CRAB: Cross-environment Agent Benchmark for Multimodal Language Model Agents

Anonymous Author(s)
Affiliation
Address
email

Abstract

The development of autonomous agents increasingly relies on Multimodal Language Models (MLMs) to perform tasks described in natural language with GUI environments, such as websites, desktop computers, or mobile phones. Existing benchmarks for MLM agents in interactive environments are limited by their focus on a single environment, lack of detailed and generalized evaluation methods, and the complexities of constructing tasks and evaluators. To overcome these limitations, we introduce CRAB, the first agent benchmark framework designed to support cross-environment tasks, incorporating a graph-based fine-grained evaluation method and an efficient mechanism for task and evaluator construction. Our framework supports multiple devices and can be easily extended to any environment with a Python interface. Leveraging CRAB, we developed a cross-platform CRAB Benchmark-v0 comprising 100 tasks in computer desktop and mobile phone environments. We evaluated four advanced MLMs using different single and multi-agent system configurations on this benchmark. The experimental results demonstrate that the single agent with GPT-4o achieves the best completion ratio of 35.26%.

1 Introduction

The development of autonomous agents for human-centric interactive systems—such as desktop OS [51], websites [56, 15], smartphones [52, 47], and games [38, 39]—has long been an important goal of AI research, aiming to convert natural language instructions into concrete operations. Traditionally, these challenges have been addressed using reinforcement learning [27]. Recently, Large Language Models (LLMs) have demonstrated remarkable proficiency in natural language understanding and commonsense reasoning, making them vital tools for developing autonomous agents. This utility is further enhanced by Multimodal Language Models (MLMs), which improve the ability to interpret visual information from GUIs [5].

To effectively develop MLM-based autonomous agents for real-world applications, it is essential to create suitable benchmarks for standardized performance evaluation. However, existing benchmarks still have limitations in terms of interaction methods, platform diversity, evaluation metrics, static task dataset that prevent them from closely mirroring complex real-world applications. First, existing benchmarks that interact with the environments through pre-collected observation data from system environments [36, 26, 6] fail to capture the dynamic nature of real-world scenarios without interactive exploration where data and conditions can change unpredictably. Second, existing benchmarks are typically evaluated on a single platform, either Web, Android, or Desktop OS [34, 47, 46]. However,

Submitted to the 38th Conference on Neural Information Processing Systems (NeurIPS 2024) Track on Datasets and Benchmarks. Do not distribute.

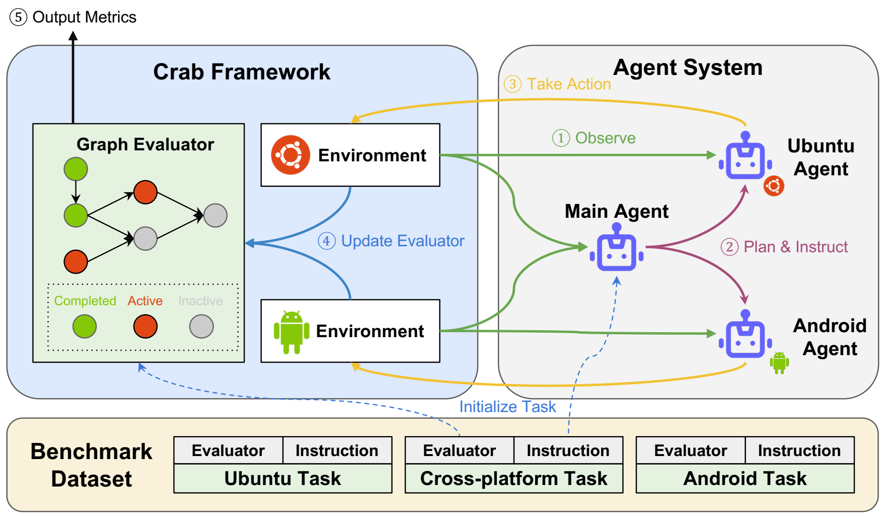


Figure 1: **Architecture of the Crab Framework demonstrating a benchmarking workflow for a multi-agent system.** A task is initialized by assigning instructions to the main agent and a graph evaluator inside the benchmark system. The workflow progresses through a cycle where the main agent observes, plans, and instructs the sub-agents, who then execute actions within their respective environments. The graph evaluator monitors the status of tasks within the environments, continuously updating and outputting the task completion metrics throughout the workflow.

the practical applications usually involve tasks that span multiple platforms. For example, using a smartphone to take a photo and sending it to a desktop for editing with a graphics editor is a common real-world task across multiple platforms. Third, existing evaluation methods are generally either goal-based or trajectory-based [34, 47]. Goal-based methods typically employ coarse-grained binary metrics, solely evaluating whether the final system state aligns with the task's goal. In contrast, trajectory-based methods can offer more fine-grained metrics by assessing the agent's action trajectory against a ground truth trajectory yet ignore the possibility of multiple valid pathways to complete a task, making the evaluation results less fair. Lastly, task creation within these complex systems are not static and extensible with fixed templates [36, 46], which limits the diversity and scope of tasks.

We propose a benchmark that closely mirrors real-world situations and an evaluation method that more accurately reflects an agent's performance on complex tasks. To this end, we introduce CRAB, a novel **CR**oss-environment **A**gent **B**enchmark framework. CRAB provides a comprehensive framework for evaluating cross-environment tasks in interactive environments, where the agent needs to operate simultaneously across various devices and platforms, adapting to varied system conditions to complete tasks efficiently. To the best of our knowledge, CRAB is the first autonomous agent benchmark framework that incorporates the **cross-environment tasks**. Moreover, we propose a novel evaluation method called **graph evaluator**. Unlike traditional goal-based and trajectory-based evaluation, our graph evaluator checks the intermediate procedures of completing a task by decomposing the task into multiple sub-goals. Each sub-goal is assigned a judge function to verify its completeness, and each is considered a node in the graph evaluator. The graph structure describes the sequential and parallel relationships between the sub-goals. Therefore, it offers fine-grained metrics similar to trajectory-based evaluations while accommodating multiple valid pathways to a solution, making it more suitable for evaluating tasks that involve various correct approaches. To solve the increasing complexity in cross-environment task construction. We also propose a highly extensible graph-based task construction method called **sub-task composition**. Combining multiple sub-tasks in a graph with task targets allows for efficient construction of various cross-environment tasks with corresponding graph evaluators. Table 1 compares CRAB with existing frameworks.

Based on CRAB framework, we develop a benchmark CRAB Benchmark-v0 with two collaborated environments that include an Android emulator and an Ubuntu desktop virtual machine. We

Table 1: **Comparison of existing agent benchmark frameworks.** The columns details key features of each framework: *Interactive Environment* indicates the presence of either interactive environments or static datasets; *Multimodal Observation* specifies the availability of vision-based observations; *Cross-platform* denotes support for multiple platforms; *Evaluation* describes the evaluation metrics, categorized as *Goal-based* (checking environment state according solely on the final goal), *Trajectory-based* (comparing agent action trajectory with a gold actions sequence), *Multiple* (varied across tasks), or *Graph-based* (a DAG with each node as an intermediate checkpoint); *Task Construction* shows the task construction method, including *Handmade* (handcrafted by human), *LLM-inspired* (using LLM to generate task drafts but still verified and annotated by human), *Template* (generated by filling in the blanks in task templates), or *Sub-task Composition* (composing multiple sub-tasks to construct tasks and evaluators).

	Interactive Environment	Multimodal Observation	Cross-platform	Evaluation	Task Construction
MINIWOB++ [34]	Web	✓	✗	Goal-based	Handmade
METAGUI [36]	✗	✗	✗	Trajectory-based	Handmade
GAIA [26]	✗	✗	✗	Goal-based	Handmade
MIND2WEB [6]	✗	✗	✗	Goal-based	LLM-inspired
AGENTBENCH [23]	Multi-isolated	✗	✗	Multiple	Handmade
INTERCODE [49]	Code	✗	✗	Goal-based	Handmade
WEBARENA [56]	Web	✓	✗	Goal-based	Template
WEBSHOP [50]	Web	✓	✗	Goal-based	Template
OMNIACT [12]	✗	✗	✗	Trajectory-based	Handmade
VWEBARENA [15]	Web	✓	✗	Goal-based	Template
ANDROIDARENA [47]	Android	✓	✗	Trajectory-based	LLM-inspired
OSWORLD [46]	Desktop OS	✓	✗	Goal-based	Template
CRAB	Desktop OS & Android	✓	✓	Graph-based	Sub-task Composition

have developed a total of 100 real-world tasks, encompassing both cross-environment and single-environment tasks across multiple levels of difficulty. These tasks address a wide array of common real-world applications and tools, including but not limited to calendars, email, maps, web browsers, and terminals, and facilitate common collaboration between smartphones and desktops. Considerable time has been invested in verifying the accuracy and comprehensiveness of the instructions for sub-tasks, as well as the generalization and correctness of their evaluators. Most tasks are constructed using a careful composition of sub-tasks, while some tasks are crafted manually to accommodate specific multi-environment collaboration scenarios. We test 4 popular MLMs, including GPT-4 Turbo, GPT-4o, Claude 3 Pro and Gemini 1.5 Pro, across different structures of single agent and multi-agent systems, totaling 9 different agent settings in our benchmarks. The experimental results show that the single agent with GPT-4o model achieves the best completion ratio of 35.26%, underscoring the necessity for ongoing development of more effective autonomous agents. Our proposed metrics successfully distinguish between different methods better than previous metrics. We further analyze the different termination reasons that reflect the problems inherent in the function calling feature of current models and communication within the multi-agent system.

2 Related Work

Leveraging LLMs as reasoning units has become an effective approach [42, 10, 45] for building autonomous agents, including embodied agents [39, 35, 4], social simulations [30, 20], web navigation [24], game playing [16, 37], office assistants [18], and code generation [54], among others. With common knowledge of Graphical User Interfaces (GUI) and operating systems, GUI agents [44, 41, 40, 55, 28] are becoming a productive research direction for developing autonomous agents capable of operating systems with GUI interfaces to accomplish complex tasks. GUI agents can typically operate multiple applications within a system, making them more versatile than the aforementioned agents, which are often limited to a single application. Various benchmarks have been developed to evaluate the performance of these GUI agents in interactive environments, which can generally be categorized into three types: web, mobile phone, and desktop.

The web environment is one of the earliest environments used to benchmark agents due to its simplicity, ease of reproduction, straightforward construction, and ease of parsing by agents. One of the earliest examples is Miniwob++ [34], initially designed for evaluating reinforcement learning agents. It quickly became a foundational benchmark for evaluating GUI agents. However, its web page designs are overly simplistic and lack modern features, limiting its ability to assess agents' performance on real-world websites. With the rise of LLMs as agent reasoning engines, more complex web environments, such as WebShop [50] Mind2Web [6] and WebArena [56], have been introduced for benchmarking language model agents, offering realistic and reproducible environments and corresponding web-based tools to simulate sufficiently complex web tasks and cross-environment interactions. Building on these works, Visual WebArena [15] focuses on evaluating multimodal language model agents by incorporating tasks that require visual understanding. Although web environments contain various real world scenarios, it is impossible to replace native applications for complex tasks like multimedia editing, programming, etc.

Intelligent assistants have long been a commercial feature in mobile operating systems, making the motivation to develop mobile agents clear. Additionally, mobile phone operations and observations are generally simpler than those on personal computers, which has made mobile devices a popular environment for benchmarking GUI agents. Several task datasets existed even before the rise of GUI agents. MetaGUI [36] introduced a dataset that focuses on GUI-based task-oriented dialogue systems (GUI-TOD), dividing mobile system control tasks into dialogues and GUI operation traces, while AITW [32] builds the operation traces of challenging multi-step tasks on involving apps and websites on mobile devices based on screenshots. Android Arena [47] underlines the collaboration among android applications and expands simple android tasks into cross-App and constrained tasks, which verifies the potential of LLM-based complicated android system control. AITZ [53] constructs datasets with Chain-of-Thought (CoT) considerations, adding semantic annotations based on visual models at each step and developing operational procedures for selected tasks. In addition, Mobile Agent Bench [43] collects app event signals via Android accessibility services, builds a benchmark with well-annotated operation trajectories, and organizes tasks into different levels of difficulty.

Desktop environments typically have a more complex action space, observation space, and operational logic, making task creation and verification more difficult. Additionally, they are highly customizable and lack generalized tools that can serve as a bridge for agents to interact with the system, which complicates the creation of reproducible environments. OMNIACT [12] is a static benchmark that captures data from multiple desktop operating systems, incorporating visual information from the OS screen UI through segmentation and corresponding tagging. OSWorld [46] provides an interactive and reproducible environment based on XML and screenshots with a standard format. However, both of these works rely on the Python library PyAutoGUI[1] and code generation for operation, which limits the generalizability.

While these benchmarks aim to evaluate an agent's capacity across a wide range of applications, they are built on human-annotated trajectories, which lack scalability. Most tasks are derived from question-and-answer platforms like Stack Overflow or based on annotators' daily usage. While these resources are realistic, they may not effectively test the generalizability of the agent, as the texts are highly likely to appear in the training data. Furthermore, the evaluation methods of previous benchmarks often rely either on full task trajectories or only on the final goals, making it difficult to capture the entire process or to account for partially completed tasks.

