



ScreenAgent : A Vision Language Model-driven Computer Control Agent

**Runliang Niu¹, Jindong Li¹, Shiqi Wang¹, Yali Fu¹, Xiyu Hu¹,
Xueyuan Leng¹, He Kong¹, Yi Chang^{1,2}, Qi Wang^{1,2*}**

¹ School of Artificial Intelligence, Jilin University

² Engineering Research Center of Knowledge-Driven Human-Machine Intelligence,
Ministry of Education, China

†

Abstract

Existing Large Language Models (LLM) can invoke a variety of tools and APIs to complete complex tasks. The computer, as the most powerful and universal tool, could potentially be controlled directly by a trained LLM agent. Powered by the computer, we can hopefully build a more generalized agent to assist humans in various daily digital works. In this paper, we construct an environment for a Vision Language Model (VLM) agent to interact with a real computer screen. Within this environment, the agent can observe screenshots and manipulate the Graphics User Interface (GUI) by outputting mouse and keyboard actions. We also design an automated control pipeline that includes planning, acting, and reflecting phases, guiding the agent to continuously interact with the environment and complete multi-step tasks. Additionally, we construct the ScreenAgent Dataset, which collects screenshots and action sequences when completing a variety of daily computer tasks. Finally, we trained a model, ScreenAgent, which achieved computer control capabilities comparable to GPT-4V and demonstrated more precise UI positioning capabilities. Our attempts could inspire further research on building a generalist LLM agent. The code is available at <https://github.com/niuzaisheng/ScreenAgent>.

1 Introduction

Large language models (LLM), such as ChatGPT and GPT-4, have recently demonstrated exceptional performance in natural language processing tasks like generation, understanding, and dialogue. They have also significantly revolutionized research in other artificial intelligence fields. In particular, the development of these technologies paves the way for the study of intelligent LLM agents, which are capable of performing complex tasks [Wang *et al.*, 2023b]. A LLM agent is

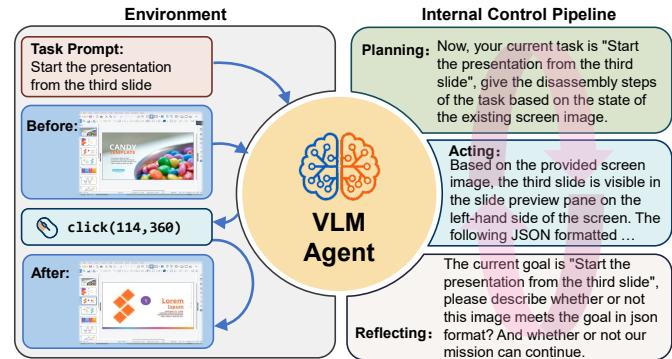


Figure 1: We have constructed a realistic computer-controlled environment and designed a control pipeline for the agent. The VLM agent retrieves instruction prompts and real computer states from the environment, then runs its internal control flow, going through the planning, acting, and reflecting phases. It outputs the next action operation, utilizes function calls to perform actions, induces changes in the computer environment, and achieves genuine real-time interaction between the agent and the environment.

an AI entity with a large language model as its core computational engine. It possesses capabilities like Perception, Cognition, Memory, and Action, enabling the agent to perform highly proactive autonomous behaviors [Wang *et al.*, 2023c]. In LLM agent-related research, how to enable agents to learn to effectively use tools for expanding their action space has drawn extensive attention.

With the growing prevalence of electronic devices, such as personal computers, smartphones, tablets, and smart electronic instruments, our lives are becoming more intertwined with the digital world. Daily activities often require frequent interaction with electronic device screens. If an agent can free people from manual operations, and seamlessly navigate these devices by controlling screens according to user needs, it would mark a significant step towards achieving more general and autonomous intelligence [Yin *et al.*, 2023]. Indeed, a screen interaction agent must possess powerful visual information processing capabilities, and the ability to execute computer control instructions as shown in Fig. 1. Therefore, to achieve such a goal, it is necessary to first create a real interactive environment for the VLM agent, then guide the model and environment to form a continuous in-

*corresponding author

†{niurl19.jdLi21, shiqiw23, fuyl23, xyhu23, lengxy22, konghe19}@mails.jlu.edu.cn, {yichang, qiwang}@jlu.edu.cn

teractive pipeline, and train the agent to improve its performance. However, it's highly challenging to implement these functions within a unified framework and achieve satisfactory results from both the project engineering and theoretical research perspectives.

Despite the recent progress achieved by several works, some aspects still need to be further explored. For instance, CogAgent [Hong *et al.*, 2023] specializes in GUI understanding and planning, showcasing remarkable proficiency in addressing diverse cross-modal challenges. However, CogAgent lacks the capability of a complete thinking chain, preventing it from forming a comprehensive tool invocation similar to ChatGLM [Du *et al.*, 2022; Zeng *et al.*, 2022]. Later, AppAgent [Yang *et al.*, 2023a] focuses on smartphone operations, learning navigation, and acquiring new application usage through autonomous exploration or by observing human demonstrations. While AppAgent lacks a planning process and sacrifices the freedom of operation. It guides the agent to click by labeling each element, thereby limiting the method of tapping. As a result, current Vision Language Models (VLM agents) are typically unable to interact with real computer or mobile environments to generate and execute continuous manipulative commands.

To address the above-mentioned issues, we propose ScreenAgent, an entirely automated agent designed to handle continuous screen operations. This agent is primarily achieved through three components, namely planning, execution, and reflection. In particular, the reflecting module is inspired by Kolb's renowned experiential Learning Cycle theory [Kolb, 2014], which enables the agent to perform reflective behaviors, making the entire pipeline more comprehensive and aligned with human action and thought processes. It autonomously assesses the execution status of the current action, providing feedback based on the ongoing state. This capability enhances its performance for subsequent actions, enabling our agent to possess the capability of a continuous thinking chain. Consequently, our agent can understand the next steps and engage in complete tool invocation to execute a series of continuous manipulative commands. The major contributions are summarized as follows:

- We present a Reinforcement Learning (RL) environment that enables the VLM agent to directly interact with a real computer screen via VNC protocol. By observing the screenshot, our agent can interact with the GUI through basic mouse and keyboard operations.
- We develop an automated pipeline that encompasses the planning phase, acting phase, and reflecting phase. This integrated pipeline facilitates the agent's continuous interaction with the environment, distinguishing our agent from others.
- We propose the ScreenAgent dataset, which includes action sequences for completing generic tasks on Linux and Windows desktops. Moreover, we provide a fine-grained scoring metric to comprehensively evaluate the various capabilities that are necessary for a VLM agent in computer-controlling tasks.
- We test GPT-4V and two state-of-the-art open-source VLMs on our test set. The results demonstrate that GPT-

4V is capable of controlling computers, but it lacks precise positioning capabilities. We thus trained a ScreenAgent to enable precise positioning and achieved comparable results to GPT-4V in all aspects. Our work can facilitate further research on building a generalist agent.

2 Related Work

2.1 Multimodal Large Language Models

LLMs have demonstrated powerful contextual understanding and text generation capabilities, enabling the implementation of complex multi-turn question-answering systems. LLaMA [Touvron *et al.*, 2023] is a series of foundational language models spanning from 7 billion to 65 billion parameters, with Vicuna-13B [Chiang *et al.*, 2023], an open-source chatbot, being refined through fine-tuning on the LLaMA architecture. GPT-4 is an advancement by OpenAI following the success of GPT-3 and it introduces several noteworthy improvements. GPT-4V(ision) [Yang *et al.*, 2023b], building upon GPT-4, has added multimodal capabilities. LLaVA [Liu *et al.*, 2023b] and LLaVA-1.5 [Liu *et al.*, 2023a] connect the pre-trained CLIP [Radford *et al.*, 2021] visual encoder with the Vicuna, achieving multimodal capabilities. Fuyu-8B¹ does not use an image encoder but opts for a pure decoder Transformer architecture. CogVLM [Wang *et al.*, 2023e] is a powerful open-source Visual Language Model that supports image understanding at a resolution of 490×490 and multi-turn dialogues. In addition, Monkey [Li *et al.*, 2023] introduces an efficient training method that enhances input resolution capability.

2.2 Computer Control Environment & Dataset

In simulated environments, agents can be trained to emulate clicking and typing. WebNav [Nogueira and Cho, 2016] creates a navigation environment with links, testing the agent's sequential decision-making ability. MiniWoB++ [Liu *et al.*, 2018] provides a lot of simplified ATARI-like web-browser-based tasks as a Reinforcement Learning (RL) environment. WebShop [Yao *et al.*, 2023] offers tasks for controlling the browser to complete the purchase process. SWDE [Hao *et al.*, 2011] preserves webpage HTML files to train information extraction models. WebSRC [Chen *et al.*, 2021] is a QA-style dataset that contains a large number of questions and answers about webpage screenshots. Mind2Web [Deng *et al.*, 2023] introduces a dataset for developing generalist web agents. Seq2act [Li *et al.*, 2020a] integrates three datasets for Android, AndroidHowTo, Rico-SCA, and PixelHelp. Screen2Words [Wang *et al.*, 2021] is a large-scale screen summarization dataset for Android UI screens. META-GUI [Sun *et al.*, 2022] is a dataset for training a multi-modal conversational agent on mobile GUI. [Burns *et al.*, 2022] provides a dataset of unknown command feasibility on Android.

