# Look It Up Before Looking Ahead: Tutorials Reduce the Benefit of World Models in Computer-Use Agents

**Anh (Joe) Nguyen** and **Stefan Lee**
Oregon State University

## Abstract

Recent work has improved computer-use agents (CUAs) by retrieving task-specific tutorials and leveraging look-ahead planning from world models, but the contribution of each component has not been systematically isolated. To study this, we construct OSWorld-Tutorial, a dataset of 360 tasks with curated multimodal tutorials. We then propose RAG-PJ, a retrieval-augmented policy-judge framework that conditions both action generation and selection on task-specific tutorials. Comparing RAG-PJ against text-based and oracle ground-truth visual rollouts, we find that look-ahead planning offers no significant improvement when tutorials are present. This suggests prior gains from world-model augmentation are largely attributable to tutorial understanding itself. Error analysis reveals the bottleneck is not rollout fidelity, but the judge's ability to reason over future states and recognize task completion. This highlights the need to improve not just CUA world models, but also reasoning over their predicted futures.[1]

## 1 Introduction

People rely on desktop and web software for everyday tasks, from editing documents to configuring browsers and manipulating media. Computer-use agents (CUAs) aim to automate these tasks by interacting with computer interfaces given natural-language user instructions. Modern CUAs are typically built on top of MLLMs trained on static offline data. As software changes over time, these MLLM-based CUAs may lack current procedural knowledge about new environments. This mismatch can make agents degrade when software updates change user interfaces or functionality (Ishmam and Marino, 2026).

A natural approach to address this gap is to provide CUAs with up-to-date task-specific tutorials.

[1]Code and our dataset will be publicly available.

**Task instruction.** Can you make Bing the main search engine when I look stuff up on the internet?

Tutorial: Set your default search engine

1. On your computer, open Chrome.
2. At the top right, select More ⋮ > Settings.
3. Select Search engine.
4. Under "Search engine," select Change.
5. Select a new default search engine.

Figure 1: A Chrome task example with a retrieved web tutorial. The tutorial provides updated step-by-step browser instructions with inline UI icons.

These tutorials contain environment descriptions and step-by-step instructions in the form of text and screenshots, and are commonly available online for many software tasks. An example tutorial is given in Figure 1.

This mirrors how humans learn unfamiliar software tasks: by consulting updated tutorials when software changes. Recent works have integrated tutorials retrieval as components in CUA systems (Yang et al., 2026a; Mei et al., 2026; Xu et al., 2025a); however, systematic evaluation of tutorials' stand-alone contribution has not been presented.

A parallel body of work has considered test-time scaling for CUA through look-ahead planning (Mei et al., 2026; Luo et al., 2026; Chae et al., 2025). These methods use a learned world model to predict the outcome of action candidates, over one or more time steps, and then select between candidates conditioned on these predicted rollouts. These methods differ from standard CUA by (a) proposing and then selecting between multiple actions via an MLLM-as-a-judge framework, and (b) augmenting the judge with predicted rollouts from the world model. Recently, tutorial-augmented variants of this technique have been introduced

(Mei et al., 2026). However, it has confounding model choices that conflate the contributions of the tutorial-conditioned judge and rollouts.

To fill this gap, we study the effect of tutorial information on CUA with and without look-ahead planning, while retaining the MLLM-as-a-judge framework. Intuitively, high-quality tutorials already provide guidance for actions and demonstrations of their likely effects that can be used by the judge. This can potentially reduce the need for costly world model training and rollouts. We propose a simple yet effective tutorial-only baseline: RAG-PJ, a retrieval-augmented policy-judge framework for tutorial-based CUAs. RAG-PJ provides a task-specific tutorial to both the policy, which proposes actions, and the judge, which selects among them using the tutorial and execution history.

To systematically evaluate the effect of tutorials across a range of tasks, we first build OSWorld-Tutorial, a comprehensive dataset of 360 tasks with curated tutorials for the widely-used OSWorld-Verified benchmark (Xie et al., 2024). We evaluate RAG-PJ with methods leveraging text-based rollouts (Mei et al., 2026) and oracle ground-truth visual rollouts—an upper bound of what a learned visual world model like (Luo et al., 2026) could achieve. We find no significant gain from these look-ahead planning methods when tutorials are available. Analyzing failures of RAG-PJ, we find that the agent's failure to understand *task success and progress*, even when ground-truth visual futures are available, is a primary bottleneck to higher performance. This indicates that progress in CUA may require not just better world models, but also stronger reasoning capabilities over the futures those models simulate.

In summary, our contributions are as follows:

- We introduce RAG-PJ, a strong tutorial-conditioned policy-judge baseline for CUAs that isolates the effect of tutorials from world models.
- We build OSWorld-Tutorial, a comprehensive dataset of 360 tasks with curated tutorials for the OSWorld environment (Xie et al., 2024).
- We provide a controlled comparison between tutorial-only judging, text-based world-model rollout, and ground-truth visual rollout – demonstrating that rollout-based models do not improve over RAG-PJ given task-specific tutorials.

## 2 Related Work

**Agents that use tutorials and manuals.** Tutorials or manuals can describe both step-by-step instructions (*what to do*) and how an environment changes over time (the *dynamics of the environment*). Many works study how agents can use tutorials in different domains, including 1) games (Nguyen and Lee, 2025; Zhang et al., 2024; Lin et al., 2024; Dainese et al., 2023; Chen et al., 2024c), 2) robotics (Zhang et al., 2025b; Ren et al., 2023), and 3) embodied environments (Chen et al., 2024b). While our work targets Computer-use Agent (CUA) domain, some of these works share a similar goal: using tutorials at inference time to help agents handle new tasks or environments. In particular, our work is closely related to (Nguyen and Lee, 2025; Zhang et al., 2024; Dainese et al., 2023). However, these works focus on symbolic 2D environments, where language is restricted to a single sentence per entity—typically 3–5 sentences total—and covers only environmental dynamics. Meanwhile, we focus on real-world GUI environments with longer tutorials that cover both task instructions and environmental dynamics.

**World models for CUA agents.** Recent work has explored world models (Ha and Schmidhuber, 2018) to improve planning for CUA agents. These include techniques for text-based (Chae et al., 2025; Li et al., 2025; Cao et al., 2026; Mei et al., 2026) and image-based (Luo et al., 2026; Xiang et al., 2025) future state prediction. Like R-WoM (Mei et al., 2026), we focus on settings where multimodal task-specific tutorials are available. However, R-WoM's results do not disentangle the effect of tutorials and MLLM-as-a-judge when comparing the effect of world model rollouts. Further, R-WoM is evaluated on a subset of OSWorld tasks ($\approx 85$), which is not publicly available. In contrast, we construct a comprehensive dataset of tutorial-task pairs and design controlled experiments to study the impact of tutorials, action selection via a judge, and rollouts on task success separately.

Several works study whether pretrained MLLMs can reliably serve as world models (Li et al., 2026; Xu et al., 2026) – finding that generated rollouts may not yet be faithful proxies to real environments. Further, prior work has examined the effect of oracle world models to provide upper bounds on look-ahead performance in embodied (Qian et al., 2026) and compute-use tasks (Gonzalo et al., 2025). We extend this analysis, considering the effect of

1) oracle image-based rollouts in the presence of task-specific tutorials, and 2) look-ahead rollout per time step of interaction.

**CUA agents that use tutorials and manuals.** Prior work uses tutorials either to create training data as in (Ou et al., 2024; Xu et al., 2025b; Zhang et al., 2025a,c; Song et al., 2026a) or leverages them at inference time (Xu et al., 2025a; Agashe et al., 2025a; Mei et al., 2026; Yang et al., 2026a) like our proposed work. In this paradigm, the CUA system retrieves a relevant tutorial and uses it to guide action generation (policy) or selection (judge). RAG-GUI (Xu et al., 2025a) and Agent S (Agashe et al., 2025a) retrieve text-only tutorials to extract useful guidance for policy-only agents. We make use of multi-modal tutorials that may contain both images and text.

Most similar to our approach are OS-Symphony (Yang et al., 2026a) and R-WoM (Mei et al., 2026). OS-Symphony uses a multi-agent system including a search agent to find online tutorials for each task, then it uses another agent to implement a history reflection method through screenshot duplication and milestone detection as the policy. However, it lacks proper evaluation against other tutorial-used baselines, thus the effectiveness of this tutorial-used strategy remains unclear. Our contributions are orthogonal to OS-Symphony as our tutorial-conditioned judge could be added to its policy.

R-WoM instead provides tutorials to an MLLM-based world model to produce text-based next-state descriptions, given the current state and current action. Similar to R-WoM, our method RAG-PJ is a RAG-based MLLM-as-judge that uses tutorials to judge policy actions. In contrast, we show that multi-modal tutorials alone suffice to judge agent actions, without explicit reasoning over their future consequences. This eliminates the need for expensive world model rollout within the judge.

## 3 OSWorld-Tutorial

To robustly evaluate the impact of tutorials, we construct the OSWorld-Tutorial dataset by augmenting 360 tasks from OSWorld-Verified (Xie et al., 2024) benchmark. Each task is paired with a natural language tutorial that may contain both text and images sourced from the web.

**Generating tutorials.** To collect tutorials, we recruit three annotators with experience in software tasks similar to those in OSWorld-Verified. For each task, we first launch it in a virtual machine to get an initial screenshot. We then provide the task instruction together with this screenshot to Google Gemini Pro (Google, 2026), along with the tutorial-generation prompt shown in Appendix A. This process produces a response with references to relevant web-based tutorials as well as a generated synthetic instruction.

The annotators then review the retrieved tutorials and select the most helpful one for each task. To verify tutorials from publicly available articles, annotators follow the tutorial's instructions to complete the task and confirm whether it is correct and helpful. When completing the task from the tutorial is possible, we retain the task-tutorial pair for the dataset. However, we find that many OSWorld-Verified tasks require combining multiple tutorials while others have no relevant tutorials available online. For these tasks, we use the synthetic tutorial generated by Google Gemini Pro directly, without further verification.

**Dataset statistics.** In total, we collect 360 tasks with verified tutorials, covering all 8 categories of tasks in OSWorld-Verified. Among these, 32% of the tutorials are retrieved from the web, while 68% are synthesized by Google Gemini Pro. We follow Agent-S3 (Gonzalo et al., 2025) and OSWorld-MPC (Jia et al., 2026) to exclude eight Google Drive tasks. Recently, these tasks present technical challenges with using Google accounts and Google Drive credentials in the sandbox. We notice that some web tasks have just been upgraded to use more aggressive bot detection systems. These block automated access even through residential proxies provided by OSWorld-Verified. We therefore exclude one task that is blocked by website-level bot-detection or access restrictions, see Appendix A.

## 4 Tutorial-conditioned CUAs

As in standard computer-use settings, we consider an agent interacting with a digital environment to complete tasks specified by users in natural language. The agent observes the environment through screenshots and interacts by executing mouse and keyboard commands – mirroring how a human user would interact. Beyond this typical setup, we also assume there exists a task-relevant tutorial consisting of text and possibly images that can at least partially guide the agent through the required tasks. Our key question then is how and

where to integrate this tutorial to improve agent performance. Before discussing our design, we review existing agent architectures.

### 4.1 Preliminaries

**Problem definition.** More formally, we cast the computer-use agent task as a standard finite-horizon partially observable Markov decision process (POMDP) $\langle S, O, A, T, R, L, M \rangle$, conditioned on the task instruction $L$ and a multimodal tutorial $M$. The computing environment itself (typically a virtual machine) serves as a mostly-deterministic[2] transition function $T : S \times A \rightarrow S$ mapping between underlying system states $S$ in response to agent actions $A$. As the agent only accesses the environment visually, the observation $o_t \in O$ at each time step $t$ is simply the current screenshot. Actions correspond to movements or clicks of the mouse and keystrokes as well as a stop action. The reward function $R : S \times A \rightarrow \{0, 1\}$ is sparse and indicates task success when the agent issues the stop action or reaches the horizon $H$. We consider a computer-use agent to be a stochastic policy $p(a_t \mid o_{\leq t}, L, M)$, which produces a distribution over actions given a sequence of observations, the instruction, and the tutorial. Abstracting the form of this policy for now, we would like to find a parameterization of this policy which achieves high expected cumulative reward (i.e., task success quickly) over a wide task distribution.

**Agent architectures.** Given the need for visual and language reasoning, a common choice for instantiating the policy $\pi$ is with a pretrained multimodal large language model (MLLM). Where literature diverges is in whether a single MLLM is used to directly map from observations / instructions to actions in an end-to-end manner (as in UI-TARS (Qin et al., 2025a)) or a more modular framework of multiple MLLM and specialized models interacting to induce an action distribution (as in Agent-S2 (Agashe et al., 2025b) or OS-Symphony (Yang et al., 2026a)). A common modular design is to separate a natural language policy which outputs generalized actions like "click(The change button to next to the current search engine.)" that are then passed to a specialized grounding model to convert to low-level parameterized actions like "click(731,244)". We consider models in this family and denote the policy as $\pi(\hat{a}_t \mid \cdot)$ and the grounding model as $g(a_t \mid \hat{a}_t, \cdot)$.

**Judges and world models.** Beyond modularity in the action space, further works introduce a judge model $j(\hat{a}_t^1, ..., \hat{a}_t^K \mid o_{\leq t}, L, M)$ which chooses between multiple action candidates sampled from $\pi(\hat{a}_t \mid \cdot)$ – passing only the selected action to the grounding model for execution (Yang et al., 2026b; **?**). This sample-then-select paradigm has shown promising results when coupled with look-ahead planning using a learned world model (Ha and Schmidhuber, 2018) to estimate the outcome of each action over a fixed number of time steps (Mei et al., 2026). Given a trained world model $w : O \times A \rightarrow O$ that predicts the next observation given past observations and the current action, the outcome of each of the $K$ candidate actions can be estimated as $\tilde{o}_t^1, ..., \tilde{o}_t^K$ in a one-step look-ahead. The policy $\pi$ can then be executed to generate a single next action for each predicted observation. This loop can be executed until some fixed look-ahead horizon is reached to generate a rollout trajectory $\xi^j = (o_t, \hat{a}_t^j, \tilde{o}_{t+1}^j, \hat{a}_{t+1}^j, \ldots, \tilde{o}_{t+k}^j)$ for each action candidate $\hat{a}_t^j$. The judge model can then select between actions based on their rollout trajectories. The success of look-ahead planning relies on the capabilities of the world model, diversity and quality of the base candidate actions, and the selection ability of the judge – making it prone to cascading errors. Further, it greatly increases computational cost by requiring multiple policy and world model executions per step.

### 4.2 RAG-PolicyJudge (RAG-PJ)

Given this context, we ask whether the extra compute dedicated to judge and world models is necessary when strong policy guidance is provided in the form of a tutorial. To answer this question, we design a simple tutorial-conditioned baseline RAG-PolicyJudge (RAG-PJ) shown in overview in Figure 2. We adapt a two-staged GUI agent framework (Yang et al., 2026b; Gou et al., 2025; Song et al., 2026b) consisting of a policy and a grounding model as described above. In this architecture, we can easily ablate the inclusion of the tutorial in the policy / judge.

**Policy and Grounding Models.** We instantiate the policy as an MLLM and ablate base models in the experiments section. For memory conservation, we represent the trajectory at time $t$ as the current observation $o_t$, the three most recent screenshots

[2] In our setting, $T$ is deterministic for most tasks. For examples of non-deterministic tasks, see Appendix I.

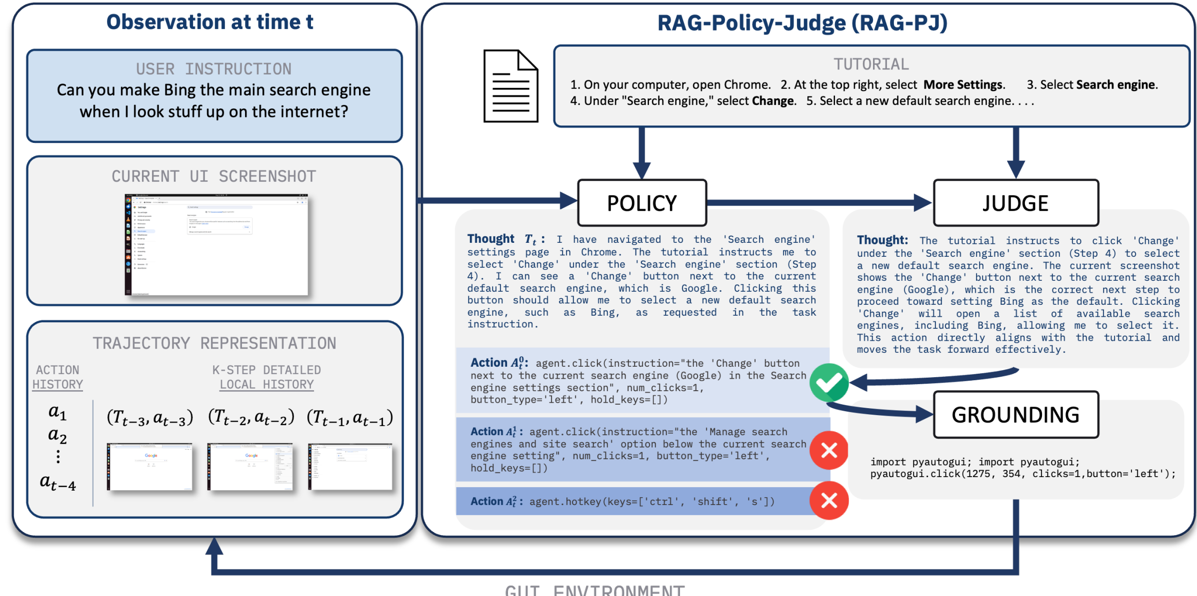



Figure 2: Overview of RAG-PJ. Given the user instruction, current screenshot, and a representation of the trajectory so far, RAG-PJ invokes a tutorial-conditioned Policy LLM to produce intermediate thoughts and candidate generic language actions. A tutorial-conditioned Judge LLM is then called to select between these actions, and the selected action is grounded to keyboard/mouse inputs by a Grounding model before being sent to the GUI environment.

with their immediate chain-of-thought tokens from the policy for these steps, and prior actions (as in GTA1 (Yang et al., 2026b)). Additionally, the user prompt and tutorial are always available at each time step. We adopt the action space from Agent-S2 (Agashe et al., 2025b), which is also used by our baselines. The policy prompt is available in Appendix C. We adopt UI-Venus-1.5 (Venus Team et al., 2026) as the grounding model with the prompt provided in Appendix E. The grounding model does not access the tutorial or user prompt.

**Judge Model.** We instantiate the judge using the same MLLM base as the policy itself. The judge evaluates the quality of candidate actions from the policy via a ranking-based prompt (see Appendix D). The judge is provided the candidate actions, user prompt, the same trajectory encoding as the policy, and the tutorial. It outputs selections by ranking action IDs in preference order and the top choice is passed to the grounding model to be parameterized and executed in the GUI environment.

