Appendices

In this supplementary material, we present additional details and clarifications that are omitted in the main text due to space constraints.

- Appendix A Limitations.
- Appendix B Dataset Details.
- Appendix C Implementation Details.
- Appendix D Inverse Dynamics Model Architecture.
- Appendix E Inverse Dynamics Model Training.
- Appendix F More Results.

A. Limitations

While our framework demonstrates strong performance, several opportunities for extension remain. Our inverse dynamics model (IDM) currently focuses on a core set of primitive actions such as `click`, `type`, `move`, and `scroll`. It does not yet handle keyboard shortcut or hotkey based actions (*e.g.*, `Ctrl+C` or `Ctrl+Z`), as such events are rarely captured in web based state transition data and therefore lack corresponding visual context. Nevertheless, most hotkey operations have direct graphical interface equivalents, such as selecting menu items or clicking toolbar icons, allowing the IDM to retain near complete functional coverage of real world tasks. Future work could incorporate application based data collection or key event logging to expand the action space and support shortcut based interactions more efficiently.

Second, our retrieval framework retrieves demonstrations at the granularity of full tasks. While effective, this may not always align with the granularity needed by an agent during execution. Future work could explore mechanisms to automatically merge shorter tasks into longer workflows, or split lengthy tutorials into more targeted subtrajectories. Such advances would enable finer-grained retrieval and more flexible trajectory construction, ultimately improving the adaptability of our approach.

We view these limitations not as fundamental barriers but as natural opportunities to further enhance the scalability and generality of our framework.

B. Dataset Details

B.1. Applications by Category

We selected seven categories: Productivity, Programming, Design, Screen Editing, Audio Production, System Utilities, and Science & Data. These categories span a broad range of realistic computer use. Productivity tools (*e.g.*, Microsoft Office, Google Workspace) cover everyday document and collaboration tasks, while Programming environments (*e.g.*, VS Code, Jupyter) capture software development workflows. Design (*e.g.*, Photoshop, Figma), Screen Editing (*e.g.*, Premiere Pro, OBS Studio), and Audio Production (*e.g.*, Audacity, FL Studio) extend to creative domains with specialized interfaces. System Utilities (*e.g.*, Task Manager, Finder, Docker) test low-level system interaction, and Science & Data tools (*e.g.*, MATLAB, Tableau, SPSS) represent analytical and visualization tasks.

Table 7. Applications grouped by category.

Category	Applications
Productivity	Microsoft Office, Google Workspace, Notion, Evernote, OneNote, Trello, Asana, ClickUp, Monday.com, Slack, Microsoft Teams
Programming	VS Code, PyCharm, IntelliJ IDEA, Eclipse, Android Studio, Xcode, Jupyter Notebook, Google Colab, RStudio, Sublime Text, Atom, GitHub Desktop
Design	Adobe Photoshop, Adobe Illustrator, Adobe XD, Figma, Sketch, Canva, CorelDRAW, Inkscape, Gimp
Screen Editing	Adobe Premiere Pro, Final Cut Pro, DaVinci Resolve, Camtasia, OBS Studio, ScreenFlow, Filmora, iMovie
Audio Production	Audacity, Adobe Audition, FL Studio, Logic Pro X, Ableton Live, Pro Tools, Cubase, GarageBand
System Utilities	Windows Task Manager, PowerShell, macOS Finder, Activity Monitor, Disk Utility, Linux Terminal, Docker, VirtualBox, CCleaner, WinRAR, 7-Zip
Science & Data	MATLAB, Mathematica, SPSS, SAS, Tableau, Power BI, Google Colab, Jupyter Notebook, Stata, RapidMiner

Applications within each category were chosen for their widespread adoption, abundant tutorial availability on YouTube, and ability to showcase the diverse interaction challenges agents must master. While we focused on these applications, our method is not restricted to them: additional data can be generated from any new tutorial videos available on the web. The distribution of applications is in Table 7.

C. Implementation Details

C.1. State-Transition Data Collection

This section provides additional details of the automated pipeline used to gather the 600K state-transition pairs (O_t, a_t, O_{t+1}) described in the main paper. The system consists of four components: an entry-point sampler, a browser controller, an action executor, and a transition logger. All data were collected using Playwright with a fixed viewport of 1280×720 pixels and 48 parallel workers.

Entry-point sampling. We sample URLs from the March 2025 snapshot of the Common Crawl index [1] (CC-MAIN-2025-13). We retain only HTML documents, filter out pages smaller than 5 KB, and discard URLs that trigger login walls or CAPTCHA challenges. To reduce domain imbalance, we cap the number of visits to 30 pages per domain. Each browsing session begins from a randomly chosen entry point, ensuring high visitation diversity across the crawl.

Action sampling policy. At each timestep, the pipeline samples an action from our six primitives: `click`, `release`, `scroll`, `type`, `wait`, and `move`. We empirically bias the sampling distribution toward common web

[1] https://commoncrawl.org/blog/march-2025-crawl-archive-now-available

interactions: click (∼45%), scroll (∼20%), move (∼15%), type (∼8%), wait (∼2%), and release (∼10%). Coordinates for `click` and `move` actions are drawn uniformly over the viewport and jittered around visible UI elements obtained from Playwright's accessibility tree. Scroll magnitudes follow a fixed value (500 px). Text for `type` actions is sampled from a curated list of ∼5K frequently used English words [2].

Transition recording. For each action, we record the pre- and post-action screenshots (O_t, O_{t+1}), the action primitive, all associated parameters, the current page URL, and a timestamp. Transitions where O_t and O_{t+1} differ by fewer than 50 nonzero pixels (excluding `wait`) are removed to avoid degenerate interactions. Each action is executed with a 5 s timeout; failed executions or delayed page loads are logged and the transition is discarded.

Error handling and session resets. The system automatically detects HTTP 4xx/5xx responses, navigation loops, frozen page states, and browser crashes. Upon such events, the session is terminated and restarted with a fresh entry point. To promote diversity, each session is limited to 15 actions before being reset, preventing oversampling from any single page.

Scale and runtime. Data collection was run on CPU-only machines (32 cores, 64 GB RAM). Across 48 parallel workers, the pipeline collected approximately 600K high-quality transitions over 3.5 days. After filtering, the resulting dataset covers a wide range of layouts, domains, and dynamic interaction patterns encountered in real web environments.

C.2. Video Retrieval

To build a large-scale dataset of application demonstrations, we require a method to identify relevant tutorial videos from the web. YouTube is a natural source since it contains abundant tutorials across productivity, programming, design, and other domains. However, naively searching by task description may yield irrelevant or entertainment-focused videos. To address this, we designed a dedicated prompt for generating targeted search queries.

The prompt (shown below) instructs a language model to act as an expert in YouTube search, taking as input a task description and a list of related applications. It outputs a short and effective query that emphasizes tutorials, how-to videos, and instructional content. By constraining queries to be concise and domain-specific, this approach improves retrieval precision and reduces noise from unrelated videos.

[2] https://www.vocabulary.com/lists/154147

Prompt for Video Retreival Query Generation

You are an expert at creating YouTube search queries. Given a task instruction and related applications, create a concise, effective search query that will find relevant tutorial videos.

Task: {instruction}
Related Applications: {related_apps}

Create a search query that would find helpful tutorial videos for this task. Focus on tutorial, how-to, or instructional content. Keep it concise (under 10 words).
Search query:

C.3. Video Filtering

After retrieving candidate tutorials, many videos still contain irrelevant or low-quality content such as talking-head introductions, presentation slides, or animated transitions. To ensure that our dataset is composed of high-quality screen recordings that clearly demonstrate application use, we apply a filtering step.

We design a prompt that instructs a language model to act as a visual classifier. Given a single frame from a video, the model assigns both a categorical label (*e.g.*, clean screencast, zoomed screencast, talking head) and a quality score between 0.0 and 1.0. We retain only those videos where the average frame score exceeds 0.8, which empirically yields a reliable set of clean tutorial screencasts. This threshold balances recall and precision: it removes noisy or non-screencast content while retaining a broad coverage of genuine tutorials.

Prompt for Video Filtering

You are a visual classifier helping to filter video tutorial frames for clean screencast content.
Your task is to classify an input image (a single frame from a video) and provide a quality score.

Classify the image into one of these categories:
1. Clean Screencast: Full desktop screen showing software interface, application window, code editor, browser, or terminal. Clear, unzoomed view of the entire screen or application window.

2. Zoomed Screencast: Screenshot that has been zoomed in or cropped, showing only part of the screen or interface elements.

3. Animated/Transition: Frames with animations, transitions, intro/outro effects, or visual effects that are not static screencast content.

4. Talking Head: Person's face or upper body from webcam, typically in corner or overlay.

5. Slide/Presentation: Static presentation slide, diagram, or text-heavy content.

6. Other: Content that doesn't fit the above categories.

For each classification, also provide a quality score from 0.0 to 1.0: - 1.0: Perfect clean screencast - 0.8-0.9: Good screencast with minor issues - 0.6-0.7: Acceptable screencast - 0.4-0.5: Poor quality or partially zoomed - 0.0-0.3: Very poor or not screencast

Return your response in this format: Category: [category name] Quality: [score] Reason: [brief explanation]

C.4. In-context Learning

To evaluate W&L under in-context learning (ICL), we augment the default prompts used by state-of-the-art OSWorld agents with a small number of structured demonstration trajectories. The OSWorld repository provides model-specific system prompts in OS-World Github [3], which define the action format, reasoning style, and interface conventions. We prepend our exemplars to these prompts without modifying any downstream agent logic.

For a given evaluation task, we retrieve three relevant trajectories from our video corpus and convert each into a sequence of *(observation, thought, action)* triples. These triples are formatted using the exact conventions defined by OSWorld agents, including the same observation style, reasoning structure, and action specification. We prepend the resulting exemplars to the agent prompt by placing them inside a dedicated instruction block that appears immediately *before* the OSWorld rules and reasoning instructions. This placement ensures that the agent first learns the desired behavior pattern from the exemplars, and then applies the original constraints that follow.

ICL Example Prompt Block

Example Trajectories
Below are some example task trajectories showing how to observe, think, and act. Study these examples carefully to understand:

- how to provide detailed observations of the screen state,
- how to structure your reasoning process step by step,
- how to format action code with clear comments,
- how to choose appropriate coordinates based on visual elements,
- when to use special codes like `WAIT`, `DONE`, or `FAIL`,
- how to recover from errors by adjusting your approach.

These examples demonstrate similar tasks to what you'll be performing. Use them as a reference for the level of detail expected in your observations and the reasoning process for choosing actions. Pay attention to how coordinates are selected based on element positions and how actions are adapted when previous attempts don't succeed.

Example {`example_num`}: {`task_description`}
Step {`step_num`}:
Screenshot: `[Image showing the current state]`
Thought: {`thought_text`}
Action:

```
{action_code}
```

Example of Full ICL Prompt with O3 Agent. In Appendix F.2 we show how a set of exemplar trajectories is prepended to the O3 agent instructions. The exemplars appear after the initial instruction, followed immediately by the rule and reasoning sections used by the OSWorld agent.