2.3 Large Language Model-Driven Agents

With the advancement of large language models, the capabilities of intelligent agents have also been enhanced. WebGPT [Nakano *et al.*, 2021] conducts fine-tuning on GPT-3 to

¹<https://www.adept.ai/blog/fuyu-8b>

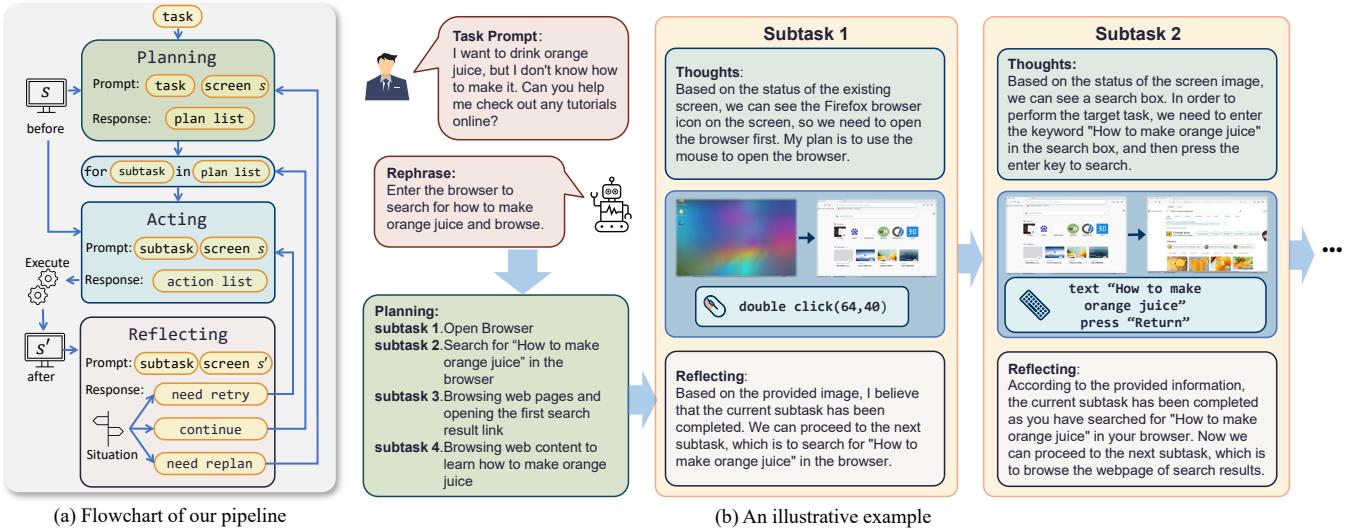


Figure 2: The overview of our computer control pipeline, which includes planning, acting, and reflecting phases. Sub-figure (a) presents the flowchart of our pipeline, while sub-figure (b) provides an illustrative example. Based on the user's task prompt, the agent initially decomposes the task into subtasks. In each subtask, the agent first describes the screen and generates mouse and keyboard operations in a function-call style. In the reflection phase, the agent decides whether to proceed to the next subtask, retry the current subtask, or reformulate the entire plan.

address extended questions within a text-based web-browsing environment, enabling the model to explore and navigate the web for answers. ToolFormer [Schick *et al.*, 2023] integrates an assortment of utilities, featuring a calculator, Q&A system, and search engine, among others. Voyager [Wang *et al.*, 2023a] stands as the inaugural embodiment of a Large Language Model-powered, lifelong learning agent within the Minecraft environment. RecAgent [Wang *et al.*, 2023d] proposes that agents can generate high-level thoughts through the operation of memory reflection. ProAgent [Ye *et al.*, 2023] introduces a novel paradigm in process automation that seamlessly integrates agents powered by Large Language Models. CogAgent [Hong *et al.*, 2023], an 18-billion-parameter visual language model, is meticulously designed for GUI comprehension and navigation. AppAgent [Yang *et al.*, 2023a] proposes a multi-modal agent framework for learning operations on smartphone applications.

3 Framework

In this section, we introduce our Reinforcement Learning (RL) environment and the autonomous control flow within the Agent. Through this environment, a VLM agent can interact with a real computer screen, observe screen images, select actions, and autonomously complete specific tasks.

3.1 Computer Control Environment

We construct a computer control environment to assess the capabilities of VLM agents. This environment connects to a desktop operating system through remote desktop (VNC) protocol and allows the sending of mouse and keyboard events to the controlled desktop. The formal definitions of this environment are defined as follows:

- **A-Action Space.** We define an action as a form of a function call. If the agent outputs an action in the required JSON-style format, the action will be parsed and executed by the environment. All action types and corresponding action attributes are defined in Table 1.
- **S-State Space.** The screenshot image is utilized as the state space of the environment. The environment will collect screenshots s and s' , denoting the state before and after each action, respectively.
- **R-Reward Function.** Due to the highly open-ended nature of the task, the reward function in this environment is flexibly opened to different interfaces, which can integrate different existing or future reward models.

Through remote control, the agent can perform arbitrary tasks on the screen, which creates a highly challenging open environment having a large state and action space.

3.2 Control Pipeline

To guide the agent to continually interact with the environment and complete multi-step complex tasks. We designed a control pipeline including the Planning, Acting, and Reflecting phases. The whole pipeline is depicted in Fig. 2. The pipeline will ask the agent to disassemble the complex task, execute subtasks, and evaluate execution results. The agent will have the opportunity to retry some subtasks or adjust previously established plans to accommodate the current occurrences. The details are depicted as follows:

Planning Phase. In the planning phase, based on the current screenshot, the agent needs to decompose the complex task relying on its own common-sense knowledge and computer knowledge.

Acting Phase. In the acting phase, based on the current screenshot, the agent generates low-level mouse or keyboard

Action Type	Attributes
Mouse	Move Mouse Position(width:int, height:int)
	Click Mouse Button(left/middle/right), Mouse Position(width:int, height:int)
	Double Click Mouse Button(left/middle/right), Mouse Position(width:int, height:int)
	Scroll Up Scroll Repeat(int)
	Scroll Down Scroll Repeat(int)
Drag	Drag End Position(width:int, height:int)
Keyboard	Press Keyboard Key or Combined-keys (string)
	Text Keyboard Text(string)
Wait Action	Wait Time(float)
Plan Action	Element(string)
Evaluate Action	Situation(success/retry/reformulate) Advice(string)

Table 1: All supported action types and action attributes.

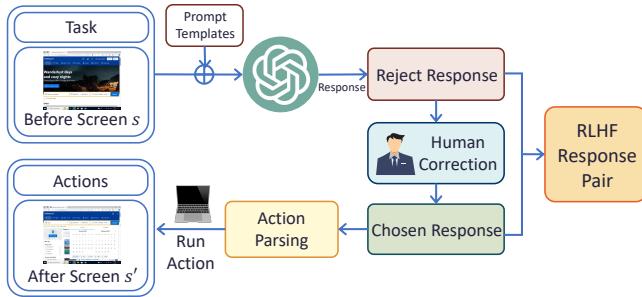


Figure 3: Data annotation process. We invoke GPT-4V to generate an original response, and annotators correct this response as the golden labeled response. The environment parses executable actions from the text and sends them for real computer execution. The original response and the golden labeled response form a pair, which can be utilized for training in future Reinforcement Learning from Human Feedback (RLHF) processes.

actions in JSON-style function calls. The environment will attempt to parse the function calls from the agent’s response, and convert them to device actions defined in the VNC protocol. Then our environment will send actions to the controlled computer. The environment will capture the after-action screen as input for the next execution phase.

Reflecting Phase. The reflecting stage requires the agent to assess the current situation based on the after-action screen. The agent determines whether needs to retry the current sub-task, go on to the next sub-task, or make some adjustments to the plan list. This phase is crucial within the control pipeline, providing some flexibility to handle a variety of unpredictable circumstances.

All prompts and templates in the pipeline are provided in Appendix A.

4 ScreenAgent Dataset & CC-Score

Existing computer-controlled datasets typically have a narrow range of applicability scenarios. For instance, building upon the foundational premise that it is easy to obtain UI element metadata through HTML or developer modes, WebNav [Nogueira and Cho, 2016], Mind2Web [Deng et

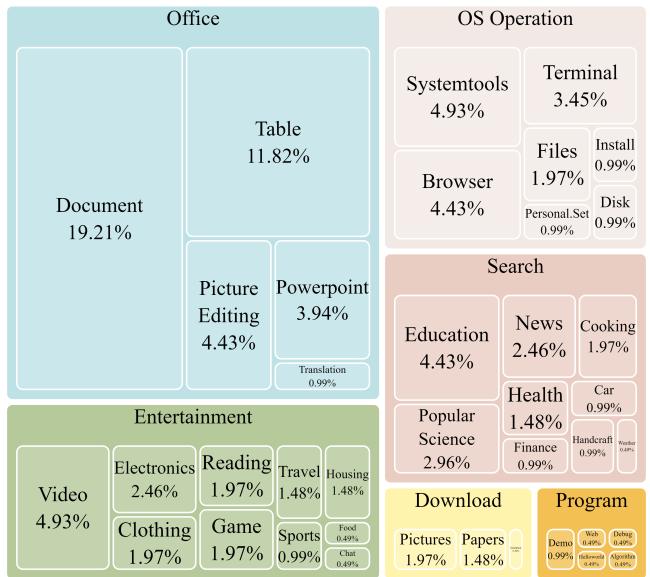


Figure 4: Task type statistics in ScreenAgent training set.

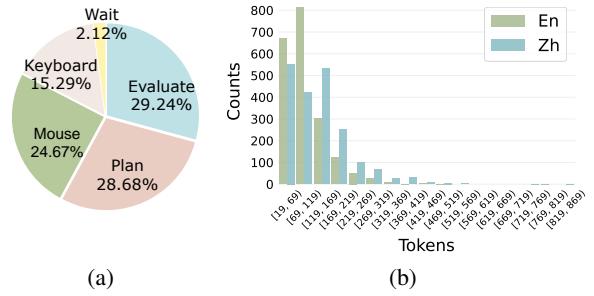


Figure 5: The statistical information of ScreenAgent training set: (a) Distribution of action types; (b) Chosen response token number distribution.

al., 2023], and SWDE [Hao et al., 2011] mainly focused on web browsing, while Seq2act [Li et al., 2020a] and Screen2Words [Wang et al., 2021] are tailored for Android. However, the mouse and keyboard are also common and universal interfaces to control a computer. To fill the gap in this type of control method, we build an interactive annotation process (shown in Fig. 3) to construct the ScreenAgent Dataset which is collected from Linux and Windows operating systems for completing specific tasks. This dataset endeavors to cover a wide range of daily computer usage scenarios, including daily office, booking, information retrieval, card games, entertainment, programming, system operations, and so on. As illustrated in Fig. 4, the ScreenAgent Dataset encompasses 39 sub-task categories across 6 themes. The dataset has 273 complete task sessions, with 203 sessions (3005 screenshots) for training and 70 sessions (898 screenshots) for testing. Fig. 5 shows important statistical information about the dataset. More statistical information and samples are provided in Appendix C.