3 Definitions

3.1 Problem Formulation

Consider autonomous agents performing a task on a digital device (i.e. desktop computer). Such a device typically has input devices (i.e. mouse and keyboard) for human interaction and output devices (i.e. screen) to allow human observation of its state. In CRAB, we represent this type of device as an

[1]`https://github.com/asweigart/pyautogui`

environment. Formally, this environment is defined as a reward-free Partially Observable Markov Decision Process (POMDP), denoted by the tuple $M := (\mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{O})$, where $\mathcal{S}$ represents the state space, $\mathcal{A}$ the action space, $\mathcal{T} : \mathcal{S} \times \mathcal{A} \to \mathcal{S}$ the transition function, and $\mathcal{O}$ the observation space. Considering the collaborative nature of multiple devices in real-world scenarios, we can combine multiple environments into a set $\mathbf{M} = M_1, M_2, ..., M_n$, where n is the number of environments and each environment $M_j = (\mathcal{S}_j, \mathcal{A}_j, \mathcal{T}_j, \mathcal{O}_j)$. We define a task that requires operations across multiple environments as a **cross-environment task**. This task is formalized as a tuple $(\mathbf{M}, I, R)$, in which $\mathbf{M}$ is the environment set, I is the task objective in the form of natural language instructions, and R is the reward function of the task. An **agent system**, designed to complete a task represented by an instruction I, can be modeled as a policy $\pi((m, a) \mid (I, H, o_1, ..., o_n))$, which defines the probability of taking action a in environment m when receiving observation $(o_1, ..., o_n)$ from environment $(M_1, ..., M_n)$ with a history action trajectory H. An **agent** within the agent system should have a fixed back-end MLM and system prompt, and retain its chat history. An agent system is composed of either a single agent responsible for planning, reasoning, and action-taking or multiple agents connected through a communication strategy to collaborate.

3.2 Graph of Task Decomposition

Decomposing a complex task into several simpler sub-tasks has been proved to be an effective prompting method for LLMs [13]. Some studies represent sub-tasks in a graph structure. For instance, PLaG [19] uses a graph-based structure to enhance plan reasoning within LLMs, while DyVal [57] employs directed acyclic graphs (DAGs) to facilitate dynamic evaluation of LLMs. By introducing this concept into cross-environment tasks, naturally, decomposing a cross-environment task into sub-tasks with in different environments that have both sequential and parallel connections forms a DAG. Therefore, we introduce the **Graph of Decomposed Tasks** (GDT), where each node in the DAG is a sub-task, formalized as a tuple $(\boldsymbol{m}, \boldsymbol{i}, \boldsymbol{r})$, where $\boldsymbol{m}$ specifies the environment in which the sub-task is performed, $\boldsymbol{i}$ provides the subtask natural language instruction, and $\boldsymbol{r}$ represents the reward function. The reward function evaluates the state of m and returns a boolean value to determine if the sub-task is completed. The edges within GDT represent the sequential relationship between sub-tasks. An example GDT is shown in Fig. 2.

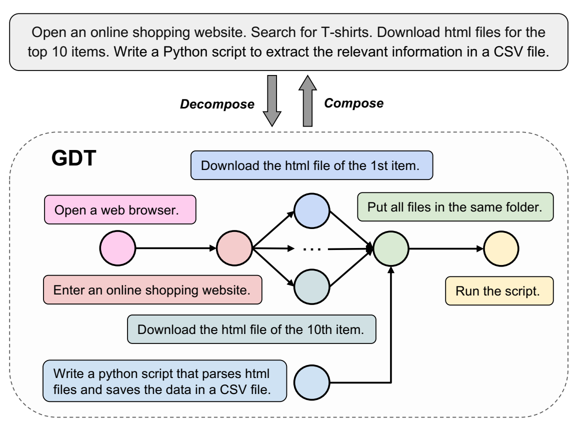


Figure 2: **Graph of Decomposed Tasks.**

4 The Crab Framework

4.1 Cross-environment Agent Interaction

Compared to single-environment tasks, cross-environment tasks offer three main advantages for benchmarking agents. First, cross-environment tasks reflect real-world scenarios where humans use multiple devices simultaneously to accomplish tasks. Second, these tasks require sophisticated message processing and information transfer between environments. Such tasks demand that the agent plan actions, construct outputs for each environment, and remember what needs to be transferred, showcasing a high-level understanding of environments and tasks. Lastly, role-playing multi-agent systems have proven to be effective in executing complex tasks [17, 9]. The underlying principle of their effectiveness is the division of responsibilities. Cross-environment tasks are suited to multi-agent, as they can be divided by distinct observation spaces, action spaces, and specialized knowledge in each environment, as shown in Fig. 1. CRAB uses a unified interface for agents to operate in all environments. Implementation details are in the Appendix A.2.

4.2 Graph Evaluator

Inspired by the "decomposing" idea from GDT (Sec. 3.2), we propose a novel integrated approach, the *Graph Evaluator*, which provides fine-grained metrics and supports multiple valid paths. To build a graph evaluator for a given task, we begin by decomposing the task into a GDT, where each sub-task is associated with an intermediate environment state critical to completing the overall task. Nodes in the graph evaluator activate when they either have no incoming edges or after all their preceding tasks are completed, ensuring a sequential order of tasks. After an agent takes an action, the system checks these active nodes to verify if the target state of each node is reached. A node completion triggers successor nodes to activate and verify the state. This cycle repeats until no new nodes activate, showing that the system's task sequence aligns with the current state of the environment. Unlike trajectory-based methods, which compare sequences of agent actions, the Graph Evaluator does not rely on the specific actions taken by the agent, allowing it the freedom to choose any path. Instead, it concentrates on the key intermediate states of the environment necessary for reaching the final goal.

Given a Graph Evaluator synchronized with the environment state, it becomes possible to track agent progress through the current status of sub-task completions. Beyond the traditional **Success Rate (SR)**, which marks a task as *success* only when all sub-tasks are completed, we introduce three metrics aiming at assessing both performance and efficiency of agents, leveraging the detailed sub-task status provided by the graph evaluator. Specifically, the **Completion Ratio (CR)** measures the proportion of completed sub-task nodes relative to the total nodes in the graph, calculated as C / N, where C is the number of completed nodes and N is the total number of nodes. This metric offers a straightforward measure of an agent's progress on a given task. The **Execution Efficiency (EE)**, calculated as CR / A, where A denotes the count of executed actions. It evaluates how efficiently actions are executed relative to the completion of nodes, reflecting the agent's task execution efficiency. Lastly, the **Cost Efficiency (CE)**, calculated as CR / T, where T is the total number of model tokens used, evaluates the efficiency of resource consuming by the agent.

4.3 Task and Evaluator Construction

Despite the graph evaluator offers detailed evaluations, one challenge is the complexity in creating each evaluator. Creating a graph evaluator requires: (1) adequately decomposing a task into multiple sub-tasks, each with a well-defined graph structure; and (2) engaging an expert of the target platform to carefully craft an evaluator for each sub-task. To efficiently create graph evalautors, we connect sub-tasks as GDTs to formulate new tasks. There are two primary challenges in constructing GDT: (1) Sub-tasks still require manual creation, necessitating a method to quickly generate them on a large scale; (2) Properly modeling the sequential and parallel relationships between sub-tasks, ensuring that the edges connecting sub-task nodes are semantically meaningful and systematically applicable. A template-based approach is commonly used to address the first issue by generating a large number of tasks efficiently. To tackle the second challenge, we employ the message transferring concept (Sec. 4.1). Specifically, if a sub-task α produces an output message that serves as an input for another sub-task β, then α can be considered a legitimate prerequisite of β, allowing us to connect α and β with an directed edge in the GDT. To further refine our approach, we introduce a *sub-task template* structure. Each sub-task is described using a natural language instruction template that includes several replaceable input attributes and an optional output, where each input attributes and output have a fixed type. To generate a GDT, input attributes can be filled with either a hand-crafted value corresponding to their type or linked to a task with the same output type as the input type. From the evaluator's perspective, each sub-task template is linked to an evaluator generator that uses the input attribute value to generate evaluator subgraphs. Once a GDT is constructed, the graph evaluator is created by interlinking each subgraph. The description for the composed task is initially generated by GPT-4 using the sub-task descriptions as prompts and subsequently refined and polished by human reviewers.

5 Experiments

5.1 Benchmark

We build an agent benchmark CRAB Benchmark-v0 featuring with cross-environment, graph evaluator, and task generation through CRAB framework, including an Android smartphone emulator and a Ubuntu Linux desktop virtual machine. Both environments are reproducible and standalone. Detailed environment implementation, observation space and action space are provided in Appendix A.1. we meticulously construct 16 sub-task templates for the Android environment and 19 sub-task templates for the Ubuntu environment. The Ubuntu templates encompass a variety of tasks such as Command Line Interface (CLI) operations, file system management, search engine usage, desktop configurations, and map navigation. Conversely, the Android sub-task templates are primarily focused on the storage and transmission of messages via various applications. Each sub-task template is linked to a graph evaluator consisting of one to four nodes. Each sub-task is verified by at least two related field experts. The dataset has 29 android tasks, 53 Ubuntu tasks and 18 cross-platform tasks. Besides, the sub-task pool has 19 in Ubuntu and 17 in Android.

5.2 Baseline Agent System

At the core of MLM Agents are back-end Multimodal Language Models that provide natural language and image understanding, basic device knowledge, task planning, and logical reasoning abilities. To run in CRAB Benchmark-v0, the back-end model needs to support: (1) Accept multimodal mixed input, as the system provides both screenshots and text instructions as prompts; (2) Handle multi-turn conversations, as most tasks require the agent to take multiple actions, necessitating the storage of history messages in its context; (3) Generate structured output through function calling, ensuring the proper use of provided actions with type-correct parameters. We selected four MLMs that meet these criteria for our experiments: GPT-4o (gpt-4o-2024-05-13) [29], GPT-4 Turbo (gpt-4-turbo-2024-04-09) [1], Gemini 1.5 Pro (May 2024 version) [33], Claude 3 Opus (claude-3-opus-20240229) [2]. To examine how different multi-agent structures impact performance, we design three agent system structures. In the **single agent** structure, one agent manages all responsibilities, including observation analysis, planning, reasoning, and format the output action. The **multi-agent by functionality** structure splits tasks between a main agent, responsible for analysis and planning, and a tool agent that translates instructions into actions without accessing environmental observations. This division allows the main agent to concentrate on high-level tasks without managing functional call formats. Meanwhile, in the **multi-agent by environment** setup, responsibilities are further distributed. A main agent processes all environmental observations for high-level planning, while each environment-specific sub-agent executes actions based on the main agent's instructions, incorporating observations from their respective environments.

For all agents, we utilized the default API parameters and retained two turns of historical messages. The interaction turns are limited to 15 and the task will terminated because reaching max turns. The agent can also terminate the task ahead if it thinks the task is completed. The screenshots do not descale and passed through PNG format with the highest quality that the APIs provide. Detailed agent and prompt designs are shown in Appendix B. In the experiment, we deployed four cloud machines cloned from the same disk image to ensure a consistent environment for all agents. Running a single agent setting in the benchmark requires at least 30 hours to complete on one machine. This duration depends on the API call times and the necessity for manual resets in certain tasks.

5.3 Results

The primary outcomes are detailed in Table 2. The GPT-4o and GPT-4 Turbo models, developed by OpenAI, achieve the highest average success rates and completion ratios among the tested models. Specifically, GPT-4o slightly outperforms GPT-4 Turbo. This result suggests a tiny difference in their underlying architectures or training data, but GPT-4o possibly be trained on more GUI data. Claude 3 outperforms Gemini 1.5 in all settings, according to CR. The multi-agent structures' performances on

Table 2: **Evaluation results on CRAB Benchmark-v0.** The *Model* column identifies the backend masked language models (MLMs) used. The *Structure* column describes the configuration of the agent system: *Single* means *single agent*; *By Func* is *multi-agent by functionality*; *By Env* indicates *multi-agent by environment*. We provide traditional metric of *Success Rate* (SR) alongside newly introduced metrics: *Completion Ratio* (CR), *Execution Efficiency* (EE), and *Cost Efficiency* (CE). Note that Gemini 1.5 Pro has an invalid CE because the Gemini API does not support retrieving token counts at the start time of experiments. The *Termination Reason* shows the ratio of reasons why the agent stops when it does not complete the task. *False Completion* (FC) indicates that the agent believes it has completed the task, but it actually has not; *Reach Step Limit* (RSL) means the agent has reached the step limit but has not completed the task; *Invalid Action* (IA) refers to the agent producing outputs that do not follow instructions, which may include invalid formats, nonexistent actions, or invalid action parameters.

Agent system		**Metrics**				**Termination Reason**		
Model	**Structure**	**SR(%) ↑**	**CR(%) ↑**	**EE(%) ↑**	**CE(%) ↑**	**FC(%)**	**RSL(%)**	**IA(%)**
GPT-4o	Single	**14.00**	**35.26**	**3.66**	5.26×10^{-4}	7.00	59.00	20.00
GPT-4o	By Func	13.00	32.48	3.29	5.20×10^{-4}	12.00	54.00	21.00
GPT-4o	By Env	**14.00**	33.74	3.40	2.71×10^{-4}	8.00	49.00	29.00
GPT-4 Turbo	Single	11.00	31.52	3.60	$\mathbf{6.45 \times 10^{-4}}$	7.00	64.00	18.00
GPT-4 Turbo	By Func	13.00	29.99	3.53	4.79×10^{-4}	11.00	41.00	35.00
Gemini 1.5 Pro	Single	6.00	17.19	1.69	\	3.00	55.00	36.00
Gemini 1.5 Pro	By Func	6.00	14.53	1.50	\	10.00	33.00	51.00
Claude 3 Opus	Single	6.00	21.39	2.66	4.51×10^{-4}	7.00	53.00	34.00
Claude 3 Opus	By Func	5.00	18.79	1.90	3.31×10^{-4}	29.00	32.00	34.00

all back-end MLMs are slightly lower than the single agent, indicating that current autonomous agents mainly rely on back-end model performance. Regarding termination reason, multi-agent structures have higher possibility to take invalid action and incorrectly complete the task, this can caused by the hallucination when main agent generating the instruction messages or misunderstanding of the sub-agents when receiving these messages. We analyze the reasons for the poorer performance of multi-agent structures in Appendix C.2. In terms of execution efficiency, the GPT-4 series show strong performance. However, when evaluating cost efficiency, GPT-4 Turbo exhibited a lower CE value compared to GPT-4o, suggesting that GPT-4 Turbo is more cost-effective.