C.5. Models

For in-context learning evaluations we query API-based models using their latest public versions: Google Gemini 2.5 Flash (`gemini-2.5-flash`), OpenAI o3 (`o3-2025-04-16`), and Anthropic Claude 4 Sonnet (`claude-4-sonnet-20250514`). We use deterministic decoding with temperature set to 0.0.

For IDM training, we use the AdamW optimizer with a learning rate of $3e{-}4$, batch size 256, and cosine learning rate decay. Training is run for 15 epochs on $8{\times}$A100 GPUs (80GB) with gradient clipping at 1.0 and mixed-precision (bfloat16). For supervised fine-tuning of CUAs, we follow the official training recipes from UI-TARS-1.5 and Qwen 2.5-VL, adapting batch size to fit the same hardware setup.

C.6. Action Space and Primitive Composition

The inverse dynamics model (IDM) predicts atomic GUI interaction primitives from observed state transitions. The primitive action space consists of *Click*, *Release*, *Move*, *Scroll*, *Type*, and *Wait*. These primitives serve as a low-level interaction vocabulary and are not executed directly in the environment.

Instead, predicted primitive sequences are converted into executable environment actions through a deterministic composition layer implemented using `pyautogui`. This layer maps temporally ordered primitive sequences to environment-compatible commands. For instance, the sequence `Click`→`Release` corresponds to a standard mouse click, while `Click`→`Move`→`Release` corresponds to drag-based interactions such as `moveto` or

[3] https://github.com/xlang-ai/OSWorld/blob/main/mm_agents/prompts.py

`dragto`. The composition stage enforces a consistent interface between the model output space and the environment action space.

Primitive distinctions are introduced only when they induce observable changes in the visual state. When multiple interaction sequences are visually indistinguishable under the observation stream, they are resolved to the minimal primitive sequence consistent with the observed transition. This constraint avoids introducing latent action distinctions that cannot be inferred from visual evidence.

To ensure compatibility with downstream execution, the IDM primitive space is aligned with a subset of the agent and environment action spaces through the deterministic mapping described above. This alignment constrains the prediction space to executable interactions and mitigates degenerate solutions during training by preventing the model from collapsing onto non-executable or redundant action representations.

D. Inverse Dynamics Model Architecture

D.1. Vision Encoder

Model Selection. We employ SigLIP-2 Base (patch size 16) with the NaFlex variant as our vision encoder [34]. This architecture was specifically chosen for its superior performance on OCR and document-understanding benchmarks, which closely aligns with the requirements of computer interface understanding where precise text recognition and spatial localization are critical.

Input Processing. Each observation O_t is a 1000×1000 RGB screenshot. We utilize the NaFlex preprocessing strategy with `max_num_patches=1024`, which preserves the square aspect ratio while resizing the input to approximately 704×704 pixels before patch extraction. This configuration yields exactly 1024 visual tokens (a 32×32 grid of 16×16 patches), providing sufficient spatial resolution for identifying small UI elements such as buttons, icons, and text labels while maintaining computational efficiency.

Feature Extraction. For the pair (O_t, O_{t+1}), we process each observation independently through the frozen SigLIP-2 encoder to obtain visual embeddings $\mathbf{v}_t, \mathbf{v}_{t+1} \in \mathbb{R}^{1024\times768}$. These embeddings preserve spatial information critical for coordinate prediction.

Rationale for NaFlex. Compared to the fixed-resolution SigLIP-2 variant at 512×512 (which would downsample our 1000×1000 input by 48%), NaFlex provides: (1) Higher effective resolution (∼704×704) for better preservation of fine-grained UI details, (2) Superior performance on text-heavy and structured visual content [34], and (3) Minimal aspect ratio distortion for square inputs. While fixed-resolution variants benefit from additional distillation stages, preliminary experiments showed NaFlex's advantages in OCR-like capabilities outweigh this trade-off for UI understanding.

D.2. Transformer Backbone

Temporal Feature Fusion. The patch-level features from (O_t, O_{t+1}) are concatenated along the sequence dimension, yielding a combined representation of shape $\mathbb{R}^{2048\times768}$. We prepend a learnable `[CLS]` token to this sequence. To distinguish temporal information, we add learnable temporal embeddings $\mathbf{e}_t, \mathbf{e}_{t+1} \in \mathbb{R}^{768}$ to the respective halves of the sequence.

Architecture. Our backbone consists of 4 transformer layers with the following specifications:

- **Hidden dimension**: 768 (matching SigLIP-2 output)
- **Attention heads**: 12
- **MLP hidden dimension**: 3072 (4× expansion)
- **Activation function**: GELU
- **Dropout**: 0.1 applied to attention weights and MLP outputs
- **Layer normalization**: Pre-norm configuration with $\epsilon = 10^{-6}$
- **Attention mechanism**: Standard scaled dot-product attention with causal masking disabled (full bidirectional attention)

The final `[CLS]` token representation $\mathbf{h}_{\text{cls}} \in \mathbb{R}^{768}$ serves as the global action embedding, while the patch-level outputs $\mathbf{h}_{\text{patch}} \in \mathbb{R}^{2048\times768}$ are used for coordinate prediction.

D.3. Prediction Heads

D.3.1. Action Classification Head

A simple two-layer MLP predicts the action type:

$$\mathbf{p}_{\text{action}} = \text{softmax}(\mathbf{W}_2 \cdot \text{ReLU}(\mathbf{W}_1\mathbf{h}_{\text{cls}} + \mathbf{b}_1) + \mathbf{b}_2) \quad (1)$$

where $\mathbf{W}_1 \in \mathbb{R}^{512\times768}$, $\mathbf{W}_2 \in \mathbb{R}^{6\times512}$, and $\mathbf{p}_{\text{action}} \in \mathbb{R}^6$ represents the probability distribution over the six action primitives: `click`, `release`, `scroll`, `type`, `wait`, and `move`.

D.3.2. Coordinate Prediction Head

Architecture. For location-based actions, we predict normalized coordinates discretized into 1000 bins for each axis. We employ separate prediction heads for x and y coordinates:

For the x-coordinate:

$$\mathbf{h}_x = \text{AvgPool}(\mathbf{h}_{\text{patch}}) \quad \text{(over spatial dimension)} \quad (2)$$

$$\mathbf{p}_x = \text{softmax}(\mathbf{W}_x \cdot \text{ReLU}(\text{LayerNorm}(\mathbf{h}_x))) \quad (3)$$

where $\mathbf{W}_x \in \mathbb{R}^{1000\times768}$ and $\mathbf{p}_x \in \mathbb{R}^{1000}$. The y-coordinate head is parameterized identically with separate weights $\mathbf{W}_y$.

Discretization Strategy. Screen coordinates are normalized to $[0, 1]$ and discretized into 1000 uniform bins. This

formulation offers several advantages: (1) Transforms regression into classification, enabling the use of cross-entropy loss which empirically provides stronger gradients than regression losses (*e.g.*, L1, L2), (2) Naturally handles multi-modal distributions when multiple valid click locations exist, (3) Provides calibrated uncertainty estimates via the output probability distribution.

Resolution Alignment. The 1000-bin discretization aligns naturally with our 1000×1000 input resolution, where each bin corresponds to approximately a 1-pixel region. This granularity is sufficient for UI interaction, as most clickable targets span multiple pixels.

D.3.3. Language Head

Architecture. For text-entry actions (`type`), we employ a lightweight autoregressive decoder based on GPT-2 [30].

Visual Conditioning. The decoder attends to the visual features via cross-attention. Specifically, at each decoder layer ℓ, we insert a cross-attention sublayer:

$$\text{CrossAttn}^{(\ell)}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V \quad (4)$$

where Q is derived from the decoder's hidden states, and K, V are projections of the concatenated visual features $[\mathbf{v}_t; \mathbf{v}_{t+1}]$ from the SigLIP-2 encoder. This design enables the model to ground its text generation in the observed visual transition.

Training Strategy. During training, we use teacher forcing with cross-entropy loss on the ground-truth text sequence.

E. Inverse Dynamics Model Training

This part provides the full definition of the training losses used for the inverse dynamics model (IDM) in W&L. The IDM predicts three types of supervision depending on the action: an action primitive, optional pixel coordinates, and optional text tokens. For completeness, we describe the outputs, losses, and training objective in detail below.

E.1. Model Outputs

Given two consecutive observations (O_t, O_{t+1}), where O_t and O_{t+1} are screen images before and after executing action a_t, the IDM produces the following outputs:

- **Action primitive**: A categorical distribution $p_\theta(a_t)$ over the set of supported primitives (e.g., `click`, `scroll`, `type`, `wait`, `move`).
- **Coordinate predictions**: For actions requiring a spatial position, the model predicts distributions $p_\theta(x_t)$ and $p_\theta(y_t)$ over discretized pixel bins $\{0, \ldots, 999\}$.
- **Typed text**: For text-entry actions, the IDM predicts the sequence of typed tokens $w_{1:L}$ autoregressively through $p_\theta(w_\ell \mid w_{<\ell}, O_t, O_{t+1})$.

Each action is routed only to the heads that apply to its supervision.

E.2. Overall Objective

The entire model is trained end-to-end using a multi-task objective that adds the losses from the applicable prediction heads:

$$\mathcal{L} = \lambda_{\text{cls}}\mathcal{L}_{\text{cls}} + \lambda_{\text{coord}}\mathcal{L}_{\text{coord}} + \lambda_{\text{lang}}\mathcal{L}_{\text{lang}}, \quad (5)$$

where λ_{cls}, λ_{coord}, and λ_{lang} are scalar weights (set to 1 by default). Loss terms are included only for actions with the corresponding supervision signal.

E.2.1. Classification Loss

Every action includes a primitive label a_t^{cls}, which is supervised using a standard cross-entropy loss:

$$\mathcal{L}_{\text{cls}} = -\log p_\theta(a_t^{\text{cls}} \mid O_t, O_{t+1}). \quad (6)$$

This term teaches the IDM to infer the underlying categorical action from the state transition.

E.2.2. Coordinate Loss

For location-based actions (such as `click`, `move`, or `type`), the IDM predicts normalized pixel coordinates (x_t, y_t), each discretized into 1000 bins. We define the coordinate loss as the sum of independent cross-entropy terms for the two axes:

$$\mathcal{L}_{\text{coord}} = -\log p_\theta(x_t) - \log p_\theta(y_t). \quad (7)$$

This term is omitted for actions that do not require coordinates.

E.2.3. Language Modeling Loss

For text-entry actions, the IDM predicts the token sequence $w_{1:L}$ using a left-to-right autoregressive decoder. The text loss is the usual negative log-likelihood over the sequence:

$$\mathcal{L}_{\text{lang}} = -\sum_{\ell=1}^{L} \log p_\theta(w_\ell \mid w_{<\ell}, O_t, O_{t+1}), \quad (8)$$

where L is the length of the typed sequence. This loss is skipped entirely for non-text actions.