To assess an agent’s capability in the computer control task, we have designed a fine-grained evaluation metric Vision Language Computer Control Score (CC-Score) for as-

Model	Plan total 284	Action Type total 650	Mouse Action Type total 232	Mouse Button total 209	Mouse Position total 218	Keyboard Keys or Text total 134	Reflecting Situation Assessment total 546
LLaVA-1.5	<u>0.78</u>	0.75	0.71	0.74	0.72	0.45	0.98
CogAgent-VQA	0.00	0.03	0.06	0.06	0.05	0.01	0.39
CogAgent-Chat (original output)	0.00	0.00	0.00	0.00	0.00	0.00	0.30
CogAgent-Chat (help by GPT-3.5)	0.29	0.38	0.44	0.45	0.42	0.17	0.76
GPT-4V(ision)	0.87	0.86	<u>0.85</u>	<u>0.85</u>	<u>0.83</u>	<u>0.77</u>	1.00
ScreenAgent	0.72	<u>0.83</u>	0.91	0.92	0.91	0.82	1.00

Table 2: Proportion of successful function call on ScreenAgent test-set.

Model	CC-Score	Plan (BLEU)	Action Type (F1)	Mouse Action Type (F1)	Mouse Button (F1)	Mouse Position (Accuracy)	Keyboard Keys or Text (BLEU)	Reflecting Situation Assessment (F1)
LLaVA-1.5	0.51	0.29	0.91	0.90	0.96	0.03	0.70	<u>0.52</u>
CogAgent-Chat (help by GPT-3.5)	0.33	<u>0.32</u>	0.83	0.86	0.02	0.07	0.74	0.51
GPT-4V(ision)	0.63	0.47	0.98	0.96	0.99	<u>0.12</u>	0.92	0.60
ScreenAgent	<u>0.61</u>	0.31	0.98	<u>0.94</u>	0.97	0.51	0.87	0.52

Table 3: Comparison of VLM fine-grained score in all successful matched action on ScreenAgent Test-Set.

sessing the similarity of control action sequences. This metric takes into account both the sequential order and actions’ attribution. We developed specific similarity metrics for every action type. For mouse actions, the metrics include four aspects of consistency: action type, mouse operation type, mouse button, and whether the click coordinates are within the annotated feasible bounding box. For text and keyboard actions, the metrics involve two aspects: action type consistency and the BLEU score of the input text, single key, or keyboard shortcut combination. For the entire action sequence, we employ an alignment algorithm that identifies the maximum matching pairs of predicted action and labeled action, while maintaining the sequence order. This approach maximizes the overall score, which is used as the measure of sequence similarity. Ultimately, the CC-Score encompasses the normalized scores of predicted and labeled sequences, the F1 values for each action attribute classification, and the unigram similarity values for text types. The implementation details of CC-Score are provided in Appendix D.

5 Experiment

In the experimental phase, we assessed OpenAI GPT-4V performance on the ScreenAgent test set, along with evaluations of three open-source VLMs. Furthermore, one of these models underwent fine-tuning to potentially augment its proficiency in screen control tasks. Subsequently, we conducted a thorough analysis of the outcomes and identified several typical cases to elucidate the inherent challenges of our task.

5.1 Evaluation Results on ScreenAgent Test-Set

Apart from GPT-4V, we selected several recently released SoTA VLMs for testing, including LLaVA-1.5 [Liu *et al.*,

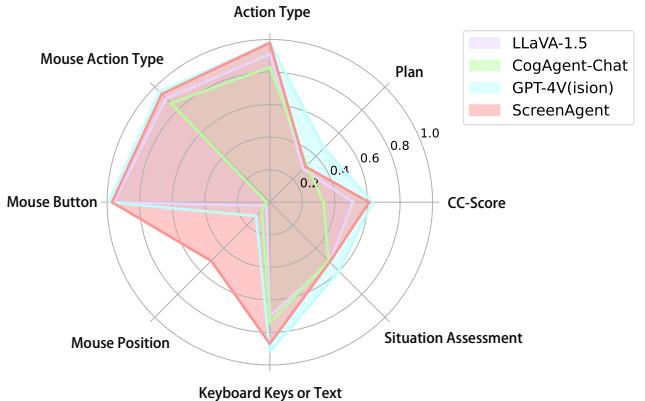


Figure 6: ScreenAgent can complete computer control tasks most excellently compared with other VLMs/Agents.

2023a] and CogAgent [Hong *et al.*, 2023]. LLaVA-1.5 is a 13B-parameter multimodal model, unfortunately, it only supports up to 336×336 px image size inputs. CogAgent is an 18B-parameter visual language model designed for GUI comprehension and navigation. Leveraging dual image encoders for both low-resolution and high-resolution inputs, CogAgent demonstrates proficiency at a resolution of 1120×1120 px, allowing it to discern minute elements and text.

We test the models’ capabilities from two aspects: The ability to follow instructions to output the correct function call format, shown in Table 2, and the ability to complete specific tasks assigned by the user, shown in Table 3.

Following instructions and executing correct function calls is the most fundamental skill for an LLM agent when learning to use external tools. Table 2 presents the success rate

Dataset	Samples	1	2	3	4
COCO	42404	20%	10%	-	-
Widget Captions	41221	20%	10%	-	-
Mind2Web	12846	30%	40%	50%	30%
ScreenAgent Dataset	3005	30%	40%	50%	70%

Table 4: Training data proportions and division of four training phases. Percentages indicate the proportion of samples from this data set at each phase.

of these function calls for each attribute key. This assessment focuses on whether the model can accurately execute various functions encompassing the attribute items expected by manual action annotations. Note that, this evaluation does not consider the consistency of the attribute values with the golden labeling; it solely examines if the model’s output includes the necessary attribute keys. From the table, GPT-4V and LLaVA-1.5 achieved higher scores, while CogAgent and its upstream model CogAgent-VQA underperformed. CogAgent-VQA and CogAgent-Chat almost entirely disregarded the JSON format action definitions in our prompts, resulting in a very low score on successful function calls. Therefore, rendering them completely incapable of interacting with our environment. To ensure fairness in comparison, we utilize OpenAI GPT-3.5 to extract action into JSON-style function calls from the original CogAgent-Chat responses, indicated as "CogAgent-Chat (helped by GPT-3.5)". Even so, its scores are significantly lower than those of LLaVA-1.5 and GPT-4V, although CogAgent has been trained on Mind2Web web browsing simulation datasets.

Table 3 displays the fine-grained scores of predicted attribute values for each action within the successfully parsed function calls. As can be seen, GPT-4V remains the best performer, with action type prediction F1 score of 0.98. This implies that it can accurately select appropriate mouse or keyboard actions. Additionally, it can precisely choose the mouse action type, typing text, or pressing keys consistent with the golden label actions.

The ability for precise positioning is crucial in computer-controlling tasks. As indicated by the "Mouse Position" column in Table 3, current VLMs have not yet achieved the capability for precise positioning required for computer manipulation. GPT-4V refuses to give precise coordinate results in its answers, and two open-source models also fail to output the correct coordinates with our pipeline prompt template.

Another significant challenge for all models is the reflection phase. In this phase, the agent is required to determine whether the subtask has been completed in the current state, and decide whether to proceed further or make some adjustments. This is crucial for constructing a continuous interactive process. Regrettably, all models show insufficient accuracy in this determination, with GPT-4V achieving only a 0.60 F1 score. This implies that human intervention is still necessary during task execution.

5.2 Fine-tuning Training

To demonstrate the potential for ongoing research on the task, we continue to fine-tune the CogAgent-Chat model on our ScreenAgent training set to enhance its function call ability.

Similar to the approach adopted in recent VLM works [Chen *et al.*, 2023], we mix data from multiple datasets and construct four distinct training phases, which is illustrated in Table 4. We reformulated two objective detection datasets, COCO [Lin *et al.*, 2015] and Widget Captions [Li *et al.*, 2020b], into mouse-click tasks to enhance the model’s localization ability. For Mind2Web, we implemented a series of complex data augmentations to align with our task objectives. The details of the data construction are outlined in Appendix B.

After vision fine-tuning, ScreenAgent achieved the same level of following instructions and making function calls as GPT-4V on our dataset, as shown in Table 2. In Table 3, ScreenAgent also reached a comparable level to GPT-4V. Notably, our ScreenAgent far surpasses existing models in the precision of mouse clicking. This indicates that vision fine-tuning effectively enhances the model’s precise positioning capabilities. Additionally, we observed that ScreenAgent has a significant gap compared to GPT-4V in terms of task planning, highlighting GPT-4V’s common-sense knowledge and task-planning abilities.

5.3 Case Study

To evaluate our ScreenAgent model on computer control tasks, we provide two cases. In Fig. 7, we present a case illustrating the workflow of ScreenAgent executing a chain of actions. In Fig. 8, we compare different agents in executing the details of the three phases in the pipeline. Fig. 8 (a) shows the planning process of all the agents, where we find that our ScreenAgent produces the most concise and effective plan. Fig. 8 (b) presents four different click action tasks, each representing a step in a specific task. Results show that LLaVA clicks on the bottom-left corner on all screens, cogAgent may fail to generate click positions, and in the fourth task, only our agent can correctly click on the position. Fig. 8 (c) shows that our agent can recognize whether an action needs to be re-tried after reflection and successfully execute the action following a failure.

6 Conclusion

In this paper, we build a new environment for the screen control task. VLM agents can manipulate a real computer through direct mouse and keyboard control commands. To encourage the agent to continuously interact with the environment and accomplish complex multi-step tasks, we designed a control pipeline that includes planning, acting, and reflecting phases. Furthermore, we unveil a new dataset that covers a wide range of everyday digital works. We propose a fine-grained metric to assess the agent’s computer-controlling capabilities with both action-level and task-level evaluation. We tested OpenAI GPT-4V and two state-of-the-art VLM models on the test set. The results indicate that GPT-4V has the potential to act as a computer-controlling agent, but it lacks precise positioning capabilities. Finally, we trained the ScreenAgent model, inherited from CogAgent, to achieve comparable scores with GPT-4V but more accuracy in UI element localization. It is hoped that our work will inspire further research in building more powerful and generalized agents. In terms

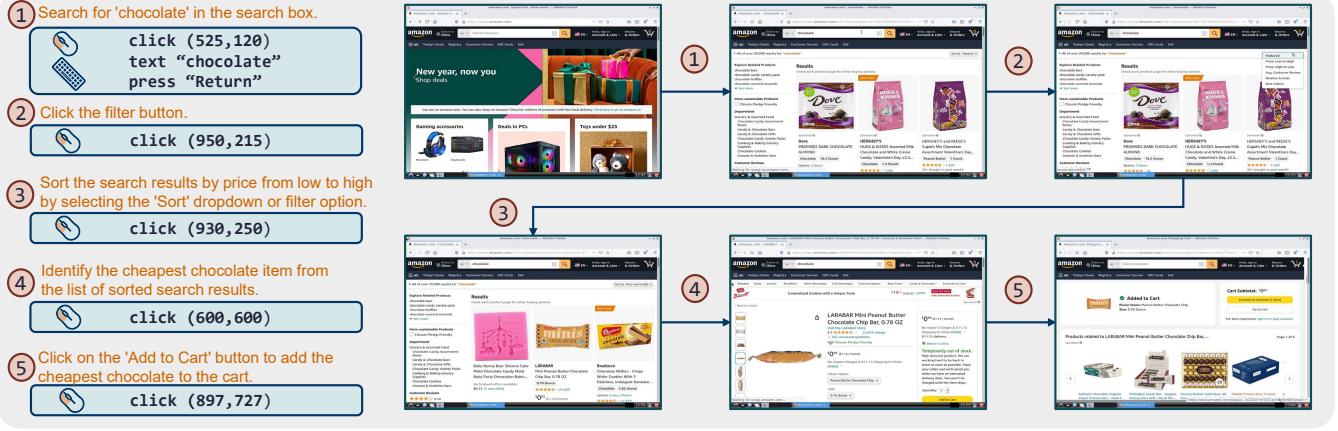


Figure 7: An example of the action task, where we assign ScreenAgent the following task: "Find and add the cheapest chocolate to the cart on Amazon". The diagram delineates the actions that the agent needs to perform, along with the alterations in the computer environment before and after the execution of these actions.