## 5 Experimental Settings

We evaluate our method and relevant baselines in OSWorld-Tutorial. To avoid confounding effects of noisy retrieval, we use the annotated tutorial for each task throughout the following experiments – *essentially using an oracle tutorial retrieval system.*

**Evaluation metric.** We evaluate agents using the OSWorld-Verified metric of task success rate within a fixed step budget, using both the original maximum of 15 steps (Xie et al., 2024; Kuntz et al., 2025; Ye et al., 2025; Wang et al., 2025) and the extended 50-step budget adopted in follow-up work (Abhyankar et al., 2025; OpenAI, 2025; Jia et al., 2026).

**Baselines.** We consider a suite of baselines that let us examine the interactions of tutorials with judge and look-ahead methods. Except when explicitly noted, all methods share the same input and output space as RAG-PJ:

– `Policy Only` drops the judge from RAG-PJ but otherwise uses the same model adapted from the official implementation of GTA-1 (Yang et al., 2026b). When including the tutorial we denote this model as `Policy Only + RAG`.

– `Policy + Judge (GTA-1)` is a standard policy-judge framework that does not leverage tutorials or world model look-ahead.

– `R-WoM H=#` is a re-implementation of the recent tutorial-augmented look-ahead method from Mei et al. (2026) described in Section 4.1 which uses an MLLM-based world model to produce text-based next-state descriptions. We set $H = 1, 2$ and 3 to adjust how many steps are rolled

| # | Method | Qwen3-VL-8B 15 steps | Qwen3-VL-8B 50 steps | Qwen3-VL-32B 15 steps | Qwen3-VL-32B 50 steps |
|---|---|---|---|---|---|
| *Policy only* | | | | | |
| 1 | Policy Only | 28.3% ±1.3% | 33.6% ±2.0% | 38.8% ±0.4% | 42.5% ±1.5% |
| 2 | + RAG | 37.7% ±2.8% | 42.0% ±1.6% | 44.5% ±0.6% | 48.0% ±1.9% |
| *Policy + Judge* | | | | | |
| 3 | Policy + Judge (GTA-1) | 29.6% ±0.8% | 37.8% ±2.0% | 39.9% ±2.0% | 44.1% ±1.5% |
| 4 | **RAG-PJ (Ours)** | **40.2% ±1.4%** | **45.1% ±2.6%** | **47.8% ±1.4%** | **51.3% ±2.4%** |
| *Policy + RAG Judge + Text-based rollout (R-WoM Mei et al. (2026))* | | | | | |
| 5 | R-WoM $H$=1 | 29.6% ±0.9% | 34.3% ±1.1% | 38.3% ±0.6% | – |
| 6 | +Policy RAG | 40.7% ±0.9% | 42.9% ±0.09% | 47.3% ±0.1% | – |
| 7 | R-WoM $H$=2 | 30.3% ±0.3% | 34.0% ±1.2% | 40.2% ±1.7% | – |
| 8 | +Policy RAG | 40.5% ±0.8% | 44.7% ±1.5% | 47.3% ±1.3% | – |
| 9 | R-WoM $H$=3 | 29.6% ±1.3% | 35.9% ±0.3% | 40.0% ±0.9% | – |
| 10 | +Policy RAG | 38.4% ±1.8% | 44.0% ±1.3% | 47.0% ±0.3% | – |
| *RAG Policy + RAG Judge + Ground-truth visual rollout* | | | | | |
| 11 | GT-rollout $H$=1 | 40.6% ±0.7% | 44.4% ±2.0% | 46.5% ±1.2% | – |
| 12 | GT-rollout $H$=2 | 39.4% ±3.5% | 45.4% ±1.1% | 48.2% ±2.1% | – |

Table 1: Success rate of RAG-PJ and other baselines on 360 tasks from OSWorld-Tutorial, over the maximum runtime of 15 steps and 50 steps. All results are reported as mean ± sample standard deviation over 3 runs. "RAG" indicates a method uses a tutorial for a task. We note our method (denoted by **Bold**) consistently performs *similarly* to other baselines that use a tutorial and world model rollout. As world model rollouts in Qwen3-VL-32B with 50 steps are computationally expensive, we skip experiments in these settings, which is denoted by "–".

out during look-ahead search. An official implementation is not yet available so we catalog details of our re-implementation based on the paper in Appendix F to aid reproducibility.

– `GT-rollout H=#`: an oracle experiment in which we use ground truth screenshots as future rollouts. Similar to `R-WoM H=#`, we set $H = 1, 2$ indicating how many look-ahead steps are used in rollout.

**Base MLLMs.** We use Qwen3-VL-8B and Qwen3-VL-32B [3] as our main MLLM bases for policy / judge / world model, which are commonly used in recent CUA agents works (Xue et al., 2026; Yan et al., 2025; Bai et al., 2026; Wang et al., 2026).

# 6 Results

Given a task and corresponding tutorial, we are interested in the following questions with which we organize this section: (1) how do tutorials affect policy-judge frameworks? (Section 6.1); (2) how much do current text-based world models for CUA agents help the judge? (Section 6.2); (3) how much does ground-truth visual world model help the judge? (Section 6.3).We provide key experimental support and claims for each below.

[3]Qwen3-VL-8B-Instruct and Qwen3-VL-32B-Instruct-AWQ from HuggingFace official version of Qwen3-VL

| # | Method | Qwen3-VL-8B 15 steps | Qwen3-VL-8B 50 steps |
|---|---|---|---|
| 1 | Policy + Judge (GTA-1) | 30.3 ±0.6 | 37.8 ±2.0 |
| 2 | Policy + RAG Judge | 31.0 ±0.9 | 38.5 ±2.9 |
| 3 | RAG Policy + Judge | 40.2 ±1.4 | 43.0 ±1.2 |
| 4 | **RAG Policy + RAG Judge (RAG-PJ)** | **40.3** ±1.0 | **45.1** ±2.6 |

Table 2: Ablation of where tutorials are used in the policy–judge pipeline on OSWorld-Tutorial. All results are reported as mean ± sample standard deviation over 3 runs. The best performance is achieved when tutorials are used in both the policy and the judge.

## 6.1 How do tutorials affect policy-judge framework performance?

Our framework enables including tutorial information as conditioning in the policy, judge, or both.

***Finding 1: Tutorials help improve task success rate over non-tutorial baselines.*** As shown in Table 2, including tutorials in the judge (`row 2`) or policy (`row 3`) improves over the non-tutorial baseline (`row 1`) by up to ∼10%@15 and ∼5%@50. Notably, tutorial-conditioning in the policy leads to the majority of the gains – suggesting the guidance

in producing action candidates is a major mechanism for performance improvement. When compared to `Policy Only + RAG` from Table 1 `row 2`, we observe that policy conditioning alone is not sufficient to account for all the gains (37.7%@15 vs. 40.2%@15 ) which indicates judge action selection is still a positive factor. Finally, including tutorials in both yields our proposed RAG-PJ which further improves performance in the 50-step setting to 45% – achieving similar performance to look-ahead planning methods.

### 6.2 Do predicted text-based rollouts improve judgment when tutorials are present?

Tutorials provide strong priors about appropriate actions and their effects; yet, prior work using tutorials, R-WoM (Mei et al., 2026), have reported improvements from look-ahead using text-based world models. However, comparison with tutorial-conditioned policy-judge models without look-ahead was not provided.

***Finding 2: Current text-based world model degrades judge performance when tutorials are present.*** We report the success rate of R-WoM in OSWorld-Tutorial in Table 1 and find that it does not improve compared to a judge without world model rollout. Specifically, R-WoM without policy-side tutorial conditioning (`rows 5, 7, 9`) performs comparably to the non-tutorial Policy + Judge baseline (`row 3`) and under-performs Policy + RAG Judge (Table 2 `row 2`)regardless of rollout horizon (H=1,2,3), suggesting the text-based rollout provides no benefit and may degrade performance. When policy-side RAG is added (`rows 6, 8, 10`), R-WoM performs comparably to RAG-PJ (`row 4`) – indicating that gains again stem from tutorial conditioning rather than rollout-augmented judging. Further, we observe no consistent trend across rollout horizons: H=2 and H=3 do not improve over H=1, and in some settings longer rollout horizons slightly degrade performance, suggesting compounding errors across simulated steps add noise to the judging process.

### 6.3 Do ground-truth visual rollouts help?

Motivated by the observation that text-based rollouts failed to improve system performance, we consider an oracle rollout setting to establish an upper bound on world model performance. To avoid introducing additional confounders, we opt to directly use screenshots collected from the environment after executing actions. For rollouts across multiple action candidates at a given time step, we save and reset state of the virtual machine hosting the environment before collecting each rollout. For more details, see Appendix G. This provides an upper bound on the potential benefit of a learned visual world model.

***Finding 3: Visual ground-truth rollouts do not aid the judge in the presence of task-specific tutorials.*** Comparing tutorial-conditioned policy variants of text-based and oracle visual rollouts (Table 1 `rows 11, 12` vs. `rows 6,8`) we find no substantial differences. Likewise, the ground truth visual rollouts are comparable to RAG-PJ where tutorials alone are used. As before, this result suggests that the judge can already make effective decisions based on the retrieved tutorials, without needing to reason over future consequences.

***Finding 4: Agents fail to use ground-truth future information because they struggle to understand task success and progress.*** To understand why ground-truth visual rollouts do not improve success, we analyze failed tasks across all seeds in this setting with the help of Gemini 3.5 Flash (Google DeepMind, 2026) and summarize the failure reasons and their frequency in Figure 3. We provide more details about the error analysis implementation in Appendix H. Results from Figure 3 indicate the top two main reasons for failure contribute most faults.

The most frequently failure category is the agent (both the policy and the judge) failing to understand *task success: if the current state satisfies the task instruction or not*. For instance, a task is not completed in a current state, yet the policy and the judge think the current state already satisfies the instruction. The judge therefore selects ending the task prematurely. In the other end, if the task already satisfies the instruction, the agent should stop acting. However, the judge fails to recognize this, even with the help of future information from rollouts, and thus continues to act and fail. We show an example of this in the appendix.

Second most frequently tagged is the judge failing to reason over the future information about *task progress: what has been done and what is left to do*. Even if the policy generates good action candidates, the judge fails to select the one that leads to task progress. The judge often falls back to repeating the same failed action, instead of recognizing the failure and choosing other candidates.

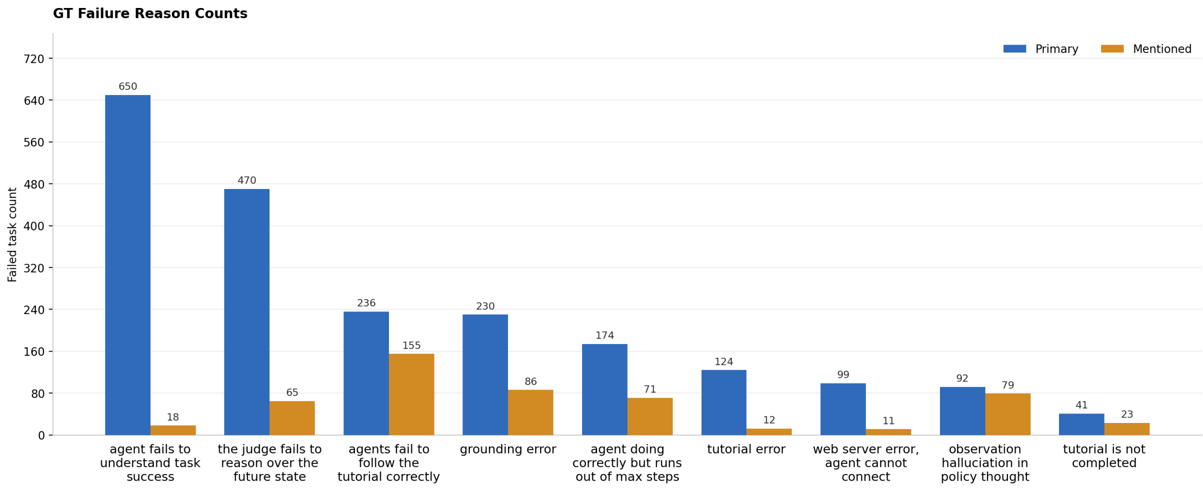



Figure 3: Error analysis of 2116 failed tasks with ground-truth visual rollouts. The categories summarize the primary failure reason assigned to each sampled trajectory.

We show examples of these failures in Figure 16, 20 and 17. Overall, we show that in the majority of failure cases, the judge lacks the ability to *reason over future states generated by world models*, not because of the quality of the world model itself. Examples of other less significant failure reasons are shown in Appendix H.

***Implication: CUAs not only need better world models, but also better mechanisms for reasoning over future rollouts.*** Based on the above analysis, we argue that improving world-model quality alone, such as next-state prediction fidelity, may be insufficient to improve look-ahead performance. What is missing from current world model methods for CUA is *a good reward prediction* for rollout states, which is necessary for the judge to select among action candidates based on their future trajectories. This finding is consistent with Qian et al. (2026), who also find that effectively leveraging future information from world models is a major bottleneck for agents in embodied environments. Furthermore, we also need to develop better methods for the judge to effectively understand future rollouts from world models, such as better prompting techniques or training paradigms that encourage learning from future states.

## 7 Discussion and Risks

**Compute used.** We use total 272 hours of GPU training with 2 H100s and 1 H200 GPU.

**Risks.** Like all assistive technologies, advances in computer-use agents make it easier for individuals to take action – both good and ill. Given how modern infrastructures relies on digital interfaces, this may have significant risks for personal digital security, coordinated astroturfing, or model errors causing users harm.

## 8 Conclusion

We investigated the interplay between tutorial retrieval and look-ahead planning in computer-use agents, motivated by the observation that these components have not been analyzed in isolation. To enable this analysis, we constructed OSWorld-Tutorial, a dataset of 360 tasks paired with curated multimodal tutorials spanning all eight OSWorld task categories, and proposed RAG-PJ, a strong tutorial-conditioned policy-judge baseline. Our controlled experiments reveal a consistent pattern: when task-specific tutorials are available, look-ahead planning with world model rollouts provides no benefit over tutorial-conditioned judging alone and may in fact hinder action selection. This held for both text-based rollouts and oracle ground-truth visual rollouts, suggesting the bottleneck is in judge's ability to reason over future states rather than rollout fidelity. Error analysis corroborates that failures stem primarily from agents misidentifying task completion and from judges failing to leverage available rollouts. Together, these findings suggest that previously reported gains from world-model augmentation in tutorial-equipped systems may be attributable largely to improved policy and judge behavior from tutorial conditioning itself.

# 9 Limitations

Several limitations bound the scope of our conclusions. First, our experiments use oracle tutorial retrieval—each task is paired with its annotated tutorial at inference time—which avoids the noise of real retrieval pipelines. Performance of RAG-PJ and the baselines may degrade when tutorials must be retrieved from noisy web sources, and the relative benefit of look-ahead planning under noisy retrieval remains an open question. Second, 68% of the tutorials in OSWorld-Tutorial are synthesized by Google Gemini Pro rather than sourced from human-verified web articles. These synthetic tutorials are not verified for correctness, and our error analysis (Figure 3) identifies incorrect MLLM-generated tutorials as a notable failure mode. Results on verified web tutorials may differ from those on synthesized tutorials.

## Acknowledgments

We thank everyone from VIRL lab and Huazheng's lab (Oregon State University). The first author was personally supported by Nguyen Thi Ngoc Anh, Amanda Putiza, Ngo Thi Bich Lan, Tran Thanh Nhu, Bui Thuy Tien, and Nguyen Hoang Kieu Anh. This work is supported by NSF CAREER Award 2339676. We also thank the anonymous reviewers for their valuable feedback and suggestions.

## References

Reyna Abhyankar, Qi Qi, and Yiying Zhang. 2025. OSWorld-human: Benchmarking the efficiency of computer-use agents. In *ICML WCUA Workshop*.

Saaket Agashe, Jiuzhou Han, Shuyu Gan, Jiachen Yang, Ang Li, and Xin Eric Wang. 2025a. Agent S: An open agentic framework that uses computers like a human. In *ICLR*.

Saaket Agashe, Kyle Wong, Vincent Tu, Jiachen Yang, Ang Li, and Xin Eric Wang. 2025b. Agent S2: A compositional generalist-specialist framework for computer use agents. In *COLM*.

Hao Bai, Alexey Taymanov, Tong Zhang, Aviral Kumar, and Spencer Whitehead. 2026. WebGym: Scaling training environments for visual web agents with realistic tasks. *arXiv*.

Yilin Cao, Yufeng Zhong, Zhixiong Zeng, Liming Zheng, Jing Huang, Haibo Qiu, Peng Shi, Wenji Mao, and Wan Guanglu. 2026. MobileDreamer: Generative sketch world model for GUI agent. *arXiv*.

Hyungjoo Chae, Namyoung Kim, Kai Tzu-Iunn Ong, Minju Gwak, Gwanwoo Song, Jihoon Kim, Sunghwan Kim, Dongha Lee, and Jinyoung Yeo. 2025. Web agents with world models: Learning and leveraging environment dynamics in web navigation. In *ICLR*.

Dongping Chen, Ruoxi Chen, Shilin Zhang, Yinuo Liu, Yaochen Wang, Huichi Zhou, Qihui Zhang, Yao Wan, Pan Zhou, and Lichao Sun. 2024a. MLLM-as-a-judge: Assessing multimodal LLM-as-a-judge with vision-language benchmark. *arXiv [cs.CL]*.

Minghao Chen, Yihang Li, Yanting Yang, Shiyu Yu, Binbin Lin, and Xiaofei He. 2024b. AutoManual: Constructing instruction manuals by LLM agents via interactive environmental learning. In *NeurIPS*.

Xiong-Hui Chen, Ziyan Wang, Yali Du, Shengyi Jiang, Meng Fang, Yang Yu, and Jun Wang. 2024c. Policy learning from tutorial books via understanding, rehearsing and introspecting. In *NeurIPS*.

Nicola Dainese, Pekka Marttinen, and Alexander Ilin. 2023. Reader: Model-based language-instructed reinforcement learning. In *EMNLP*, pages 16583–16599.

Gonzalez-Pumariega Gonzalo, Tu Vincent, Lee Chih-Lun, Yang Jiachen, Li Ang, and Xin Eric Wang. 2025. The unreasonable effectiveness of scaling agents for computer use. *arXiv*.

Google. 2026. Google Gemini (version 3.1 pro) [Large language model]. https://gemini.google.com.

Google DeepMind. 2026. Gemini 3.5 Flash model card. https://deepmind.google/models/model-cards/gemini-3-5-flash/. Accessed: 2026-05-25.

Boyu Gou, Ruohan Wang, Boyuan Zheng, Yanan Xie, Cheng Chang, Yiheng Shu, Huan Sun, and Yu Su. 2025. Navigating the digital world as humans do: Universal visual grounding for GUI agents. In *ICLR*.

David Ha and Jürgen Schmidhuber. 2018. World models. *arXiv*.

Md Farhan Ishmam and Kenneth Marino. 2026. Time-Warp: Evaluating web agents by revisiting the past. *arXiv*.

Hongrui Jia, Jitong Liao, Xi Zhang, Haiyang Xu, Tianbao Xie, Chaoya Jiang, Ming Yan, Si Liu, Wei Ye, and Fei Huang. 2026. OSWorld-MCP: Benchmarking MCP tool invocation in computer-use agents. In *ICLR*.

