3. **Summary Quality:**

- Be SPECIFIC about what the function does (not just "clicked a button")

- Reference the UI elements involved (button names, menu paths, etc.)

- Describe the OBSERVABLE effect or outcome

- Keep it concise (1-2 sentences) but informative

4. **Examples of Good Summaries:**

- "Applied bold formatting to selected text by clicking the Bold button in the toolbar, resulting in the text appearing in bold weight."

- "Opened the Save As dialog from File menu and successfully saved the document with a new filename to the Desktop location."

- "Inserted a table with 3 rows and 2 columns using the Insert ¿ Table menu option, which now appears in the document."

5. **Examples of Poor Summaries:**

- "Clicked a button." (too vague)

- "Did something with text." (not specific)

- "Explored the menu." (doesn't describe the function)

UTPUT FORMAT (JSON):

{{ "function-summary": "One or two sentences clearly describing the unit function, its trigger (buttons/menus), and its observable effect" }}

Return ONLY valid JSON. The function-summary field is CRITICAL and must be high-quality.

Feedback Prompt: Error Attribution (Action)

You are a feedback model for GUI exploration with expertise in action sequences and UI interaction patterns.
CONTEXT:

Exploring: {*node-path*}

Instruction: {instruction}

Actions: {history-actions-text}

Stop reason: {history-summary}

Classification: {reasoning}

Screenshots: BEFORE (state before actions) and AFTER (state after actions)

SITUATION 3: Reasonable Instruction, Problematic Actions

The exploration instruction is valid, but the ACTION SEQUENCE has issues (wrong targets, missing steps, ineffective execution).

{*fine-grained-actions-text*}

YOUR TASK:

Provide ACTIONABLE feedback to fix the action sequence. If the task requires fine-grained actions, you MUST select the appropriate action from the list above and provide its specific instructions.

CRITICAL REQUIREMENTS:

1. **Use Your Prior Knowledge:** Apply your understanding of typical UI interaction patterns to diagnose what went wrong.

2. **Careful Analysis:**

- Compare what SHOULD have happened vs. what DID happen

- Identify specific problems: wrong element grounded, missing prerequisites, incorrect order, etc.

- Determine if a fine-grained action is needed

3. **Fine-Grained Action Selection:**

- If the task requires a fine-grained action, you MUST:

* Choose the MOST APPROPRIATE action from the available fine-grained actions list above

* Include the EXACT action-instruction steps from that action in your feedback

* Include the action-primitive code if helpful for understanding

- DO NOT suggest retrying without providing the fine-grained action template

4. **Feedback Structure When Fine-Grained Action Required:**

a) **Problem Diagnosis**: What went wrong and why

b) **Selected Fine-Grained Action**: Name of the chosen action (e.g., “FineGrainedActions.select-text-span”)

c) **Action Instructions**: Copy the exact action-instruction steps from the selected action

d) **Action Primitives**: Copy the exact action-primitive code from the selected action

e) **FINE-GRAINED-REQUIRED: Yes** - Mark explicitly

5. **Feedback Structure When No Fine-Grained Action Needed:**

a) **Problem Diagnosis**: What went wrong

b) **Suggested Fix**: Concrete correction steps using standard actions

c) **FINE-GRAINED-REQUIRED: No**

OUTPUT FORMAT (JSON):

{{

“feedback”: “Your detailed feedback following the structure above”,

“fine-grained-required”: true or false,

“selected-fine-grained-action”: “Action name if applicable, otherwise omit this field”

}}

Example with fine-grained action:

{{

“feedback”: The agent is instructed to select a target text span, however, due to the fine-grained nature of the action, the action model failed to produce the action successfully.

Selected Fine-Grained Action:

FineGrainedActions.select-text-span Action Instructions:

1. Move the cursor to the start of the target text span in the editor.

2. Press and hold the left mouse button.

3. Drag the cursor to the end of the target text span.

4. Release the left mouse button to finalize the selection.

Action Primitives:
import pyautogui; pyautogui.moveTo(START-X, START-Y, duration=0.1);

pyautogui.mouseDown(); pyautogui.moveTo(END-X, END-Y, duration=DURATION);

pyautogui.mouseUp(); FINE-GRAINED-REQUIRED: Yes”,

“fine-grained-required”: true,

“selected-fine-grained-action”: “FineGrainedActions.select-text-span”

}}

Example without fine-grained action:

{{

“feedback”: “Problem: The agent clicked the wrong button.

Suggested Fix: Click the correct ‘Save’ button instead of ‘Cancel’.