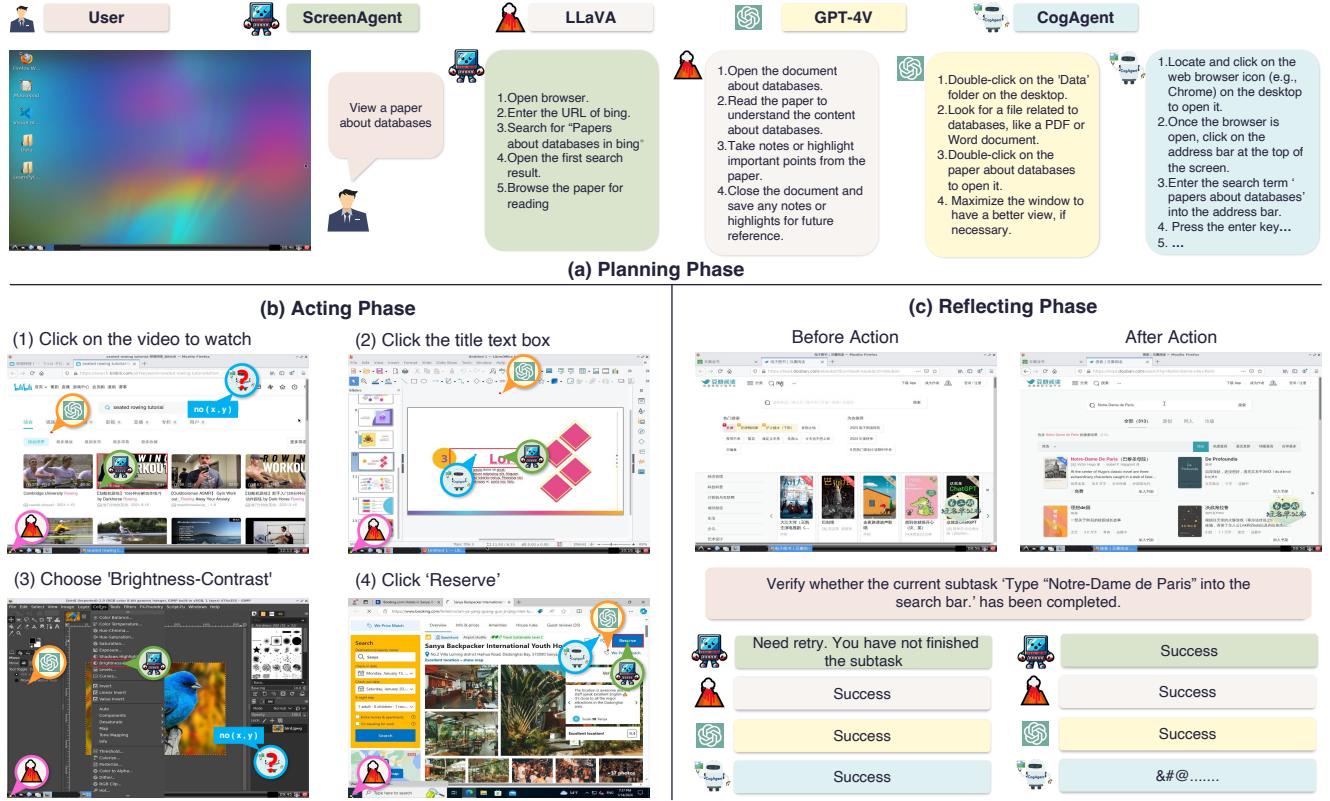


Figure 8: An example to show the execution results of multiple VLM agents among the Planning, Acting, and Reflecting phases in our pipeline.

of technical limitations, due to the input restrictions of VLM, our model can only process single-frame images, not videos or multi-frame images. The VLM's language capability is limited by the abilities of its foundation language model. We also found that even GPT-4V has limited support for non-English text on the screen.

Ethical Statement

The rational use of automated agents with autonomous decision-making capabilities brings significant societal benefits, including improving the accessibility of computers, reducing duplication of human effort on digital work, and aiding in computer education. However, the potential negative

impacts of these agents, such as employment impact, fraud and abuse, privacy issues, and the risk of misoperation, cannot be overlooked. The method could potentially be used to bypass the CAPTCHA test for automatic account registration, spreading misinformation, or conducting illegal activities. We are focused on and committed to the responsible use of AI technology.

References

- [Burns *et al.*, 2022] Andrea Burns, Deniz Arsan, Sanjna Agrawal, Ranjitha Kumar, Kate Saenko, and Bryan A. Plummer. A dataset for interactive vision language navigation with unknown command feasibility. In *European Conference on Computer Vision (ECCV)*, 2022.
- [Chen *et al.*, 2021] Xingyu Chen, Zihan Zhao, Lu Chen, Danyang Zhang, Jiabao Ji, Ao Luo, Yuxuan Xiong, and Kai Yu. Websrc: A dataset for web-based structural reading comprehension, 2021.
- [Chen *et al.*, 2023] Jun Chen, Deyao Zhu, Xiaoqian Shen, Xiang Li, Zechun Liu, Pengchuan Zhang, Raghuraman Krishnamoorthi, Vikas Chandra, Yunyang Xiong, and Mohamed Elhoseiny. Minigpt-v2: large language model as a unified interface for vision-language multi-task learning, 2023.
- [Chiang *et al.*, 2023] Wei-Lin Chiang, Zuhuan Li, Zi Lin, Ying Sheng, Zhanghao Wu, Hao Zhang, Lianmin Zheng, Siyuan Zhuang, Yonghao Zhuang, Joseph E Gonzalez, et al. Vicuna: An open-source chatbot impressing gpt-4 with 90%* chatgpt quality. See <https://vicuna.lmsys.org> (accessed 14 April 2023), 2023.
- [Deng *et al.*, 2023] Xiang Deng, Yu Gu, Boyuan Zheng, Shijie Chen, Samuel Stevens, Boshi Wang, Huan Sun, and Yu Su. Mind2web: Towards a generalist agent for the web, 2023.
- [Du *et al.*, 2022] Zhengxiao Du, Yujie Qian, Xiao Liu, Ming Ding, Jiezhong Qiu, Zhilin Yang, and Jie Tang. Glm: General language model pretraining with autoregressive blank infilling. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 320–335, 2022.
- [Hao *et al.*, 2011] Qiang Hao, Rui Cai, Yanwei Pang, and Lei Zhang. From one tree to a forest: a unified solution for structured web data extraction. In *Proceedings of the 34th International ACM SIGIR Conference on Research and Development in Information Retrieval, SIGIR ’11*, page 775–784, New York, NY, USA, 2011. Association for Computing Machinery.
- [Hong *et al.*, 2023] Wenyi Hong, Weihan Wang, Qingsong Lv, Jiazheng Xu, Wenmeng Yu, Junhui Ji, Yan Wang, Zihan Wang, Yuxiao Dong, Ming Ding, et al. Cogagent: A visual language model for gui agents. *arXiv preprint arXiv:2312.08914*, 2023.
- [Kolb, 2014] David A Kolb. *Experiential learning: Experience as the source of learning and development*. FT press, 2014.
- [Li *et al.*, 2020a] Yang Li, Jiacong He, Xin Zhou, Yuan Zhang, and Jason Baldridge. Mapping natural language instructions to mobile ui action sequences. In *Annual Conference of the Association for Computational Linguistics (ACL 2020)*, 2020.
- [Li *et al.*, 2020b] Yang Li, Gang Li, Luheng He, Jingjie Zheng, Hong Li, and Zhiwei Guan. Widget captioning: Generating natural language description for mobile user interface elements, 2020.
- [Li *et al.*, 2023] Zhang Li, Biao Yang, Qiang Liu, Zhiyin Ma, Shuo Zhang, Jingxu Yang, Yabo Sun, Yuliang Liu, and Xiang Bai. Monkey: Image resolution and text label are important things for large multi-modal models. *arXiv preprint arXiv:2311.06607*, 2023.
- [Lin *et al.*, 2015] Tsung-Yi Lin, Michael Maire, Serge Belongie, Lubomir Bourdev, Ross Girshick, James Hays, Pietro Perona, Deva Ramanan, C. Lawrence Zitnick, and Piotr Dollár. Microsoft coco: Common objects in context, 2015.
- [Liu *et al.*, 2018] Evan Zheran Liu, Kelvin Guu, Panupong Pasupat, Tianlin Shi, and Percy Liang. Reinforcement learning on web interfaces using workflow-guided exploration. In *International Conference on Learning Representations (ICLR)*, 2018.
- [Liu *et al.*, 2023a] Haotian Liu, Chunyuan Li, Yuheng Li, and Yong Jae Lee. Improved baselines with visual instruction tuning, 2023.
- [Liu *et al.*, 2023b] Haotian Liu, Chunyuan Li, Qingyang Wu, and Yong Jae Lee. Visual instruction tuning. *arXiv preprint arXiv:2304.08485*, 2023.
- [Nakano *et al.*, 2021] Reiichiro Nakano, Jacob Hilton, Suchir Balaji, Jeff Wu, Long Ouyang, Christina Kim, Christopher Hesse, Shantanu Jain, Vineet Kosaraju, William Saunders, et al. Webgpt: Browser-assisted question-answering with human feedback. *arXiv preprint arXiv:2112.09332*, 2021.
- [Nogueira and Cho, 2016] Rodrigo Nogueira and Kyunghyun Cho. End-to-end goal-driven web navigation. In *Advances In Neural Information Processing Systems*, pages 1903–1911, 2016.
- [Radford *et al.*, 2021] Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, et al. Learning transferable visual models from natural language supervision. In *International conference on machine learning*, pages 8748–8763. PMLR, 2021.
- [Schick *et al.*, 2023] Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools. *arXiv preprint arXiv:2302.04761*, 2023.
- [Sun *et al.*, 2022] Liangtai Sun, Xingyu Chen, Lu Chen, Tianle Dai, Zichen Zhu, and Kai Yu. Meta-gui: Towards multi-modal conversational agents on mobile gui. *arXiv preprint arXiv:2205.11029*, 2022.