The completion ratio metric reveals a notable performance difference between models. For instance, even though Claude (single agent) and Gemini (multi-agent by functionality) have the same success rates, their completion ratios differ by up to 6.86%. This highlights the value of the completion ratio in assessing the effectiveness of different methods. We provide more detailed analyses and comparisons of agent configurations in Appendix C.

6 Conclusion

We propose the CRAB framework introducing cross-environment automatic task performing problem, featuring advanced graph-based task generation and evaluation methods, which reduce the manual effort in task step and provide a more dynamic and accurate agent assessments. Based on the framework, we propose CRAB Benchmark-v0, including a set of high quality cross-environment tasks for a smart phone and desktop, equipped with visual prompting strategy. We test various backend models and agent system structures on the dataset. The result reflects preference of different agent settings. Despite our work contributing to better cross-environment agent research, there are still some limitations. We build sub-tasks upon the original apps in the Ubuntu system and the Android system on Pixel, which cannot cover a wider range of applications. Moreover, the visual information is not used in the evaluation on the sub-tasks in Android System. Future works can focus on expanding datasets and environments and testing more models, prompts, structure of agents upon the benchmark.

Acknowledgement

We express our gratitude to Yuhui Wang for refining the expressions in our paper and providing invaluable advice on writing. We would like to also thank Beichen Huang for the helpful discussions on solving virtualization technology issues.

References

[1] Josh Achiam et al. *GPT-4 Technical Report*. Mar. 4, 2024. URL: `http://arxiv.org/abs/2303.08774`. preprint.

[2] Anthropic. *The Claude 3 Model Family: Opus, Sonnet, Haiku*. `https://www-cdn.anthropic.com/de8ba9b01c9ab7cbabf5c33b80b7bbc618857627/Model_Card_Claude_3.pdf`. Year.

[3] Fabrice Bellard. "QEMU, a fast and portable dynamic translator." In: *USENIX annual technical conference, FREENIX Track*. Vol. 41. 46. California, USA. 2005, pp. 10–5555.

[4] Yongchao Chen et al. "Autotamp: Autoregressive task and motion planning with llms as translators and checkers". In: *arXiv preprint arXiv:2306.06531* (2023).

[5] Kanzhi Cheng et al. *SeeClick: Harnessing GUI Grounding for Advanced Visual GUI Agents*. Jan. 17, 2024. URL: `http://arxiv.org/abs/2401.10935`. preprint.

[6] Xiang Deng et al. *Mind2Web: Towards a Generalist Agent for the Web*. 2023. arXiv: `2306.06070 [cs.CL]`.

[7] Johan Edstedt et al. "RoMa: Robust Dense Feature Matching". In: 2024.

[8] Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart. "Exploring Network Structure, Dynamics, and Function Using NetworkX". In: *Proceedings of the 7th Python in Science Conference*. Ed. by Gaël Varoquaux, Travis Vaught, and Jarrod Millman. Pasadena, CA USA, 2008, pp. 11–15.

[9] Sirui Hong et al. "Metagpt: Meta programming for multi-agent collaborative framework". In: *arXiv preprint arXiv:2308.00352* (2023).

[10] Wenlong Huang et al. "Language models as zero-shot planners: Extracting actionable knowledge for embodied agents". In: *International Conference on Machine Learning*. PMLR. 2022, pp. 9118–9147.

[11] Hanwen Jiang et al. "OmniGlue: Generalizable Feature Matching with Foundation Model Guidance". In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. 2024.

[12] Raghav Kapoor et al. *OmniACT: A Dataset and Benchmark for Enabling Multimodal Generalist Autonomous Agents for Desktop and Web*. Feb. 28, 2024. URL: `http://arxiv.org/abs/2402.17553`. preprint.

[13] Tushar Khot et al. "Decomposed Prompting: A Modular Approach for Solving Complex Tasks". In: *The Eleventh International Conference on Learning Representations*. 2023. URL: `https://openreview.net/forum?id=_nGgzQjzaRy`.

[14] Avi Kivity et al. "kvm: the Linux virtual machine monitor". In: *Proceedings of the Linux symposium*. Vol. 1. 8. Dttawa, Dntorio, Canada. 2007, pp. 225–230.

[15] Jing Yu Koh et al. *VisualWebArena: Evaluating Multimodal Agents on Realistic Visual Web Tasks*. Jan. 24, 2024. URL: `http://arxiv.org/abs/2401.13649`. preprint.

[16] Yihuai Lan et al. "Llm-based agent society investigation: Collaboration and confrontation in avalon gameplay". In: *arXiv preprint arXiv:2310.14985* (2023).

[17] Guohao Li et al. "Camel: Communicative agents for" mind" exploration of large scale language model society". In: (2023).

[18] Hongxin Li et al. "SheetCopilot: Bringing Software Productivity to the Next Level through Large Language Models". In: *Advances in Neural Information Processing Systems* 36 (2024).

[19] Fangru Lin et al. *Graph-Enhanced Large Language Models in Asynchronous Plan Reasoning*. Feb. 5, 2024. URL: `http://arxiv.org/abs/2402.02805`. preprint.

[20] Jiaju Lin et al. "Agentsims: An open-source sandbox for large language model evaluation". In: *arXiv preprint arXiv:2308.04026* (2023).

[21] Nelson F. Liu et al. *Lost in the Middle: How Language Models Use Long Contexts*. Nov. 20, 2023. URL: http://arxiv.org/abs/2307.03172. Pre-published.

[22] Shilong Liu et al. *Grounding DINO: Marrying DINO with Grounded Pre-Training for Open-Set Object Detection*. arXiv.org. Mar. 9, 2023. URL: https://arxiv.org/abs/2303.05499v4.

[23] Xiao Liu et al. "AgentBench: Evaluating LLMs as Agents". In: *The Twelfth International Conference on Learning Representations*. 2024. URL: https://openreview.net/forum?id=zAdUB0aCTQ.

[24] Xing Han Lù, Zdeněk Kasner, and Siva Reddy. *WebLINX: Real-World Website Navigation with Multi-Turn Dialogue*. arXiv.org. Feb. 8, 2024. URL: https://arxiv.org/abs/2402.05930v1.

[25] Michael M. McKerns et al. *Building a Framework for Predictive Science*. Feb. 6, 2012. URL: http://arxiv.org/abs/1202.1056. preprint.

[26] Grégoire Mialon et al. *GAIA: A Benchmark for General AI Assistants*. Nov. 21, 2023. URL: http://arxiv.org/abs/2311.12983. preprint.

[27] Volodymyr Mnih et al. "Human-Level Control through Deep Reinforcement Learning". In: *Nature* 518.7540 (Feb. 2015), pp. 529–533. ISSN: 1476-4687. URL: https://www.nature.com/articles/nature14236.

[28] Runliang Niu et al. *ScreenAgent: A Vision Language Model-Driven Computer Control Agent*. Feb. 8, 2024. URL: http://arxiv.org/abs/2402.07945. preprint.

[29] OpenAI. *GPT-4 omni*. https://openai.com/index/hello-gpt-4o/. 2024.

[30] Joon Sung Park et al. "Generative agents: Interactive simulacra of human behavior". In: *Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology*. 2023, pp. 1–22.

[31] Guilherme Potje et al. "XFeat: Accelerated Features for Lightweight Image Matching". In: *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. 2024.

[32] Christopher Rawles et al. *Android in the Wild: A Large-Scale Dataset for Android Device Control*. 2023. arXiv: 2307.10088 [cs.LG].

[33] Machel Reid et al. *Gemini 1.5: Unlocking Multimodal Understanding across Millions of Tokens of Context*. Apr. 25, 2024. URL: http://arxiv.org/abs/2403.05530. preprint.

[34] Tianlin Shi et al. "World of Bits: An Open-Domain Platform for Web-Based Agents". In: *Proceedings of the 34th International Conference on Machine Learning*. Ed. by Doina Precup and Yee Whye Teh. Vol. 70. Proceedings of Machine Learning Research. PMLR, June 2017, pp. 3135–3144. URL: https://proceedings.mlr.press/v70/shi17a.html.

[35] Chan Hee Song et al. "Llm-planner: Few-shot grounded planning for embodied agents with large language models". In: *Proceedings of the IEEE/CVF International Conference on Computer Vision*. 2023, pp. 2998–3009.

[36] Liangtai Sun et al. *META-GUI: Towards Multi-Modal Conversational Agents on Mobile GUI*. Nov. 24, 2022. URL: http://arxiv.org/abs/2205.11029. preprint.

[37] Weihao Tan et al. *Towards General Computer Control: A Multimodal Agent for Red Dead Redemption II as a Case Study*. 2024. arXiv: 2403.03186 [cs.AI].

[38] Oriol Vinyals et al. "Grandmaster Level in StarCraft II Using Multi-Agent Reinforcement Learning". In: *Nature* 575.7782 (Nov. 2019), pp. 350–354. ISSN: 1476-4687. URL: https://www.nature.com/articles/s41586-019-1724-z.

[39] Guanzhi Wang et al. *Voyager: An Open-Ended Embodied Agent with Large Language Models*. May 25, 2023. URL: http://arxiv.org/abs/2305.16291. preprint.

[40] Junyang Wang et al. *Mobile-Agent-v2: Mobile Device Operation Assistant with Effective Navigation via Multi-Agent Collaboration*. 2024. arXiv: 2406.01014 [cs.CL]. URL: https://arxiv.org/abs/2406.01014.

[41] Junyang Wang et al. *Mobile-Agent: Autonomous Multi-Modal Mobile Device Agent with Visual Perception*. arXiv.org. Jan. 29, 2024. URL: https://arxiv.org/abs/2401.16158v2.

[42] Lei Wang et al. "A survey on large language model based autonomous agents". In: *Frontiers of Computer Science* 18.6 (2024), pp. 1–26.

[43] Luyuan Wang et al. *MobileAgentBench: An Efficient and User-Friendly Benchmark for Mobile LLM Agents*. 2024. arXiv: 2406.08184 [cs.AI]. URL: https://arxiv.org/abs/2406.08184.

[44] Zhiyong Wu et al. *OS-Copilot: Towards Generalist Computer Agents with Self-Improvement*. arXiv.org. Feb. 12, 2024. URL: https://arxiv.org/abs/2402.07456v2.

[45] Zhiheng Xi et al. *The Rise and Potential of Large Language Model Based Agents: A Survey*. arXiv.org. Sept. 14, 2023. URL: https://arxiv.org/abs/2309.07864v3.

[46] Tianbao Xie et al. *OSWorld: Benchmarking Multimodal Agents for Open-Ended Tasks in Real Computer Environments*. Apr. 11, 2024. URL: http://arxiv.org/abs/2404.07972. preprint.

[47] Mingzhe Xing et al. *Understanding the Weakness of Large Language Model Agents within a Complex Android Environment*. Feb. 9, 2024. URL: http://arxiv.org/abs/2402.06596. preprint.

[48] Jianwei Yang et al. *Set-of-Mark Prompting Unleashes Extraordinary Visual Grounding in GPT-4V*. Nov. 6, 2023. URL: http://arxiv.org/abs/2310.11441. preprint.

[49] John Yang et al. *InterCode: Standardizing and Benchmarking Interactive Coding with Execution Feedback*. Oct. 30, 2023. URL: http://arxiv.org/abs/2306.14898. preprint.

[50] Shunyu Yao et al. "Webshop: Towards scalable real-world web interaction with grounded language agents". In: *Advances in Neural Information Processing Systems* 35 (2022), pp. 20744–20757.

[51] Chaoyun Zhang et al. *UFO: A UI-Focused Agent for Windows OS Interaction*. Mar. 1, 2024. URL: http://arxiv.org/abs/2402.07939. preprint.

[52] Chi Zhang et al. *AppAgent: Multimodal Agents as Smartphone Users*. Dec. 21, 2023. URL: http://arxiv.org/abs/2312.13771. preprint.

[53] Jiwen Zhang et al. *Android in the Zoo: Chain-of-Action-Thought for GUI Agents*. 2024. arXiv: 2403.02713 [cs.CL].

[54] Shun Zhang et al. "Planning with large language models for code generation". In: *arXiv preprint arXiv:2303.05510* (2023).

[55] Longtao Zheng et al. "Synapse: Trajectory-as-Exemplar Prompting with Memory for Computer Control". In: *The Twelfth International Conference on Learning Representations*. 2024. URL: https://openreview.net/forum?id=Pc8AU1aF5e.

[56] Shuyan Zhou et al. *WebArena: A Realistic Web Environment for Building Autonomous Agents*. Oct. 24, 2023. URL: http://arxiv.org/abs/2307.13854. preprint.

[57] Kaijie Zhu et al. "DyVal: Dynamic Evaluation of Large Language Models for Reasoning Tasks". In: *The Twelfth International Conference on Learning Representations*. 2024. URL: https://openreview.net/forum?id=gjfOL9z5Xr.

A Benchmark Detail

Section A.1 shows the system design and implementation strategies of environments and evaluators. Seciton A.2 is the crab framework implementation details at code level. Section A.3 describes the our experiment settings in detail. Section A.4 describes the specific data format defined in our framework. Fig. 3 shows the structure of modules inside CRAB Benchmark-v0.