Thomas Kuntz, Agatha Duzan, Hao Zhao, Francesco Croce, Zico Kolter, Nicolas Flammarion, and Maksym Andriushchenko. 2025. OS-harm: A benchmark for measuring safety of computer use agents. In *NeurIPS*.

Shufan Li, Konstantinos Kallidromitis, Akash Gokul, Yusuke Kato, Kazuki Kozuka, and Aditya Grover. 2025. MobileWorldBench: Towards semantic world modeling for mobile agents. *arXiv*.

Yixia Li, Hongru Wang, Jiahao Qiu, Zhenfei Yin, Dongdong Zhang, Cheng Qian, Zeping Li, Pony Ma, Guanhua Chen, Heng Ji, and Mengdi Wang. 2026. From word to world: Can large language models be implicit text-based world models? In *ICLR LLA Workshop*.

Jessy Lin, Yuqing Du, Olivia Watkins, Danijar Hafner, Pieter Abbeel, Dan Klein, and Anca Dragan. 2024. Learning to model the world with language. In *ICML*.

Dezhao Luo, Bohan Tang, Kang Li, Georgios Papoudakis, Jifei Song, Shaogang Gong, Jianye Hao, Jun Wang, and Kun Shao. 2026. ViMo: A generative visual GUI world model for app agents. In *ICLR*.

Kai Mei, Jiang Guo, Shuaichen Chang, Mingwen Dong, Dongkyu Lee, Xing Niu, and Jiarong Jiang. 2026. R-WoM: Retrieval-augmented world model for computer-use agents. In *ICLR*.

Anh Nguyen and Stefan Lee. 2025. Language-conditioned world model improves policy generalization by reading environmental descriptions. In *NeurIPS LAW Workshop*.

OpenAI. 2025. Introducing operator. https://openai.com/index/introducing-operator/. Accessed: 2026-05-15.

Tianyue Ou, Frank F Xu, Aman Madaan, Jiarui Liu, Robert Lo, Abishek Sridhar, Sudipta Sengupta, Dan Roth, Graham Neubig, and Shuyan Zhou. 2024. Synatra: Turning indirect knowledge into direct demonstrations for digital agents at scale. In *NeurIPS*.

Cheng Qian, Emre Can Acikgoz, Bingxuan Li, Xiusi Chen, Yuji Zhang, Bingxiang He, Qinyu Luo, Dilek Hakkani-Tür, Gokhan Tur, Yunzhu Li, and Heng Ji. 2026. Current agents fail to leverage world model as tool for foresight. *arXiv*.

Yujia Qin, Yining Ye, Junjie Fang, Haoming Wang, Shihao Liang, Shizuo Tian, Junda Zhang, Jiahao Li, Yunxin Li, Shijue Huang, Wanjun Zhong, Kuanye Li, Jiale Yang, Yu Miao, Woyu Lin, Longxiang Liu, Xu Jiang, Qianli Ma, Jingyu Li, and 16 others. 2025a. Ui-tars: Pioneering automated gui interaction with native agents. *Preprint*, arXiv:2501.12326.

Yujia Qin, Yining Ye, Junjie Fang, Haoming Wang, Shihao Liang, Shizuo Tian, Junda Zhang, Jiahao Li, Yunxin Li, Shijue Huang, Wanjun Zhong, Kuanye Li, Jiale Yang, Yu Miao, Woyu Lin, Longxiang Liu, Xu Jiang, Qianli Ma, Jingyu Li, and 16 others. 2025b. UI-TARS: Pioneering automated GUI interaction with native agents. *arXiv*.

Allen Z Ren, Bharat Govil, Tsung-Yen Yang, Karthik Narasimhan, and Anirudha Majumdar. 2023. Leveraging language for accelerated learning of tool manipulation. In *CoRL*.

Chan Hee Song, Yiwen Song, Palash Goyal, Yu Su, Oriana Riva, Hamid Palangi, and Tomas Pfister. 2026a. Watch and learn: Learning to use computers from online videos. In *CVPR*.

Linxin Song, Yutong Dai, Viraj Prabhu, Jieyu Zhang, Taiwei Shi, Li Li, Junnan Li, Silvio Savarese, Zeyuan Chen, Jieyu Zhao, Ran Xu, and Caiming Xiong. 2026b. CoAct-1: Computer-using agents with coding as actions. In *ICLR*.

Venus Team, Changlong Gao, Zhangxuan Gu, Yulin Liu, Xinyu Qiu, Shuheng Shen, Yue Wen, Tianyu Xia, Zhenyu Xu, Zhengwen Zeng, Beitong Zhou, Xingran Zhou, Weizhi Chen, Sunhao Dai, Jingya Dou, Yichen Gong, Yuan Guo, Zhenlin Guo, Feng Li, and 8 others. 2026. UI-venus-1.5 technical report. *arXiv [cs.CV]*.

Xinyuan Wang, Bowen Wang, Dunjie Lu, Junlin Yang, Tianbao Xie, Junli Wang, Jiaqi Deng, Xiaole Guo, Yiheng Xu, Chen Henry Wu, Zhennan Shen, Zhuokai Li, Ryan Li, Xiaochuan Li, Junda Chen, Boyuan Zheng, Peihang Li, Fangyu Lei, Ruisheng Cao, and 23 others. 2025. OpenCUA: Open foundations for computer-use agents. In *NeurIPS*.

Yinjie Wang, Tianbao Xie, Ke Shen, Mengdi Wang, and Ling Yang. 2026. RLAnything: Forge environment, policy, and reward model in completely dynamic RL system. *arXiv*.

Jiannan Xiang, Yun Zhu, Lei Shu, Maria Wang, Lijun Yu, Gabriel Barcik, James Lyon, Srinivas Sunkara, and Jindong Chen. 2025. UISim: An interactive image-based UI simulator for dynamic mobile environments. In *NeurIPS LAW Workshop*.

Tianbao Xie, Danyang Zhang, Jixuan Chen, Xiaochuan Li, Siheng Zhao, Ruisheng Cao, Toh Jing Hua, Zhoujun Cheng, Dongchan Shin, Fangyu Lei, Yitao Liu, Yiheng Xu, Shuyan Zhou, Silvio Savarese, Caiming Xiong, Victor Zhong, and Tao Yu. 2024. OSWorld: Benchmarking multimodal agents for open-ended tasks in real computer environments. In *NeurIPS*.

Ran Xu, Kaixin Ma, Wenhao Yu, Hongming Zhang, Joyce C Ho, Carl Yang, and Dong Yu. 2025a. Retrieval-augmented GUI agents with generative guidelines. In *EMNLP*, pages 17866–17875, Suzhou, China. Association for Computational Linguistics.

Weikai Xu, Kun Huang, Yunren Feng, Jiaxing Li, Yuhan Chen, Yuxuan Liu, Zhizheng Jiang, Heng Qu, Pengzhi Gao, Wei Liu, Jian Luan, Xiaolin Hu, and Bo An. 2026. How mobile world model guides GUI agents? *arXiv*.

Yiheng Xu, Dunjie Lu, Zhennan Shen, Junli Wang, Zekun Wang, Yuchen Mao, Caiming Xiong, and Tao Yu. 2025b. AgentTrek: Agent trajectory synthesis via guiding replay with web tutorials. In *ICLR*.

Taofeng Xue, Chong Peng, Mianqiu Huang, Linsen Guo, Tiancheng Han, Haozhe Wang, Jianing Wang,

Xiaocheng Zhang, Xin Yang, Dengchang Zhao, Jinrui Ding, Xiandi Ma, Yuchen Xie, Peng Pei, Xunliang Cai, and Xipeng Qiu. 2026. EvoCUA: Evolving computer use agents via learning from scalable synthetic experience. *arXiv*.

Tianci Xue, Weijian Qi, Tianneng Shi, Chan Hee Song, Boyu Gou, Dawn Song, Huan Sun, and Yu Su. 2025. An illusion of progress? assessing the current state of web agents. *arXiv [cs.AI]*.

Haolong Yan, Jia Wang, Xin Huang, Yeqing Shen, Ziyang Meng, Zhimin Fan, Kaijun Tan, Jin Gao, Lieyu Shi, Mi Yang, Shiliang Yang, Zhirui Wang, Brian Li, Kang An, Chenyang Li, Lei Lei, Mengmeng Duan, Danxun Liang, Guodong Liu, and 1 others. 2025. Step-GUI technical report. *arXiv*.

Bowen Yang, Kaiming Jin, Zhenyu Wu, Zhaoyang Liu, Qiushi Sun, Zehao Li, Jingjing Xie, Zhoumianze Liu, Fangzhi Xu, Kanzhi Cheng, Qingyun Li, Yian Wang, Yu Qiao, Zun Wang, and Zichen Ding. 2026a. OS-symphony: A holistic framework for robust and generalist computer-using agent. In *ACL*.

Yan Yang, Dongxu Li, Yutong Dai, Yuhao Yang, Ziyang Luo, Zirui Zhao, Zhiyuan Hu, Junzhe Huang, Amrita Saha, Zeyuan Chen, Ran Xu, Liyuan Pan, Silvio Savarese, Caiming Xiong, and Junnan Li. 2026b. GTA1: GUI test-time scaling agent. In *ICLR*.

Jiabo Ye, Xi Zhang, Haiyang Xu, Haowei Liu, Junyang Wang, Zhaoqing Zhu, Ziwei Zheng, Feiyu Gao, Junjie Cao, Zhengxi Lu, Jitong Liao, Qi Zheng, Fei Huang, Jingren Zhou, and Ming Yan. 2025. Mobile-agent-v3: Fundamental agents for GUI automation. *arXiv*.

Alex Zhang, Khanh Nguyen, Jens Tuyls, Albert Lin, and Karthik Narasimhan. 2024. Language-guided world models: A model-based approach to AI control. In *ACL SpLU-RoboNLP Workshop*.

Bofei Zhang, Zirui Shang, Zhi Gao, Wang Zhang, Rui Xie, Xiaojian Ma, Tao Yuan, Xinxiao Wu, Song-Chun Zhu, and Qing Li. 2025a. TongUI: Building generalized GUI agents by learning from multimodal web tutorials. *arXiv*.

Jian Zhang, Hanbo Zhang, Anxing Xiao, and David Hsu. 2025b. Robot operation of home appliances by reading user manuals. In *CoRL*.

Ziyun Zhang, Xinyi Liu, Xiaoyi Zhang, Jun Wang, Gang Chen, and Yan Lu. 2025c. UI-evol: Automatic knowledge evolving for computer use agents. In *ICML WCUA Workshop*.

# A Building the OSWorld-Tutorial Dataset

```
Give detailed instructions for completing the given task in Ubuntu with <program> <version>,
assuming the given start state steps are already complete.

Just output the instructions. Do not add any unnecessary conversational content to the end or
beginning. Don't ask any questions, including at the end. You should not solve the task: you are
providing instructions for solving the task. Give me more than 1 method if possible, put the most
 effective method on top. Do not close anything unless it is instructed by user.

For tasks that require timing (date, month), give me generic instructions and do not condition
them on today's date.

If the task is infeasible (cannot be done with <program> alone), say "Infeasible." followed by a
brief explanation of why. Do NOT suggest workarounds, alternative tools, or command-line
solutions --- just state why it is infeasible and stop.


Task:
<task>
<the first screenshot>
```

Figure 4: Tutorial-generation prompt used to retrieve or synthesize task-specific tutorials. The placeholders are replaced with the program category, task instruction, and initial task screenshot from each OSWorld task. For `"multiapp"` tasks, we set `<version>` to `None`.

**Overview.** We recruit three student annotators with experience in software tasks similar to those in OSWorld to collect a tutorial for each task. For each task, we first launch it in a virtual machine and provide the task instruction together with the initial screenshot to the tutorial-generation prompt shown in Figure 4. We submit this prompt to Google Gemini Pro (Google, 2026). It searches the web for relevant tutorials and synthesizes a comprehensive summary over the retrieved results. The annotators then review the retrieved tutorials and select the most helpful one for each task. To verify tutorials from publicly available articles, they follow its instructions to complete the task and confirm that it is correct and helpful.

We find that many OSWorld tasks require combining multiple tutorials, while others have no relevant tutorials available online. For these tasks, we use the response generated by Google Gemini Pro directly as the tutorial. *We do not verify the correctness of AI-synthesized tutorials.*

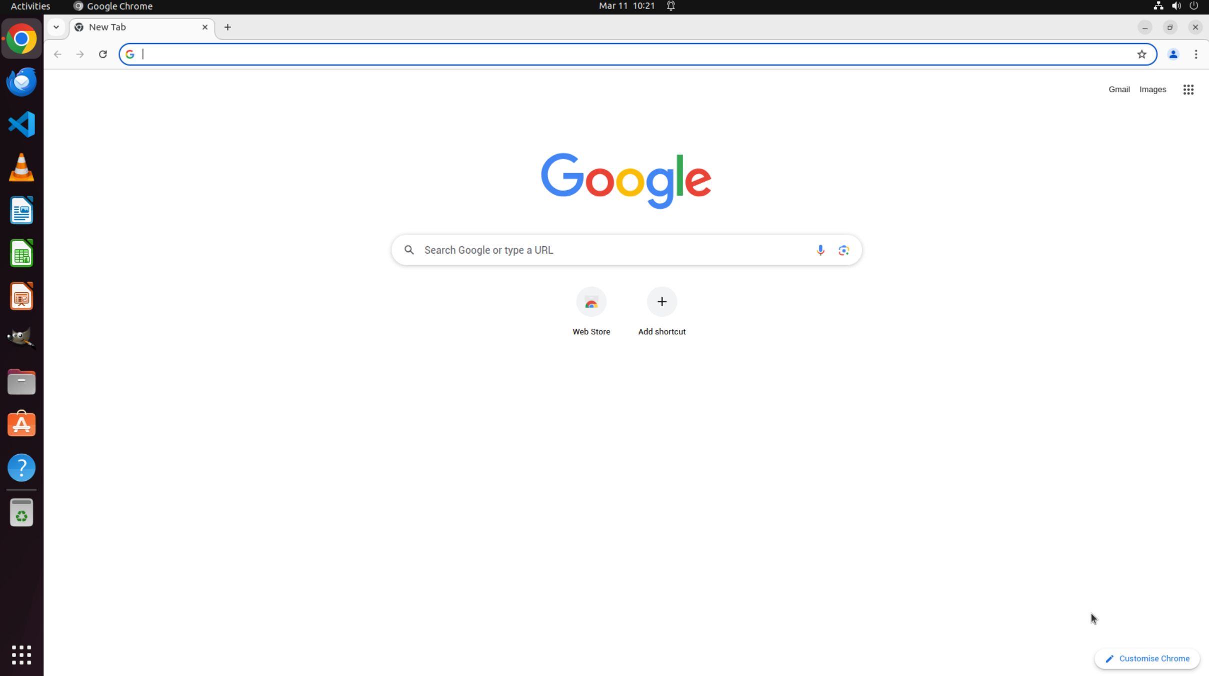
Figures 5 and 6 show additional examples of retrieved publicly available tutorials and MLLM-generated tutorials. Each figure shows task instruction, initial screenshot, and the corresponding tutorial content.

**Statistics.** In total, we collect 360 tasks with verified tutorials, covering all 8 categories of tasks in OSWorld. Among these, 32% of the tutorials are retrieved from the web, while 68% are synthesized by Google Gemini Pro. We follow Agent-S3 (Gonzalo et al., 2025) and OSWorld-MPC (Jia et al., 2026) to exclude eight Google Drive tasks. Recently, these tasks present technical challenges with using Google accounts and Google Drive credentials in the sandbox. We notice that some web tasks have just been upgraded to use more aggressive bot detection systems, which block automated access even through residential proxies provided by OSWorld. We therefore exclude one task that is blocked by website-level bot-detection or access restrictions, as listed as task id in Figure 7.

```
# Excluded task IDs - one per line, comments start with #
# tripadvisor.com - "Access is temporarily restricted" behavioral bot detection.
b7895e80-f4d1-4648-bee0-4eb45a6f1fa8
```

Figure 7: Excluded task IDs for websites that block automated access through bot detection, CAPTCHAs, or access-denial systems.

# B RAG-PJ implementation details




Task instruction. Enable the “Do Not Track” feature in Chrome to enhance online privacy.

Turn “Do Not Track” on or off

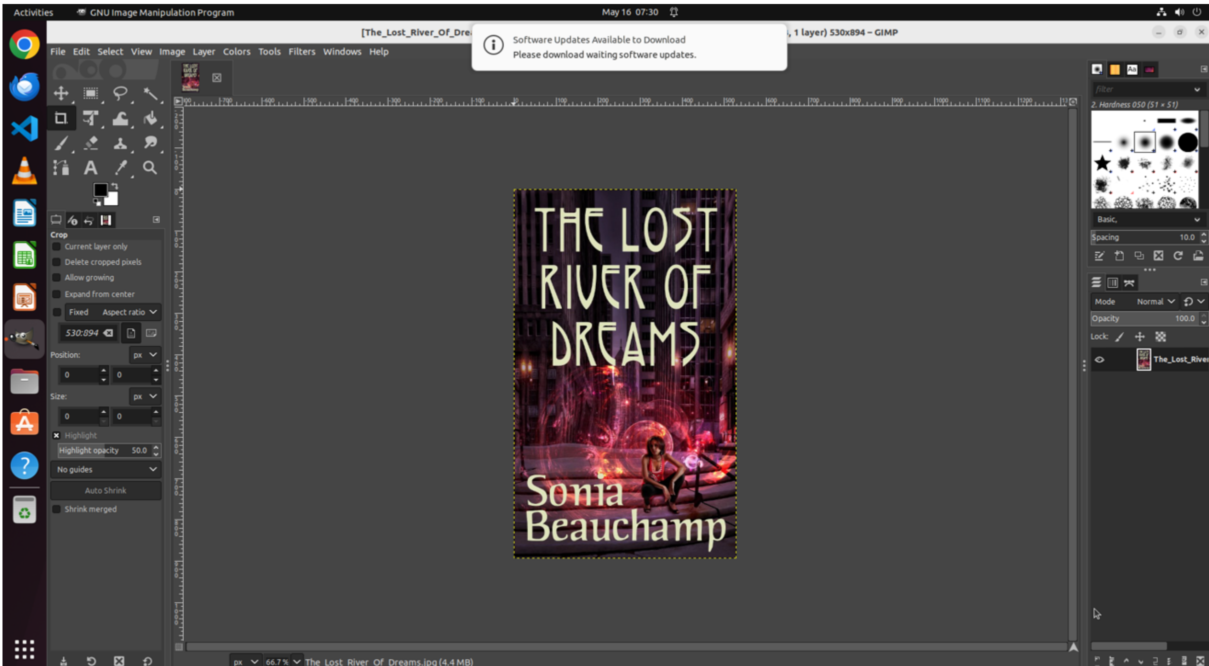
When browsing the web on computers or Android devices, users can send a request to websites not to collect or track browsing data. The setting is turned off by default.

However, what happens to the data depends on how a website responds to the request. Many websites still collect and use browsing data for security, content, services, ads, recommendations, and reporting statistics.

Most websites and web services, including Google’s, do not change their behavior when they receive a Do Not Track request.