- [Touvron *et al.*, 2023] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023.
- [Wang *et al.*, 2021] Bryan Wang, Gang Li, Xin Zhou, Zhourong Chen, Tovi Grossman, and Yang Li. Screen2words: Automatic mobile ui summarization with multimodal learning. In *The 34th Annual ACM Symposium on User Interface Software and Technology, UIST ’21*, page 498–510, New York, NY, USA, 2021. Association for Computing Machinery.
- [Wang *et al.*, 2023a] Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Voyager: An open-ended embodied agent with large language models. *arXiv preprint arXiv:2305.16291*, 2023.
- [Wang *et al.*, 2023b] Lei Wang, Chen Ma, Xueyang Feng, Zeyu Zhang, Hao Yang, Jingsen Zhang, Zhiyuan Chen, Jiakai Tang, Xu Chen, Yankai Lin, et al. A survey on large language model based autonomous agents. *arXiv preprint arXiv:2308.11432*, 2023.
- [Wang *et al.*, 2023c] Lei Wang, Chen Ma, Xueyang Feng, Zeyu Zhang, Hao Yang, Jingsen Zhang, Zhiyuan Chen, Jiakai Tang, Xu Chen, Yankai Lin, Wayne Xin Zhao, Zhewei Wei, and Ji-Rong Wen. A survey on large language model based autonomous agents, 2023.
- [Wang *et al.*, 2023d] Lei Wang, Jingsen Zhang, Hao Yang, Zhiyuan Chen, Jiakai Tang, Zeyu Zhang, Xu Chen, Yankai Lin, Ruihua Song, Wayne Xin Zhao, Jun Xu, Zhicheng Dou, Jun Wang, and Ji-Rong Wen. When large language model based agent meets user behavior analysis: A novel user simulation paradigm. *arXiv preprint arXiv:2306.02552*, 2023.
- [Wang *et al.*, 2023e] Weihan Wang, Qingsong Lv, Wenmeng Yu, Wenyi Hong, Ji Qi, Yan Wang, Junhui Ji, Zhuoyi Yang, Lei Zhao, Xixuan Song, et al. Cogvlm: Visual expert for pretrained language models. *arXiv preprint arXiv:2311.03079*, 2023.
- [Yang *et al.*, 2023a] Zhao Yang, Jiaxuan Liu, Yucheng Han, Xin Chen, Zebiao Huang, Bin Fu, and Gang Yu. Appagent: Multimodal agents as smartphone users. 2023.
- [Yang *et al.*, 2023b] Zhengyuan Yang, Linjie Li, Kevin Lin, Jianfeng Wang, Chung-Ching Lin, Zicheng Liu, and Li-juan Wang. The dawn of lmms: Preliminary explorations with gpt-4v (ision). *arXiv preprint arXiv:2309.17421*, 9(1), 2023.
- [Yao *et al.*, 2023] Shunyu Yao, Howard Chen, John Yang, and Karthik Narasimhan. Webshop: Towards scalable real-world web interaction with grounded language agents, 2023.
- [Ye *et al.*, 2023] Yining Ye, Xin Cong, Shizuo Tian, Jiannan Cao, Hao Wang, Yujia Qin, Yaxi Lu, Heyang Yu, Huadong Wang, Yankai Lin, et al. Proagent: From robotic process automation to agentic process automation. *arXiv preprint arXiv:2311.10751*, 2023.
- [Yin *et al.*, 2023] Shukang Yin, Chaoyou Fu, Sirui Zhao, Ke Li, Xing Sun, Tong Xu, and Enhong Chen. A survey on multimodal large language models. *arXiv preprint arXiv:2306.13549*, 2023.
- [Zeng *et al.*, 2022] Aohan Zeng, Xiao Liu, Zhengxiao Du, Zihan Wang, Hanyu Lai, Ming Ding, Zhuoyi Yang, Yifan Xu, Wendi Zheng, Xiao Xia, et al. Glm-130b: An open bilingual pre-trained model. *arXiv preprint arXiv:2210.02414*, 2022.



ScreenAgent : A Vision Language Model-driven Computer Control Agent

Appendix

A Agent Prompt Details

Below are the prompt templates sent at different stages, the pipeline controller fills these `{{variables}}` in them according to the context.

Planning phase prompt template:

```
You are familiar with the Linux operating system.  
You can see a computer screen with height: {{ video_height }}, width: {{ video_width }}, and the current task is "{{ task_prompt }}", you need to give a plan to accomplish this goal.  
Please output your plan in json format, e.g. my task is to search the web for "What's the deal with the Wheat Field Circle?", the steps to disassemble this task are:  
```json  
[
 {"action_type": "PlanAction", "element": "Open web browser."},
 {"action_type": "PlanAction", "element": "Search in your browser for \"What's the deal with the Wheat Field Circle?\""},
 {"action_type": "PlanAction", "element": "Open the first search result"},
 {"action_type": "PlanAction", "element": "Browse the content of the page"},
 {"action_type": "PlanAction", "element": "Answer the question \"What's the deal with the Wheat Field Circle?\" according to the content."}
]
```\nAnother example, my task is "Write a brief paragraph about artificial intelligence in a notebook", the steps to disassemble this task are:  
```json  
[
 {"action_type": "PlanAction", "element": "Open Notebook"},
 {"action_type": "PlanAction", "element": "Write a brief paragraph about AI in the notebook"}
]
```\n{% if advice_%}  
Here are some suggestions for making a plan: {{ advice_ }}  
{% endif %}  
Now, your current task is "{{ task_prompt }}", give the disassembly steps of the task based on the state of the existing screen image.
```

Acting phase sends prompt:

```
You're very familiar with the Linux operating system and UI operations. Now you need to use the Linux operating system to complete a mission.  
Your goal now is to manipulate a computer screen, video width: {{ video_width }}, video height: {{ video_height }}, the overall mission is: "{{ task_prompt }}".  
We have developed an implementation plan for this overall mission:  
{% for item in sub_task_list %}  
    {{ loop.index }}. {{ item }}  
{% endfor %}  
  
The current subtask is "{{ current_task }}".
```

```
You can use the mouse and keyboard, the optional actions are:
````json
[
 {"action_type": "MouseAction", "mouse_action_type": "click", "mouse_button": "left", "mouse_position": {"width": int, "height": int} },
 {"action_type": "MouseAction", "mouse_action_type": "double_click", "mouse_button": "left", "mouse_position": {"width": int, "height": int} },
 {"action_type": "MouseAction", "mouse_action_type": "scroll_up", "scroll_repeat": int},
 {"action_type": "MouseAction", "mouse_action_type": "scroll_down", "scroll_repeat": int},
 {"action_type": "MouseAction", "mouse_action_type": "move", "mouse_position": {"width": int, "height": int} },
 {"action_type": "MouseAction", "mouse_action_type": "drag", "mouse_button": "left", "mouse_position": {"width": int, "height": int} },
 {"action_type": "KeyboardAction", "keyboard_action_type": "press", "keyboard_key": "KeyName in keysymdef"},
 {"action_type": "KeyboardAction", "keyboard_action_type": "press", "keyboard_key": "Ctrl+A"},
 {"action_type": "KeyboardAction", "keyboard_action_type": "text", "keyboard_text": "Hello, world!"},
 {"action_type": "WaitAction", "wait_time": float}
]
````
```

Where the mouse position is relative to the top-left corner of the screen, and the keyboard keys are described in [keysymdef.h].

Please make output execution actions, please format them in json, e.g.
My plan is to click the Start button, it's on the left bottom corner, so my action will be:

```
````json
[
 {"action_type": "MouseAction", "mouse_action_type": "click", "mouse_button": "left", "mouse_position": {"width": 10, "height": 760} }
]
````
```

Another example, my plan is to open Notepad and I see Mousepad app on the screen, so my action will be:

```
````json
[
 {"action_type": "MouseAction", "mouse_action_type": "double_click", "mouse_button": "left", "mouse_position": {"width": 60, "height": 135} }
]
````
```

{% if advice_%}

Here are some suggestions for performing this subtask: "{{ advice_ }}".

{% endif %}

The current subtask is "{{ current_task }}", please give the detailed next actions based on the state of the existing screen image.

Reflecting phase sends prompt:

You're very familiar with the Linux operating system and UI operations.
Your current goal is to act as a reward model to judge whether or not this image meets the goal, video width: {{ video_width }}, video height: {{ video_height }}

```

, the overall mission is: "{{ task_prompt }}".  

We have developed an implementation plan for this overall mission:  

{%
  for item in sub_task_list %
    {{ loop.index }}. {{ item }}
  % endfor %

Now the current subtask is: "{{ current_task }}".  

Please describe whether or not this image meets the current subtask, please  

answer json format:  

Here are a few options, if you think the current subtask is done well, then  

output this:  

```json {"action_type": "EvaluateSubTaskAction", "situation": "sub_task_success"
"} ```

The mission will go on.

If you think the current subtask is not done well, need to retry, then output

this:

```json {"action_type": "EvaluateSubTaskAction", "situation": "need_retry", "advice": ""I don't think you're clicking in the right place."} ```

You can give some suggestions for implementation improvements in the "advice" field.  

If you feel that the whole plan does not match the current situation and you  

need to reformulate the implementation plan, please output:  

```json {"action_type": "EvaluateSubTaskAction", "situation": "need_reformulate",
", "advice": "I think the current plan is not suitable for the current

situation, because the system does not have installed"} ```

You can give some suggestions for reformulating the plan in the "advice" field.