Together, these losses encourage the IDM to reconstruct action primitives, spatial arguments, and typed text directly from observed UI transitions, providing a unified formulation for all interaction types.

F. More Results

F.1. What is the effect of data scale for supervised fine-tuning?

We study how scaling the number of training trajectories affects the performance of Qwen 2.5-VL on OSWorld. As shown in Table 9, success rates increase from 1.9% with the

Table 8. Detailed OSWorld category-wise task successes. W&L provides the strongest improvements in domains with abundant specialized tutorials (*e.g.*, Chrome, Gimp, VLC), while gains are smaller in domains requiring heavy text entry, rare actions, or fine-grained control.

Setting	Category	Model	chrome	gimp	lo_calc	lo_impress	lo_writer	multi_apps	os	thunderbird	vlc	vs_code	Total
ICL	General Models	Gemini 2.5 Flash	8	8	4	3	5	9	10	6	5	12	70
		+ **W&L**	**10 (+3)**	**10 (+2)**	4	5	5	9	10	**8 (+2)**	8 (+3)	12	**81 (+11)**
		o3	6	10	5	5	7	15	15	4	7	9	83
		+ **W&L**	**9 (+3)**	**13 (+2)**	**7 (+1)**	7	7	**18 (+1)**	15	4	**9 (+2)**	9	**98 (+9)**
		Claude 4 Sonnet	25	13	15	22	14	27	11	11	7	14	159
		+ **W&L**	**27 (+2)**	**15 (+2)**	15	22	14	27	11	11	**9 (+2)**	14	**169 (+6)**
	Agentic Framework	Jedi	26	21	19	21	15	32	13	12	10	13	182
		+ **W&L**	**29 (+3)**	**23 (+2)**	19	**23 (+2)**	15	32	13	12	**12 (+2)**	13	**191 (+9)**
SFT	Open-Weight Models	UI-TARS-1.5-7B	11	15	6	14	9	5	8	4	6	15	93
		+ **W&L**	**13 (+2)**	**17 (+2)**	**8 (+2)**	**16 (+2)**	9	**7 (+2)**	8	**4 (+2)**	**7 (+2)**	15	**104 (+14)**
		Qwen 2.5-VL 7B	4	1	0	0	2	0	0	2	2	0	7
		+ **W&L**	**12 (+8)**	**10 (+9)**	**3 (+3)**	**1 (+1)**	2	**1 (+1)**	**5 (+5)**	**4 (+2)**	**6 (+4)**	**4 (+4)**	**48 (+41)**

Table 9. Data scaling results on OSWorld with Qwen 2.5-VL. Performance improves as training data increases from 10k to 25k and the full dataset.

Model	Base	10k	25k	Full
Qwen 2.5-VL	1.9	3.3	4.9	13.0

base model to 3.3% with 10k trajectories, 4.9% with 25k trajectories, and 13.0% with the full dataset. The improvement is closer to exponential than linear, suggesting that a minimum critical mass of data is required before substantial gains emerge.

We hypothesize that this behavior arises because Qwen must learn both *grounding* and *planning* from the video-derived trajectories. With limited data, the model struggles to acquire either capability robustly, leading to only small improvements. Once enough trajectories are available, however, Qwen begins to effectively integrate grounding of UI states with coherent planning patterns, producing sharper gains. This indicates that further scaling of high-quality trajectories could unlock even larger benefits.

F.2. Which application domains benefit most from our data?

To better understand the strengths and limitations of our approach, we break down results by application domain on OSWorld. Table 8 reports task successes for general-purpose models (o3, Claude 4 Sonnet), the Jedi agentic framework, and the open-source model UI-TARS-7B, both with and without W&L exemplars or training data.

The largest improvements are observed in `chrome`, `gimp`, and `vlc`. These domains benefit strongly from specialized procedural knowledge that is well covered by online tutorials, such as configuring browser settings, editing images, or adjusting media player preferences. The presence of abundant, step-by-step demonstrations in these categories enables our pipeline to extract high-quality trajectories that transfer effectively to downstream agents.

By contrast, the gains are smaller in domains such as `vscode` and `os`, which often require extensive text entry or code manipulation—capabilities that are less easily captured by our current action set. Improvements are also limited in `thunderbird` and LibreOffice applications (`lo.calc`, `lo.writer`, `lo.impress`), where high-quality tutorials are scarce and tasks sometimes involve fine-grained interactions such as dragging objects or manipulating small interface elements.

Overall, this breakdown highlights a key property of our approach: it yields the largest benefits in domains where web tutorials are both plentiful and aligned with the action space of the agent, while leaving room for future extensions in text-heavy or fine-grained interaction domains.

Combined ICL Prompt for o3

You are an agent which follow my instruction and perform desktop computer tasks as instructed. You have good knowledge of computer and good internet connection and assume your code will run on a computer for controlling the mouse and keyboard. You are on Ubuntu operating system and the resolution of the screen is 1920x1080. For each step, you will receive a screenshot of the current computer screen and must predict the corresponding action.

Example Trajectories
Below are some example task trajectories showing how to observe, think, and act. Study these examples carefully to understand:
- how to provide detailed observations of the screen state,
- how to structure your reasoning process step by step,
- how to format action code with clear comments,
- how to choose appropriate coordinates based on visual elements,
- when to use special codes like `WAIT`, `DONE`, or `FAIL`,
- how to recover from errors by adjusting your approach.

These examples demonstrate similar tasks to what you will be performing. Use them as a reference for the level of detail expected in your observations and the reasoning process for choosing actions.

Example 1: Open a settings panel
Step 1:
Screenshot: `[Image showing desktop with a gear-shaped Settings icon in the left sidebar]`
Thought: `The Settings icon is visible in the sidebar. Clicking it should open the system settings window.`
Action:

```
# Click the gear-shaped "Settings" icon in the left sidebar
pyautogui.click(x=120, y=340, duration=1)
```

The following rules are important:
- If previous actions did not achieve the expected result, do not repeat them. Adjust your action or coordinates based on the new screenshot.
- Do not predict multiple clicks at once. Base each action only on what is currently visible.
- You cannot complete the task by outputting text. You must interact through mouse and keyboard. Return `Fail` when the task cannot be done.

Observation. Provide a detailed observation of the current computer state based on the full screenshot in an "Observation:" section. Include any potentially relevant information such as pop-ups, notifications, or error messages. You *must* output the observation before the thought.

Thought. Think step by step and describe the full reasoning process behind your next action. Your thought should include:
- a progress assessment and reflection on previous attempts,
- identification of errors or mismatches,
- enumeration of possible next actions based on visible state,
- analysis and selection of the best next action.

You *must* output the thought before the action code.

Action Format. Use `pyautogui` to perform actions, but do **not** use `pyautogui.locateCenterOnScreen` or `pyautogui.screenshot()`. Return exactly one line of Python code, preceded by a clear instruction comment. Return the code inside a code block:

```
```python
# instruction
pyautogui.click(x=..., y=..., duration=1)
```
```

Combined ICL Prompt for o3 (continued)

Special tokens.

- `WAIT`: when a pause is needed,
- `DONE`: when the task is completed,
- `FAIL`: when the task cannot be completed.

You have a maximum of 100 steps and the current step is {current_step} out of {max_steps}. My computer's password is '{CLIENT_PASSWORD}'. Use it when necessary for sudo operations. First, briefly reflect on the current screenshot and prior actions, then return only the code or special token.

Table 10. **W&L labeling example**

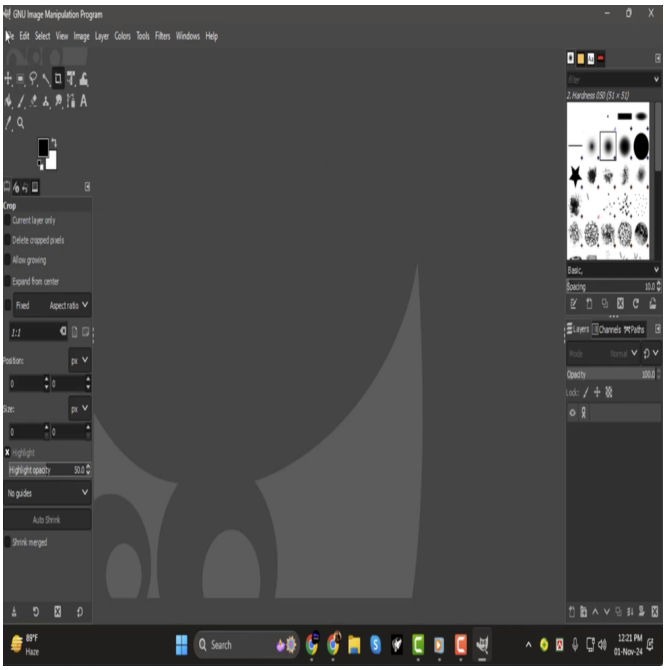
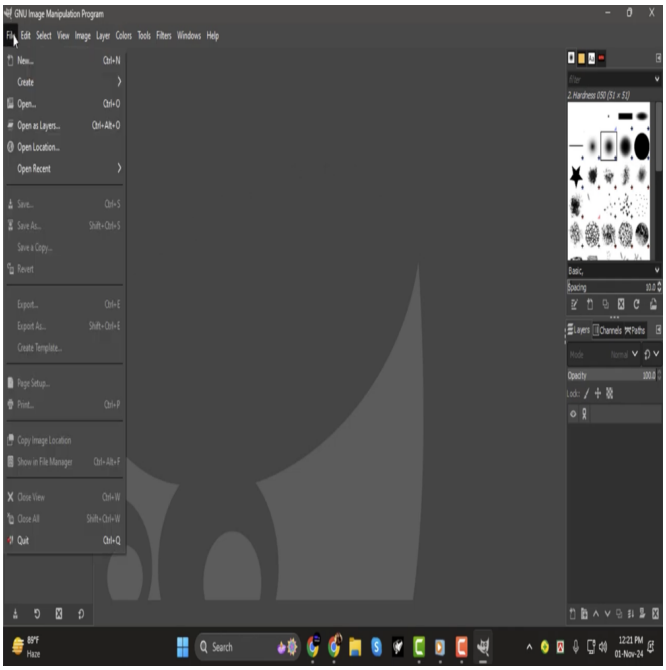
Task: Resize image in GIMP
Observation 1 (GIMP)

Action 1
click (8, 45)
Observation 2 (GIMP)

Action 2
move (34, 147)