1. On your computer, open Chrome.
2. At the top right, select More ⋮ › Settings.
3. Select Privacy and security › Third-party cookies.
4. Turn Send a “Do not track” request with your browsing traffic on or off.

Figure 5: **Chrome task example with a retrieved web tutorial.** The top panel shows the initial screenshot; the lower panel shows the retrieved markdown tutorial with inline images preserved. *Task ID:* `030eeff7-b492-4218-b312-701ec99ee0cc`. *Source:* Google Chrome Help.




Task instruction. Could you assist me in placing my photo on the desktop and renaming it to `export.jpg`?
Category: `gimp`.

GIMP treats standard image formats such as `.jpg` as exports rather than saves.

1. In the top menu, go to File > Export As... or press Ctrl+Shift+E.
2. In the Name box at the top, delete the existing text and type `export.jpg`.
3. On the left-hand sidebar, under Places, click Desktop to ensure it is being saved to the right spot.
4. Click the Export button in the bottom right.
5. An "Export Image as JPEG" window will pop up. Click Export again to confirm the settings.

Figure 6: **GIMP task example with an MLLM-synthesized tutorial.** The top panel shows the initial screenshot; the lower panel shows the synthesized tutorial for exporting the image as `export.jpg` on the desktop. *Task ID:* `77b8ab4d-994f-43ac-8930-8ca087d7c4b4`.

**History information.** Following GTA1 (Yang et al., 2026b) [4], for both the policy and the judge, we include screenshots, intermediate thought tokens, and actions for the most recent three steps in the execution history. For other steps, we only include the actions. We find that including intermediate thought tokens for these steps does not significantly improve performance, while it does increase the input length and inference time for both the policy and the judge. We include the ablation study on the effect of including intermediate thought tokens in the execution history in Table 3.

| MLLM base | 15 steps | |
|---|---|---|
| | w/ thought | w/o thought |
| Qwen3-VL-8B | 28.5% $\pm$ 1.6% | 28.1% $\pm$ 1.2% |
| Qwen3-VL-32B | 39.0% $\pm$ 1.2% | 38.8% $\pm$ 0.4% |

Table 3: Ablation of including intermediate thought tokens in the execution history. We use the setting vanilla policy without tutorial retrieval. Each result is mean and sample standard deviation over 3 runs. We find that including intermediate thought tokens does not significantly improve performance.

**Policy.** We use the default decoding hyperparameters for our policies[5], shown in Table 4. We use the prompt template shown in Figure 8 for the policy, which includes the retrieved tutorial and execution history as context for action generation.

| Hyperparameter | Value |
|---|---|
| `temperature` | 0.7 |
| `top_p` | 0.8 |
| `top_k` | 20 |
| `repetition_penalty` | 1.0 |
| `presence_penalty` | 1.5 |
| `max_output_length` | 16384 |

Table 4: Default decoding hyperparameters used for the RAG-PJ policy.

**Judge.** Following WebJudge (Xue et al., 2025) and MLLM-as-the-judge (Chen et al., 2024a), we use deterministic decoding for the judge, shown in Table 5. We use the prompt template shown in Figure 9 for the judge, which includes the retrieved tutorial and execution history as context for action evaluation. We also apply the same set of hyperparameters for other judges in our baselines, including R-WoM judge (ranker) and ground-truth judge.

| Hyperparameter | Value |
|---|---|
| `temperature` | 0 |
| `top_p` | 1 |

Table 5: Default decoding hyperparameters used for the RAG-PJ judge.

**Grounding.** We use UI-Venus-1.5 (Venus Team et al., 2026) as the grounding model for RAG-PJ. We use the prompt template shown in Figure 10 for the grounding model.

# C Policy Prompt

[4]Note that the official implementation of GTA1 `https://github.com/Yan98/GTA1` is different from GTA1's paper description. In the implementation, it only uses thought tokens and actions, and does not use observation caption descriptions for history information. We follow the implementation instead of the paper description.

[5]Hugging Face model cards: Qwen3-VL-8B-Instruct; Qwen3-VL-32B-Instruct-FP8.

```
You are a desktop computer agent on Ubuntu (1920x1080) with internet access. You are given a task
 instruction and a tutorial for this task if available. Each step you receive: (1) history of
past steps (Thought and/or Action); (2) recent screenshots. Your task is to choose the best next
action(s) to advance toward the overall goal, based on the provided information.

# ACTION SPACE:
def click(self, instruction: str, num_clicks: int = 1, button_type: str = 'left', hold_keys: List
 = []):
'''Click the target element.
instruction: describe target (one clear sentence, include visual details to differentiate it from
 other elements).
num_clicks: 1 or 2 (double).
button_type: "left"|"right"|"middle".
hold_keys: e.g. ["ctrl"] for Ctrl+click.
Example: agent.click(instruction="the selected file in the file explorer", num_clicks=1,
button_type="right", hold_keys=[]) # right click
Example: agent.click(instruction="the file icon on the desktop", num_clicks=2, button_type="left",
 hold_keys=[]) # double click
'''

def done(self, return_value: Union[Dict, str, List, Tuple, int, float, bool, NoneType] = None):
'''Call ONLY when task is fully complete. Visually verify the result on screen first.
Example: agent.done(return_value=None)
'''

def drag_and_drop(self, starting_description: str, ending_description: str, hold_keys: List = []):

'''Drag mouse from start to end. Prefer cut/paste for file ops. Use for multi-line text selection,
 drawing.
Example: agent.drag_and_drop(starting_description="the file icon on the desktop",
ending_description="the Documents folder in the sidebar", hold_keys=[])'''

def fail(self):
'''Call ONLY as a LAST RESORT when task is truly impossible (e.g. missing required app/file,
hardware limitation).
Example: agent.fail()
'''

def highlight_text_span(self, starting_phrase: str, ending_phrase: str):
'''Select text between starting_phrase and ending_phrase.
Example: agent.highlight_text_span(starting_phrase="Hello", ending_phrase="world") to select "
Hello abc world" from "hihihih Hello abc world xyz"
'''

def hover(self, instruction: str):
'''Move the cursor to the target WITHOUT clicking. Sometimes in some software, it is useful to
hover over an element to reveal a submenu.
Example: agent.hover(instruction="the 'Mode' row in the open Image menu")
'''

def hold_and_press(self, hold_keys: List, press_keys: List):
'''Hold hold_keys and press press_keys in sequence.
Example: agent.hold_and_press(hold_keys=["ctrl"], press_keys=["a"])'''

def hotkey(self, keys: List):
'''Press hotkey combo. Prefer over clicking menu items (faster). Do NOT use for switching apps--
use switch_applications.
Example: agent.hotkey(keys=["ctrl", "c"])'''

def open(self, app_or_filename: str):
'''Open app or file by name (do not open manually). Do NOT use for already open apps--use
switch_applications.
Use only the filename, NOT the full path.
Available: Google Chrome, LibreOffice Writer, LibreOffice Calc, LibreOffice Impress, Thunderbird,
 VSCode, VLC Media Player, GIMP.
Example: agent.open(app_or_filename="Google Chrome")
Example: agent.open(app_or_filename="report.xlsx")'''
```

```
def scroll(self, instruction: str, clicks: int, shift: bool = False):
'''Scroll at the described element. clicks: positive=up, negative=down (or left/right if shift=
True). clicks=1 ~ 3 lines; small 1-5, section 5-10, long 10-20. Never exceed 50 in one action;
use multiple smaller scrolls instead.
Example: agent.scroll(instruction="the main content area", clicks=-5, shift=False)'''

def set_cell_values(self, cell_values: Dict[str, Any], app_name: str, sheet_name: str):
'''Set cell values in a spreadsheet. ALWAYS use for spreadsheet data--do NOT use click+type for
cells.
cell_values: e.g. {"A1": "Name", "B1": 42}. Types: float, int, string, bool, formulas (start with
 "=").
app_name: the spreadsheet FILE name as shown in the window title (e.g. "data.xlsx"), not "
LibreOffice Calc".
sheet_name: the sheet tab name (e.g. "Sheet1").
Example: agent.set_cell_values(cell_values={"A6": "Hello World"}, app_name="data.xlsx",
sheet_name="Sheet1")'''

def switch_applications(self, app_code):
'''Switch to an app by its code from the list of open applications.
Example: agent.switch_applications(app_code="google-chrome")'''

def type(self, element_description: str, text: str = '', overwrite: bool = False, enter: bool =
False):
'''Type text into a specific element. Grounding clicks the element first. Always specify
element_description even for focused fields.
overwrite: True to clear existing text first.
enter: True to press Enter after typing.
Example: agent.type(element_description="the terminal window", text="cd ~/project && make build",
 enter=True)
'''

def wait(self, time: float):
'''Wait time seconds. Use only when progress bar or load indicator is visible.
Example: agent.wait(time=2.0)'''

# POLICY RULES:
- Do not repeat failed actions from history.
- Act only on visible elements. Output exactly ONE valid `agent.*(...)` call; no multiple calls
or semicolons.
- Prefer the shortest reliable method: terminal or agent.hotkey() over GUI when equivalent.
- Save with Ctrl+S before agent.done().
- "Authentication required" → Cancel.
- Follow the user instruction exactly. Do not change unrelated UI, close unrelated Chrome tabs,
or be distracted by irrelevant page/file content.
- Before agent.done(), visually verify the task is complete and values/text/formatting match the
instruction. If uncertain, take another step to confirm, verify the result.
- Do NOT call agent.fail() unless the tutorial says infeasible or you have tried fundamentally
different approaches.
- On the last step, call agent.done() or agent.fail().
- Never kill/restart the desktop environment, log out, or restart the system.
- When adjusting image/media properties, use moderate values unless the instruction specifies
exact amounts or "maximum"/"extreme".
Password: '{CLIENT_PASSWORD}' for sudo. Use this actual value when typing passwords (not the
literal placeholder text)."""

# OUTPUT FORMAT:
Thought: Must have the following sections:
(1) Step by Step Progress Assessment---analyze completed task parts, reflect on potential errors
or unexpected results, predict a recovery step if previous action was incorrect.
(2) Next Action(s) Analysis---list possible next actions based on current state, evaluate options
 considering current state and previous actions, propose most logical next action(s).
(3) Tutorial Reference: QUOTE the exact relevant sentence(s) from the tutorial. If none applies,
say ``No tutorial reference for this step.'' If the current method isn't working, switch to
another method from the tutorial.

Actions: After the Thought, output exactly {K} action lines separated by {DELIMITER}.
Each line must start with "Action:" and contain exactly one agent.*(...) call.
You must output all {K} candidates; do not stop after the first one.
```

```
Example:
Thought: {thinking in here}
Action: ```agent.fail()``` {DELIMITER}
Action: ```agent.hotkey(keys=["ctrl", "s"])``` {DELIMITER}
Action: ```agent.type(element_description="the filename field", text="output.txt", enter=True)```

# TUTORIAL
{a retrieved tutorial, None if not available}

# TASK INSTRUCTION
{task instruction}

# HISTORY
history of past steps (Thought, Action, and screenshots for the most recent K=3 steps, and only
actions for the rest of the trajectory)

# CURRENT SCREENSHOT
{current screenshot}

# STEP
You are at step {current_step} of the max step {max_steps}.

# OUTPUT FORMAT
Output exactly {K} candidate actions separated by
{DELIMITER}'. Follow the policy output format from the system message.
```

Figure 8: Policy prompt used to generate candidate actions from the task instruction, retrieved tutorial, interaction history, and current screenshot.

# D Judge Prompt

```
You are an expert at evaluating computer-use agent planning on Ubuntu.
You are given a task instruction and a tutorial for this task if available. Each step you receive:
 (1) history of past steps (Thought and Action); (2) recent screenshots, and (3) a list of
candidate actions that the agent can take at the current step. Your task is to evaluate the
candidate actions and select the one that best advances toward the overall goal, based on the
provided information.

# ACTION SPACE:
Allowed agent methods (you MUST only use these, with correct arguments):
- agent.click(instruction, num_clicks=1, button_type='left', hold_keys=[])
- agent.done(return_value=None)
- agent.drag_and_drop(starting_description, ending_description, hold_keys=[])
- agent.fail()
- agent.highlight_text_span(starting_phrase, ending_phrase)
- agent.hover(instruction)
- agent.hold_and_press(hold_keys, press_keys)
- agent.hotkey(keys)
- agent.open(app_or_filename)
- agent.scroll(instruction, clicks, shift=False)
- agent.set_cell_values(cell_values, app_name, sheet_name)
- agent.switch_applications(app_code)
- agent.type(element_description, text='', overwrite=False, enter=False)
- agent.wait(time)

# EVALUATION CRITERIA:
Use the actual screenshots as ground truth. Tutorial alignment is guidance unless the tutorial
starts with `Infeasible'.
1. Feasibility: Can the action actually be performed from the current screenshot? If the target
element is not visible, reject it. If an after-screenshot or rollout is shown, it must support
that the action worked.
2. Exactness: Does the action relate to the user task? No extra or unnecessary steps are
performed.
3. Effectiveness: Does the action make meaningful progress in the available screenshot(s),
```

```
rollout, or final state?
4. Completion: Is the task already done? If yes, do agent.done().
5. Tutorial alignment: Prefer candidates that match a feasible tutorial, but trust the current
screen over the tutorial when they differ.
6. No regression: Reject actions that break state, navigate away, or move away from the goal.

# RULES:
1. OVERRIDE - Infeasible tutorial: If a tutorial is provided and starts with `Infeasible', agent.
fail() is the only correct action.
2. Prefer terminal over GUI; between GUI options, prefer shorter or more efficient actions (
hotkey > several clicks).
3. Do NOT select an action the agent has already tried 2-3 times without progress.
4. If a tutorial is provided and does NOT start with `Infeasible', prefer candidates that follow
it, but trust the actual screenshot(s) over the tutorial when they differ.
5. Judge if the instruction can be EXACTLY fulfilled with agent.done() or is IMPOSSIBLE with
agent.fail().
6. On the LAST step, only agent.done() or agent.fail() can succeed.
7. Google Chrome is the ONLY feasible browser.

# TUTORIAL
{tutorial, None if not available}

# TASK INSTRUCTION
{instruction}

# HISTORY
history of past steps (Thought, Action, and screenshots for the most recent K=3 steps, and only
actions for the rest of the trajectory)

# CURRENT SCREENSHOT
{current_screenshot}

# CANDIDATE ACTIONS
{candidate_actions}

# STEP
You are on step {current_step} of maximum {max_steps} steps.

# TASK:
Respond only with valid JSON (no extra keys or comments).
{
  "reason": "Thinking about the best action to choose, based on the criteria and rules above.",
  "best_index": {index of the best action}
}
"best_index" MUST be an integer in the inclusive range [0, number_of_candidates - 1].
```

Figure 9: Judge prompt used to select the best candidate action based on the current state, task instruction, retrieved tutorial, and evaluation criteria.

# E Grounding Prompt

```
You are a GUI grounding agent. Given a screenshot and user's grounding instruction, locate the UI
 element. Examine the screenshot carefully, think step by step about which element matches, then
provide the coordinate.

Match the requested semantic label, not merely a similar-looking icon. In LibreOffice Calc, "
Background Color", "Fill Color", or "paint bucket" means the cell background/fill-color bucket
control, not the funnel-shaped AutoFilter icon. The output should just be the coordinates of a
point, in the format [x,y]. Additionally, if the task is infeasible (e.g., the task is not
related to the image), the output should be [-1,-1].

# Task instruction:
{}.

# Screenshot:
```

```
{}
```

Figure 10: Grounding prompt used by UI-Venus-1.5 to localize UI elements from grounding instructions and screenshots.

# F R-WoM reimplementation

## F.1 Implementation details

We reimplement the R-WoM method based on the description in the paper. We make the following implementation changes to R-WoM to make it work and compatible with our environment and to other baselines for a more controlled comparison. We want to stress that these changes do not worsen or even improve the performance of R-WoM, which is to use a text-based world model for rollout and a judge to evaluate the rollout.

1. We use action space defined by Agent-S2 (Agashe et al., 2025b) , instead of the original action space of OSWorld (Xie et al., 2024) used in R-WoM. This action space makes agent act more effectively and is more widely adopted by other GUI agent works, such as the baseline GTA1 (Yang et al., 2026b), Agent-S family (Agashe et al., 2025b; Gonzalo et al., 2025) and UITAR1.5 (Qin et al., 2025b).

2. We notice text-based rollout of R-WoM does not produce well-structured rollout that follows the rules in the prompt, which makes it hard for R-WoM judge (ranker) to judge from the rollout. For example, given the horizon $H = 2$, it does not stop at the state 2 as required, and it stops at rollout actions instead. Therefore, we design a new rollout prompt with stricter rules and a more structured output format, as shown in the rollout prompt in Figure 11.

3. To make it fair to compare R-WoM with other baselines such as GTA1 (Yang et al., 2026b), we add judge rules from GTA1 to R-WoM judge prompt.

4. We do not reuse policy prompt from R-WoM, since it is not compatible with other baselines and it is independent of the rollout ability of text-based world model in R-WoM. Instead, we use policy prompt from GTA1 (Yang et al., 2026b) for all methods, including R-WoM.

## F.2 Rollout prompt

```
You are a world-model assistant with extensive knowledge of desktop and web UIs.
Given the task objective, a candidate action, and the observation before the candidate action,
you must "simulate the future" and describe the plausible future states.
```

**Modified prompt**

```
# ACTION SPACE:
Allowed agent methods (you MUST only use these, with correct arguments; there is NO
right_click/double_click method):
- agent.click(instruction, num_clicks=1, button_type='left', hold_keys=[])  # right-click:
button_type='right'; double-click: num_clicks=2
- agent.done(return_value=None)
- agent.drag_and_drop(starting_description, ending_description, hold_keys=[])
- agent.fail()
- agent.highlight_text_span(starting_phrase, ending_phrase)
- agent.hover(instruction)
- agent.hold_and_press(hold_keys, press_keys)
- agent.hotkey(keys)
- agent.open(app_or_filename)
- agent.scroll(instruction, clicks, shift=False)
- agent.set_cell_values(cell_values, app_name, sheet_name)
- agent.switch_applications(app_code)
- agent.type(element_description, text='', overwrite=False, enter=False)
- agent.wait(time)
```

```
# TUTORIAL USAGE GUIDELINES

1. Use tutorials to identify efficient workflow patterns that should be predicted as likely
outcomes.
2. Provide a reference to the tutorial if the current situation matches the standard operations
in the tutorials.
3. If the current situation does not align with tutorials, rely on internal world knowledge
instead.

# ENVIRONMENT AWARENESS CHECKLIST

- Visible UI elements: text, icons, menus, modals, tooltips.
- Element states: enabled/disabled, focused/hovered, loading progress.
- Hidden or off-screen affordances revealed by scrolling or clicking.
- Cursor position, caret position, selection highlights.
- Global context: file system changes, network requests, OS dialogs.

# OUTPUT FORMAT

Produce an ordered chain from STATE 0 (current) up to STATE {k_steps}.
```

```
Additional prompt

# STRICT RULES:
- The highest allowed STATE index is {k_steps}. Never output STATE {k_steps} + 1 or any larger
  index.
- After STATE 0, output ACTION 0: followed by the candidate action exactly as provided in the
user message.
- Between every two consecutive states, you MUST output an ACTION line.
- Every STATE i (for i < {k_steps}) MUST be followed by ACTION i: before STATE i+1.
- STOP IMMEDIATELY after STATE {k_steps}.
- Do NOT output ACTION {k_steps} or any text after the last STATE.
- The response must end with STATE {k_steps}: <description>.
- Do not add explanations, code fences, headings, or extra text before or after the rollout.

# EXAMPLE OUTPUT:
{rollout_example}

## WRONG (do NOT do this):
STATE {k_steps}: <description>
ACTION {k_steps}: <any action here>

The ACTION {k_steps} line is forbidden; the response must end at STATE {k_steps}.

# TASK OBJECTIVE

{instruction}

# TUTORIAL EVIDENCE (IF ANY)

{evidence_text}

# HISTORY
history of past steps (Thought, Action, and screenshots for the most recent K=3 steps, and only
actions for the rest of the trajectory)

# CURRENT SCREENSHOT

{current_obs_caption}

# CANDIDATE THOUGHT AND ACTION

Thought: {thought}

Action: {action_code}

# CURRENT STEP

You are on step {current_step} of maximum {max_steps} steps.
```

Figure 11: R-WoM prompt in our reimplementation.