Please surround the json output with the symbols ```json and ```.

The current goal is: "{{ task_prompt }}", please describe whether or not this

image meets the goal in json format? And whether or not our mission can

continue.

```

## B Construction and Processing for the COCO, Widget Captions, and Mind2Web Datasets

To enhance the model localization capabilities, we use three additional datasets to perform a phased blending with the ScreenAgent Dataset, converting them to keyboard and mouse actions that are consistent with those of the ScreenAgent. The converting operations we performed on each dataset are described below:

### B.1 COCO & Widget Captions Dataset

The COCO and Widget Captions are two objective detection datasets. Each sample contains the image, the caption of an object in the image, and the bounding box of this object. We therefore convert this into a mouse click and drag action that requires the model to click on the center of the object or drag to draw a box from the top left corner to the bottom right corner. We use the following answer templates to construct these two tasks:

```

My plan is to click {{ task_prompt }}, so my action will be:

```json
[
  {"action_type": "MouseAction", "mouse_action_type": "click", "mouse_button": "left", "mouse_position": {"width": {{ center_width }}, "height": {{ center_height }}}}
]
```

```

```

My plan is to drag draw a box of {{ task_prompt }}, so my action will be:
````json
[
    {"action_type": "MouseAction", "mouse_action_type": "move", "mouse_position": {"width": {{ drag_start_width }}, "height": {{ drag_start_height }}}},
    {"action_type": "MouseAction", "mouse_action_type": "drag", "mouse_button": "left", "mouse_position": {"width": {{ drag_end_width }}, "height": {{ drag_end_height }}}}}
]
```

```

## B.2 Mind2Web Dataset

Mind2Web consists of sessions that consist of multiple steps to accomplish a task on one website. Each session consists of multiple actions, with screenshots of the web page before and after each action. There are three types of actions in Mind2Web: CLICK, SELECT, and TYPE. We use the following answer template to transform the Mind2Web dataset according to the same action definition as ScreenAgent.

```

To finish "{{ task_prompt }}", I need to finish the current_task"{{ current_task }}"
by this action:
{% if operation_type == 'CLICK' or operation_type == 'SELECT' %}
````json
[
    {"action_type": "MouseAction", "mouse_action_type": "click", "mouse_button": "left", "mouse_position": {"width": {{ center_width }}, "height": {{ center_height }}}}}
]
```
{% elif operation_type == 'TYPE' %}
````json
[
    {"action_type": "MouseAction", "mouse_action_type": "click", "mouse_button": "left", "mouse_position": {"width": {{ center_width }}, "height": {{ center_height }}}},
    {"action_type": "KeyboardAction", "keyboard_action_type": "text", "keyboard_text": ":{{ operation_value }}"}{% if is_last_action_in_subsession %},
    {"action_type": "KeyboardAction", "keyboard_action_type": "press", "keyboard_key": "Enter"}{% endif %}
]
```
{% endif %}

```

In addition, we have designed a planning template to enhance the ScreenAgent model's planning ability in web scenarios:

```

I can see a {{ website }} page about {{ domain }}{{ subdomain }}, and I'm now
targeting {{ task_prompt }}.
Based on the screen I'm seeing I've set up some detailed plans for this goal:
````json
[
    {% for item in sub_task_list %}
        {"action_type": "PlanAction", "element": "{{ item }}"}{% if not loop.last %},{%
            endif %}
    {% endfor %}
]
```

```

Agents can schedule multiple actions when actions can be done in the same page without jumping to another page.

## C Details of ScreenAgent Dataset

We conducted some statistical analysis of the dataset from various perspectives. As illustrated in Fig. 5(a), the dataset encompasses five types of actions, with Mouse action comprising the majority, excluding Plan action and Evaluate action. This aligns with real-world scenarios, where humans predominantly use the mouse for computer interactions.

The task prompt represents the overall objective provided by the user. Fig. 9(a) and Table 5 display the statistical information regarding the number of tokens in the task prompts. Chinese will consume more tokens than English due to the encoding efficiency of the tokenizer.

The Fig. 9(b) illustrates the distribution of the number of subtask plan elements required in the training set to complete the entire task. The most complex tasks demand up to 13 steps in planning. Approximately 60% of the tasks require a plan consisting of 3 to 5 steps, and the average number of steps in the plans for these tasks is 4.

The statistics regarding the number of actions in subtasks are also included in Table 5. This implies that, on average, 1.5 control actions are required in one interaction with the screen.

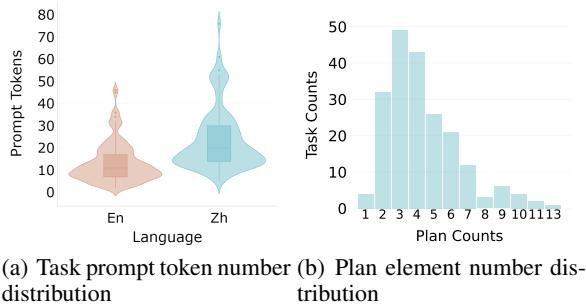


Figure 9: Some statistical information of the ScreenAgent training set.

| Type                       | # Average | # Max | # Min |
|----------------------------|-----------|-------|-------|
| Task prompt tokens(En)     | 13.2      | 46    | 2     |
| Task prompt tokens(Zh)     | 23.8      | 76    | 5     |
| Chosen response tokens(En) | 97.1      | 779   | 19    |
| Chosen response tokens(Zh) | 129.9     | 845   | 27    |
| Actions in sub-task        | 1.5       | 18    | 1     |

Table 5: The statistics about tokens in the ScreenAgent training set.

## D CC-Score

Consider two action sequences, namely, the label action sequence  $L$  and the pred action sequence  $P$ . We denote the label action sequence as  $L = \{l_1, l_2, \dots, l_n\}$  and the prediction action sequence as  $P = \{p_1, p_2, \dots, p_m\}$ . Define a score matrix  $S$  as an  $n \times m$  matrix, where  $S_{ij}$  represents the similarity score between action  $l_i$  and action  $p_j$ .

A possible alignment is a sequence  $C = \{(c_1, d_1), (c_2, d_2), \dots, (c_k, d_k)\}$ , where each element  $(c_i, d_i)$  is a pair of actions such that  $c_i \in L \cup \{\text{None}\}$  and  $d_i \in P \cup \{\text{None}\}$ . Each pair  $(c_i, d_i)$  either consists

of corresponding actions from  $L$  and  $P$ , or a combination of an action from either  $L$  or  $P$  with a null value (None). Importantly, this sequence must adhere to the constraint that the order of non-null actions in  $L$  and  $P$  within the alignment is preserved as in their original sequences. This means that if  $(c_i, d_i)$  and  $(c_j, d_j)$  are two pairs in  $C$  where  $c_i, c_j \neq \text{None}$  and  $d_i, d_j \neq \text{None}$ , and if  $i < j$ , then  $c_i$  must precede  $c_j$  in  $L$  and  $d_i$  must precede  $d_j$  in  $P$ . This constraint ensures that the alignment respects the sequential nature and integrity of the original action sequences.

For a given alignment  $C$ , its score is the sum of the similarity scores for all matched pairs of actions in the alignment, that is,  $\sum_{(i,j) \in C} S_{ij}$ . We choose the alignment with the highest score as the best matching alignment  $C^*$ .

Finally, the CC-Score for the prediction and label action sequence is calculated as:

$$\text{CC-Score}(L, P) = \frac{1}{|L|} \sum_{(i,j) \in C^*} S_{ij}$$

## E More Generated Samples of ScreenAgent

To more concretely demonstrate ScreenAgent’s performance in accomplishing the task and the ability of the process in the three phases of planning, action, and reflection, we provide additional samples in the following pages.

## F ScreenAgent Training Configurations

Table 6 shows the hyper-parameters we used to train the ScreenAgent model.

| Hyper-parameters          | Fine-tune   |
|---------------------------|-------------|
| Total steps               | 6258        |
| Warmup step ratio         | 0.02        |
| Learning rate             | 1e-5        |
| Learning rate decay style | Consine     |
| Batch size                | 8           |
| Weight decay              | 0.05        |
| Adam $\epsilon$           | 1e-8        |
| Adam $\beta$              | (0.9, 0.95) |
| Gradient clipping         | 0.1         |

Table 6: ScreenAgent model training hyper-parameters



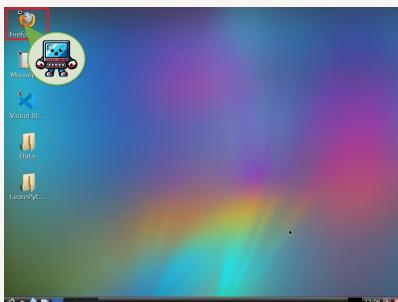
## User: Watch a tutorial video on seated rowing



**ScreenAgent(Planning Phase):**

```
```json
[{"action_type": "PlanAction", "element": "Open browser"}, {"action_type": "PlanAction", "element": "Enter the video website URL in the browser address bar"}, {"action_type": "PlanAction", "element": "Enter the search term \"seated rowing tutorial\" in the video website search box"}, {"action_type": "PlanAction", "element": "Find the appropriate seated rowing tutorial video in the search results and watch it"}]
```

```



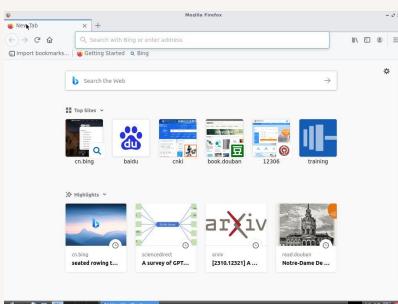
### Subtask: Open browser

#### ScreenAgent(Acting Phase):

According to the screen image provided, the Firefox Web Browser icon is located in the upper left corner of the desktop. To execute the subtask "Double-click on the Firefox Web Browser icon to open it.", the following JSON action sequence will be performed:

```
```json
[{"action_type": "MouseAction", "mouse_action_type": "double_click", "mouse_button": "left", "mouse_position": {"width": 50, "height": 40}}]
```

```

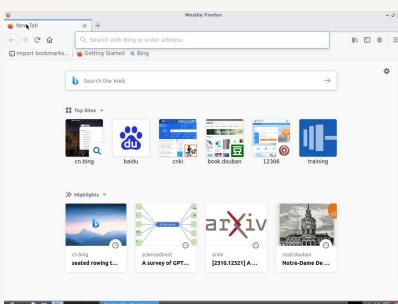


### Subtask: Open browser

#### ScreenAgent(Reflecting Phase):