Table 11. **W&L labeling example**

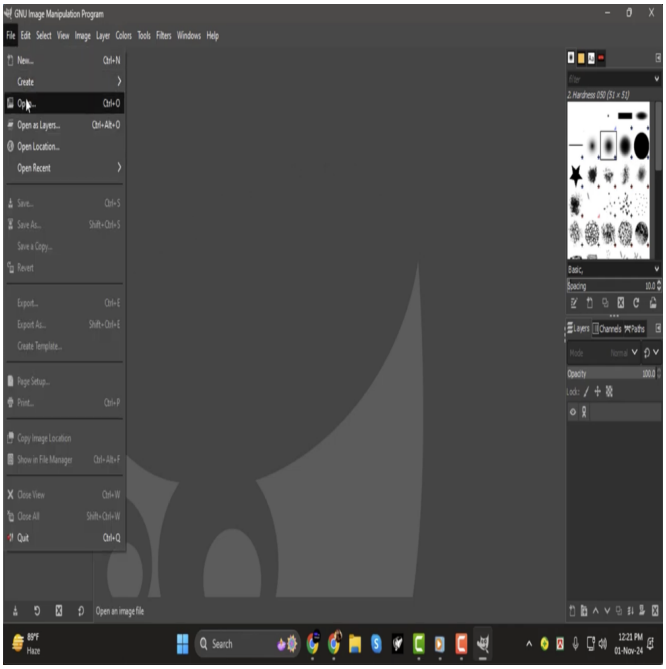
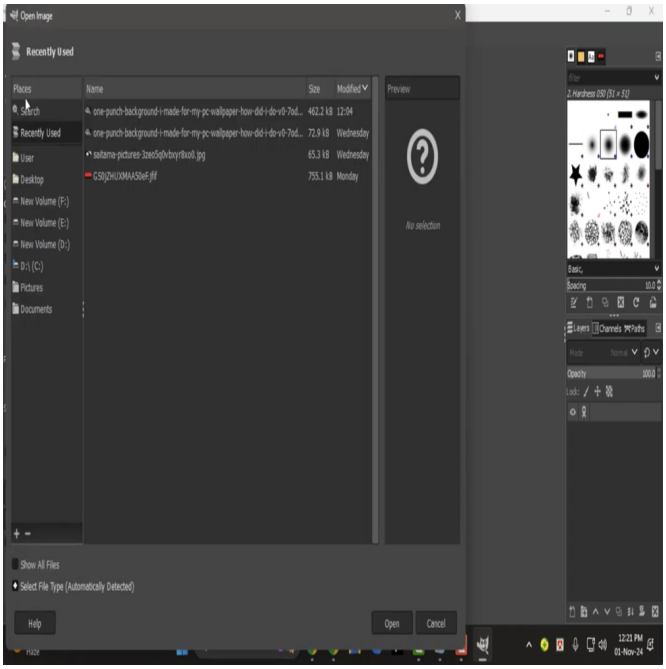
Observation 3 (GIMP)

Action 3
click (34, 147)
Observation 4 (GIMP)

Action 4
click (75, 426)

Table 12. **W&L labeling example**

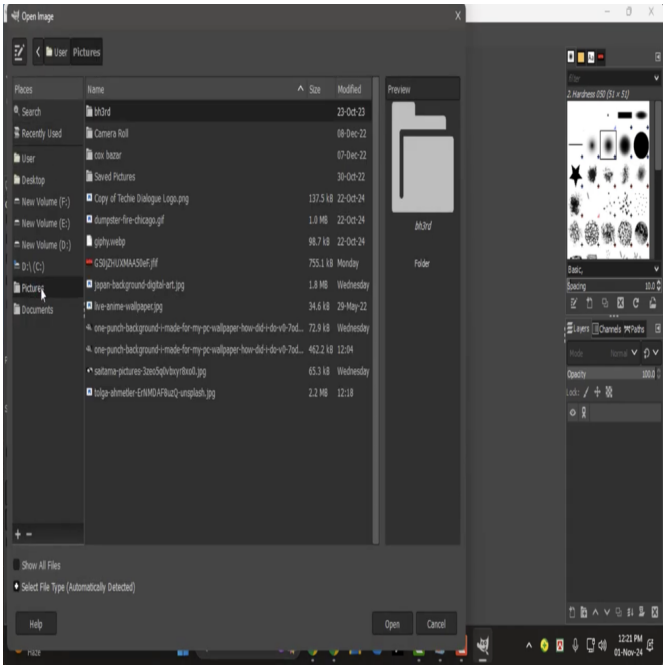
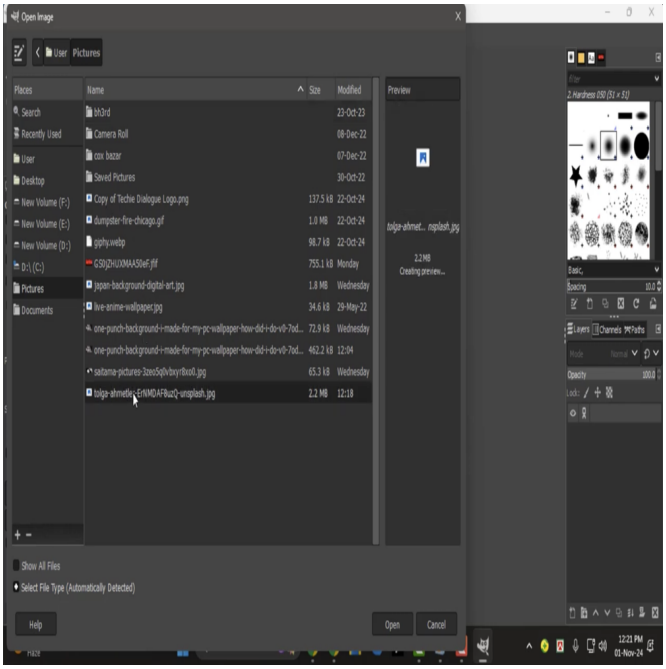
Observation 5 (GIMP)

Action 5
click (200, 594)
Observation 6 (GIMP)

Action 6
click (585, 929)

Table 13. **W&L labeling example**

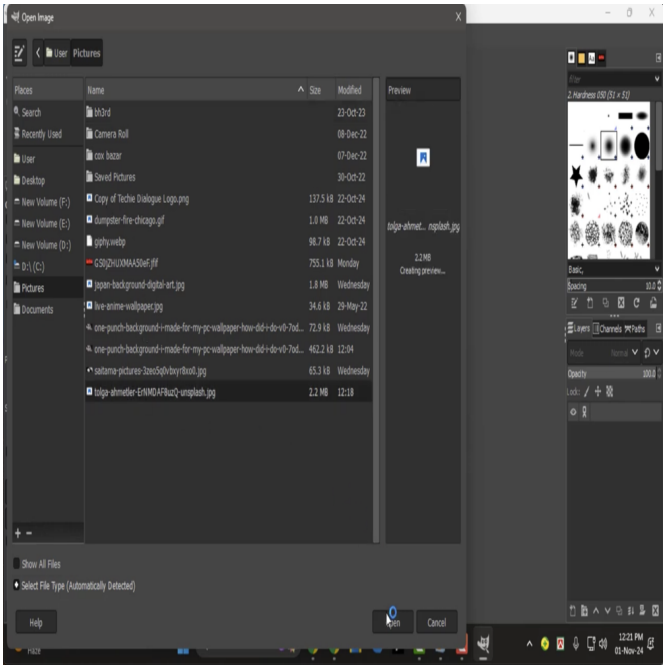
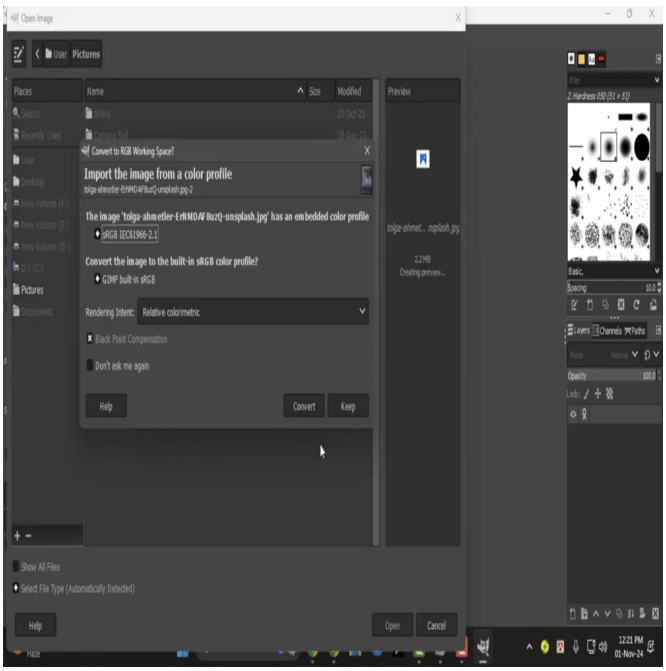
Observation 7 (GIMP)

Action 7
click (585, 929)
Observation 8 (GIMP)

Action 8
wait

Table 14. **W&L labeling example**

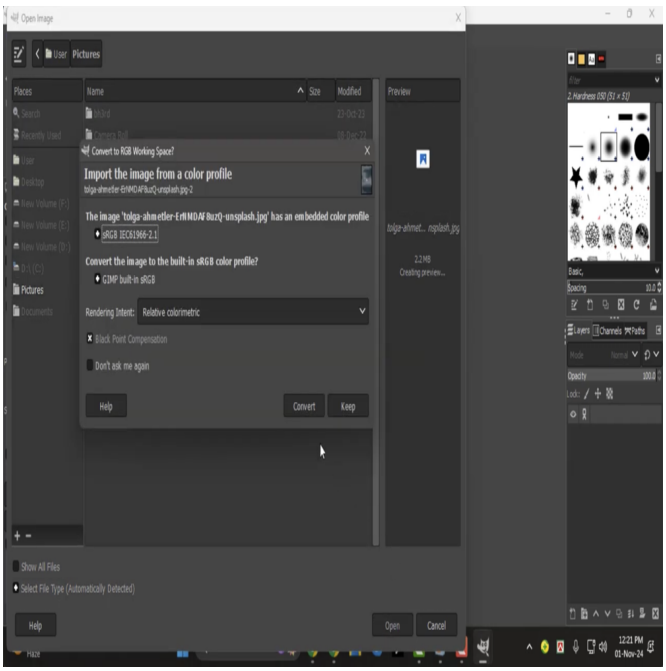
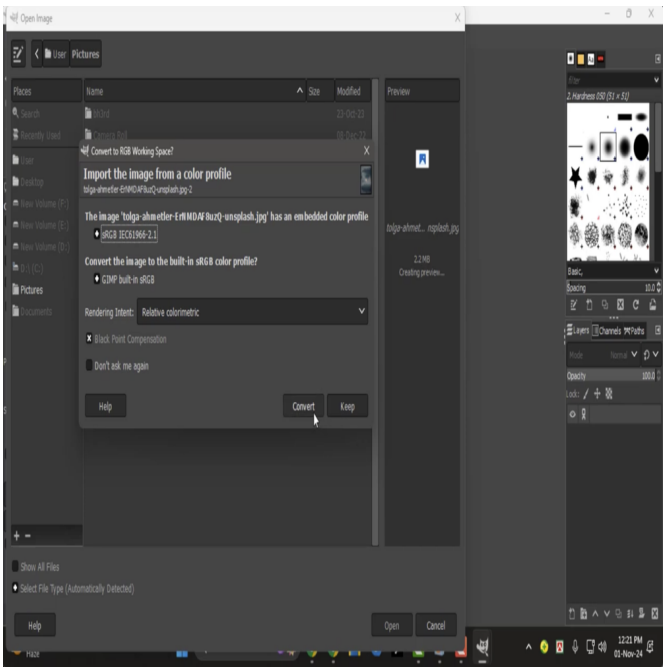
Observation 9 (GIMP)