### Rollout Example, K=1

```
ROLLOUT_EXAMPLE_K1 = """STATE 0 (current): A browser settings page is open with the search field
focused and empty.
ACTION 0: agent.typewrite(text="downloads")
STATE 1: The search field contains "downloads", and matching download-related settings are
visible."""
```

### Rollout Example, K=2

```
ROLLOUT_EXAMPLE_K2 = """STATE 0 (current): A browser settings page is open with the search field
focused and empty.
ACTION 0: agent.typewrite(text="downloads")
STATE 1: The search field contains "downloads", and matching download-related settings are
visible.
ACTION 1: agent.click(instruction="Click the Change button for download location")
STATE 2: A file chooser dialog opens with folder choices and Cancel and Select buttons."""
```

### Rollout Example, K=3

```
ROLLOUT_EXAMPLE_K3 = """STATE 0 (current): A browser settings page is open with the search field
focused and empty.
ACTION 0: agent.typewrite(text="downloads")
STATE 1: The search field contains "downloads", and matching download-related settings are
visible.
ACTION 1: agent.click(instruction="Click the Change button for download location")
STATE 2: A file chooser dialog opens with folder choices and Cancel and Select buttons.
ACTION 2: agent.click(instruction="Select the Documents folder")
STATE 3: The Documents folder is highlighted in the file chooser, ready to be selected as the new
 download location."""
```

We use different examples with different K steps of rollout.

## F.3 R-WoM judge

**R-WoM Judge Prompt**

```
You are an expert at evaluating computer-use agent planning on Ubuntu. You are given a task
instruction, a tutorial for this task if available, the agent history, the current screenshot,
candidate actions, and simulated rollouts for each candidate action. Your task is to rank the
candidate actions by how effectively they advance toward the overall goal, considering both the
current observation and their simulated future outcomes.
```

**Modified prompt**

```
# ACTION SPACE:
Allowed agent methods (you MUST only use these, with correct arguments; there is NO
right_click or double_click method):
- agent.click(instruction, num_clicks=1, button_type='left', hold_keys=[])  # right-click:
button_type='right'; double-click: num_clicks=2
- agent.done(return_value=None)
- agent.drag_and_drop(starting_description, ending_description, hold_keys=[])
- agent.fail()
- agent.highlight_text_span(starting_phrase, ending_phrase)
- agent.hover(instruction)
- agent.hold_and_press(hold_keys, press_keys)
- agent.hotkey(keys)
- agent.open(app_or_filename)
- agent.scroll(instruction, clicks, shift=False)
- agent.set_cell_values(cell_values, app_name, sheet_name)
- agent.switch_applications(app_code)
- agent.type(element_description, text='', overwrite=False, enter=False)
- agent.wait(time)
```

```
Additional prompt

# EVALUATION CRITERIA:
Use the actual screenshots as ground truth. Tutorial alignment is guidance unless the tutorial
 starts with `Infeasible'.
1. Feasibility: Can the action actually be performed from the current screenshot? If the
target element is not visible, reject it. If a rollout is shown, it must support that the
action worked.
2. Exactness: Does the action relate to the user task? No extra or unnecessary steps are
performed.
3. Effectiveness: Does the action make meaningful progress in the rollout or final simulated
state?
4. Completion: Is the task already done? If yes, rank agent.done() highest.
5. Tutorial alignment: Prefer candidates whose rollouts follow a feasible tutorial, but trust
the current screen over the tutorial when they differ.
6. No regression: Reject actions whose rollouts break state, navigate away, or move away from
the goal.

# RULES:
1. OVERRIDE - Infeasible tutorial: If a tutorial is provided and starts with `Infeasible',
agent.fail() is the only correct action.
2. Prefer terminal over GUI; between GUI options, prefer shorter or more efficient actions (
hotkey > several clicks).
3. Do NOT rank an action highest if the agent has already tried it 2-3 times without progress.
4. If a tutorial is provided and does NOT start with `Infeasible', prefer candidates that
follow it, but trust the actual screenshot(s) over the tutorial when they differ.
5. Judge if the instruction can be EXACTLY fulfilled with agent.done() or is IMPOSSIBLE with
agent.fail().
6. On the LAST step, only agent.done() or agent.fail() can succeed.
7. Google Chrome is the ONLY feasible browser.

# TASK INSTRUCTION:
{instruction}

# TUTORIAL EVIDENCE (if available):
{evidence_text}

# STEP HISTORY:
history of past steps (Thought, Action, and screenshots for the most recent K=3 steps, and only
actions for the rest of the trajectory)

# CURRENT SCREENSHOT:
{current screenshot}

# CANDIDATE ACTIONS AND ROLLOUTS:
Each block below contains one candidate action and the corresponding world-model simulation.
Evaluate each candidate's feasibility, correctness, and effectiveness, then rank them.
{rollouts_block}

# STEP:
You are at step {current_step} of the max step {max_steps}.

# TASK:
Respond only with valid JSON (no extra keys or comments).
{
  "reason": "Thinking about the ranking of the candidate actions, based on the criteria, rules,
current screenshot, tutorial, and rollouts above.",
  "ranking": [{indices of candidate actions from best to worst}]
}
"ranking" MUST contain every candidate index exactly once.
```

Figure 12: R-WoM judge prompt in our reimplementation.

# G Visual Ground-Truth implementation details

To find the ground-truth screenshot for each step in the serial option, we first use QEMU virtual machine supported by Singularity as our environment for OSWorld. We first save a QEMU snapshot of the current VM state before evaluating any candidate action. This snapshot saves the exact state of the VM, including the screen, memory, and disk, allowing us to restore it to this state after each candidate action is executed. We then execute each candidate action one at a time on the same VM, wait for the interface to settle, and capture the resulting screenshot as that candidate's ground-truth next state. After each candidate is executed, the VM is restored back to the saved snapshot, so the next candidate starts from the exact same screen state. This makes the comparison fair and deterministic on most of the tasks, with a few exception when the task involves internet access and the content on the screen may change (e.g., Google search results). See more details in Appendix I.

```
You are a desktop computer agent on Ubuntu with internet access that evaluates actions by
considering the observation before the action candidates and the potential outcomes of these
actions. Rank the candidate actions by how effectively they advance toward the goal, considering
both the current observation and their simulated future outcomes.

# ACTION SPACE
Allowed agent methods (you MUST only use these, with correct arguments; there is NO right_click/
double_click method):
- agent.click(instruction, num_clicks=1, button_type='left', hold_keys=[])  # right-click:
button_type='right'; double-click: num_clicks=2
- agent.done(return_value=None)
- agent.drag_and_drop(starting_description, ending_description, hold_keys=[])
- agent.fail()
- agent.highlight_text_span(starting_phrase, ending_phrase)
- agent.hover(instruction)
- agent.hold_and_press(hold_keys, press_keys)
- agent.hotkey(keys)
- agent.open(app_or_filename)
- agent.scroll(instruction, clicks, shift=False)
- agent.set_cell_values(cell_values, app_name, sheet_name)
- agent.switch_applications(app_code)
- agent.type(element_description, text='', overwrite=False, enter=False)
- agent.wait(time)"""

Multi-horizon rollout (depth={depth})**: Each candidate shows a trajectory of {depth} steps
executed on the real VM. The first action is the candidate's proposed action; subsequent actions
were chosen by the policy model based on the resulting screenshot. Evaluate the FULL trajectory;
prefer the candidate whose trajectory makes the most cumulative progress.

# EVALUATION CRITERIA:
Use the actual screenshots as ground truth. Tutorial alignment is guidance unless the tutorial
starts with 'Infeasible'.
1. Feasibility - Can the action actually be performed from the current screenshot? If the target
element is not visible, reject it. If an after-screenshot or rollout is shown, it must support
that the action worked.
2. Exactness - Does the action relate to the user task? No extra or unnecessary steps are
performed.
3. Effectiveness - Does the action make meaningful progress in the available screenshot(s),
rollout, or final state?
4. Completion - Is the task already done? If yes, do agent.done().
5. Tutorial alignment - Prefer candidates that match a feasible tutorial, but trust the current
screen over the tutorial when they differ.
6. No regression - Reject actions that break state, navigate away, or move away from the goal.

# TUTORIAL GROUNDING GUIDANCE (IF TUTORIAL IS PROVIDED):
Prioritize action sequences that follow the standard operations in the tutorials and have
captured the milestones and conditions to make more meaningful progress to achieve the task
objective.

# RULES:
(1) **OVERRIDE - Infeasible tutorial**: If a tutorial is provided and starts with 'Infeasible',
agent.fail() is the only correct action.
(2) Prefer terminal over GUI; between GUI options prefer shorter/more efficient actions (hotkey >
 several clicks).
```

```
(3) Do NOT select an action the agent has already tried 2-3 times without progress.
(4) If a tutorial is provided and does NOT start with 'Infeasible', prefer candidates that follow
 it, but trust the actual screenshot(s) over the tutorial when they differ.
(5) Judge if the instruction can be EXACTLY fulfilled with agent.done() or is IMPOSSIBLE with
agent.fail().
(6) On the LAST step, only agent.done() or agent.fail() can succeed.
(7) Google Chrome is the ONLY feasible browser.

Evaluate each candidate by its full observed trajectory, not just the action text.
Prefer the candidate whose resulting state makes the most progress toward the task goal.
For multi-step trajectories, evaluate cumulative progress across ALL steps and focus on the FINAL
 screenshot.

# TASK INSTRUCTION
{instruction}

# HISTORY
history of past steps (Thought, Action, and screenshots for the most recent K=3 steps, and only
actions for the rest of the trajectory)

# CURRENT SCREENSHOT
{current_screenshot}

# CANDIDATE ACTIONS
{candidate_actions}

# STEP
You are on step {current_step} of maximum {max_steps} steps.

# TASK
Based on the ACTUAL screenshots above, select the candidate that makes the most progress toward
the task goal.

Respond **only** with valid JSON (no extra keys or comments). `ranking` MUST contain each
candidate index exactly once, ordered from best to worst. The first index in `ranking` is the
selected action. Valid candidate indices are 0 to {n_candidates_minus_one}. Never output `-1`.

```json
{{
  "thought": "what is the best action to choose given the evaluation above?",
  "ranking": [x, x, x]
}}
```
```

Figure 13: Visual ground-truth judge prompt used to rank candidate action trajectories using screenshots captured from the virtual machine.

# H Ground-Truth Error Analysis

We follow the same procedure described in Qian et al. (2026) to analyze the error of the ground-truth visual rollout. In total, we sample 2116 failed tasks from all runs with ground-truth rollouts. We first sample 30 tasks and manually go through trajectory data: screenshots, policy "thought" tokens, judge "thought" tokens, and proposed actions. We then identify and summarize the following core categories of errors that lead to the failure of the agents:

1. **Wrong MLLM-generated tutorial**: MLLM-generated tutorials are not correct and the agent doesn't know how to recover from wrong steps in the tutorial. We show an example of this case in Figure 14.

2. **Policy fails to follow the tutorial correctly**: The policy itself misunderstands, skips, contradicts, or wrongly executes a tutorial step. We show an example of this case in Figure 15.

3. **Grounding error**: The grounding model maps the same action description to different regions in the visual input, which leads to wrong grounding and thus failure. We show an example of this case in Figure 19.

4. **Observation hallucination in policy**: The policy misreads the current state and thus reasons toward a wrong action. We show an example of this case in Figure 18.

5. **The judge fails to reason over future states**: GT-rollout judges can fail to reason over future states to wrongly judge the progress of the task. We show an example of this case in Figure 20 and Figure 16.

6. **The judge fails to understand task success**: We find a notable portion of cases where the GT-rollout judge fails to understand task success, which is a more specific failure mode. GT-rollout judges fail to understand whether future states satisfy the task instruction. We show an example of this case in Figure 17.

7. **Web server error**: The web server hosting the task environment has issues, such as the target website being down. This is an external factor that can lead to failure, and it is not caused by the agent's decision-making process.

8. **Agent doing correctly but runs out of max steps**: The agent is following a correct path but reaches the 15-step or 50-step limit before completing the task. We show an example of this case in Figure 21.

We then use an MLLM (Gemini 3.5 Flash (Google DeepMind, 2026)) to annotate the rest of the failed tasks. Given the trajectory data of a task, we prompt it (see Figure 26) to classify failure reasons into above categories. Because a failed task might have many root causes, we adopt Qian et al. (2026) to ask the MLLM to classify potential errors into: 1) a primary reason and 2) mentioned (secondary) reasons. Note that mentioned reasons cannot be duplicated with the primary reason.

To assess the reliability of this MLLM-as-a-judge framework, we run MLLM with above manually annotated examples by human and find that the agreement rate for the primary reason reaches 90%. Furthermore, we randomly sample 20 examples from MLLM results to examine manually. All of the results are reasonable and aligned with our predefined categories of error. Based on MLLM's result, we plot number of primary and mentioned reasons in Figure 3.

**Task ID:** `a96b564e-dbe9-42c3-9ccf-b4498073438a`

**Tutorial.** <...>

Once you are on the "Top" page, you can narrow down the timeframe to see which discussion has the most replies in history:

- Look for the dropdown menu that usually says "this week" or "this month" (located just below the main tabs).
- Change this selection to "All Time".
- **Incorrect tutorial step.** The list will refresh. Look at the "Replies" column on the right side of the list. The discussion at the very top of this list will be the one with the most replies in FlightAware history.

<...>

**Correct step.** Click "Replies" to sort the discussions by number of replies, then select the discussion at the top of the sorted list.
The list will refresh. Look at the "Replies" column on the right side of the list. The discussion at the very top of this list will be the one with the most replies in FlightAware history.

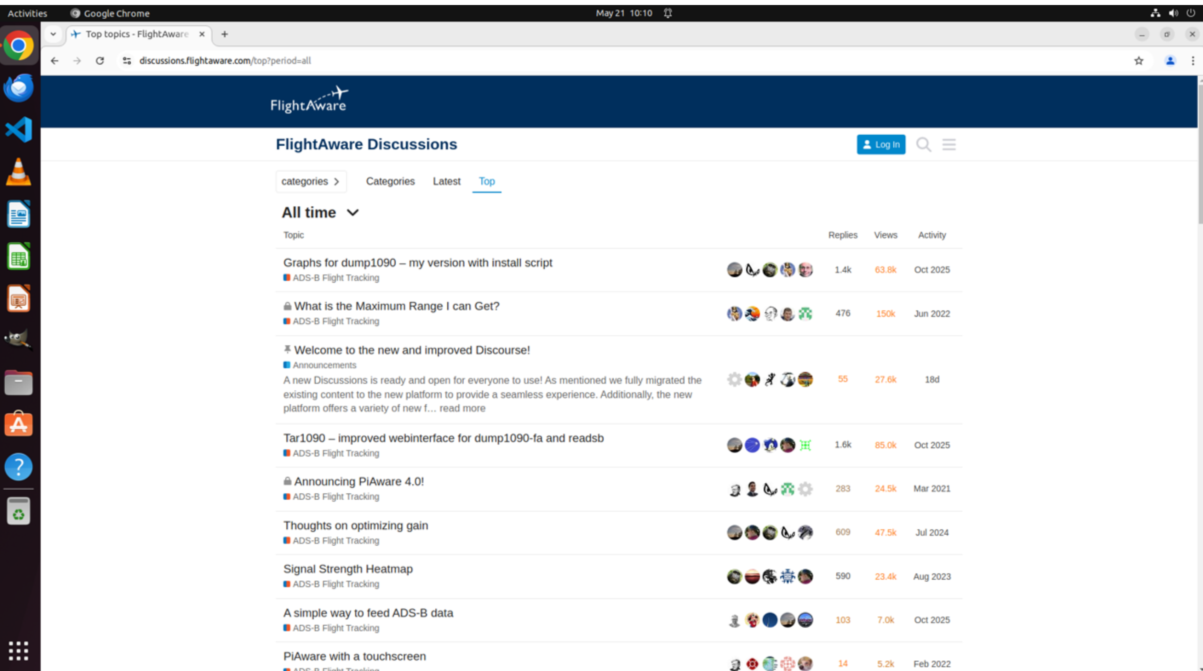



Figure 14: An example of a wrong tutorial. The top panel shows the tutorial given to the agent, and the bottom panel shows the screenshot where agents fails.

**Example:** Agents fail to follow the tutorial correctly.

**Task ID:** `eb303e01-261e-4972-8c07-c9b4e7a4922a`

**Task.** Tomorrow, I am scheduled to deliver a talk, and my PowerPoint slides and speaking notes are saved on the desktop. Help me insert my planned remarks for each slide into the “note” section of the PowerPoint as a reminder. I have completed this task for some slides; assist me in completing the remaining part.

**Failure.** The agent should copy the planned remarks from `notes.docx` and paste them into the active notes area, but instead it hallucinates and types the slide’s own content into the notes section.

**Tutorial.**

**3. Enable the Notes View**

- If the notes input area is not visible at the bottom of the main workspace, go to the top menu and select `View > Notes`.
- Alternatively, you can click the Notes tab located directly above the main slide workspace.

**4. Copy and Paste the Remarks**

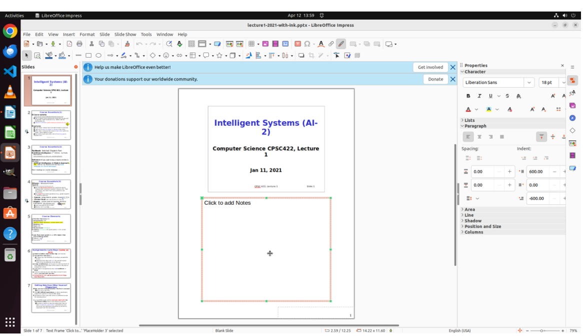
- Switch back to the `notes.docx` window.
- Highlight and copy (`Ctrl + C`) the specific text corresponding to the active slide.