```
```json
[{"action_type": "EvaluateSubTaskAction", "situation": "sub_task_success"}]
```

```



### Subtask: Enter the video website URL in the browser address bar

#### ScreenAgent(Acting Phase):

To complete the subtask "Go to the bilibili website", we need to perform the following operations:

```
```json
[{"action_type": "KeyboardAction", "keyboard_action_type": "text", "keyboard_text": "www.bilibili.com"}, {"action_type": "KeyboardAction", "keyboard_action_type": "press", "keyboard_key": "Return"}]
```

```

This will open the browser and navigate to the bilibili website.



### Subtask: Enter the video website URL in the browser address bar

#### ScreenAgent(Reflecting Phase):

```
```json
[{"action_type": "EvaluateSubTaskAction", "situation": "sub_task_success"}]
```

```

You can see that the bilibili website has been opened in the browser window, and you can continue to perform the next subtask.

Figure 10: An example of "Watch a tutorial video on seated rowing".



**Subtask:** Enter the search term "seated rowing tutorial" in the video website search box

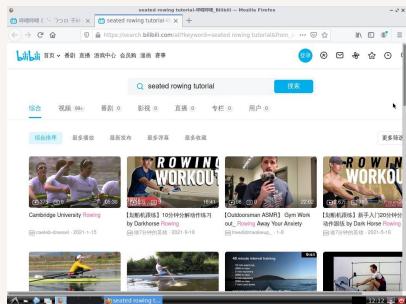
**ScreenAgent(Acting Phase):**

According to the screenshot you provided, the search box is located in the upper central area of the screen, and the text "seated rowing tutorial" is entered in the search box. Now, we need to perform the following actions to complete the subtask of "Enter 'seated rowing tutorial' in the search field":

```json

```
[{"action_type": "MouseAction", "mouse_action_type": "click", "mouse_button": "left", "mouse_position": {"width": 552, "height": 130}}, {"action_type": "KeyboardAction", "keyboard_action_type": "text", "keyboard_text": "seated rowing tutorial"}, {"action_type": "KeyboardAction", "keyboard_action_type": "press", "keyboard_key": "Return"}]
```

```



**Subtask:** Enter the search term "seated rowing tutorial" in the video website search box

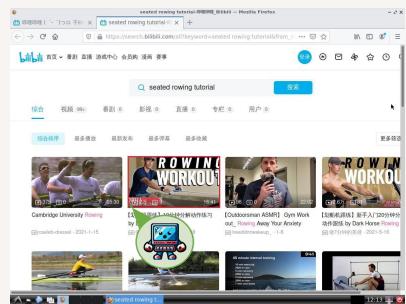
**ScreenAgent(Reflecting Phase):**

```json

```
[{ "action_type": "EvaluateSubTaskAction", "situation": "sub_task_success"}]
```

```

You can see that the search field has been filled with 'seated rowing tutorial', and the next step can be carried out.



**Subtask:** Find the appropriate seated rowing tutorial video in the search results and watch it

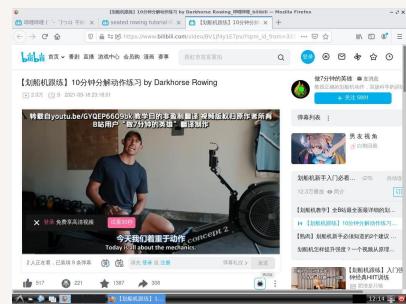
**ScreenAgent(Acting Phase):**

According to the screenshot, we can see that the search results are displayed on the screen, and the first video is a video about seated rowing. Therefore, we need to perform the following actions to click on the video:

```json

```
[{"action_type": "MouseAction", "mouse_action_type": "click", "mouse_button": "left", "mouse_position": {"width": 315, "height": 513}}]
```

```



**Subtask:** Find the appropriate seated rowing tutorial video in the search results and watch it

**ScreenAgent(Reflecting Phase):**

```json

```
[{ "action_type": "EvaluateSubTaskAction", "situation": "sub_task_success"}]
```

```

You can see that the video is playing.

Figure 11: Continue the example of "Watch a tutorial video on seated rowing".



## User:Open Event Viewer



**ScreenAgent(Planning Phase):**

```
```json
[
    {"action_type": "PlanAction", "element": "Right click"}, 
    {"action_type": "PlanAction", "element": "Select \"Event Viewer\" from the pop-up menu"}]
```
```



**Subtask: Right click**

**ScreenAgent(Acting Phase):**

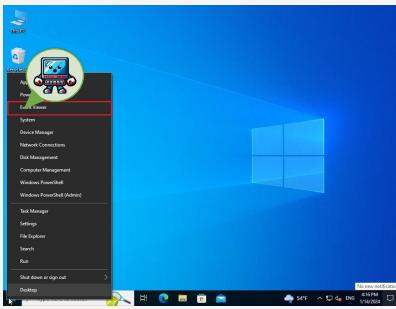
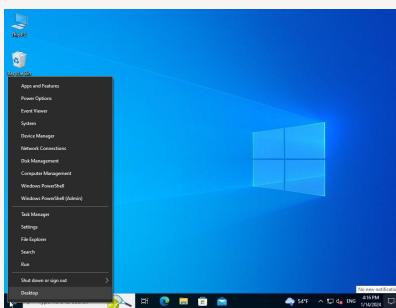
To complete the subtask "Right-click on the Start menu icon.", we can move the mouse pointer to the location of the Start menu icon and perform the right-click operation. The following is the JSON format action plan:

```
```json
[
    {"action_type": "MouseAction", "mouse_action_type": "click", "mouse_button": "right", "mouse_position": {"width": 10, "height": 750}}
]
```
```

**Subtask:Right click**

**ScreenAgent(Reflecting Phase):**

```
```json
[
    {
        "action_type": "EvaluateSubTaskAction",
        "situation": "sub_task_success"
    }
]
```
```

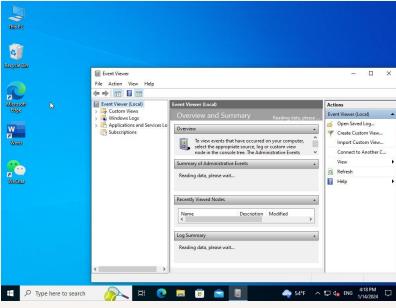


**Subtask:Select "Event Viewer" from the pop-up menu**

**ScreenAgent(Acting Phase):**

According to the screen image, we can see that the Event Viewer option has been displayed in the context menu. Therefore, our next step should be:

```
```json
[
    {"action_type": "MouseAction", "mouse_action_type": "click", "mouse_button": "left", "mouse_position": {"width": 75, "height": 260}}
]
```
```



**Subtask:Select "Event Viewer" from the pop-up menu**

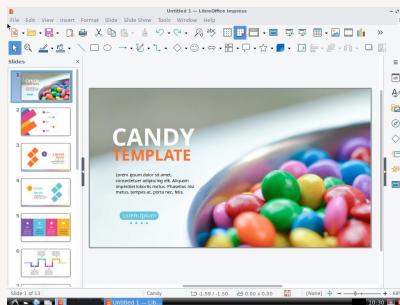
**ScreenAgent(Reflecting Phase):**

```
```json
[
    {
        "action_type": "EvaluateSubTaskAction",
        "situation": "sub_task_success"
    }
]
```
```

Figure 12: An example of a complete task in the test set.



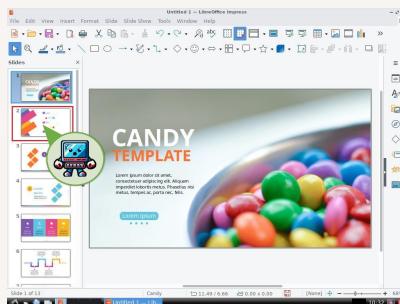
## User:Delete the second slide of a presentation



### ScreenAgent(Planning Phase):

```
```json
[
  {"action_type": "PlanAction", "element": "Select the second slide in the slide show"},
  {"action_type": "PlanAction", "element": "Right click"},  

  {"action_type": "PlanAction", "element": "Select Delete Slide"}]
```

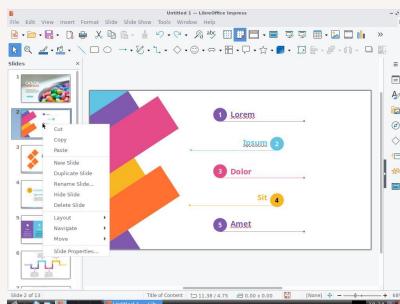


Subtask: Select the second slide in the slide show

ScreenAgent(Acting Phase):

In this case, we need to right-click on the second slide to bring up the context menu. According to the screenshot provided, the second slide is the title slide of the presentation, and the title of the slide is "CANDY TEMPLATE". We can assume that the slide is in the center of the screen. Here is the action sequence to complete this task:

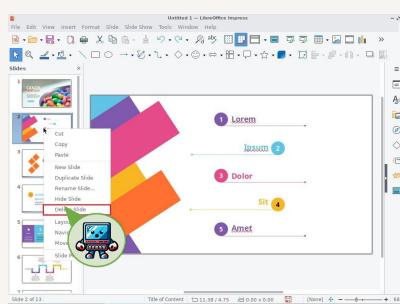
```
```json
[
 {"action_type": "MouseAction", "mouse_action_type": "click", "mouse_button": "right", "mouse_position": {"width": 100, "height": 282}}]
```



### Subtask:Select the second slide in the slide show

#### ScreenAgent(Reflecting Phase):

```
```json
[
  {
    "action_type": "EvaluateSubTaskAction",
    "situation": "sub_task_success"
  }
]
```

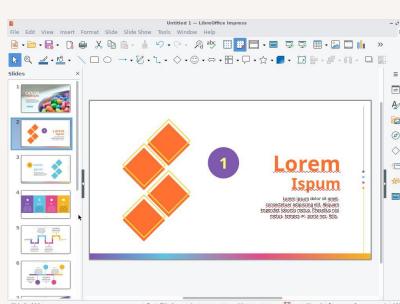


Subtask:Select Delete Slide

ScreenAgent(Acting Phase):

In this case, we need to click the "Delete Slide" option in the context menu. The following is the JSON format action sequence for this operation:

```
```json
[
 {"action_type": "MouseAction", "mouse_action_type": "click", "mouse_button": "left", "mouse_position": {"width": 150, "height": 495}}]
```



### Subtask:Select Delete Slide

#### ScreenAgent(Reflecting Phase):

```
```json
[
  {
    "action_type": "EvaluateSubTaskAction",
    "situation": "sub_task_success"
  }
]
```

Figure 13: An example of "Delete the second slide of a presentation".