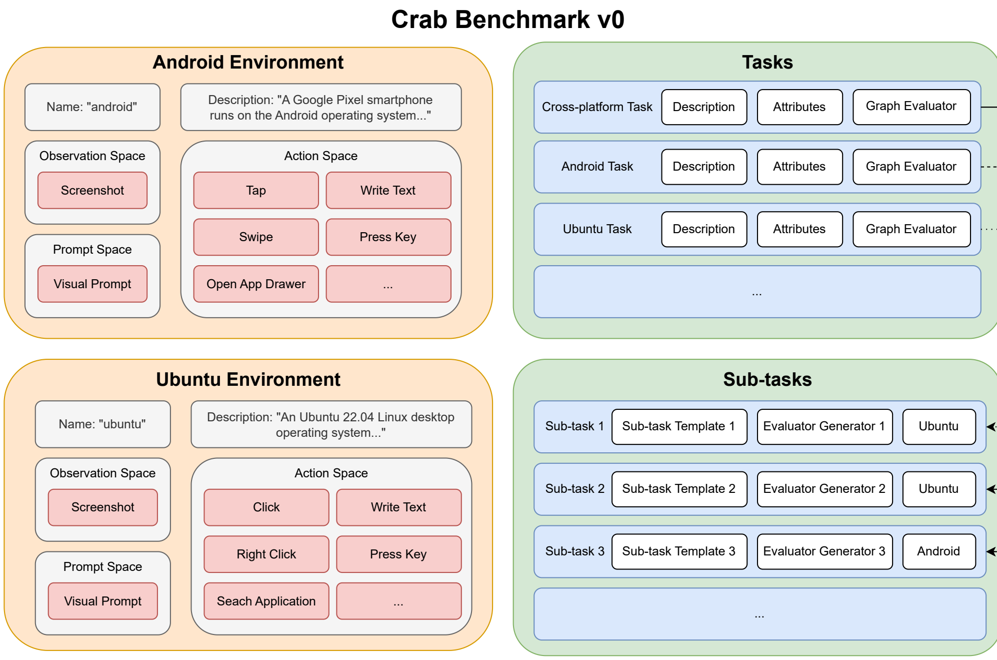


Figure 3: **Module Structure of CRAB Benchmark-v0.** The benchmark is divided into two primary sections: the left section, highlighted with warm hues, features two environments, while the right section, accentuated with cool hues, outlines various tasks. Each environment is defined by attributes including name, description, observation space, prompt method, and action space. Blocks marked in red denote actions. As for the tasks, they are composed of multiple sub-tasks and formulated by combine multiple evaluator sub-graphs derived from the sub-task evaluator generators. Arrows illustrate the compositional relationships between tasks and sub-tasks.

A.1 Overview

The Ubuntu environment is launched on a QEMU/KVM [3, 14] Virtual Machine, and the Android environment employs the Google Android Emulator[2]. Both environments utilize snapshots to ensure a consistent state across all sessions. This allows each experiment to start from an identical state, providing a controlled setup for all test agents. Interaction with the Ubuntu environment is facilitated using PyAutoGUI[3] and MSS[4], which provide high-level commands for mouse and keyboard control and screen capture, respectively. For the Android environment, we use the Android Debug Bridge (ADB)[5].

Observation Space The observation space consists solely of the current system screen for both environments, captured in image format at each step of the agent's interaction. We employ the

[2] `https://developer.android.com/studio/run/emulator`
[3] `https://github.com/asweigart/pyautogui`
[4] `https://github.com/BoboTiG/python-mss`
[5] `https://developer.android.com/tools/adb`

Set-of-Marks visual prompt method [48] to label each interactive element on the screen. Interactive elements are identified using the GroundingDINO [22] with `icon.logo.` text prompt to locate all interactive icons. Additionally, Optical Character Recognition (OCR) is utilized through EasyOCR[6] to detect and label interactive text elements. Each detected item is assigned a unique integer ID, facilitating reference within the action space.

Action Space The action spaces for Ubuntu and Android are distinct and designed to be close to the common interactions in the real devices. For Ubuntu, we define the following actions: mouse-based actions, keyboard-based actions and a shortcut action to search for applications. For Android, the action set includes tapping actions, a text action, a physical button action, and an action to open the app drawer. Additionally, we introduce two environment-irrelevant actions: completing the task and submitting an answer. Detailed descriptions for all actions are shown in Table 3.

Table 3: **Action space of CRAB Benchmark-v0.** The actions at the top of the table apply to the Ubuntu environment, those in the middle to the Android environment, and those at the bottom are relevant across all environments.

Action Name (Parameters)	**Description**
`click(elem)`	Click on `elem`.
`right_click(elem)`	Right-click on `elem`.
`write_text(text)`	Typing the specified `text`.
`press(key)`	Press a keyboard `key`.
`hotkey(keys)`	Press keyboard `keys` at the same time.
`scroll(direction)`	Scrolls page up or down.
`search_app(name)`	Search for application with `name` in the system.
`tap(elem)`	Tap on `elem`.
`long_tap(elem)`	Press and hold `elem`.
`swipe(elem,dire,dist)`	Swipe from `elem` in a specified `direction` and `distance`.
`write_text(text)`	Typing the specified `text`.
`press(key)`	Press a `key`, can be *home* or *back*.
`show_all_drawer()`	Show the app drawer to list installed applications.
`submit(answer)`	Submit `answer` if needed.
`complete()`	Tell system a task is completed.

Evaluator Design To assess the intermediate states of sub-tasks as described in Sec. 4.2, we have implemented a comprehensive suite of execution-based reward functions (evaluators) [46]. These evaluators retrieve and assess specific current states, such as the edited content of a file or a modified setting, thereby determining the successful completion of a sub-task. For each evaluator, input attributes are carefully selected to interpret software information or system settings relevant to the scenario defined for the sub-task. For instance, evaluators use file paths before and after edits as input parameters to verify the completion of file editing sub-tasks. Specifically, for sub-tasks on the Android platform, we incorporate XML-based evaluators [47]. We dump UI layout as XML path and verify whether the UI content matches the expected state. For the Ubuntu platform, we employ image matching techniques [31, 11, 7] and OCR to handle scenarios where acquiring necessary state information through conventional APIs is challenging. Image matching offers fine-grained visual correspondences by comparing keypoint features between images, allowing us to assess spatial relationships among visual elements. Using OCR and image matching, we can accurately evaluate tasks such as verifying whether an agent has successfully created a slide with specified images, text content, and layouts—tasks for which trivial evaluation methods are lacking. We utilize EasyOCR[6] and XFeat[7] as our primary tools for OCR and image matching. For tasks with real-time characteristics that may change over time, we implement crawler scripts to capture dynamic values at the moment

[6] `https://github.com/JaidedAI/EasyOCR`

[7] `https://github.com/verlab/accelerated_features`

of evaluation. These values are then compared with the results achieved by the agent upon task completion. We have a total of 59 evaluator functions.

A.2 Framework Design

CRAB offers a modular and extensible framework for evaluating agent performance in diverse tasks. At the heart of the framework lies the *action*, a unit operation representing the fundamental operation within the benchmark. The *action* is essentially an executable Python function that can be defined with explicit typed parameters and a clear description. *actions* serve not only as building blocks but also as interfaces through which agents interact with the environment. The *evaluator* is a specialized *action* restricted to returning boolean values, signifying the success or failure of an agent's task. It enhances the *actions* by analyzing the state of the environment and the sequence of *actions* executed by the agent, providing a decisive metric of task accomplishment. Additionally, multiple *evaluators* can be interconnected to form a graph evaluator for complex tasks (Sec. 4.2).

The *benchmark* is a key definition in the framework. A benchmark includes multiple *environments* and cross-environment *tasks*. The *environment* is formed by an action space and an observation space, which are both defined by a list of *actions*, and other essential parameters necessary for its configuration. This composite structure facilitates the execution and monitoring of *actions*, whether on local machines, remote servers, virtual machines, or physical devices networked together. A *task* encapsulates a natural language description and a graph evaluator.

CRAB utilizes Python functions to define all actions and evaluators, embodying a "code as configuration" philosophy. Each function's docstring outlines its description and parameter definitions, which are then presented to the agent as structured prompts. Compared to traditional methods using data interchange formats like JSON or YAML, Python code configurations provide a more structured approach and fits in modern IDE.

By decoupling actions, environments, tasks, and evaluations, CRAB facilitates a plug-and-play architecture that can adapt to various scenarios. Such a system is scalable, maintainable and expandable, allowing researchers and developers to introduce new tasks and environments without restructuring the entire framework. Our implementation uses *networkx* [8] for building graph and *dill* [25] for function serialization in our implementation.

A.3 Configuration Format by Modules

Building on the declarative and modular design of our framework, this section explains the configuration and potential extensibility of each module.

Environment The environments in CRAB are a combination of multiple different uses of actions with some environment metadata, such as name and natural language description. In CRAB Benchmark-v0, we use a computer desktop environment and a smartphone environment both based on virtual machine technology. The computer desktop environment, named *Ubuntu*, is installed from an ISO image of Ubuntu 22.04.4 LTS (Jammy Jellyfish) downloaded from the Ubuntu Official website[8]. Necessary applications such as the LibreOffice suite (Writer, Calc, and Impress) and Slack are installed later via snap and apt, according to the task dataset requirements. The smartphone environment, named *Android*, is installed using pre-defined devices (Google Pixel 8 Pro with release name *R*) provided in Google Android Studio[9]. We install additional required applications such as *Keep Notes*, *Tasks*, and *Docs* from Google Play. The descriptions of the two environments in CRAB Benchmark-v0, which are inserted in the agent prompts, are as follows:

- **Ubuntu**: An Ubuntu 22.04 Linux desktop operating system. The interface displays a current screenshot at each step and primarily supports interaction via mouse and keyboard. You

[8] `https://releases.ubuntu.com/jammy/ubuntu-22.04.4-desktop-amd64.iso`
[9] `https://developer.android.com/studio`

must use searching functionality to open any application in the system. This device includes system-related applications including Terminal, Files, Text Editor, Vim, and Settings. It also features Firefox as the web browser, and the LibreOffice suite—Writer, Calc, and Impress. For communication, Slack is available. The Google account is pre-logged in on Firefox, synchronized with the same account used in the Android environment.

- **Android**: A Google Pixel smartphone runs on the Android operating system. The interface displays a current screenshot at each step and primarily supports interaction through tapping and typing. This device offers a suite of standard applications including Phone, Photos, Camera, Chrome, and Calendar, among others. Access the app drawer to view all installed applications on the device. The Google account is pre-logged in, synchronized with the same account used in the Ubuntu environment.

Action Action implementation in CRAB Benchmark-v0 utilize the dynamic feature of Python. It provides an intuitive method to define actions through Python function. Here is an example of action `search_application` in the Ubuntu environment:

```python
@action
def search_application(name: str) -> None:
    """Search an application name.

    For exmaple, if you want to open an application named "slack",
    you can call search_application(name="slack"). You MUST use this
    action to search for applications.

    Args:
        name: the application name.
    """
    pyautogui.hotkey("win", "a")
    time.sleep(0.5)
    pyautogui.write(name)
    time.sleep(0.5)
```

Listing 1: Define "search_application" action.

We extract key information from the function through the `@action` decorator as following:

- **Name**: The action name serves as the identifier for backend models. It should semantically match the action's behavior to improve the accuracy of the agent in executing the action. The function name is extracted as the action name. In this example, `search_application` is the assigned name.
- **Description**: The description provides a natural language explanation of the action to assist the agent in understanding how to use it. The main body of the function's docstring is used as the description. For example, in this instance, the description outlines the basic usage of the action: *Search an application name*, along with an example of its usage.
- **Parameters**: The parameters are the arguments that the functions accept, offering flexibility for the agent to control the environment. Typically, a set of parameters is defined, each consisting of a name, type, and a natural language description. Parameters are extracted from the function's parameters along with their type annotations. Additionally, parameter descriptions are extracted from the `Args` section in the docstring. In this example, there is only one parameter named `name`, with a type of `str`, and its description is `the application name`.
- **Entry**: The entry represents the implementation of the function, defined within the function body to specify how the action is executed. When the agent invokes the function, the entry is executed with the provided parameters. In this example, we utilize the *pyautogui* package for keyboard control. Initially, it presses a hotkey to enter the application search panel in Ubuntu, then proceeds to type the application name provided by the parameters, finally displaying the search results.

Observation The observation space is represented by a set of actions. These observation actions are designed to be parameter-free and return an observation result. For instance, within the Ubuntu environment, the sole observation action available is the `screenshot` function, defined as follows:

```python
@action
def screenshot() -> str:
    """Capture the current screen as a screenshot."""
    with mss() as sct:
    # Capture raw pixels from the screen
    sct_img = sct.grab(sct.monitors[1])
    # Convert to PNG format
    png = tools.to_png(sct_img.rgb, sct_img.size)
    # Encode to Base64 format for easier transmission
    base64_img = base64.b64encode(png).decode("utf-8")
    return base64_img
```

Listing 2: Define the "screenshot" observation action.

This action captures the screen's current view and encodes it in Base64 format. Additionally, visual prompts are also defined by actions that utilize the output from an observation action as their input, further processing it to generate a visual prompt for the agent.

Evaluator The evaluator in CRAB Benchmark-v0 is crafted to assess the outcome of actions performed by the agent within the environment. The evaluator is defined as an action that outputs a boolean value. An example of an evaluator in the Ubuntu environment is the `check_text_in_current_window_name` function, outlined below:

```python
@evaluator(env_name="ubuntu")
def check_text_in_current_window_name(text: str) -> bool:
    try:
        out = subprocess.check_output(
            ["xdotool", "getwindowfocus", "getwindowname"], text=True
        ).strip()
    except subprocess.CalledProcessError:
        return False
    return text in out
```

Listing 3: Define "check_text_in_current_window_name" evaluator.

The evaluator function is denoted with an `@evaluator` decorator and specifies its operating environment. The function's primary role is to execute a check within the system and return a boolean value indicating success or failure based on the condition being evaluated. Here, the function aims to verify whether a specified text appears in the title of the currently focused window. This is achieved through the use of the `subprocess` module to execute system commands that fetch the window's title, checking if the provided text parameter is contained within it.

Task Following a declarative programming paradigm, the task is defined as a data model. Here is an example of a cross-platform task in the dataset:

```python
Task(
    id="a3476778-e512-40ca-b1c0-d7aab0c7f18b",
    description="Open \"Tasks\" app on Android, check the...",
    evaluator=path_graph(
        check_current_package_name("com.google.android.apps.tasks"),
        check_current_window_process("gnome-control-center"),
        check_color_scheme("prefer-dark"),
    ),
)
```

Listing 4: Define a task.