**4. Copy and Paste the Remarks (continued)**

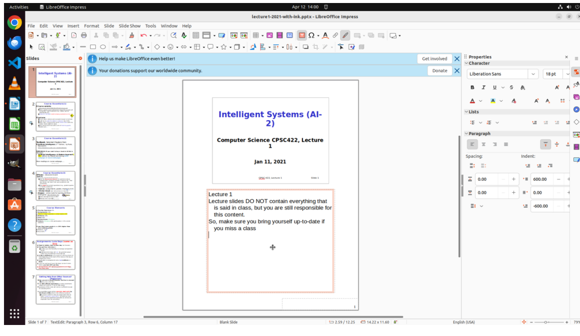
- Switch back to the presentation window and click inside the area labeled “Click to add notes” below the slide image.
- Paste the text (`Ctrl + V`).

**5. Repeat for Remaining Slides**

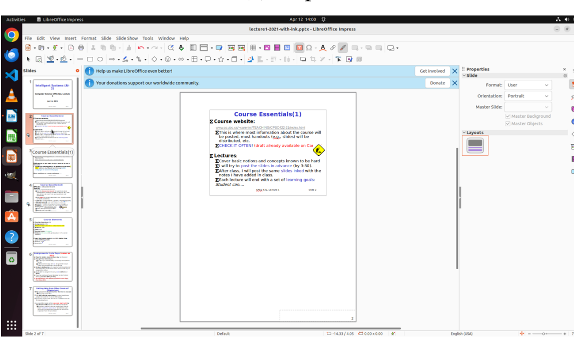
- Select the next slide in the sequence from the left pane.
- Repeat the copy-paste process for all subsequent slides until all planned remarks from the Word document are successfully transferred to the PowerPoint file.



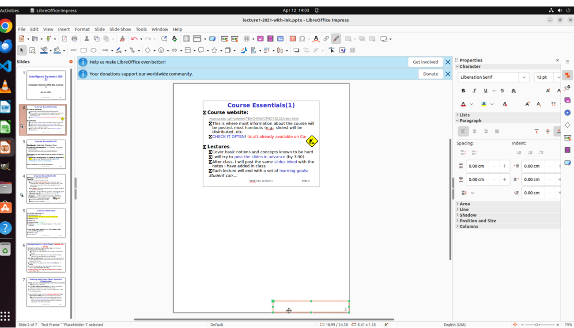

(a) Step 6.




(b) Step 7.




(c) Step 8.




(d) Step 9.

Figure 15: Example of agents failing to follow the tutorial correctly. The policy should copy planned remarks from `notes.docx` and paste them into the PowerPoint notes section, but instead types slide content directly into the notes area.

**Tutorial-consistent action (should have done):**
Copy the remaining remarks from `notes.docx`, return to the presentation, and paste them into the active notes area.

**Executed action at Step 9:**
```
agent.type(element_description="the ‘Click to add Notes’ area",
text="Course Essentials\n... Student can...", enter=True)
```

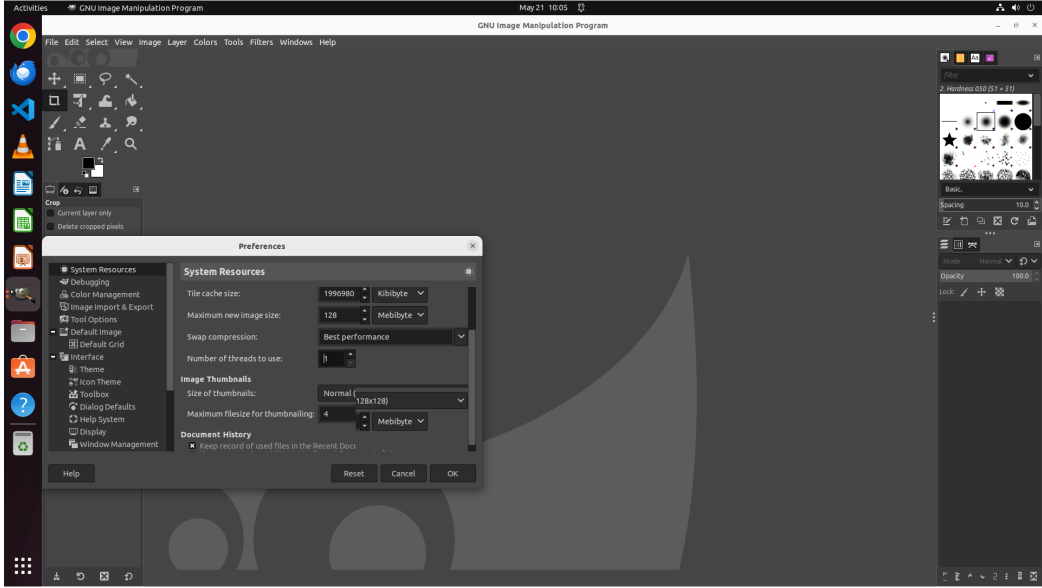
**Other candidate actions:**
```
agent.switch_applications(app_code="libreoffice-writer")
agent.done()
```

**Policy thought at Step 9.** The Notes area for Slide 2 is active and ready for pasting the remarks. I have already copied the text for Slide 2 from the notes.docx file in a previous step. The next logical action is to paste the text into this active Notes area using Ctrl+V. This aligns with the tutorial's instruction to paste the text after switching back to the presentation window.

Figure 15: Example of agents failing to follow the tutorial correctly (continued).

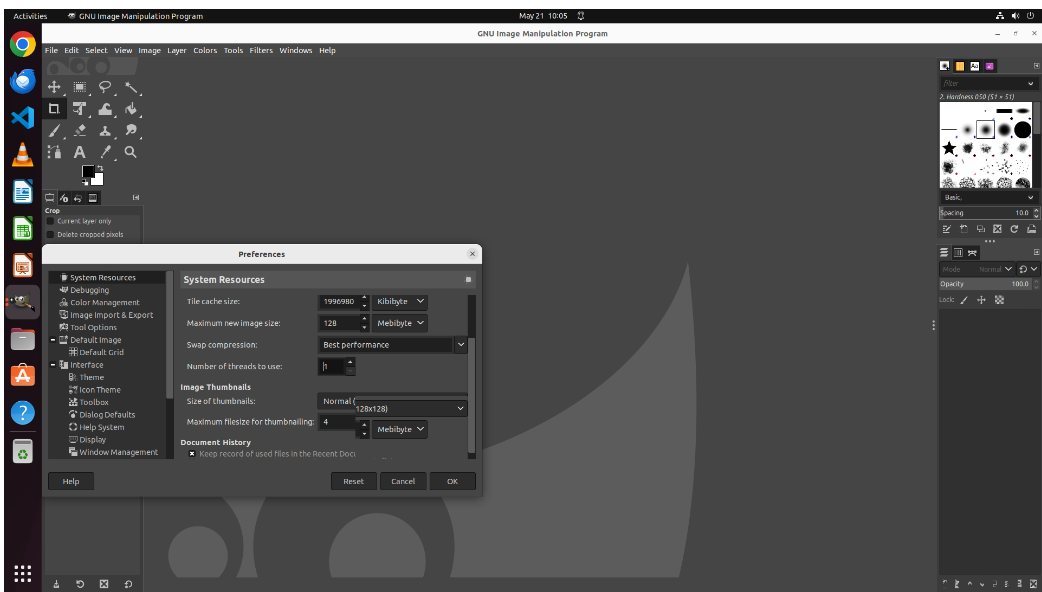
**Task ID:** `7b7617bd-57cc-468e-9c91-40c4ec2bcb3d`

**Judge tokens.** GT-Rollout judge: The goal is to set the minimum number of undo steps to 100. The current screenshot shows the “Document History” section, but the specific “Minimal number of undo levels” setting is not visible. Previous scrolling attempts have not revealed it. The “Document History” section might need to be expanded or further scrolled to locate the setting. However, the hotkey action did not yield any visible progress toward the goal. The best course of action is to continue scrolling within the “Document History” section to locate the desired setting.




(a) Current state.

**Chosen Action.**
```
agent.scroll(instruction="at the center of the windows Preference",
clicks=-5)
```



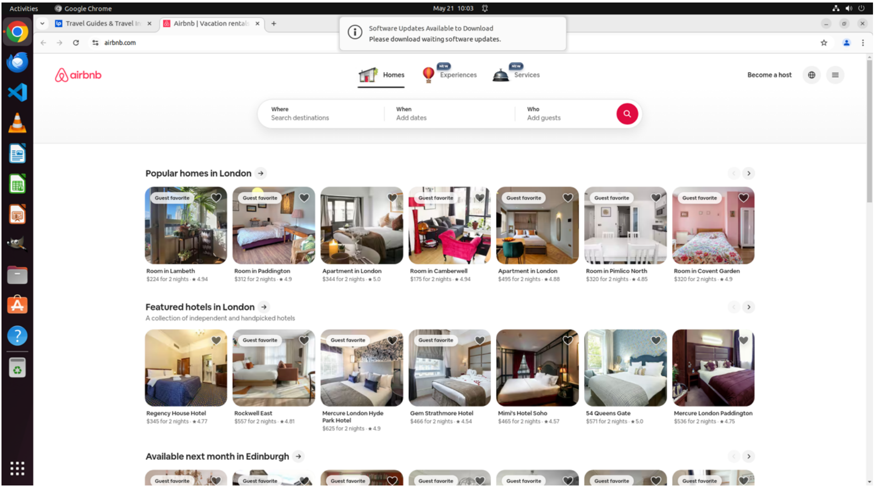

(b) GT future state.

Figure 16: An example where the judge fails to reason task process over the future. The judge recommends continued scrolling from the current state, even when it realizes the GT future state does not show progress.

**Example:** Fail to find the target state.

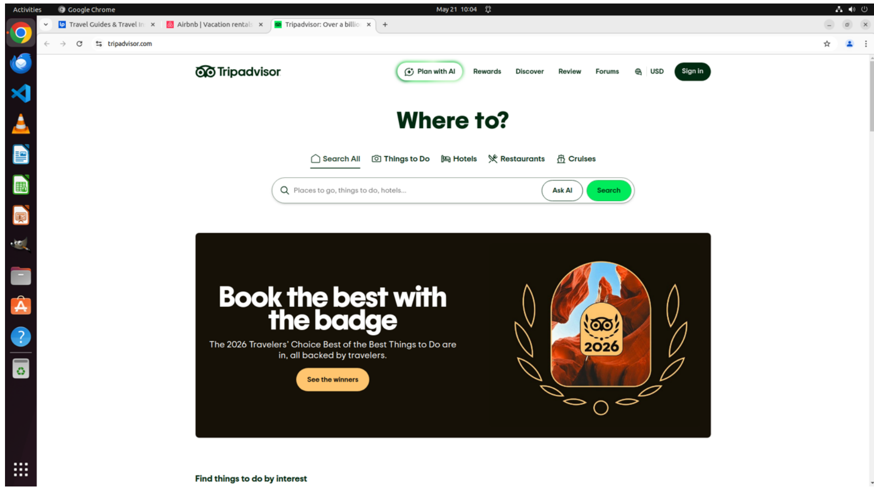
**Task ID:** `06fe7178-4491-4589-810f-2e2bc9502122`

**Task.** Can you make my computer bring back the last tab I shut down?




(a) State 1.

**Chosen Action 1.** `agent.hotkey(keys=['ctrl', 'shift', 't'])`




(b) State 2.

**Index 0 (should have chosen):**
```
agent.done()
```

**Index 1 (chosen):**
```
agent.hotkey(keys=['ctrl', 'shift', 't'])
```

**Index 2:**
```
agent.click(instruction="the 'Tripadvisor' tab in the browser",
num_clicks=1, button_type='left', hold_keys=[])
```

**Judge tokens.** GT-Rollout judge: The task is to restore the last closed tab. The tutorial confirms that `Ctrl+Shift+T` is the correct method. Candidate 1 successfully executed this shortcut and brought up the history page, which indicates the tab was restored. Candidate 0 does nothing and is not useful. Candidate 2 clicks on the current tab, which does not restore a closed tab. Therefore, Candidate 1 is the best choice as it directly fulfills the task.

Figure 17: An example where the judge fails to understand task success. The judge selects the restore-tab shortcut in State 1, observes State 2 after that action, and still selects the highlighted restore-tab option again.

**Example:** Wrong reasoning over the current state.

**Task ID:** `82bc8d6a-36eb-4d2d-8801-ef714fb1e55a`

**Task.** On next Monday, look up a flight from Mumbai to Stockholm.

**Policy thought.** The current date selected is May 21, 2026, which is a Wednesday. The task requires selecting the upcoming Monday. The next Monday after May 21 is May 26, 2026. I need to click on May 26 in the calendar to set the correct departure date.

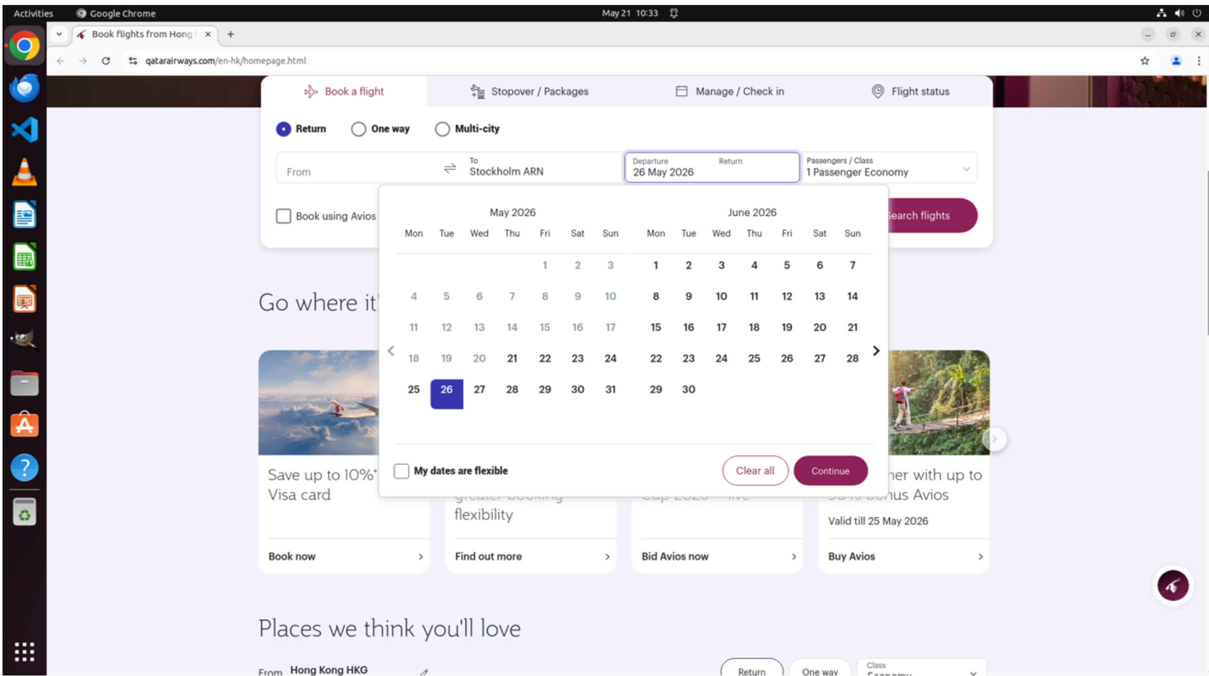


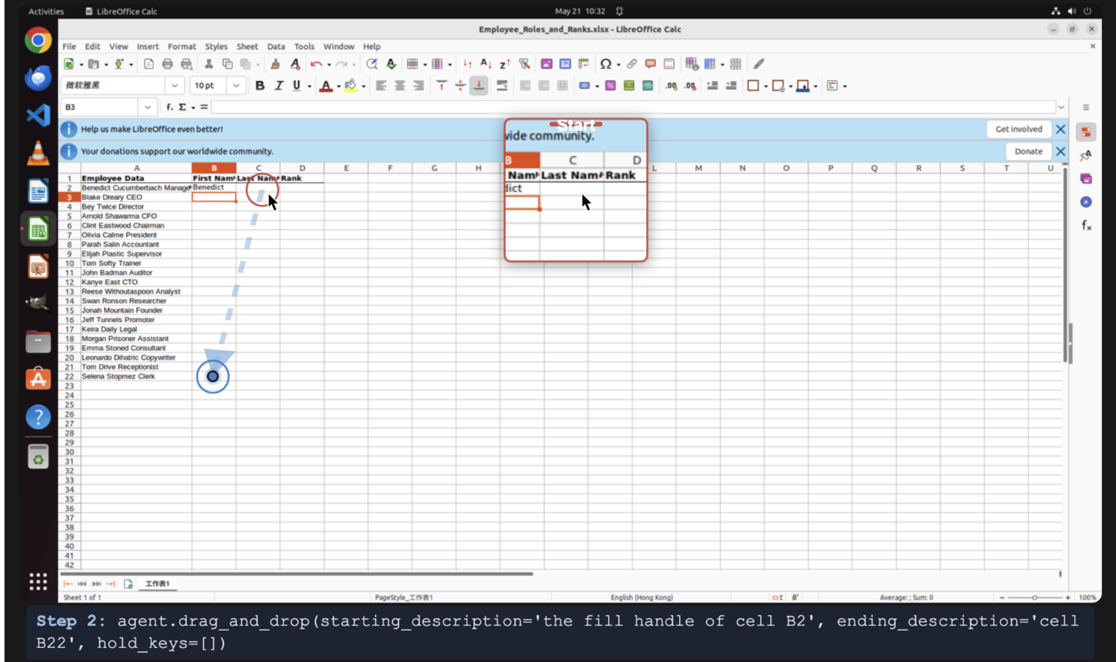

Figure 18: Wrong reasoning over the current state. The policy misreads the currently selected departure date and reasons toward an incorrect next-Monday target.

**Example:** Wrong grounding.

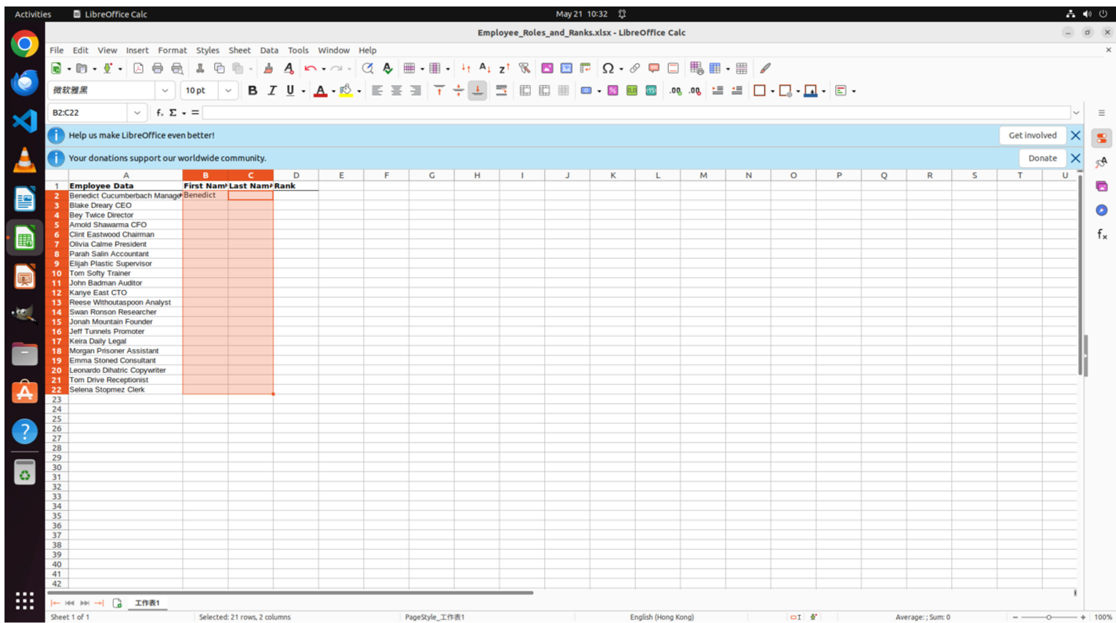
**Task ID:** `37608790-6147-45d0-9f20-1137bb35703d`

**Task.** The information are mixed in one field. Help me split them and fill in the columns of First Name, Last Name and Rank. Finish the work and don't touch the original data.