 User:Change the speed of an opened web video to 1.5x

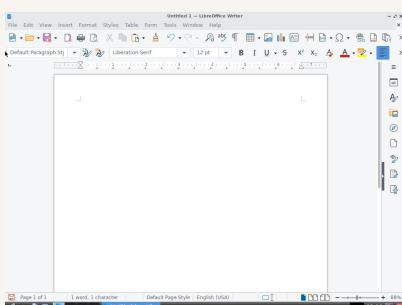


ScreenAgent(Planning Phase):

```
```json
[{"action_type": "PlanAction", "element": "Find the speed options"}, {"action_type": "PlanAction", "element": "Click the speed option"}, {"action_type": "PlanAction", "element": "Set the speed to 1.5x"}]
```

```

 User:Type "Hello, world!" and bold it

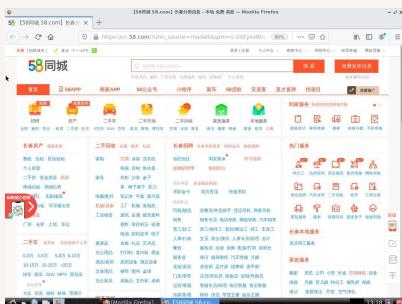


ScreenAgent(Planning Phase):

```
```json
[{"action_type": "PlanAction", "element": "Enter \"Hello, world!\" in the text editor"}, {"action_type": "PlanAction", "element": "Select all text"}, {"action_type": "PlanAction", "element": "Bold the selected text"}]
```

```

 User:Search for prices of second-hand Magotan cars on the 58.com used car platform

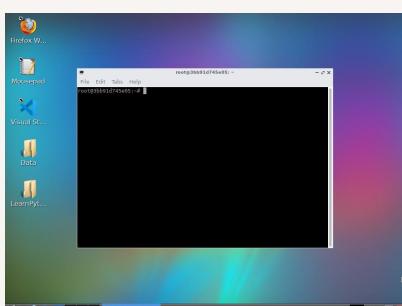


ScreenAgent(Planning Phase):

```
```json
[{"action_type": "PlanAction", "element": "Enter the text \"second-hand Magotan cars\" in the search box on the 58.com website"}, {"action_type": "PlanAction", "element": "Click the search button to perform the search"}, {"action_type": "PlanAction", "element": "View and record the price information of second-hand Magotan cars on the search results page"}]
```

```

 User:Install xeyes in the command line

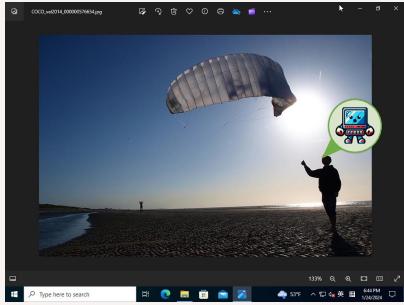


ScreenAgent(Planning Phase):

```
```json
[{"action_type": "PlanAction", "element": "Enter the installation command 'sudo apt-get install xeyes'"}, {"action_type": "PlanAction", "element": "Press Enter to execute the installation command"}, {"action_type": "PlanAction", "element": "After viewing the successful installation information, close the terminal"}]
```

```

Figure 14: Some examples of the planning stage.



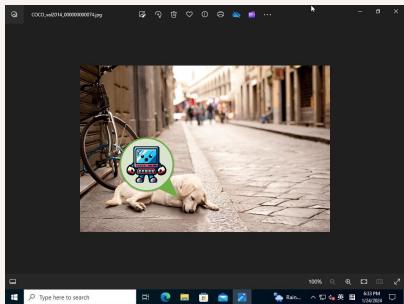
Subtask: Click the people in the picture



ScreenAgent(Acting Phase):

To complete the subtask "Click the people in the picture", I need to move the mouse cursor to the position of the people in the picture and click.

```
```json
[
 {"action_type": "MouseAction", "mouse_action_type": "click",
 "mouse_button": "left", "mouse_position": {"width": 811, "height": 400}}
]..
```



**Subtask:** Please click the dog in the picture



**ScreenAgent(Acting Phase):**

To complete "Please click the dog in the picture", I need to finish the current\_task "Click the dog" by this action:

```
```json
[
    {"action_type": "MouseAction", "mouse_action_type": "click",
     "mouse_button": "left", "mouse_position": {"width": 449, "height": 495}}
]..
```



Subtask: drag draw a box of stop sign



ScreenAgent(Acting Phase):

To finish drag draw a box of stop sign, i will click UI element by this action:

```
```json
[
 {"action_type": "MouseAction", "mouse_action_type": "move", "mouse_position": {"width": 393, "height": 42}},
 {"action_type": "MouseAction", "mouse_action_type": "drag", "mouse_button": "left", "mouse_position": {"width": 619, "height": 310}}
]..
```



**Subtask:** drag draw a box of boat



**ScreenAgent(Acting Phase):**

To finish "drag draw a box of boat", I need to finish the current task "drag draw a box of boat" by this action:

```
```json
[
    {"action_type": "MouseAction", "mouse_action_type": "move", "mouse_position": {"width": 135, "height": 308}},
    {"action_type": "MouseAction", "mouse_action_type": "drag", "mouse_button": "left", "mouse_position": {"width": 500, "height": 449}}
]..
```

Figure 15: Tested on natural images from the COCO dataset.



Subtask:Select the cutting Tool from the toolbox

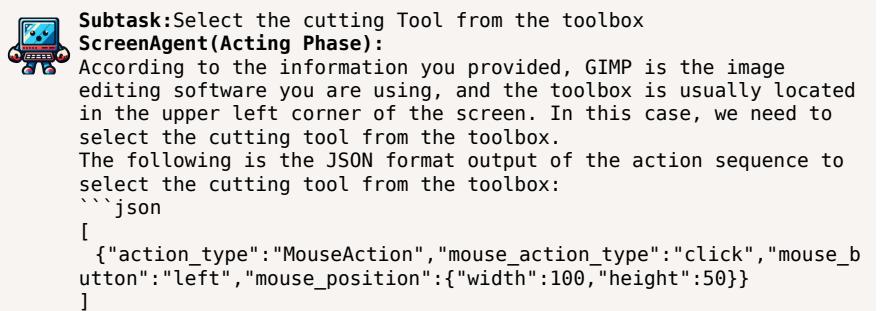
ScreenAgent(Acting Phase):

According to the information you provided, GIMP is the image editing software you are using, and the toolbox is usually located in the upper left corner of the screen. In this case, we need to select the cutting tool from the toolbox.

The following is the JSON format output of the action sequence to select the cutting tool from the toolbox:

```
```json
[
 {"action_type": "MouseAction", "mouse_action_type": "click", "mouse_button": "left", "mouse_position": {"width": 100, "height": 50}}
]
```

```



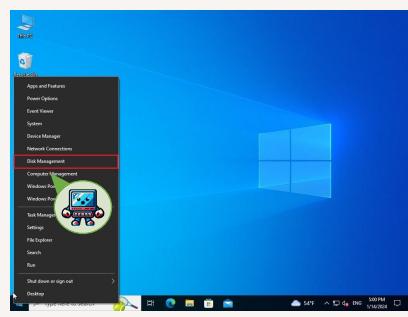
Subtask:Click on the remove or delete option to remove the smartphone from the cart

ScreenAgent(Acting Phase):

According to the screenshot provided, the smartphone is located in the shopping cart and marked as "Remove" or "Delete". You can click this option to remove the smartphone from the cart. The following is the JSON format instruction to execute this action:

```
```json
[
 {"action_type": "MouseAction", "mouse_action_type": "click", "mouse_button": "left", "mouse_position": {"width": 315, "height": 659}}
]
```

```



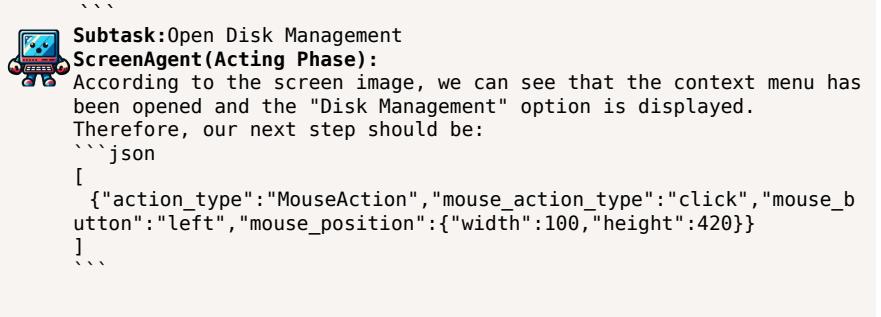
Subtask:Open Disk Management

ScreenAgent(Acting Phase):

According to the screen image, we can see that the context menu has been opened and the "Disk Management" option is displayed. Therefore, our next step should be:

```
```json
[
 {"action_type": "MouseAction", "mouse_action_type": "click", "mouse_button": "left", "mouse_position": {"width": 100, "height": 420}}
]
```

```



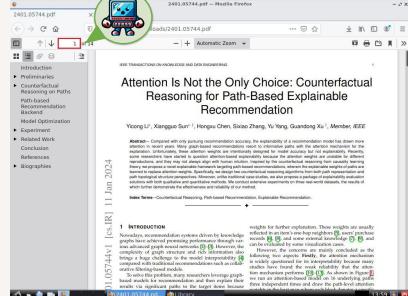
Subtask:Select 'Align Text'

ScreenAgent(Acting Phase):

In this case, you need to click the "Align Text" option in the "Alignment" section of the "Format" menu. The following is the JSON representation of the action:

```
```json
[
 {"action_type": "MouseAction", "mouse_action_type": "click", "mouse_button": "left", "mouse_position": {"width": 250, "height": 650}}
]
```

```



Subtask:Click on the page number input box located near the top of the PDF viewer

ScreenAgent(Acting Phase):

According to the screenshot provided, the page number input box is located near the top of the PDF viewer, just above the document title. To click on this input box, we can use the following actions:

```
```json
[
 {"action_type": "MouseAction", "mouse_action_type": "click", "mouse_button": "left", "mouse_position": {"width": 191, "height": 108}}
]
```

```

Figure 16: Some cases of failure. The golden label clickable areas are indicated by red bounding boxes.