Action 9
move (467, 612)
Observation 10 (GIMP)

Action 10
click (467, 612)

Table 15. **W&L labeling example**

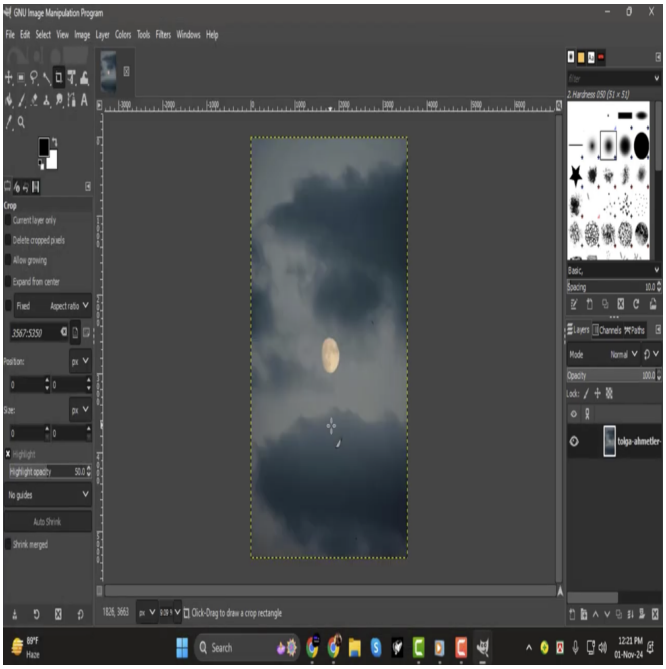
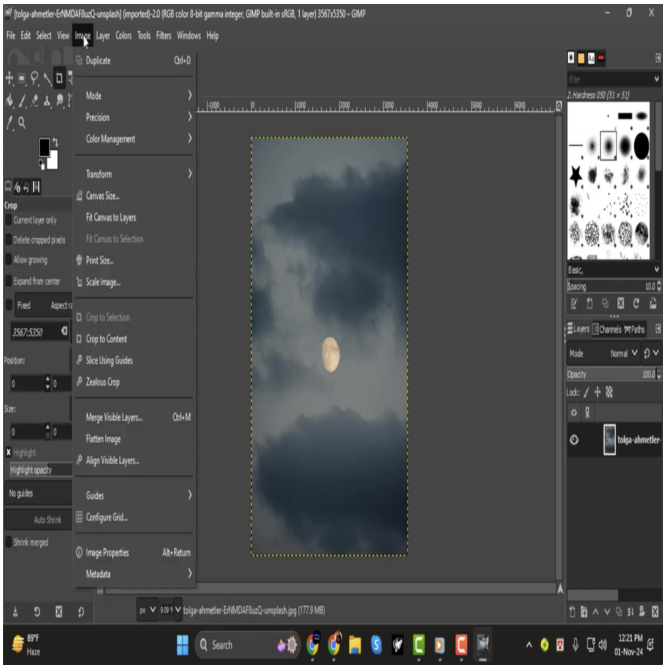
Observation 11 (GIMP)

Action 11
click (121, 46)
Observation 12 (GIMP)

Action 12
click (145, 426)

Table 16. **W&L labeling example**

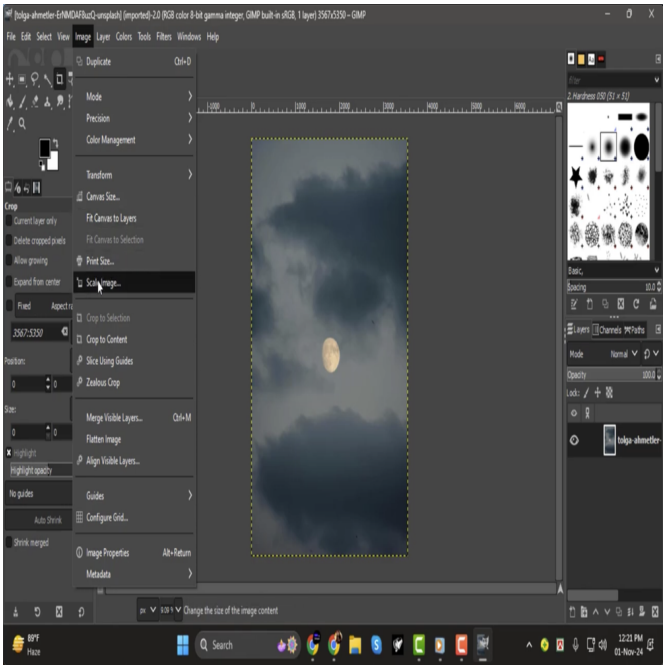
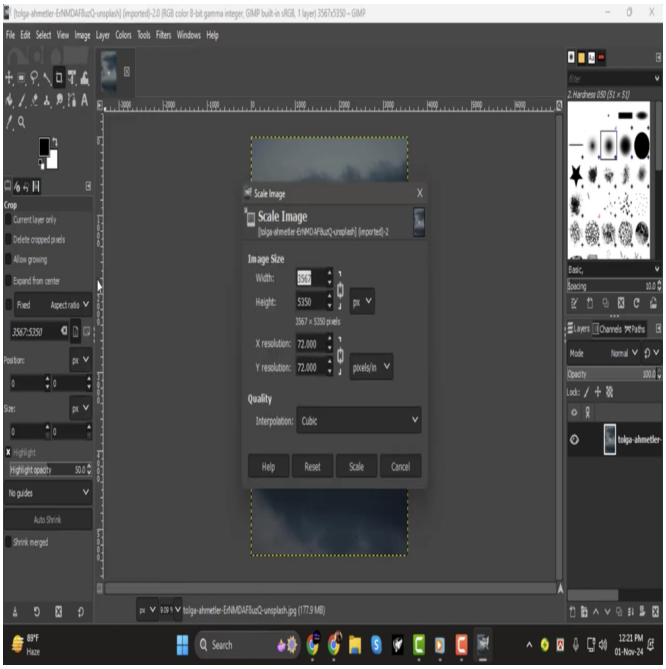
Observation 13 (GIMP)

Action 13
click (459, 411)
Observation 14 (GIMP)

Action 14
type (458, 412, ”5000”)

Table 17. **W&L labeling example**

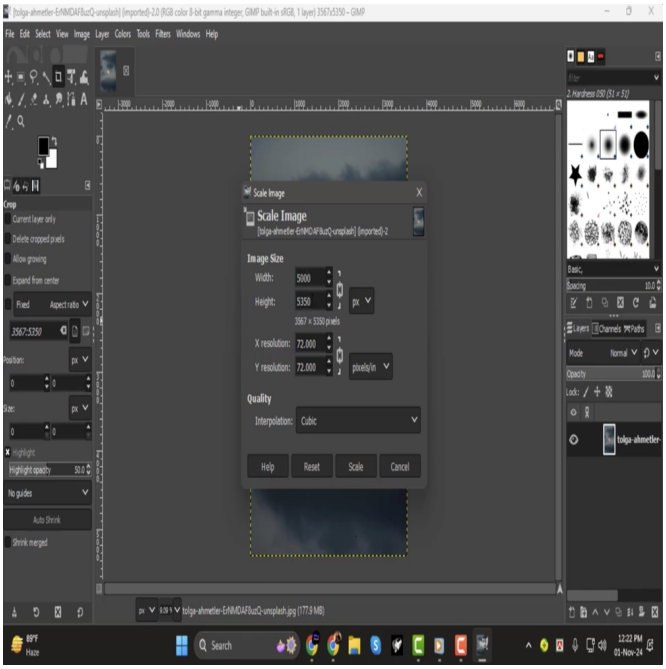
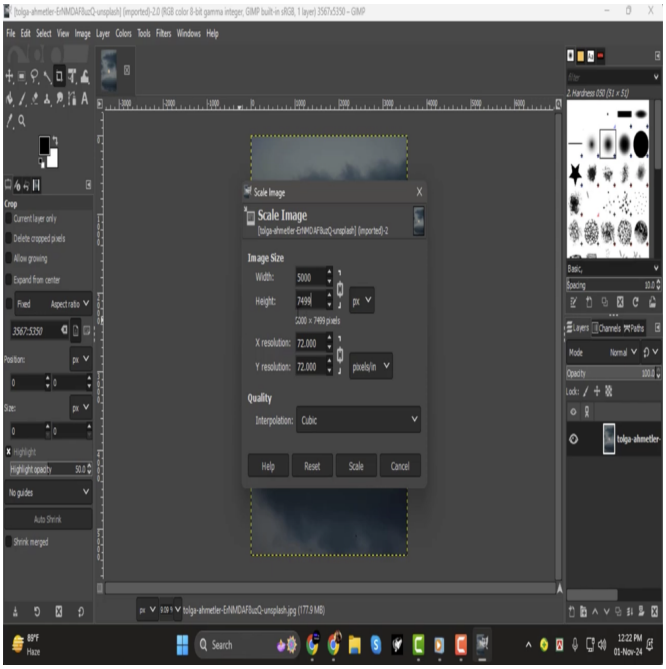
Observation 15 (GIMP)

Action 15
type (454, 446, ”7499”)
Observation 16 (GIMP)

Action 16
click (534, 704)