FINE-GRAINED-REQUIRED: No”,

“fine-grained-required”: false

}}

Return ONLY valid JSON. If fine-grained action is required, you MUST select one from the available list and provide its instructions.

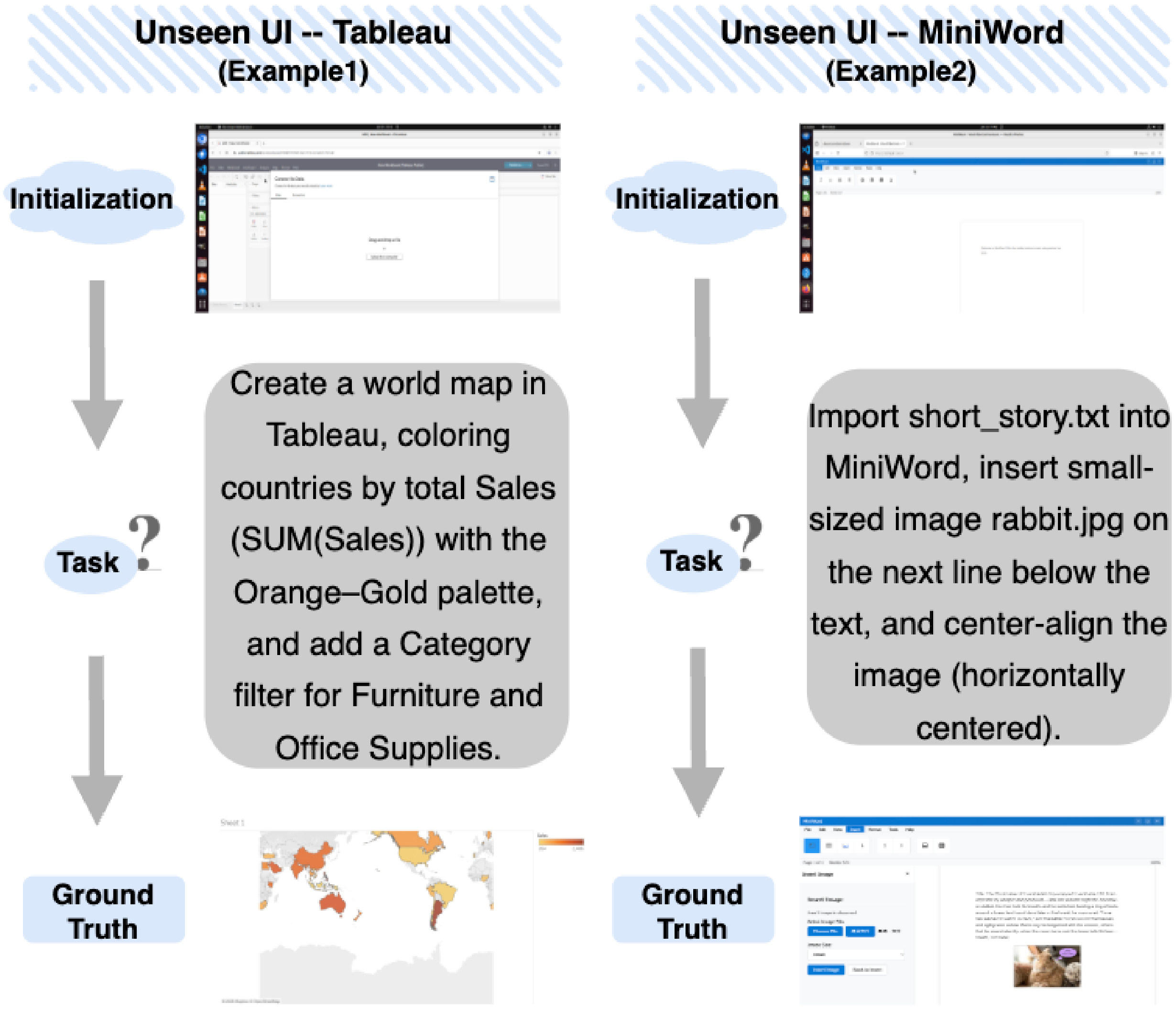

Unseen UI -- Tableau (Example1)
Unseen UI -- MiniWord (Example2)
Initialization
Initialization
Task
Create a world map in Tableau, coloring countries by total Sales (SUM(Sales)) with the Orange–Gold palette, and add a Category filter for Furniture and Office Supplies.
Task
Import short_story.txt into MiniWord, insert small-sized image rabbit.jpg on the next line below the text, and center-align the image (horizontally centered).
Ground Truth
Ground Truth

Figure 6. This image presents two representative examples from the Unseen UI category. The left example shows a task in Tableau, where the agent is required to create a world map colored by total sales and apply a category filter, illustrating generalization to a professional data visualization interface with non-standard interaction patterns. The right example shows a task in MiniWord, a custom-designed text editor with novel UI layouts, where the agent must import a text file and insert and align an image below the text. For each example, we show the initial environment state, the task instruction, and the corresponding ground-truth outcome.