In this model, each task is represented as an instance of the `Task` class, which is a subclass of `BaseModel` in *Pydantic*[10] package. Each task is uniquely identified by an ID and described by a detailed description. The evaluator component is structured as a graph evaluator, which integrates multiple evaluative functions into a directed graph using the *networkx*[11] package. Each evaluator within this graph must be appropriately parameterized to assess specific conditions relevant to the task. For example, the task demonstrated aims to open the "Tasks" app on Android and perform a series of verifications: it checks whether the correct Android app is opened, whether the current focused window's process name is `gnome-control-center`, and whether the color scheme is set to dark.

Sub-task The sub-task in CRAB is the unit component of in task construction. The following example is a sub-task template that we used to easily generate sub-tasks:

```
SubTask(
    id="0f589bf9-9b26-4581-8b78-2961b115ab49",
    description="Open \"{file_path}\" using vim in a terminal, write
    \"{content}\", then save and exit vim.",
    attribute_dict={"file_path": "file_path", "content": "message"},
    output_type="file_path",
    evaluator_generator=lambda file_path, content: path_graph(
        check_current_window_process("gnome-terminal-server"),
        is_process_open("vim"),
        is_process_close("vim"),
        check_file_content(file_path, content),
    ),
),
```

Listing 5: Define a task.

In this sub-task model, each sub-task is defined using a similar approach to the main task. The attributes of the sub-task are outlined in an `attribute_dict`, which details the types and roles of each attribute used in the sub-task's operations. The `output_type` field specifies the expected type of output from the sub-task. The types reflected in `attribute_dict` and `output_type`, play a critical role in determining the compatibility and sequential logic of compose multiple sub-tasks. The evaluator for the sub-task is dynamically generated using a lambda function, which crafts an evaluator sub-graph based on the sub-task's attributes.

A.4 Task Dataset

We use a JSON format to save the composed tasks, which includes the task ID, overall task description, sub-tasks with their attribute values, and a graph structure represented in an adjacency list. The entire task dataset is defined by the sub-task pool in Python code and the task composition JSON files categorized by task platform.

B Agent system

B.1 Agent Implementation

In this section, we outline the implementation of the agents used in our experiments, which leverage advanced multimodal language models from OpenAI, Anthropic, and Google. Each agent is designed to function in multi-environment setups, interacting with various action spaces defined by different environments.

General Framework All agents share a common architecture but are tailored to the specific APIs and capabilities of each language model provider.

[10] `https://pydantic.dev/`
[11] `https://networkx.org/`

Initialization Each agent is initialized with several key parameters, including a description, an action space, the model type, maximum tokens, history message length, and an optional environment description. The initialization process involves:

- **Action Space Conversion**: Actions defined for each environment are converted into a schema compatible with the respective API. This ensures that the actions can be correctly interpreted and executed by the language models.
- **System Message Setup**: Depending on whether the agent is configured for single or multiple environments, a system message is formatted to provide the model with context about the tasks and environments.

Interaction (Chat Method) The core functionality of each agent is encapsulated in its ability to interact with users through a chat method. This involves:

- **Content Parsing**: Input content is parsed and formatted to match the requirements of the respective API. This includes structuring user messages and any necessary contextual information.
- **Request Construction**: The request payload is constructed, incorporating the system message, chat history, and the newly parsed user input.
- **API Communication**: The constructed request is sent to the appropriate API, which generates a response. The agents handle API-specific constraints such as rate limits and response formats.
- **Response Handling**: The response from the API is processed to extract any tool calls suggested by the model. These are then appended to the chat history, maintaining a coherent conversation state.

Multi-Environment Support For agents configured to operate in multiple environments, additional logic ensures that actions are correctly associated with their respective environments. This involves modifying action names and descriptions to reflect their environmental context and handling responses accordingly.

Utilities and Shared Functions Several utility functions support the operation of these agents, facilitating tasks such as content parsing, action prompt generation, and schema conversion. These shared functions ensure consistency and reduce redundancy across the different agent implementations.

B.2 Inter-agent Communication Strategies

In this section we introduce the details of two multi-agent communications methods, which are introduced in 5.2.

Multi-agent Communication by Functionality This setting involves two agents: a main agent prompted with the task description and a tool agent with the entire action space. The main agent generates the instruction for the next step and sends it to the tool agent. The tool agent chooses the proper action with parameters and a target environment, then feeds it back to the system.

Multi-agent Communication by Environment This setting involves four agents in our benchmark setting: a main agent prompted with the task description and three tool agents, each corresponding to the environments of Android, Ubuntu, and Root, with the respective action spaces. The main agent generates the instruction for the next step and sends it to the tool agents. Each sub-environment agent receives the message containing the instruction and environment observation information. The environment agents process the message using their specialized models and action schemas, performing the required actions within their environments.

B.3 Agent Prompt

B.3.1 Single Agent

Prompt

You are a helpful assistant. Now you have to do a task as described below: `{task_description}`. And this is the description of each given environment: `{env_description}`. A unit operation you can perform is called action in a given environment. For each environment, you are given a limited action space as function calls:
`{action_descriptions}`
You may receive a screenshot of the current system. The interactive UI elements on the screenshot are labeled with numeric tags starting from 1. For each step, You must state what actions to take, what the parameters are, and you MUST provide in which environment to perform these actions. Your answer must be a least one function call. please do not output any other information. You must make sure all function calls get their required parameters.

B.3.2 Multi-Agent by Functionality

Main Agent Prompt

You are a helpful assistant. Now you have to do a task as described below: `{task_description}`. And this is the description of each given environment: `{env_description}`. A unit operation you can perform is called action in a given environment. For each environment, you are given a limited action space as function calls:
`{action_descriptions}`
You may receive a screenshot of the current system. The interactive UI elements on the screenshot are labeled with numeric tags starting from 1. For each step, You must state what actions to take, what the parameters are, and you MUST provide in which environment to perform these actions.

Tool Agent Prompt

You are a helpful assistant in generating function calls. I will give you a detailed description of what actions to take next, you should translate it into function calls. please do not output any other information.

B.3.3 Multi-Agent by Environment

Main Agent Prompt

You are a main agent, and your goal is to plan and give instructions to sub-agents in each environment to complete the final task. Now you have to do a task as described below: `{description}`. The description of each given environment: `{env_description}`. For each step, you are required to provide high-level instructions detailing the next actions to be taken. Additionally, you must specify which sub-agent in the designated environment should execute these instructions. If a sub-agent is not needed for a particular step, you may instruct it to skip that step.

Root Environment Agent Prompt

You are a sub-agent responsible for the crab benchmark root environment. Your goal is to assist the main agent in completing the whole task: `"{description}"`. You can only complete the task or submit the result when the main agent tells you the whole task has been completed. Otherwise, you can only call SKIP.

Table 4: **Evaluation results on Ubuntu tasks.**

Agent system		Metrics				Termination Reason		
Model	**Structure**	**SR(%) ↑**	**CR(%) ↑**	**EE(%) ↑**	**CE(%) ↑**	**FC(%)**	**RSL(%)**	**IA(%)**
GPT-4o	Single	10.34	26.64	2.72	4.68×10^{-4}	5.17	60.34	24.14
GPT-4o	By Func	6.90	21.90	2.07	3.86×10^{-4}	6.90	60.34	25.86
GPT-4o	By Env	8.62	20.60	2.06	2.01×10^{-4}	3.45	48.28	39.66
GPT-4 Turbo	Single	**12.07**	**28.36**	**3.82**	$\mathbf{8.79 \times 10^{-4}}$	1.72	63.79	22.41
GPT-4 Turbo	By Func	10.34	24.45	3.10	4.74×10^{-4}	8.62	34.48	46.55
Gemini 1.5 Pro	Single	1.72	7.61	0.54	\	0.00	46.55	51.72
Gemini 1.5 Pro	By Func	1.72	3.30	0.30	\	0.00	20.69	77.59
Claude 3 Opus	Single	1.72	9.54	1.41	3.42×10^{-4}	5.17	56.90	36.21
Claude 3 Opus	By Func	1.72	6.75	0.65	2.81×10^{-4}	27.59	31.03	39.66

Table 5: **Evaluation results on Android tasks.**

Agent system		Metrics				Termination Reason		
Model	**Structure**	**SR(%) ↑**	**CR(%) ↑**	**EE(%) ↑**	**CE(%) ↑**	**FC(%)**	**RSL(%)**	**IA(%)**
GPT-4o	Single	24.14	47.91	5.12	7.17×10^{-4}	13.79	58.62	3.45
GPT-4o	By Func	24.14	48.74	5.77	$\mathbf{9.19 \times 10^{-4}}$	24.14	37.93	13.79
GPT-4o	By Env	**27.59**	**53.34**	**5.93**	4.58×10^{-4}	13.79	44.83	13.79
GPT-4 Turbo	Single	10.34	30.53	2.84	3.36×10^{-4}	20.69	62.07	6.90
GPT-4 Turbo	By Func	20.69	37.01	4.32	5.92×10^{-4}	13.79	51.72	13.79
Gemini 1.5 Pro	Single	17.24	34.52	4.09	\	10.34	65.52	6.90
Gemini 1.5 Pro	By Func	17.24	35.99	3.88	\	31.03	41.38	10.34
Claude 3 Opus	Single	17.24	43.62	5.30	7.78×10^{-4}	13.79	51.72	17.24
Claude 3 Opus	By Func	13.79	42.30	4.20	5.07×10^{-4}	44.83	31.03	10.34

Sub-environment Agent Prompt

You are a sub-agent responsible for the `{environment}` environment. The description of the `{environment}` environment is: `{env_description}`. Your goal is to assist the main agent in completing the final task by performing actions in the `{environment}` environment according to the instructions from the main agent. The final task is described below: `{task_description}`. A unit operation you can perform is called action in a given environment. You can only execute action in the `{environment}` environment. For the `{environment}` environment, you are given a limited action space as function calls:
`{action_descriptions}`
The interactive UI elements on the screenshot are labeled with numeric tags starting from 1. For each step, You will receive an instruction telling you what you need to do next. After analyzing the instruction you received and the current `{environment}` system, if you think you don't need to do anything in the current `{environment}` system, you should choose SKIP action. Otherwise, you must state what actions to take, what the parameters are, and you MUST provide in which environment to perform these actions. Your answer must be function calls. Please do not output any other information. You must make sure all function calls get their required parameters.

C Further Result Analysis

This section further discusses our experimental results in detail. Section C.1 categorizes the results into three types of tasks: Ubuntu, Android, and cross-platform, and provides further analysis. Section C.3 examines three specific tasks and analyzes the performance of different agent settings on each.

C.1 Result by Platforms

Table 4, 5 and 6 show the experiment results on Ubuntu Tasks, Android Tasks, and cross-platform Tasks, respectively.

We find that certain models demonstrate a distinct preference or better alignment with specific platforms. The GPT-4o, Gemini, and Claude models, for instance, show notably better outcomes on Android platforms. This suggests potential optimizations or intrinsic features within these models that cater effectively to the Android environment's requirements. Conversely, the GPT-4 Turbo model exhibits superior performance on Ubuntu tasks, hinting at possible architectural or training aspects that are better suited for that specific environment.

In multi-agent system organized by environment, consistently yields better results in both Android and cross-platform tasks. This configuration appears to enhance the agents' ability to manage and adapt to diverse tasks more effectively, leveraging environmental specifics to optimize performance. This suggests that employing multiple agents that are either specialized or specifically configured to operate within the same environment can significantly improve task handling and overall adaptability.

Cross-platform tasks present a greater challenge for all models, as evidenced by lower Success Rates and Completion Ratios. These tasks, which necessitate functionality across different operating systems or platforms, demand a broader capability range and more sophisticated agent coordination. The importance of CR is especially critical in such environments, where it serves as a more reliable metric for distinguishing between agent models than SR. Given the presence of all Gemini and Claude agents' SR is 0.0, indicating that Completion Ratio more effectively captures an agent model's capability, thereby better reflecting its robustness and adaptability to complex requirements.

Furthermore, analyzing the reasons for task termination offers additional insights into the operational challenges these models encounter. False Completion is notably prevalent in Android tasks. Reach Step Limit remains the most frequent cause of termination, particularly in cross-platform tasks. The Claude model exhibits a significantly high Invalid Action ratio in cross-platform tasks, indicating its difficulties in managing multi-environment scenarios effectively.

Overall, these findings underscore the necessity of selecting the appropriate agent model and configuration based on specific platform and task needs. The variability in model performance across different setups also highlights the ongoing need for development and refinement of multi-agent systems to enhance their versatility and efficacy in increasingly diverse and complex operational environments.

C.2 Comparison between Single Agent and Multi-agent

The experimental results indicate that multi-agent structures perform slightly worse than single-agent systems, which is somewhat unusual. We analyse the possible reasons here.

First, comparing in False Completion Rate, we attribute the lower Success Rate (SR) of Multi-agent to a high False Completion Rate—where the agent incorrectly assumes that the task is complete. As observed in failure cases (e.g., the Cross-platform Task case study in Appendix C.3), Sub-agents often misinterpret the Main agent's instructions. Despite being required to perform a final action, the instructions lead Sub-agents to prematurely conclude that the task is complete, resulting in incorrect "complete" actions. While this issue also occurs in Multi-Env, it happens less frequently. We believe this is due to information loss during inter-agent communication. Natural language, while effective for aligning with human understanding in LLM communication, is less suited for inter-agent communication, leading to information loss during compression and interpretation, which weakens the performance of multi-agent structures.