Table 18. **W&L labeling example**

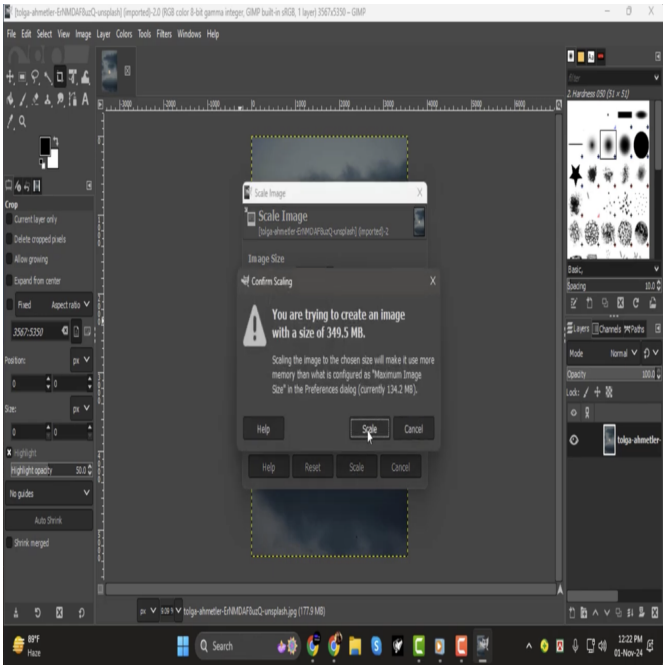
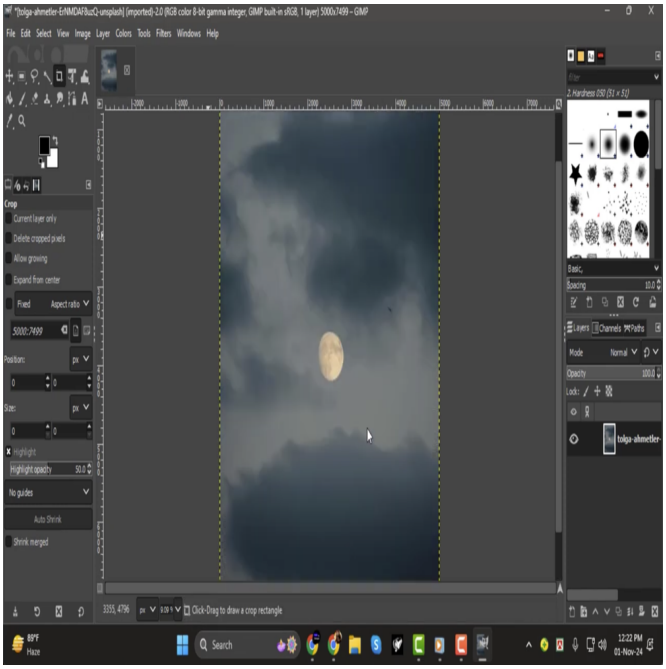
Observation 17 (GIMP)

Action 17
click (555, 647)
Observation 18 (GIMP)

Action 18
No action (Done)

Table 19. **OSWorld example (UI-TARS-1.5-7B w/ W&L)**

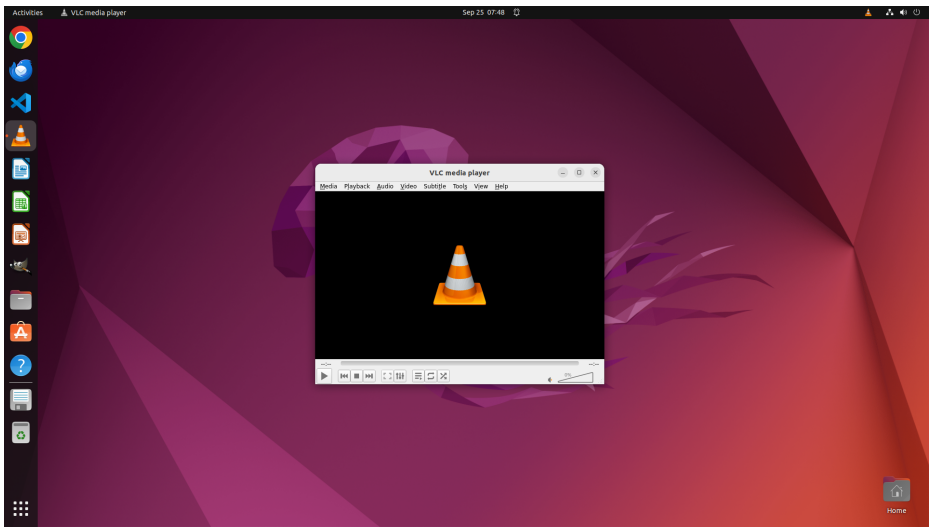
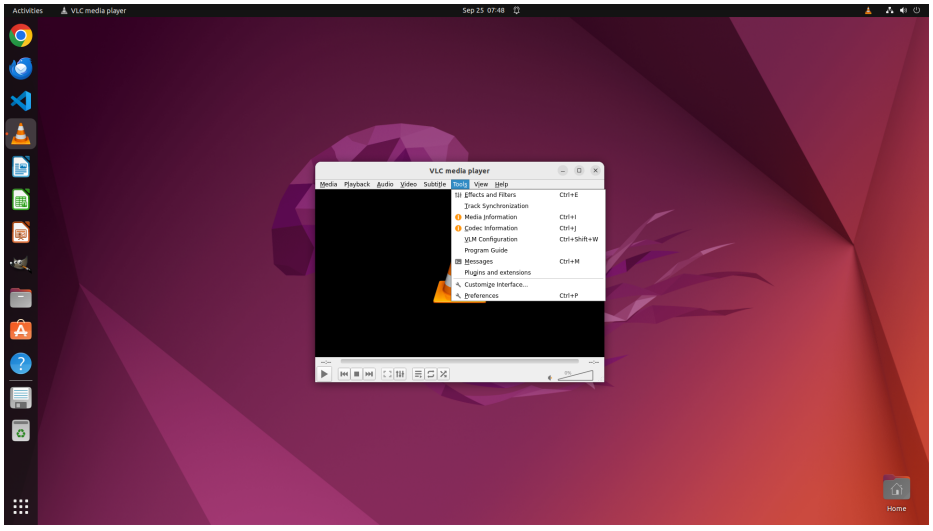
Task: Current volume is 125%. Can you increase it to 200% in VLC?
Observation 1 (VLC)

Action 1
import pyautogui pyautogui.click(941, 369)
Observation 2 (VLC)

Action 2
import pyautogui pyautogui.click(984, 602)

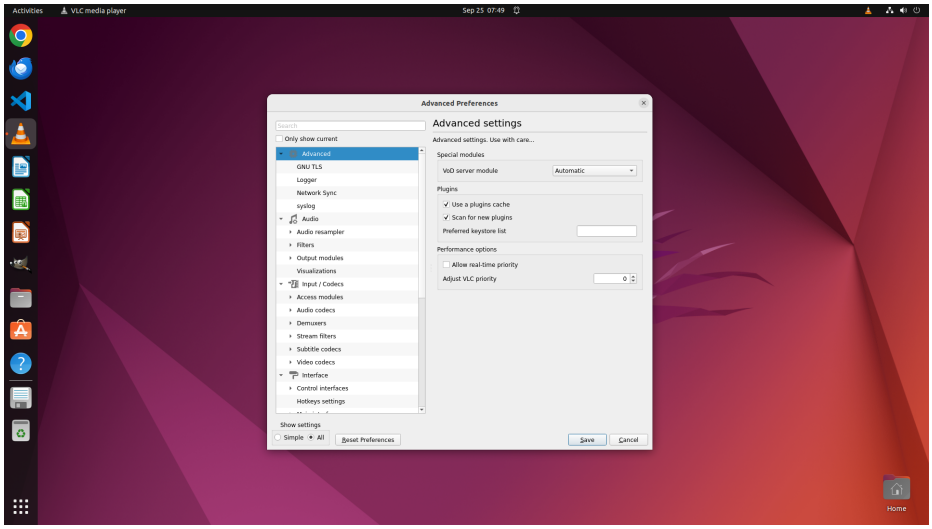
Table 20. **OSWorld example (UI-TARS-1.5-7B w/ W&L)**

Observation 3 (VLC)

Action 3

import pyautogui
pyautogui.click(635, 892)

Observation 4 (VLC)

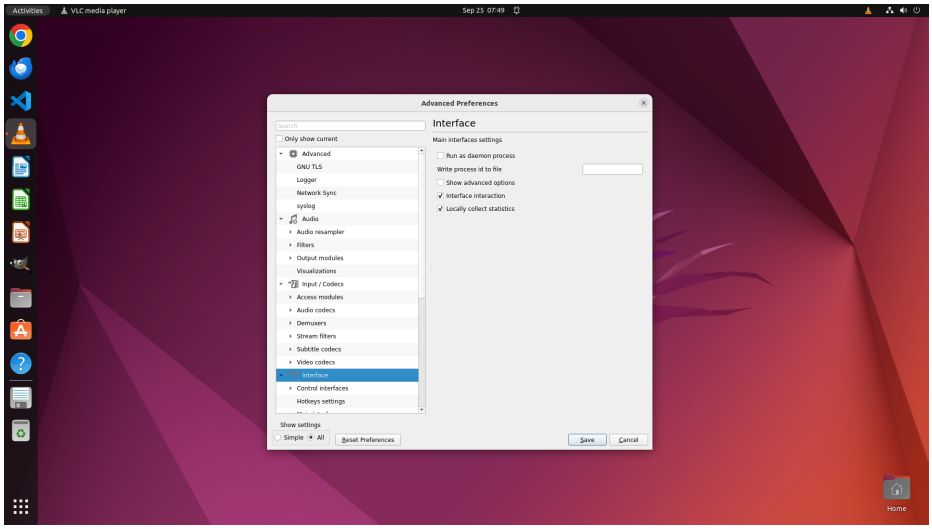


Action 4

import pyautogui
pyautogui.click(642, 763, button=”left”)

Table 21. **OSWorld example (UI-TARS-1.5-7B w/ W&L)**

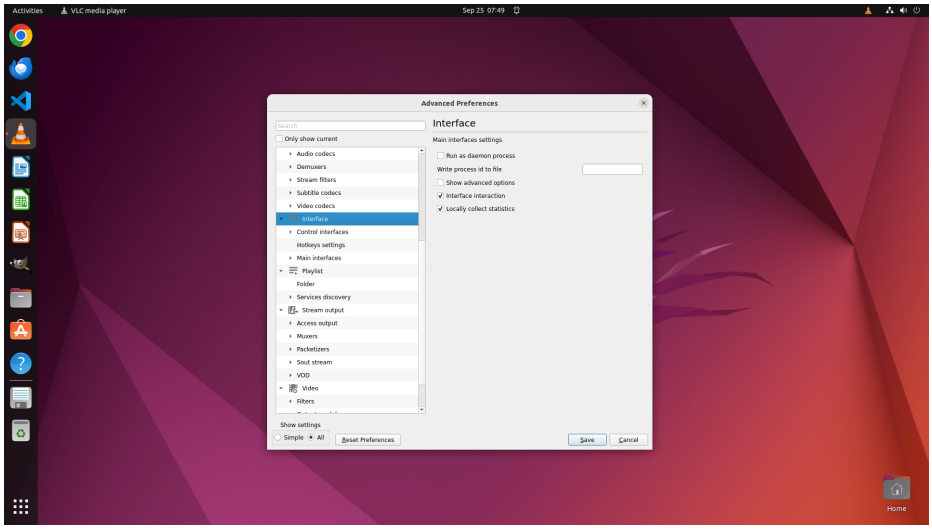
Observation 5 (VLC)



Action 5

import pyautogui
pyautogui.scroll(-50, x=722, y=596)

Observation 6 (VLC)