**Grounded action.** `agent.drag_and_drop(starting_description='the fill handle of cell B2', ending_description='cell B22', hold_keys=[])`




(a) Grounding visualization.



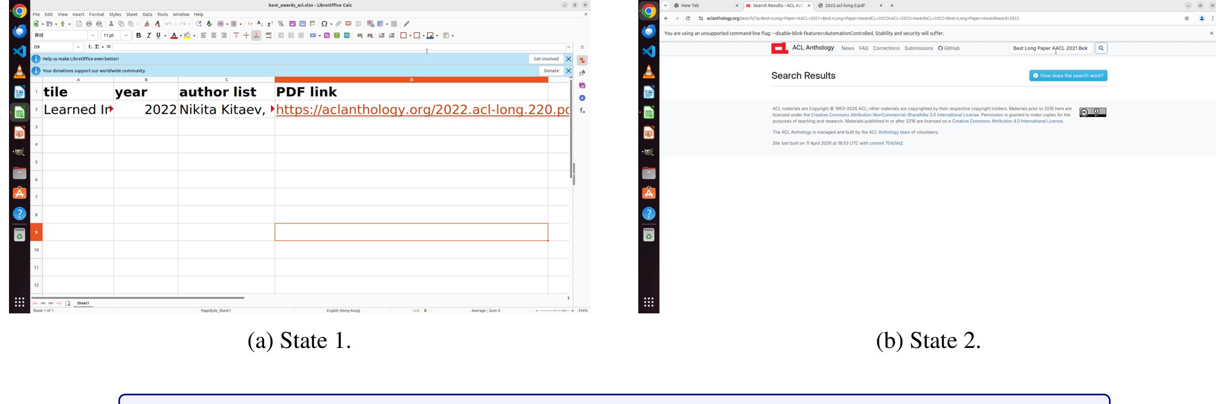

(b) Resulting state after the grounded action.

Figure 19: Wrong grounding example. The grounding model maps the drag-and-drop action to an incorrect spreadsheet region, selecting the wrong range in the resulting state.

**Example:** Failure to reason over future states.

**Task ID:** `67890eb6-6ce5-4c00-9e3d-fb4972699b06`

**Task.** I am an NLP researcher. Check out the best long paper awards of ACL from 2019 to 2022 and record the `title`, `year`, `author list`, and `PDF link` into table `best_awards_acl.xlsx` on the desktop. Separate authors only by commas, use official ACL Anthology URLs, not arXiv, and sort rows by year in descending order.




(a) State 1.

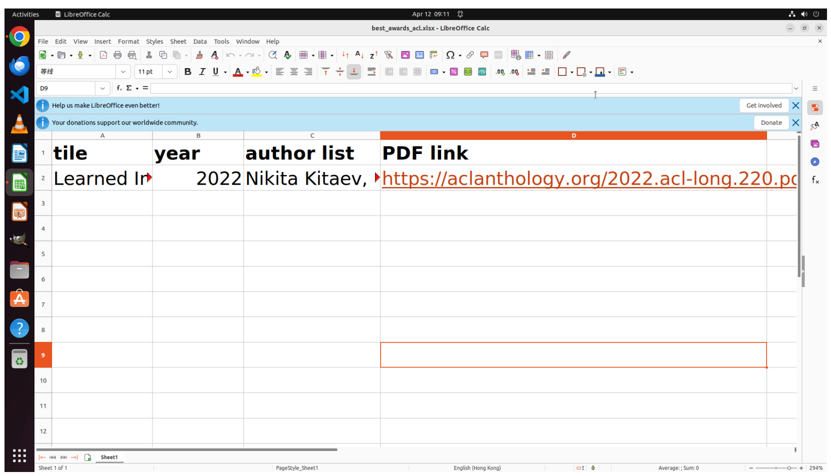
(b) State 2.

**Index 0 (should have chosen):**
```
agent.type(element_description="the search bar on the ACL Anthology website",
text="ACL 2021", enter=True)
```

**Index 1 (chosen):**
```
agent.switch_applications(app_code="libreoffice-calc")
```

**Index 2:**
```
agent.hotkey(keys=["ctrl", "s"])
```

**Judge tokens.** GT-Rollout judge: Candidate 1 switches to LibreOffice Calc, showing the spreadsheet `best_awards_acl.xlsx` with data already entered for the 2022 award, which directly progresses the task of recording award information. Candidate 0 results in a failed search, and Candidate 2 triggers a save dialog unrelated to the task. Selected: 1.




(c) State 2 future.

Figure 20: The judge fails to reason over the future. The judge selects the candidate that returns to the partially completed spreadsheet because it observes existing 2022 data, instead of preferring the candidate that continues searching for missing ACL award entries.

**Example:** Agent runs out of max steps.

**Task ID:** `6f4073b8-d8ea-4ade-8a18-c5d1d5d5aa9a`

**Task.** I now want to count the meeting cities of the three machine learning conferences in the past ten years from 2013 to 2019 (including 2013 and 2019). I have listed the names and years of the conferences in Excel. Please fill in the vacant locations.

**Failure step.** Step 50.

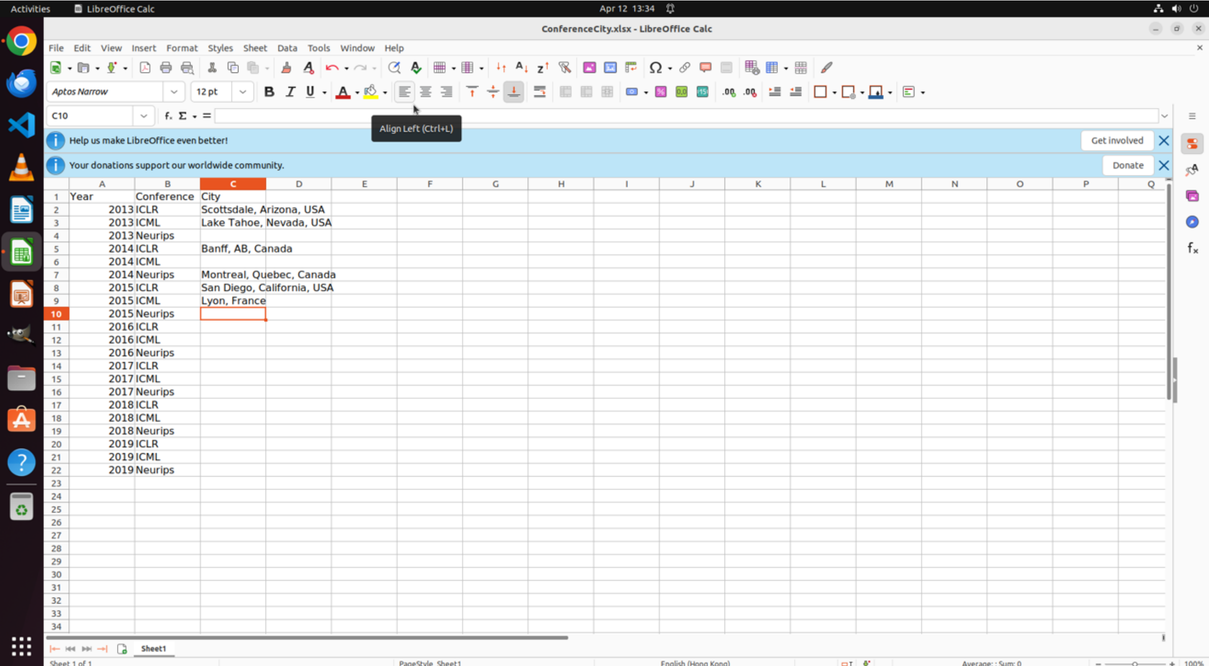



Figure 21: Out-of-max-steps example. The agent is still working on the spreadsheet at step 50 but has not completed all missing conference locations before reaching the step limit.

# I Example of stochastic dynamics in OSWorld environment

We provide examples of rollout mismatch issues in the OSWorld environment in Figures 22–25. In these cases, the screenshot captured in the rollout VM differs from the screenshot captured in the real VM after executing the same action.

# J GT-Rollout Failure Analysis Prompt

```
# SYSTEM PROMPT

You are an expert desktop UI-agent and GT-rollout failure analyst.

# OUTPUT SCHEMA
Return JSON only with this exact top-level shape:
{
  "primary_reason": {
    "category": "<allowed category string>",
    "error_step": integer or null,
    "details": string
  },
  "mentioned_reasons": [
    {
      "category": "<allowed category string>",
      "has_reason": true,
      "error_step": integer or null,
      "details": string
```

```
    }
  ],
  "step_evidence": [
    {
      "step": integer or null,
      "screenshot_ids": [integer],
      "action": string,
      "grounded_action": string,
      "details": string
    }
  ],
  "reason_presence": {
    "grounding error": boolean,
    "tutorial error": boolean,
    "tutorial is not completed": boolean,
    "observation hallucination in policy thought": boolean,
    "agent doing correctly but runs out of max steps": boolean,
    "web server error, agent cannot connect": boolean,
    "the judge fails to reason over the future state": boolean,
    "the judge fails to reason over the future state, fails to understand task success": boolean,
    "agents fail to follow the tutorial correctly": boolean
  }
}

Use only category strings exactly as listed.

# ALLOWED CATEGORY STRINGS
- "grounding error"
- "tutorial error"
- "tutorial is not completed"
- "observation hallucination in policy thought"
- "agent doing correctly but runs out of max steps"
- "web server error, agent cannot connect"
- "the judge fails to reason over the future state"
- "the judge fails to reason over the future state, fails to understand task success"
- "agents fail to follow the tutorial correctly"

# CATEGORY DEFINITIONS
- grounding error: The high-level action or selected candidate is correct, but the grounded UI
operation targets the wrong (x,y) coordinate, window, field, or application state. For example,
the agent outputs "click the 'Submit' button", but the grounded UI operation clicks a position
that is not the Submit button.
- tutorial error: The tutorial is incorrect, and the agent does not know how to recover from the
wrong tutorial step.
- tutorial is not completed: The tutorial does not provide sufficiently fine-grained actions to
complete the task, and the agent fails to infer the missing fine-grained actions by itself.
- observation hallucination in policy thought: The policy hallucinates a wrong observation in its
 thought process, which leads to a wrong next action. For example, the policy hallucinates that
it has already opened the target application and then tries to interact with that application.
- agent doing correctly but runs out of max steps: The agent is following a correct path but
reaches the maximum step limit before completing the task; this may be caused by one of the above
 reasons or another weak/unclear reason.
- web server error, agent cannot connect: The web server hosting the task environment has issues,
 such as the target website being down.
- the judge fails to reason over the future state: Use when the GT-rollout judge/selector is the
decisive failing component: it selects, approves, or keeps preferring a wrong candidate/action
because it misreads the current screenshot, candidate rollout screenshots, or future/resulting
state. If the policy proposed a correct tutorial-following action but the judge chose a different
 candidate by hallucinating that the wrong future state was successful, this category should be
primary, not tutorial-following failure. Do not also use this category for the same evidence when
 the more specific task-success category applies.
- the judge fails to reason over the future state, fails to understand task success: The policy
or GT-rollout judge fails to understand whether the current/future state satisfies the task
instruction, especially when it terminates with agent.done() despite visible evidence that the
task is still incorrect. Use this more specific category instead of the broader judge future-
state category for that same termination/task-success evidence.
- agents fail to follow the tutorial correctly: Use when the policy/agent itself misunderstands,
skips, contradicts, or wrongly executes a tutorial step, and that policy/agent behavior is the
```

```
decisive cause of failure. Do not use this as primary when the policy/proposed actions include
the correct tutorial-following action but the GT-rollout judge selects a different wrong
candidate; in that case use the judge future-state reasoning category as primary.

Do not output "other", "unknown", "unclear", or any category not listed above. If no category
fits perfectly, choose the closest listed category and explain the mismatch in details.
The primary_reason must be the latest decisive reason that directly explains the final failed
outcome. Prefer evidence from the final or near-final trajectory step, especially a judge
selection, termination decision, or executed action that preserves or confirms the failure.
Earlier mistakes should be secondary mentioned_reasons unless no later decision independently
causes, preserves, or confirms the failed state.
Critical distinction between tutorial-following failure and judge future-state failure:
- Use the tutorial-following category only when the policy/agent itself misunderstands, skips,
contradicts, or wrongly executes a tutorial step, and that policy/agent action is the decisive
cause.
- Use the judge future-state reasoning category when the GT-rollout judge selects, approves, or
keeps preferring a wrong candidate/action because it misreads the current screenshot, misreads
candidate/future screenshots, or hallucinates that an incorrect future state satisfies the task.
- If a correct tutorial-following candidate/action was available or proposed, but the judge chose
 a different candidate by claiming its resulting state was correct, the primary_reason must be
the judge future-state reasoning category. The tutorial-following category may be mentioned only
if the policy/agent also independently failed to follow the tutorial.
If the final or near-final step marks the task done even though screenshots/evaluator evidence
show the task is not successful, prefer "the judge fails to reason over the future state, fails
to understand task success" as primary over earlier action or grounding errors.
mentioned_reasons must include every listed category that has evidence in the trajectory,
including secondary causes.
Do not double-list overlapping judge categories in mentioned_reasons: if "the judge fails to
reason over the future state, fails to understand task success" applies, do not also include "the
 judge fails to reason over the future state" for the same evidence. Include both only when there
 is separate non-task-success judge-selection evidence at a different step.
reason_presence must contain every allowed category with true or false.
Keep the JSON compact: primary_reason.details and each mentioned_reasons.details must be one
short sentence.
Return at most 6 mentioned_reasons entries, only for categories with concrete evidence.
Return at most 6 step_evidence entries total, prioritizing the primary reason step and the latest
 decisive evidence near the end.
step_evidence must cite screenshot IDs from the provided Image index. Do not invent screenshot
IDs. Do not generate screenshots. The renderer will embed the referenced images.
For each step_evidence item, include the trajectory step, the high-level action proposed by the
policy, the grounded/executed action from traj.jsonl or pre_grounded fields, and one short
sentence explaining why the screenshot/action matters.
Use the evaluation JSON and result details as the source of truth for the final failed state.
Use GT-rollout candidate images/actions to decide whether the failure came from policy proposal,
grounding, rollout prediction, selection, tutorial conditioning, evaluator issues, or termination
 behavior.
Do not invent evidence. If evidence is weak, say so in details.
No markdown fences. No prose outside JSON.

# USER PROMPT

Analyze this failed OSWorld GT-rollout task.

You are given the full task/evaluation bundle, the full traj.jsonl text, and every raster image
referenced by the trajectory. Images have been preprocessed with max_image_pixels={
max_image_pixels}; they are not raw screenshots.

Task instruction:
{task_instruction}

Image index:
{image_index}

Classify the primary failure reason and every category that is mentioned by evidence in the
trajectory.
Also produce compact step_evidence entries for the screenshots/steps that justify the
classification. Refer to screenshots by Image number only.
```

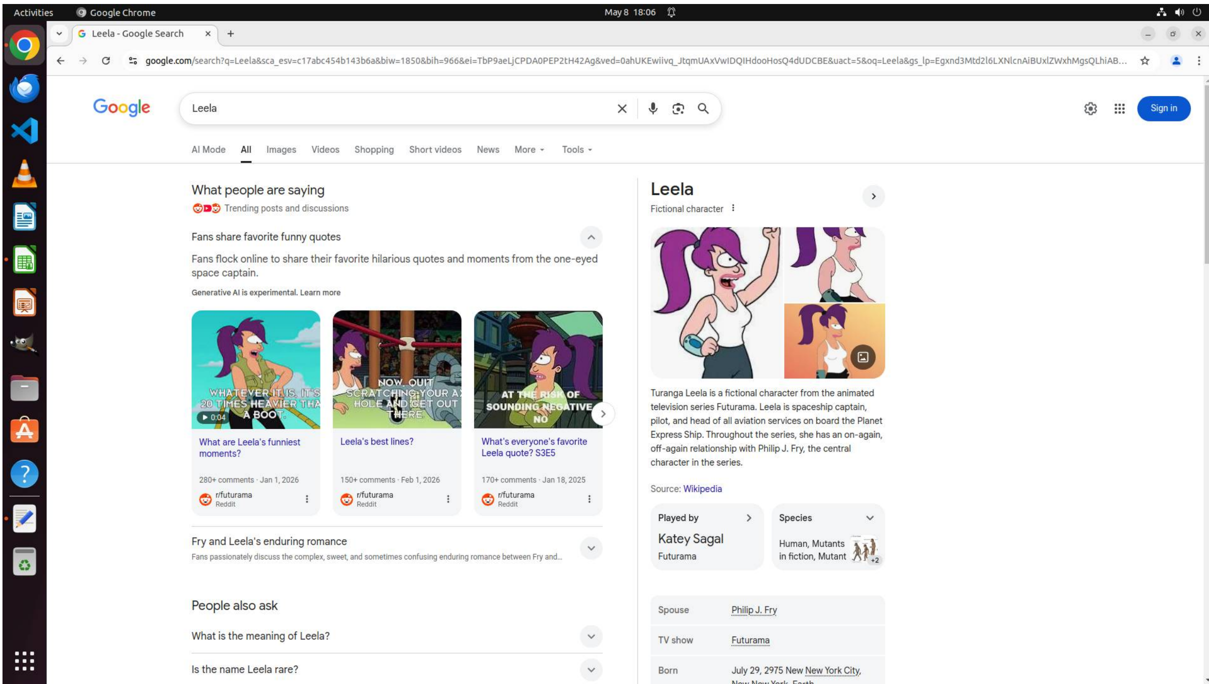
```
Task bundle and full trajectory:

{bundle_markdown}
```

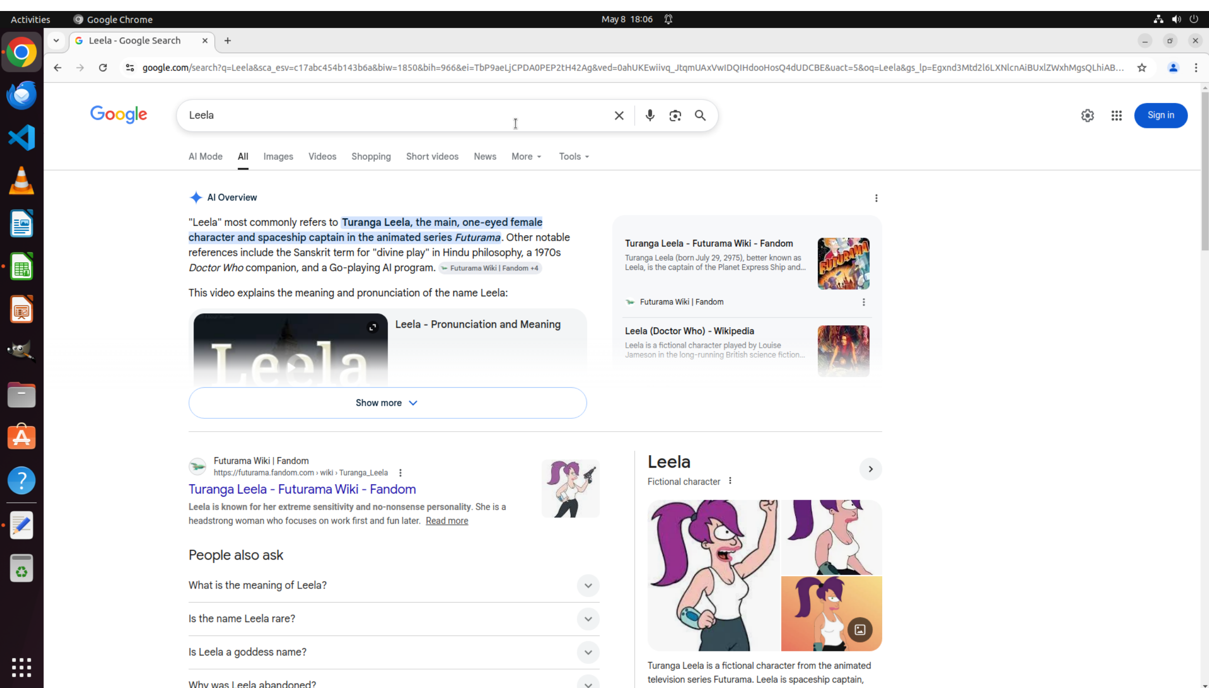
Figure 26: Gemini prompt used to classify failed GT-rollout trajectories and extract compact evidence for the failure analysis.