Next, comparing in Invalid Action Rate, we observe that in single-platform tasks, both Multi-Env and Multi-Func suffer from similar inter-agent communication issues, as indicated by their high False Completion and Invalid Action rates (Table 4 and 5). However, in cross-platform tasks (Table 6), the Single agent's Invalid Action rate is significantly higher than that of the Multi-agent structures. Cross-

Table 6: **Evaluation results on cross-platform tasks.**

Agent system		Metrics				Termination Reason		
Model	**Structure**	**SR(%) ↑**	**CR(%) ↑**	**EE(%) ↑**	**CE(%) ↑**	**FC(%)**	**RSL(%)**	**IA(%)**
GPT-4o	Single	7.69	45.53	**4.54**	$\mathbf{3.57 \times 10^{-4}}$	0.00	53.85	38.46
GPT-4o	By Func	**15.38**	43.41	3.19	2.25×10^{-4}	7.69	61.54	15.38
GPT-4o	By Env	7.69	**48.61**	3.69	1.68×10^{-4}	15.38	61.54	15.38
GPT-4 Turbo	Single	7.69	47.84	4.31	2.89×10^{-4}	0.00	69.23	23.08
GPT-4 Turbo	By Func	7.69	39.05	3.73	2.51×10^{-4}	15.38	46.15	30.77
Gemini 1.5 Pro	Single	0.00	21.25	1.49	\	0.00	69.23	30.77
Gemini 1.5 Pro	By Func	0.00	16.70	1.52	\	7.69	69.23	23.08
Claude 3 Opus	Single	0.00	24.69	2.33	1.62×10^{-4}	0.00	38.46	61.54
Claude 3 Opus	By Func	0.00	20.07	2.32	1.49×10^{-4}	0.00	38.46	61.54

platform tasks require frequent environment changes with varying action spaces, and if the model's performance output is inadequate, it often generates correct actions in the wrong environment, invalid actions in the correct environment, or correct actions in correct environment but in the wrong format. This phenomenon highlights the limitations of current general-purpose LLMs, where multi-agent structures can be advantageous. By assigning each agent a specific responsibility and a limited action space, multi-agent structures can mitigate these issues.

Last, when comparing different types of tasks, we observe that the Multi-Env structure significantly outperforms the Single Agent in Android and cross-platform tasks but underperforms in Ubuntu tasks. The key difference between the Single Agent and Multi-Env lies in the average context length each agent processes. As demonstrated by Liu et al. [21], more context does not always lead to better performance. The Single Agent is burdened with extensive knowledge across different fields, making it challenging for the model to switch between multiple environments, particularly when managing long history chat messages. In contrast, the sub-agents in the Multi-Env structure handle part of the total prompt, which enhances their performance in more complex tasks. While different backend models show varying performance across environments, resulting in some instability, the general trend is that the more complex the task, the more advantageous the multi-agent approach becomes.

In summary, the performance difference between multi-agent and single-agent structures largely depends on the task complexity. For tasks that are too complex for a single general-purpose agent, a multi-agent structure may perform better. Conversely, for simpler tasks, multi-agent structures tend to cause information loss during inter-agent communication, leading to misunderstandings among downstream agents.

To improve multi-agent system performance, we suggest to follow two approaches: (1) Developing better multi-agent structures to minimize information loss during communication, and (2) Introducing a critical agent to correct hallucinations or information loss during communication. These improvements, however, come with a trade-off, namely an increase in token costs within the agent system. Within our benchmark framework, users can utilize the error log we provide to analyze the bottlenecks of their agents and refine their designs.

C.3 Case Study

To better understand how different agents perform the same task and exhibit varied properties, we present visual results along with detailed metrics and logs for three cases by platform. The screenshots illustrate the progress of agents executing tasks according to specific natural language instructions.

C.3.1 Cross-platform Task

Task: Open the "Tasks" app on an Android device, check the first incomplete task, and then execute it as described. The first task, found incomplete in the "Tasks" app, involves **switching the system to dark mode in Ubuntu via the "Settings" application.**

This task exemplifies message passing across different environments, where the "incomplete task" serves as the critical information that the agent must relay and apply in the Ubuntu setting. These two phases—retrieving the task details via the phone and executing the task on a computer—are inseparably linked and cannot be treated as distinct tasks. The agent can only proceed to the second stage after successfully acquiring information from the first.

In this task, GPT-4o (single agent), GPT-4 Turbo (single agent), and GPT-4 Turbo (multi-agent by functionality) all successfully complete the task using the minimal steps necessary to locate and execute the task, demonstrating their efficiency in managing multiple environments simultaneously. On the other hand, both GPT-4o (multi-agent by functionality) and GPT-4o (multi-agent by environment) also perform commendably, completing the task up until the final step. However, after incorrectly performing the last step, they both erroneously conclude the task is completed and exit. This indicates a communication breakdown, where the sub-agents misinterpret the instructions from the main agent. The remaining four agents fail to complete the task. Agents equipped with the Gemini model do not even manage to open the "Tasks" app within the allocated step limit, whereas agents with the Claude model quickly open the "Tasks" app to complete the first step but fail at the task execution. The performance disparity between single-agent and multi-agent configurations in both the Gemini and Claude models highlights the variance in capability across different models and devices.

C.3.2 Ubuntu Task

Task: Create a new directory "/home/crab/assets_copy" and copy all files with the specified "txt" extension from "/home/crab/assets" to the directory "/home/crab/assets_copy".

This task can be approached through multiple methods. An agent may opt for a straightforward strategy first using the `search_application` command to find the Terminal, then using Linux commands to create the directory and copy the necessary files. Alternatively, the agent could employ a GUI-based approach, manually creating the folder and selecting files through actions like `click` and `right_click`. We evaluate various agent systems in a single-agent setting for this task. As illustrated in Table 7–10 , both GPT-4o and GPT-4 Turbo from OpenAI successfully interpret the task instructions and employ a simpler solution using Terminal commands. These agents also demonstrate superior capability in understanding the UI, selecting the correct commands, and accurately using the Terminal application to fulfill the task requirements.

Conversely, the Gemini and Claude agents, despite attempting to solve the task with Terminal, ultimately fail in different ways. Both agents struggle with precise clicking and selecting the correct icons for the intended actions, even though they share the same visual prompting mechanism as GPT-4o and GPT-4 Turbo. For instance, the Claude agent mistakenly opens the Ubuntu Desktop Guide instead of the Terminal and continues executing commands in the wrong application without realizing the error. The Gemini agent, on the other hand, unexpectedly opens the Firefox browser before correctly navigating to the Terminal but still interacts incorrectly with unrelated applications and icons. Unlike Claude, Gemini does not type in commands in the wrong applications but persists in exploring alternative methods using the Files application's UI. Despite taking significantly more steps than the GPT-4o and GPT-4 Turbo agents, neither the Claude nor the Gemini agents achieve the task's goal.

C.3.3 Android Task

Task: In Android, using the "Contacts" app, find the email of the contact named John Lauphin, then using the "Gmail" app, send an email to that contact with the subject "Hello John."

This task consists of sub-tasks across two different applications. Agents must sequentially open the two apps, retrieve the email address from the first app, and use it in the second app to send an email. This straightforward yet formal task can be completed using various methods. Agents may need to locate the contact in the Contacts app and then use the retrieved email address to send a message. We reports the performance of agents in a multi-agent setting for this challenging task. Following is the details of agents in operating the task.

GPT-4o multi-agent by functionality In steps 1-11, the agent tries to open the Contacts app but mistakenly opens Google Assistant multiple times. In steps 12-14, the agent successfully enters the Contacts app and finds the contact information. The agent then returns to the home page, and the process is terminated due to the limitation of operation steps.

GPT-4 Turbo multi-agent by functionality In steps 1-2, the agent tries to open the Contacts app but mistakenly opens Google Messages. In steps 3-5, the agent opens the Contacts app and obtains the corresponding information. In steps 6-14, the agent repeatedly opens Google Chrome and Messages apps, failing to find the Gmail app as planned.

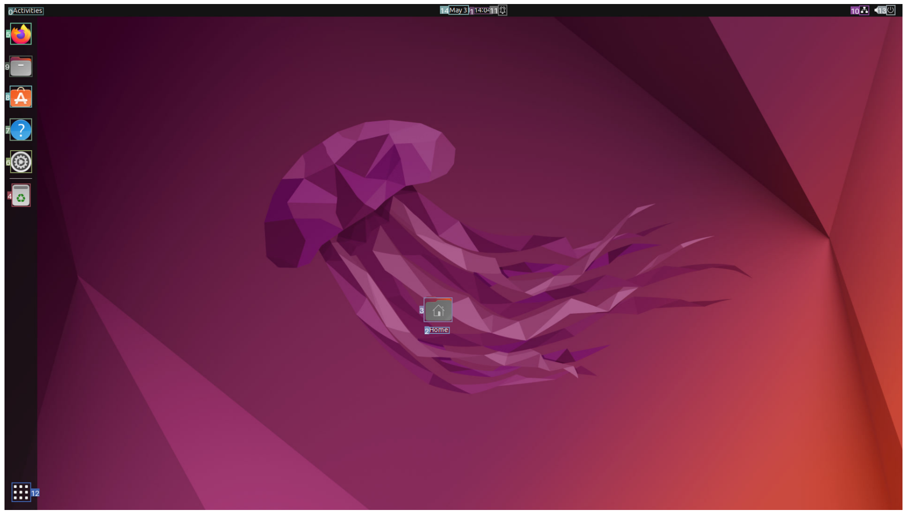
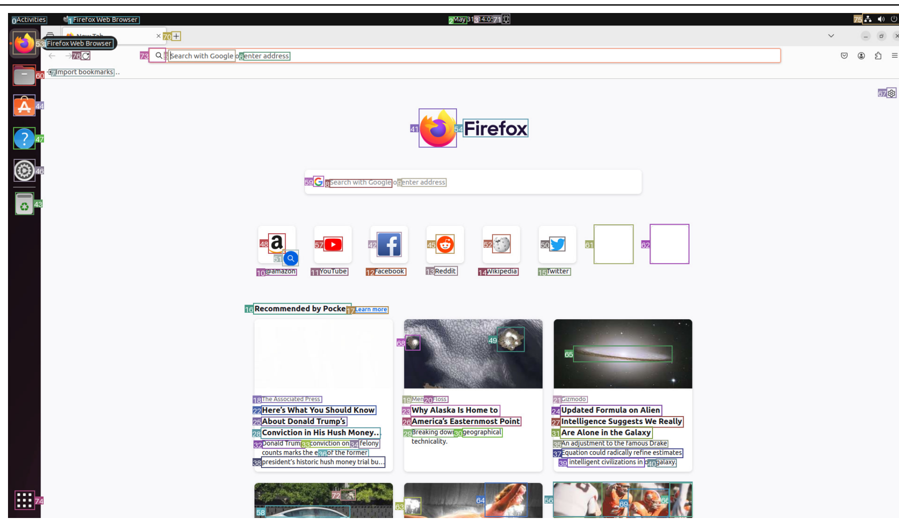
Gemini 1.5 Pro multi-agent by functionality In steps 1-2, the agent finds the Contacts app and enters it. However, the agent misunderstands the instruction, gets lost in creating a new contact with the given name, and cannot obtain the corresponding information.

Claude 3 Opus multi-agent by functionality In steps 1-7, the agent tries to open the Contacts app but mistakenly opens Google Messages multiple times. In steps 7-11, the agent tries to open the Contacts app but mistakenly opens Google Assistant. In steps 12-14, the agent successfully enters the Contacts app and finds the contact information. The agent then returns to the home page, plans to open the Gmail app, and the process is terminated due to the limitation of operation steps.

GPT-4o multi-agent by environment In steps 1-7, the agent plans to open the Contacts app, but the operation fails due to an error in opening the app drawer, which prevents the agent from finding and tapping the Contacts app. In steps 8-11, the agent successfully enters the Contacts app and obtains the information. In steps 12-14, the agent opens the Gmail app, navigates to the sending page, and tries to input the retrieved email address as the recipient.

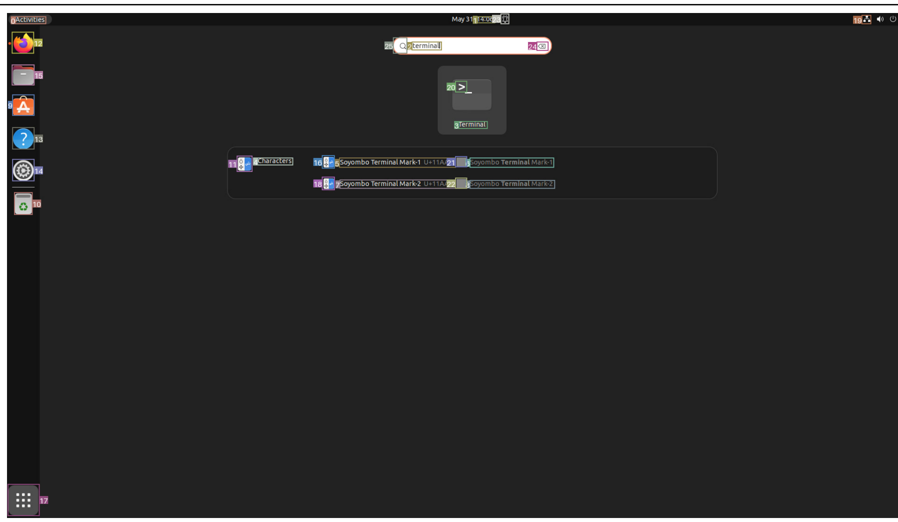
Analysis For the agents which are organized by functionality, Gemini 1.5 Pro struggles to complete the first operation. Although it recognizes and opens the Contacts app as instructed, it fails to proceed further. In contrast, Claude 3 Opus and GPT-4o successfully obtain the necessary information. In the initial phase, the multiple agents agree that opening the Contacts app is the first step. However, they often fail to find the correct position to tap, frequently opening incorrect apps such as Google Assistant and Messages. Once the agents do open the correct app, they usually find the email address of the contact quickly. Even when agents plan to go back home and open the Gmail app to send the message, due to the limitation of operations, the system ended. As shown in steps 3-5, GPT-4 Turbo quickly finishes the corresponding task after opening the correct app. However, similar to GPT-4o, GPT4-Turbo agents get stuck as they can not open the correct apps in the following steps. Besides, GPT-4o (multi-agent by environment) overcomes the issue encountered by GPT-4o (multi-agent by functionality). Even affected by not being able to access the app drawer, the system could still find and copy the corresponding information and change to the Gmail app for further operations.

Table 7: **Ubuntu task case with Gemini (Single):** Create a new directory "/home/crab/assets_copy" and copy all files with the specified "txt" extension from "/home/crab/assets" to the directory "/home/crab/assets_copy".