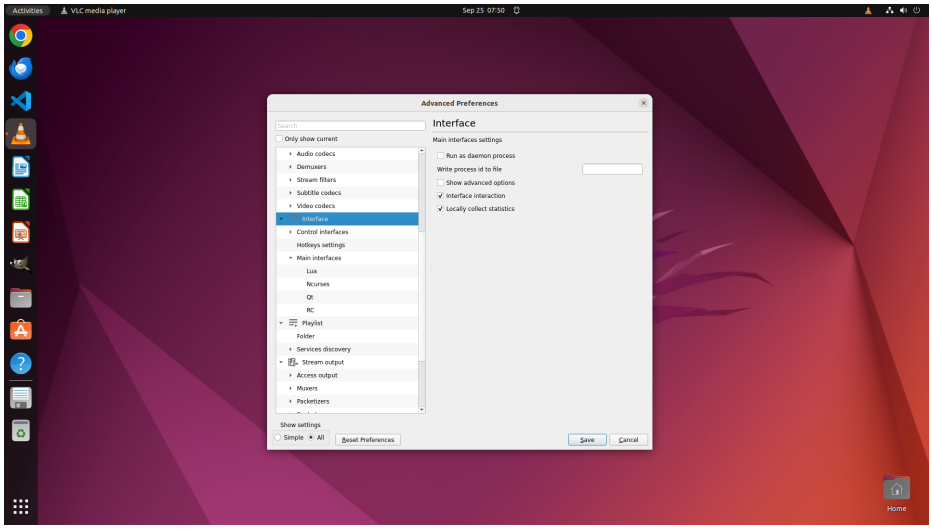


Action 6

import pyautogui
pyautogui.click(614, 518)

Table 22. **OSWorld example (UI-TARS-1.5-7B w/ W&L)**

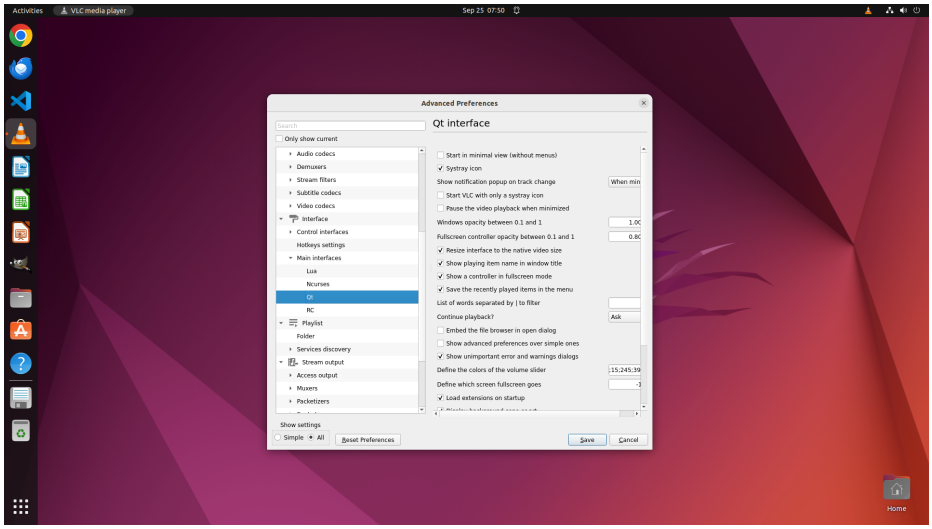
Observation 7 (VLC)



Action 7

import pyautogui
pyautogui.click(675, 604)

Observation 8 (VLC)

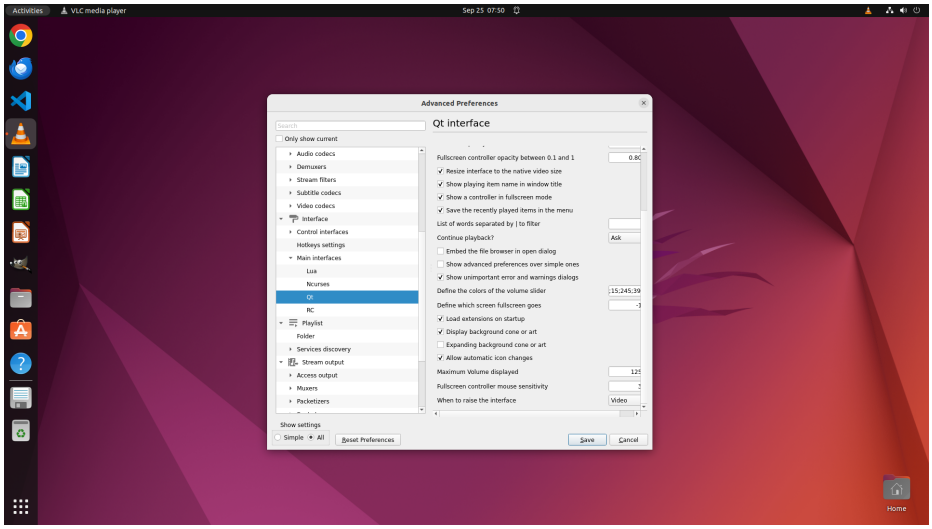


Action 8

import pyautogui
pyautogui.scroll(-50, x=1115, y=583)

Table 23. **OSWorld example (UI-TARS-1.5-7B w/ W&L)**

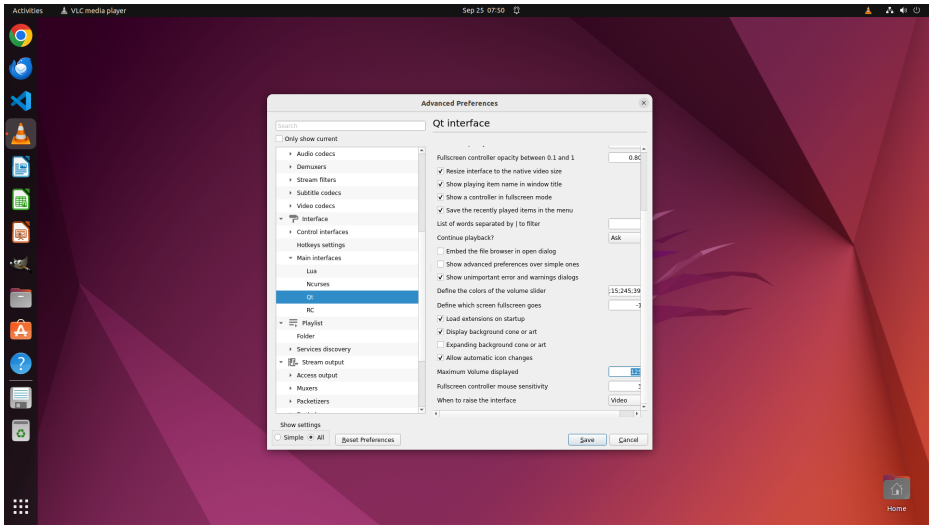
Observation 9 (VLC)



Action 9

import pyautogui
pyautogui.click(1292, 763, button=”left”)

Observation 10 (VLC)

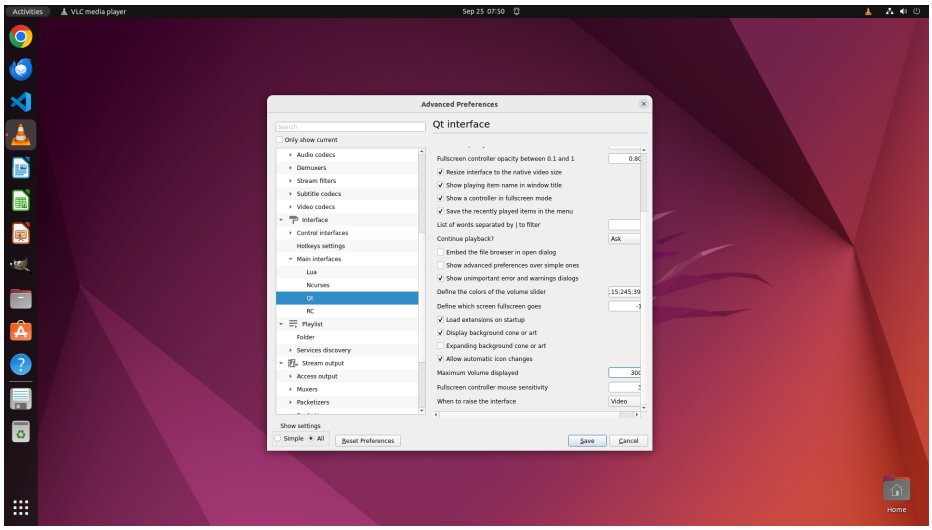


Action 10

import pyautogui
pyautogui.type(”300”)

Table 24. **OSWorld example (UI-TARS-1.5-7B w/ W&L)**

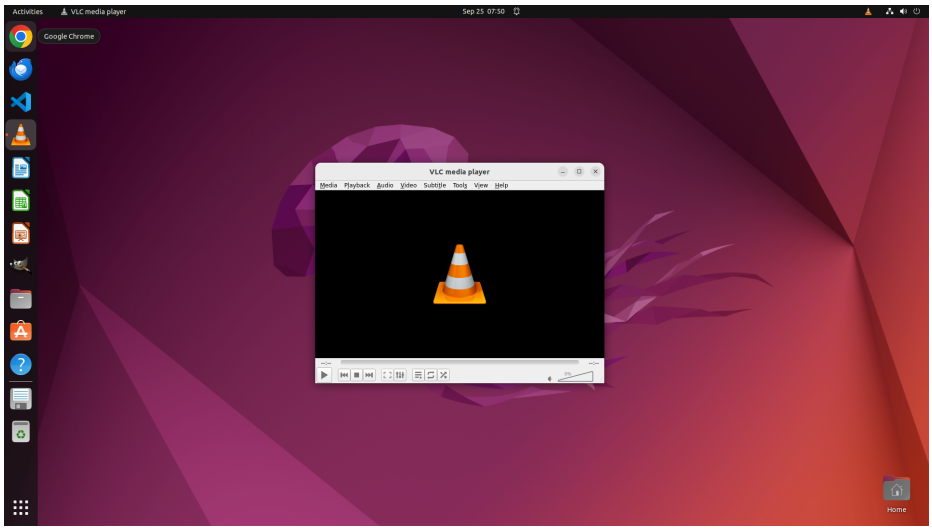
Observation 11 (VLC)



Action 11

import pyautogui
pyautogui.click(1205, 902)

Observation 12 (VLC)



Action 12

"DONE"