## K LLM Usage Statement

The authors used LLM-based assistants for limited editorial and technical support, including grammar and clarity edits, LaTeX debugging, and draft phrasing for non-substantive text. We also use LLM-assisted programming like Claude Code and Codex to implement infrastructure code for ground-truth visual rollouts and other baselines. All LLM-assisted text and code changes were inspected and revised by the authors. The authors are responsible for the final manuscript, including all claims, analyses, tables, figures, citations, and conclusions.

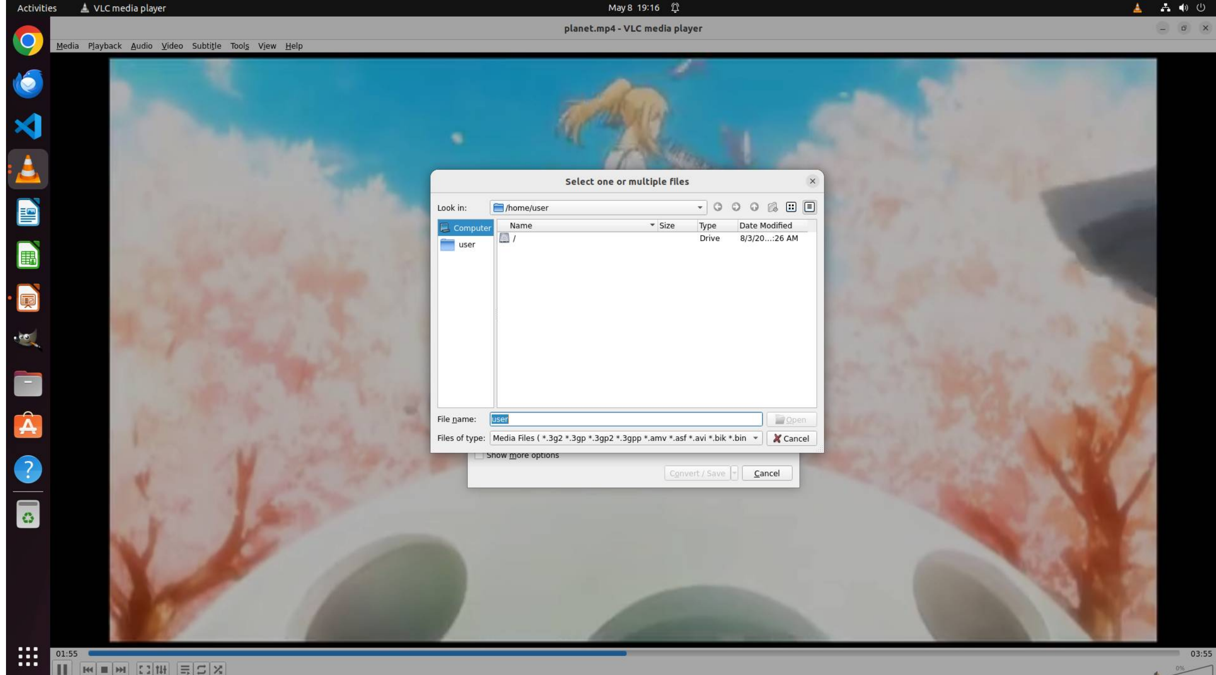



(A) Screenshot captured in a rollout VM.

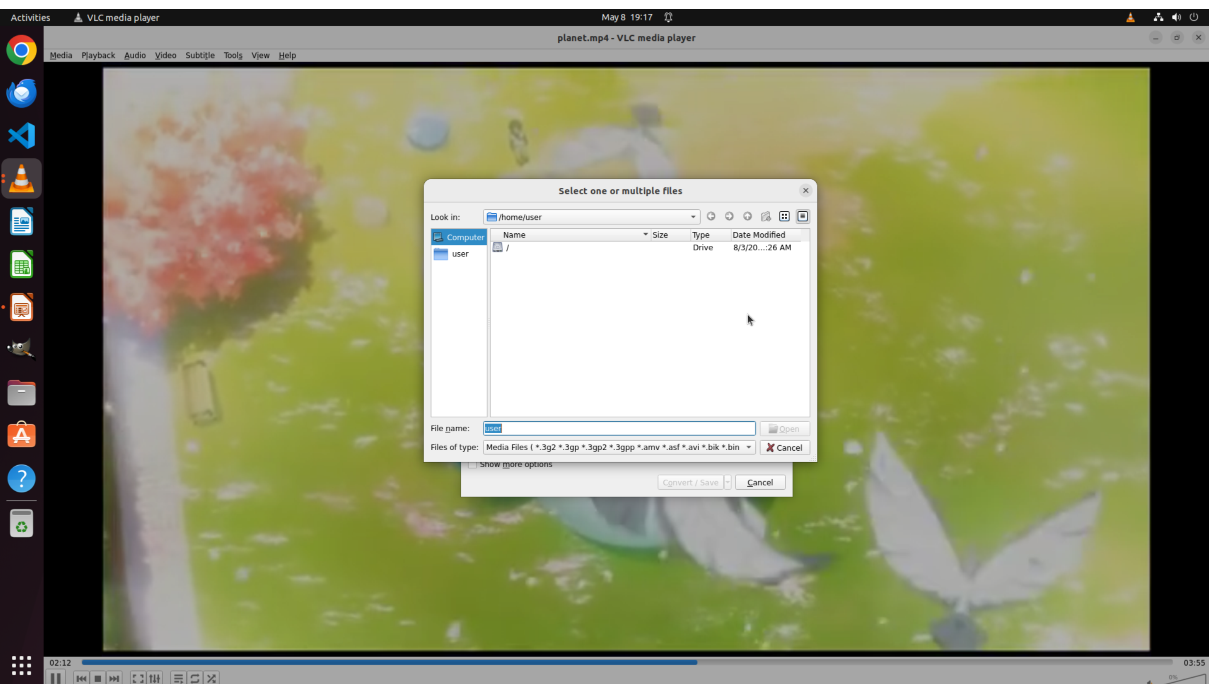



(B) Screenshot captured in a real VM.

Figure 22: Rollout vs. real VM (case 0).

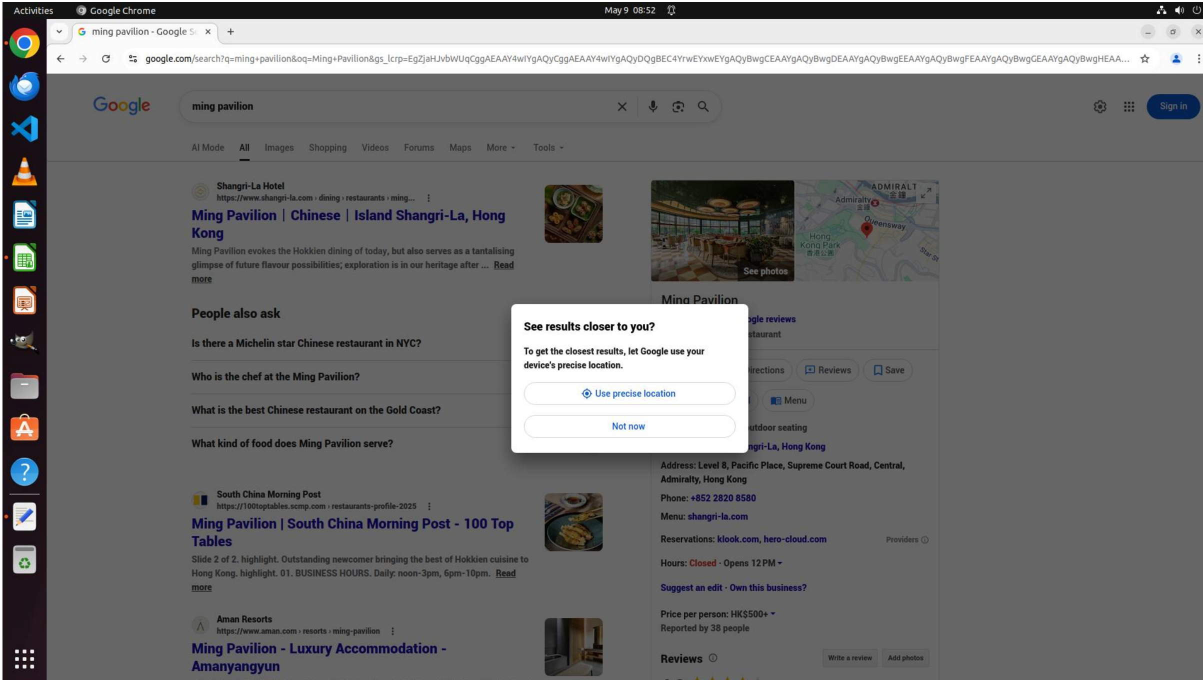



(A) Screenshot captured in a rollout VM.

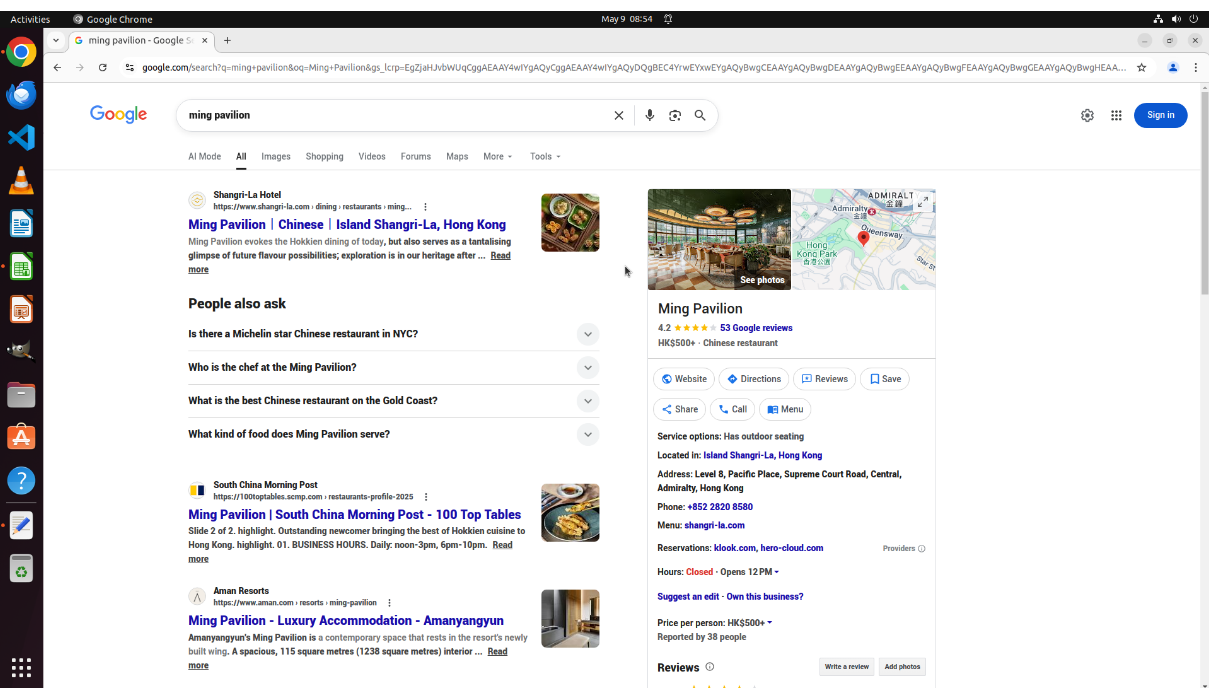



(B) Screenshot captured in a live VM.

Figure 23: Rollout vs. live VM (case 1).

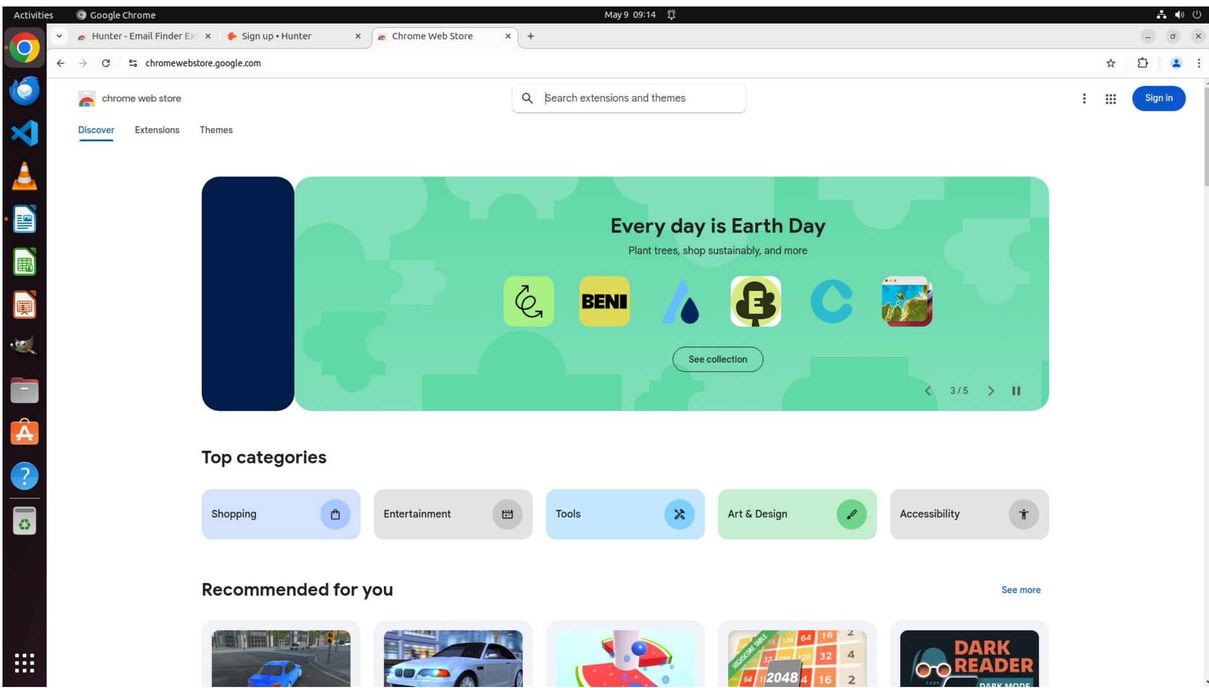



(A) Screenshot captured in a rollout VM.

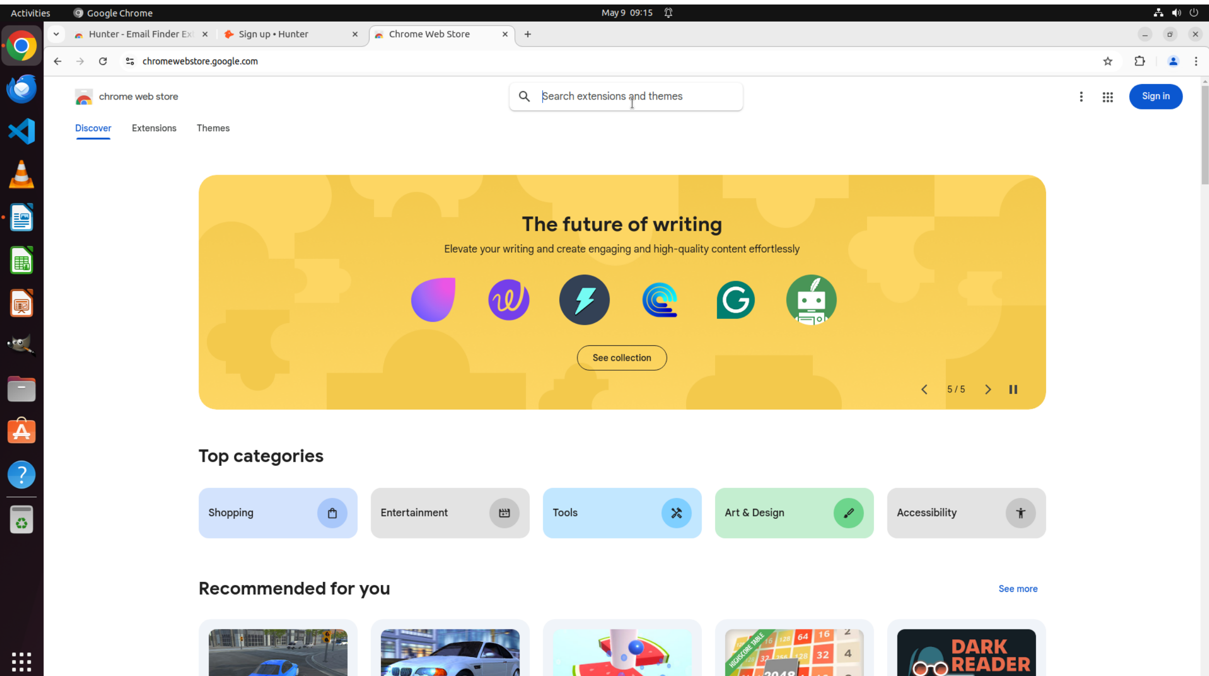



(B) Screenshot captured in a live VM.

Figure 24: Rollout vs. live VM (case 2).




(A) Screenshot captured in a rollout VM.




(B) Screenshot captured in a live VM.

Figure 25: Rollout vs. live VM (case 3).