Step	Agent Observation and Action
0	 `click(5)`
1	 `click(1)`

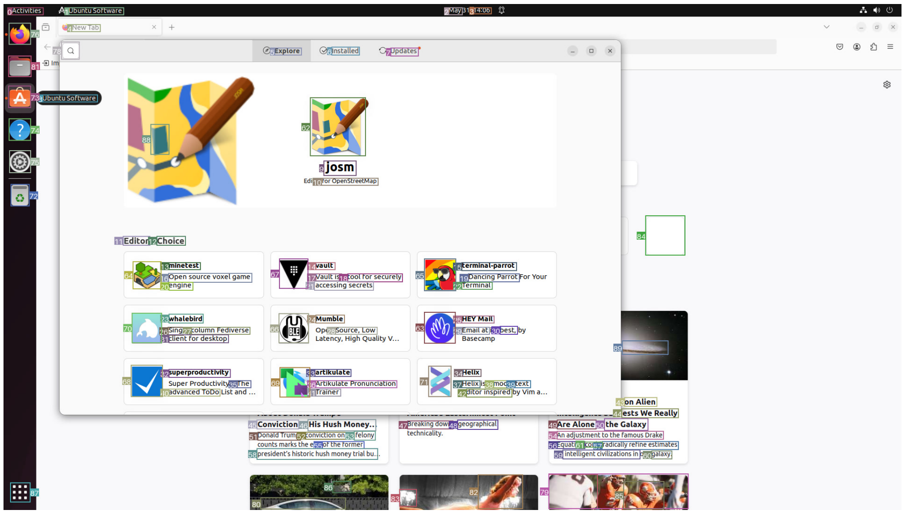
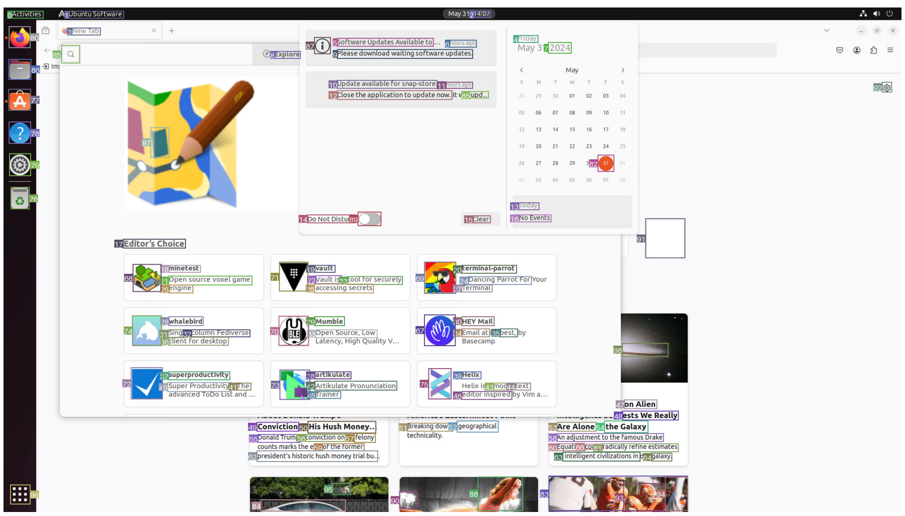
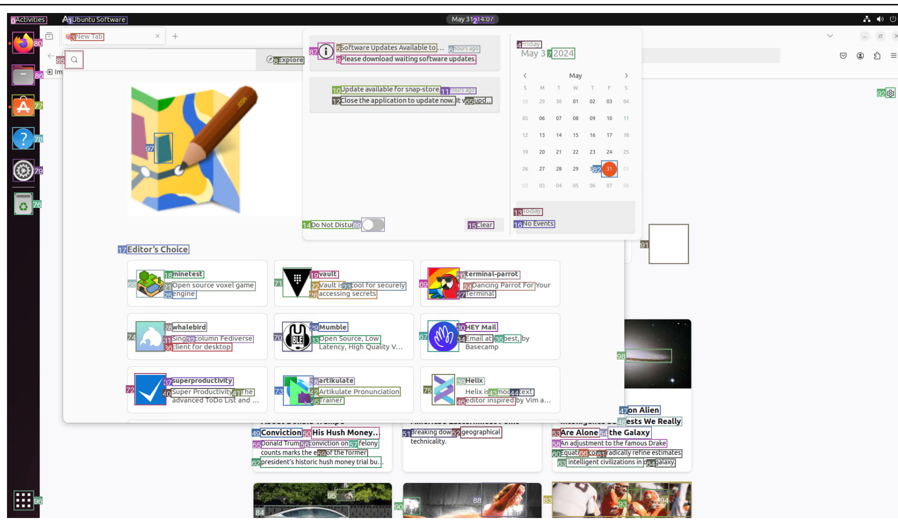
Continued on next page

Table 7 continued from previous page

2	 `click(4)`
3	 `search_application("terminal")`
4	 `click(9)`

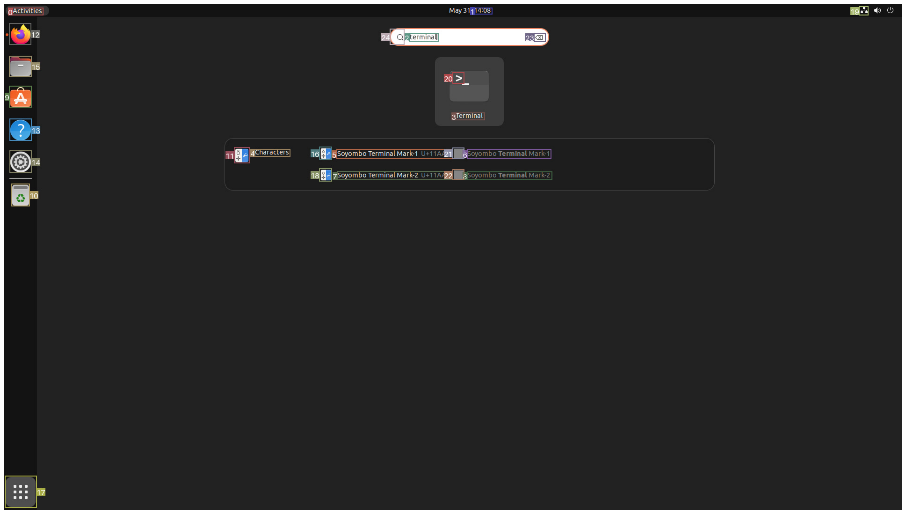
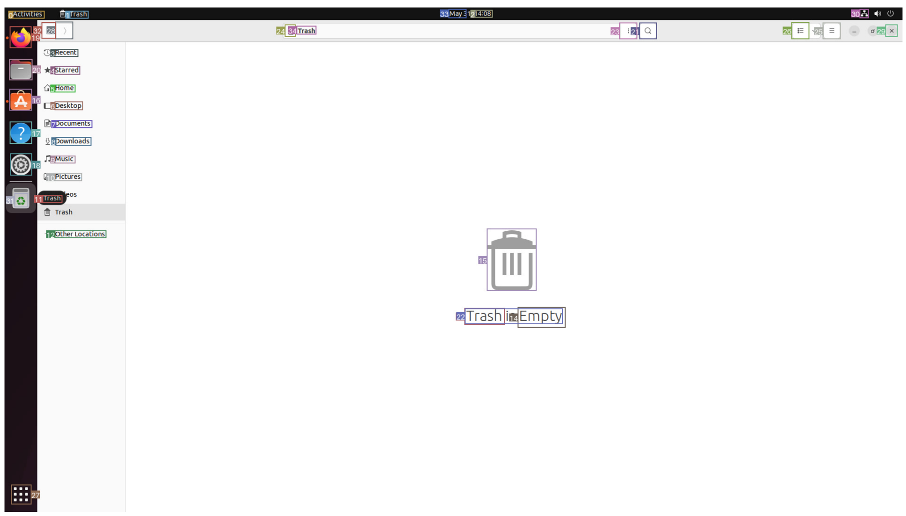
Continued on next page

Table 7 continued from previous page

5	 click(3)
6	 click(13)
7	 search_application("terminal")

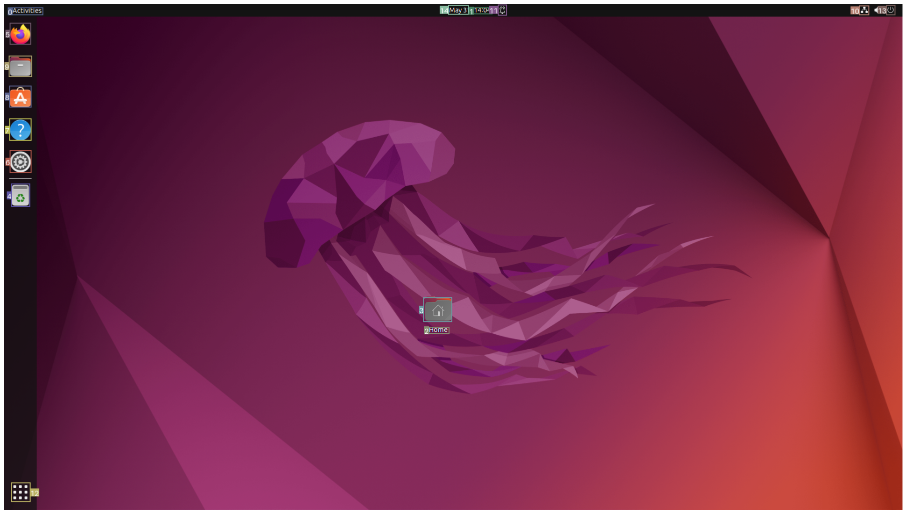
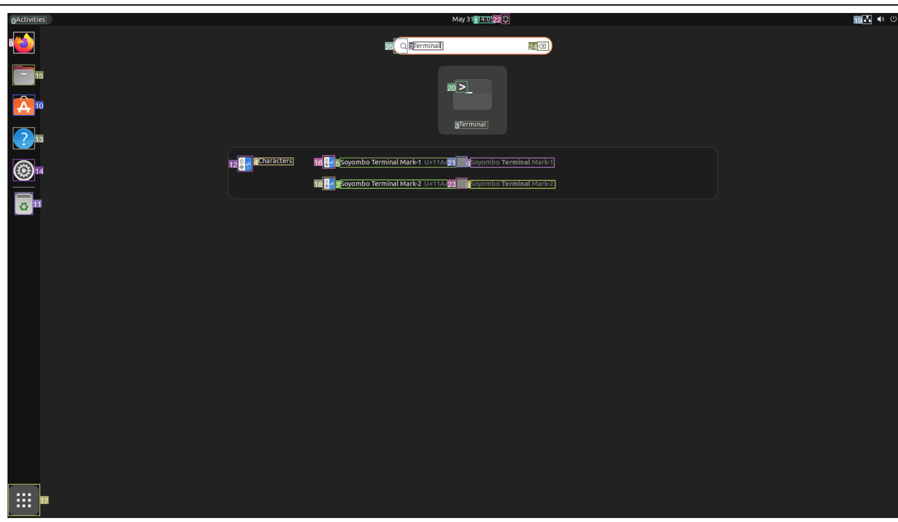
Continued on next page

Table 7 continued from previous page

8	 `click(10)`
9	 `click(6)`
10-14	The agent is stuck at this stage and keeps clicking useless elements.

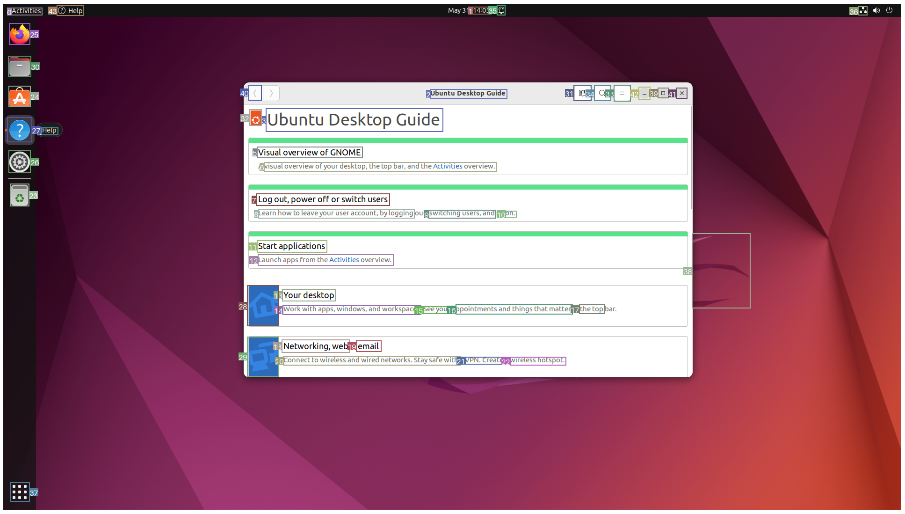
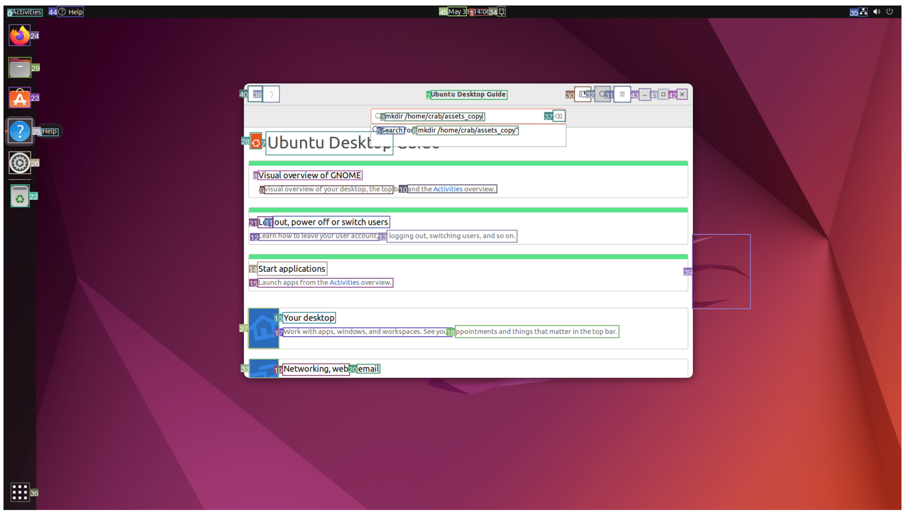
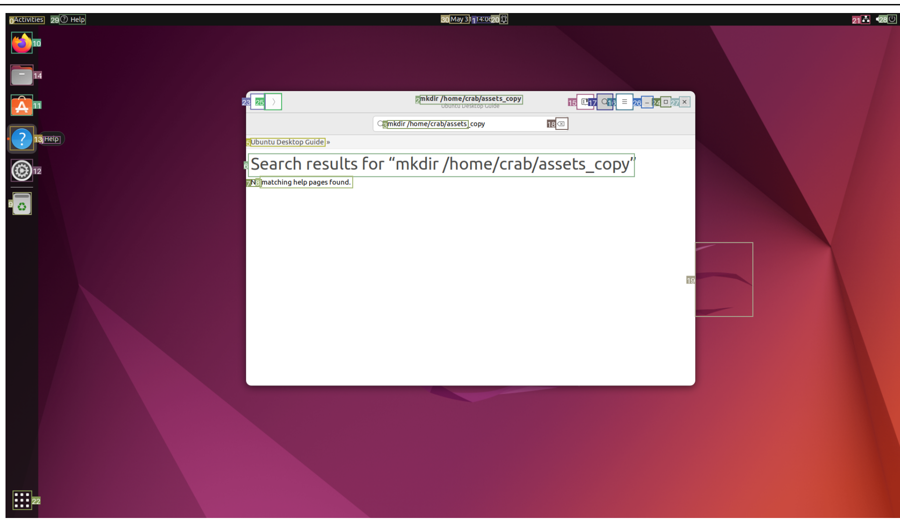
Completed Nodes / Total Nodes: 0 / 2 **Termination Reason**: Reach Step Limit