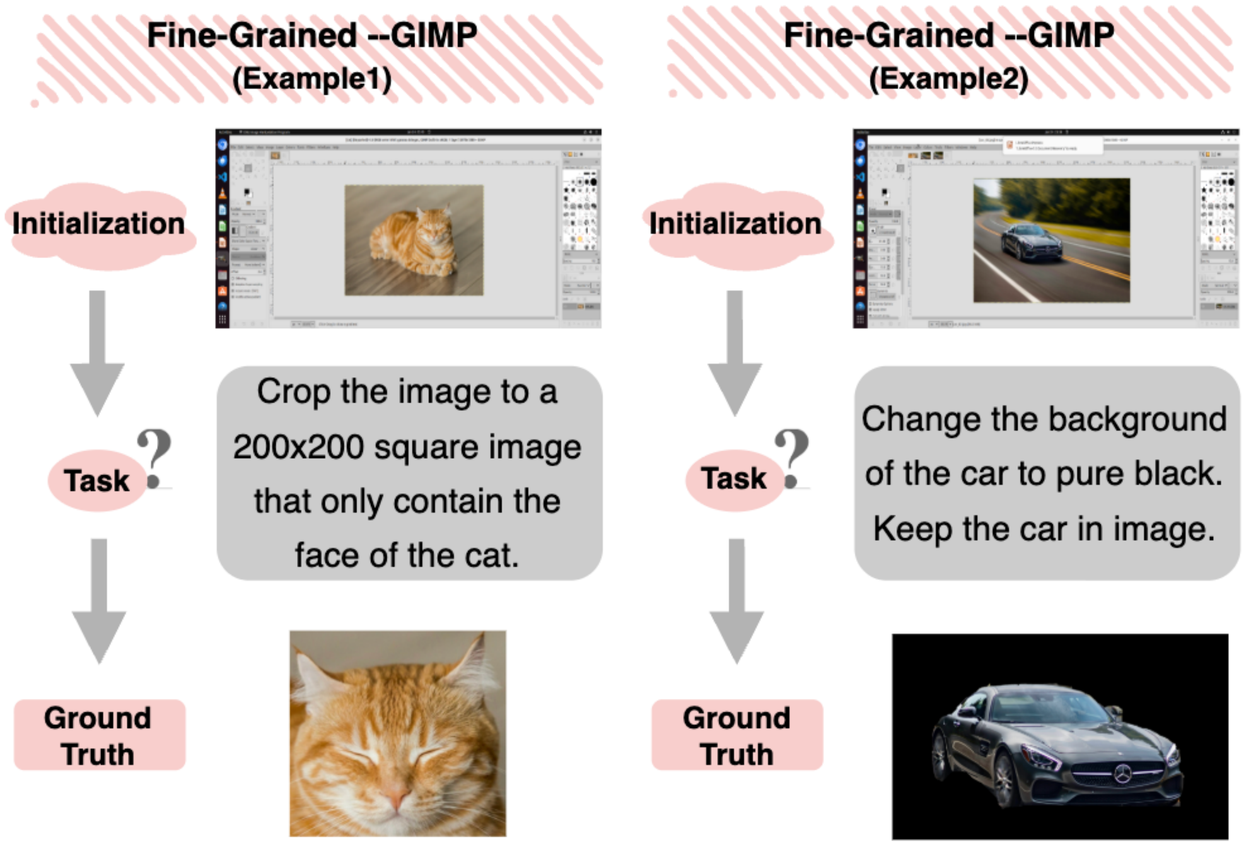

Fine-Grained --GIMP
(Example1)
Initialization
Task
Crop the image to a 200x200 square image that only contain the face of the cat.
Ground Truth
Fine-Grained --GIMP
(Example2)
Initialization
Task
Change the background of the car to pure black. Keep the car in image.
Ground Truth

Figure 7. Two examples from the *Fine-Grained Action Execution* category in OSExpert-Eval (GIMP). The left example requires cropping an image into a 200×200 square that tightly contains only the cat's face, demanding precise region selection and boundary control. The right example requires removing the background and setting it to pure black while keeping the car intact, which involves accurate foreground segmentation and careful low-level editing. For each example, we show the initial environment state, the task instruction, and the corresponding ground-truth outcome.