Table 25. **WindowsAgentArena example (UI-TARS-1.5-7B w/ W&L)**

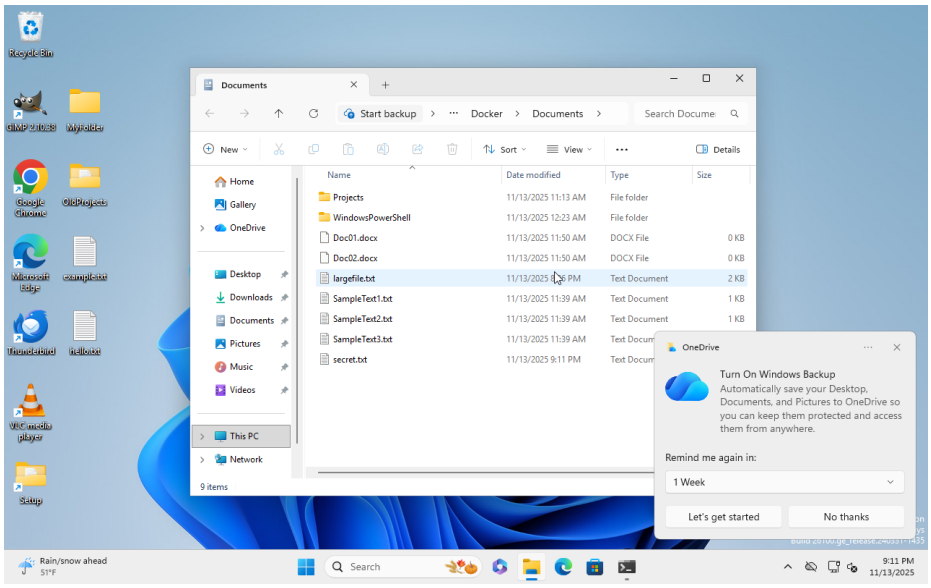
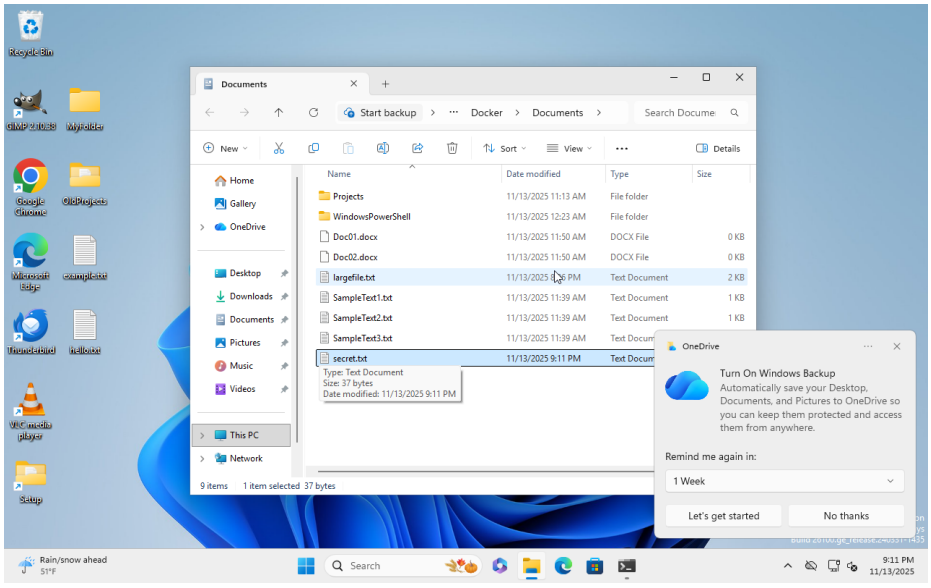
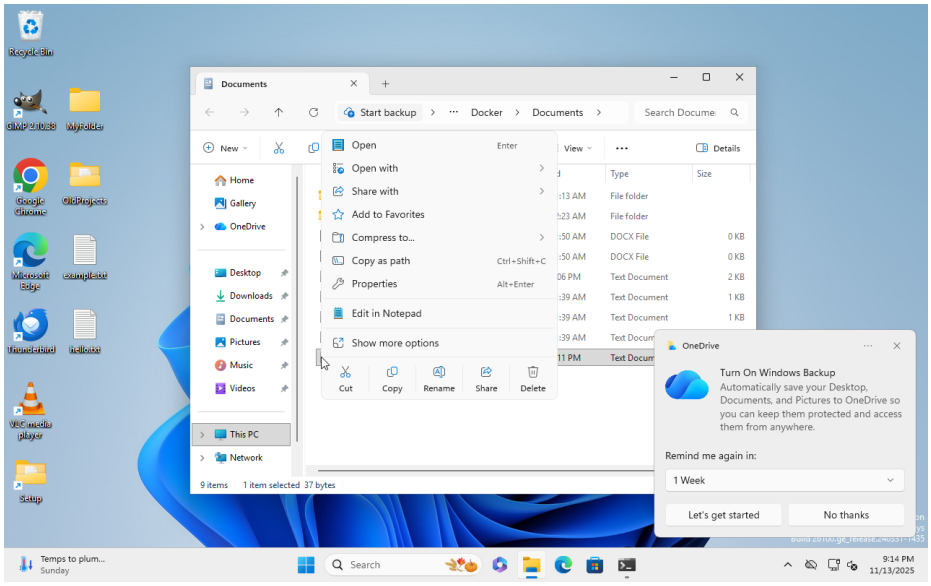
Task: Set the file "secret.txt" in the Documents folder as hidden.
Observation 1 (Windows File Explorer)

Action 1
import pyautogui pyautogui.moveTo(487, 497)
Observation 2 (Windows File Explorer)

Action 2
import pyautogui pyautogui.click(button='right')

Table 26. **WindowsAgentArena example (UI-TARS-1.5-7B w/ W&L), cont.**

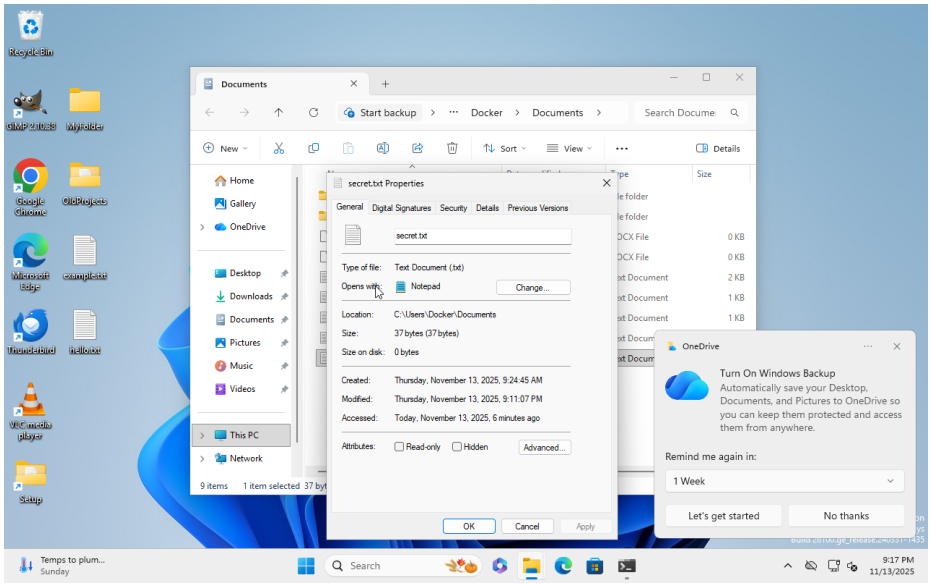
Observation 3 (Windows File Explorer)



Action 3

import pyautogui
pyautogui.click(500, 388)

Observation 4 (Windows File Explorer)

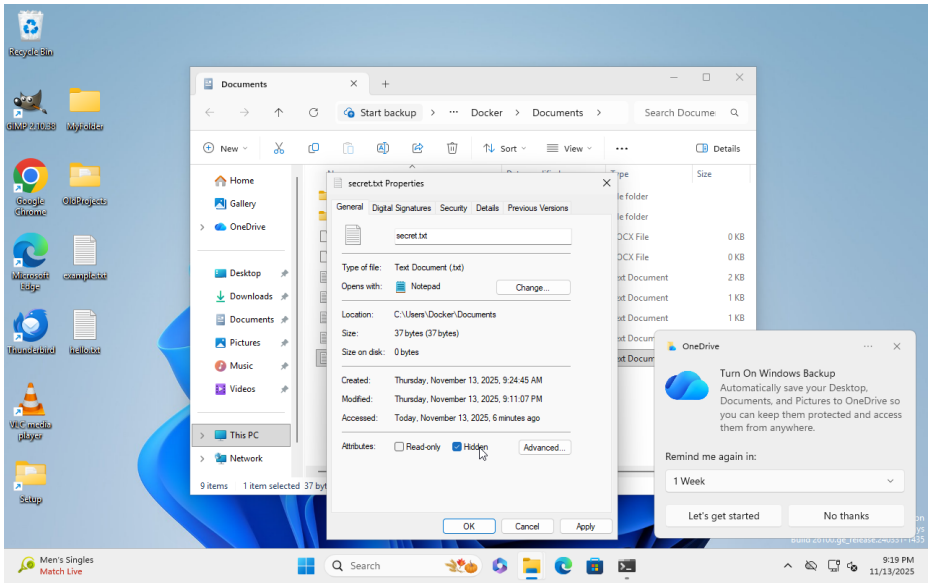


Action 4

import pyautogui
pyautogui.click(626, 611)

Table 27. **WindowsAgentArena example (UI-TARS-1.5-7B w/ W&L), cont.**

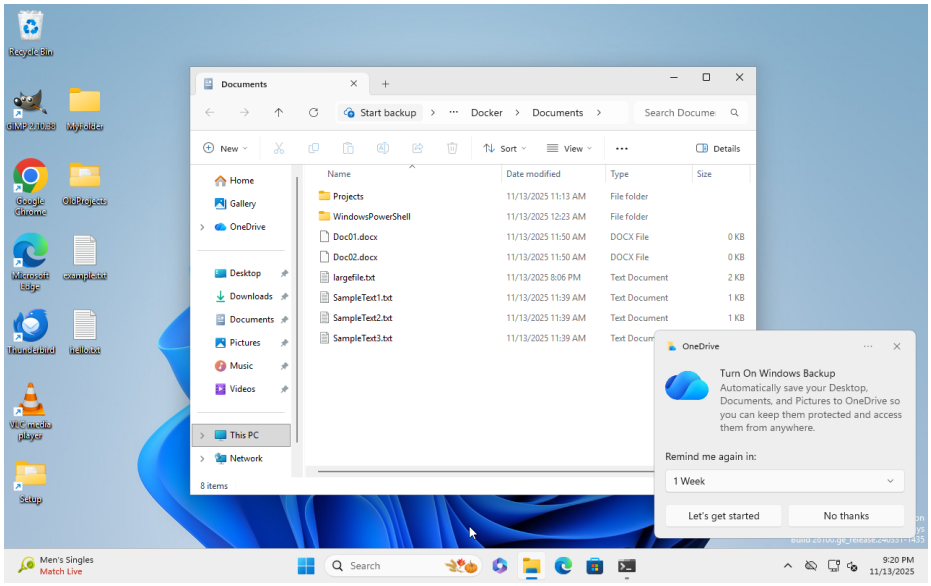
Observation 5 (Windows File Explorer)



Action 5

import pyautogui
pyautogui.click(635, 719)

Observation 6 (Windows File Explorer)



Action 6

"DONE"