Table 8: **Ubuntu task case with Claude (Single):** Create a new directory "/home/crab/assets_copy" and copy all files with the specified "txt" extension from "/home/crab/assets" to the directory "/home/crab/assets_copy".

Step	Agent Observation and Action
0	 `search_application("terminal")`
1	 `click(13)`

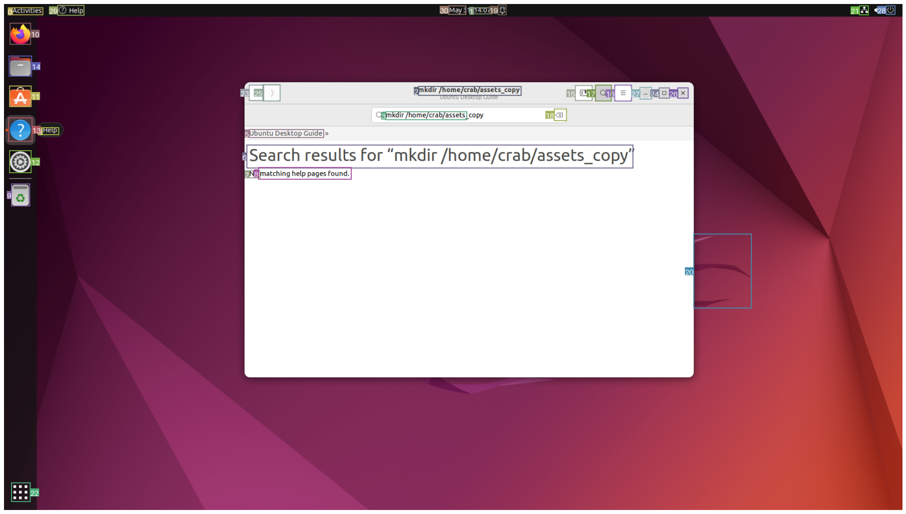
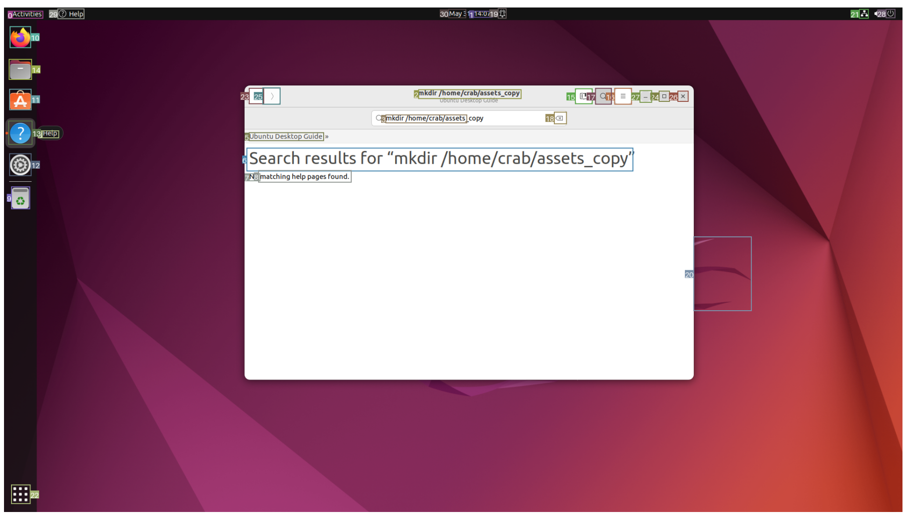
Continued on next page

Table 8 continued from previous page

2	 `write_text("mkdir /home/crab/assets_copy")`
3	 `key_press("enter")`
4	 `write_text("cd /home/crab/assets")`

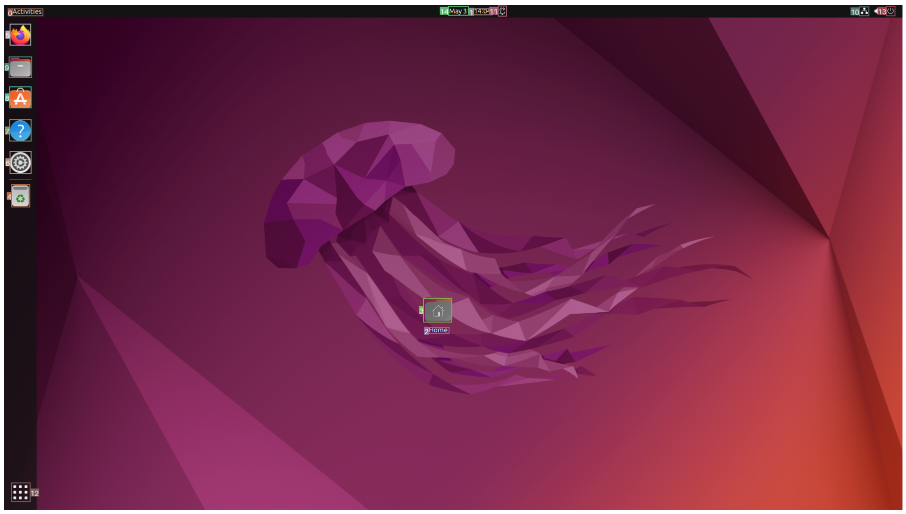
Continued on next page

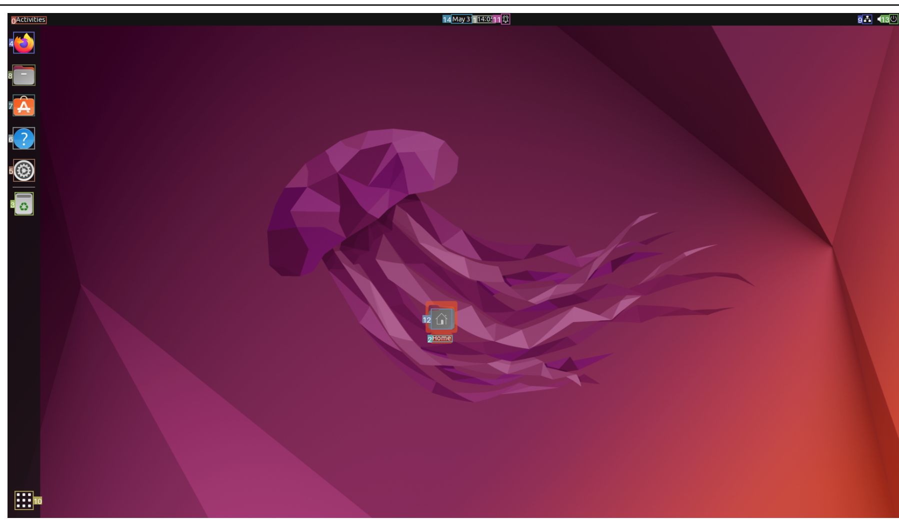
Table 8 continued from previous page

5	 `key_press("enter")`
6	 `write_text("mkdir assets_copy")`
7-14	The agent is stuck at this stage and keeps pressing keys.

Completed Nodes / Total Nodes: 0 / 2 **Termination Reason**: Reach Step Limit

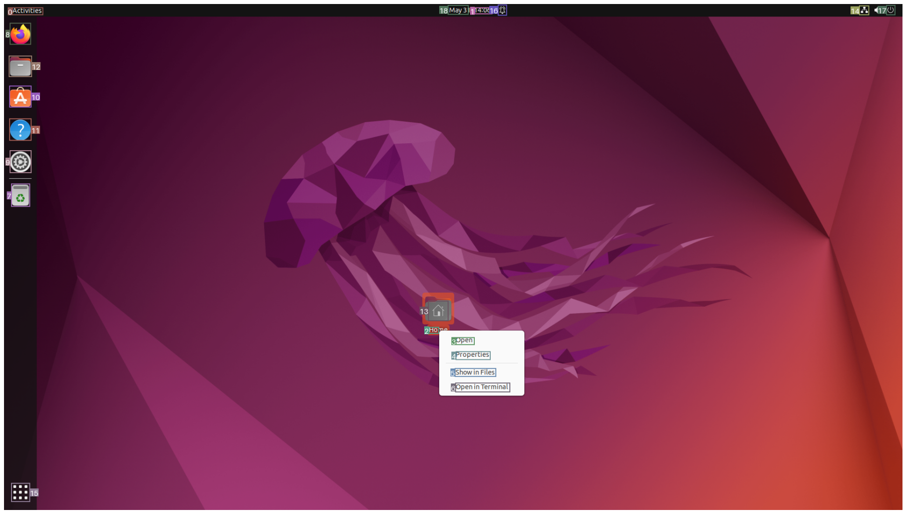
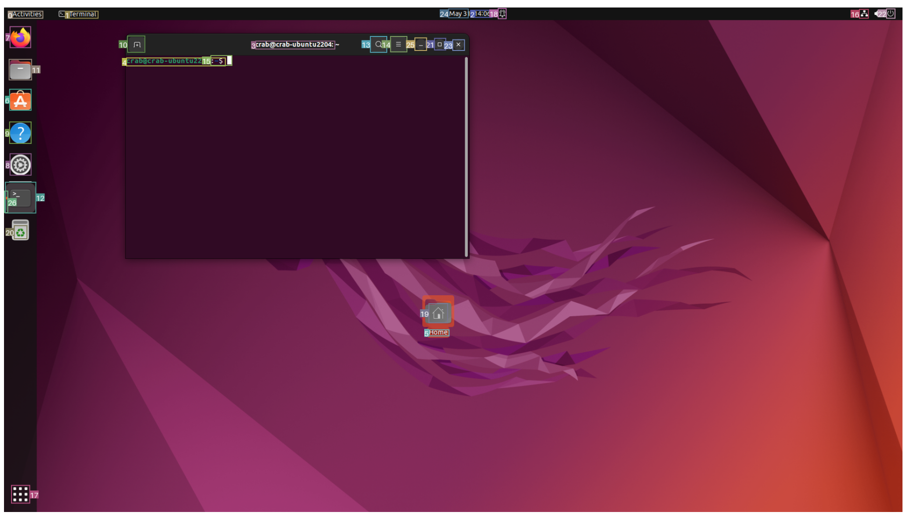
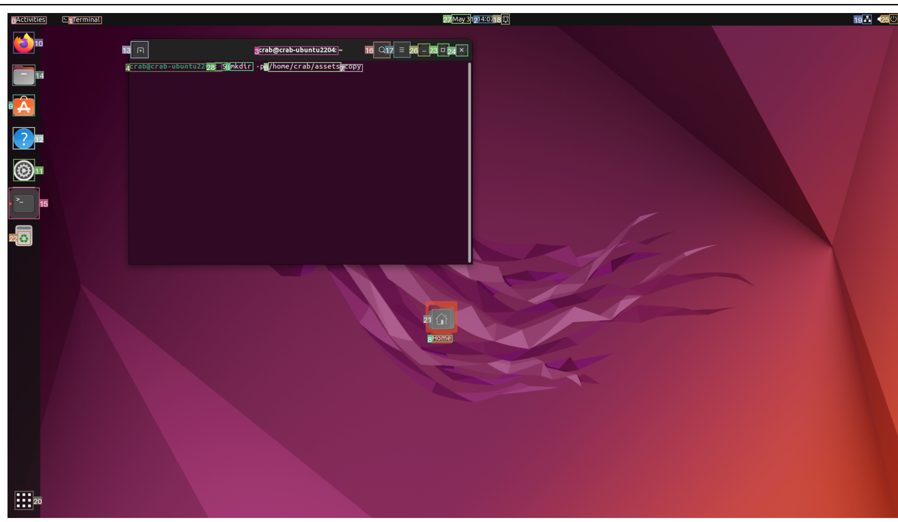
Table 9: **Ubuntu task case with GPT-4o (Single):** Create a new directory "/home/crab/assets_copy" and copy all files with the specified "txt" extension from "/home/crab/assets" to the directory "/home/crab/assets_copy".

Step	Agent Observation and Action
0	 `click(2)`

1-2	
	`click(2)`
	`right_click(2)`

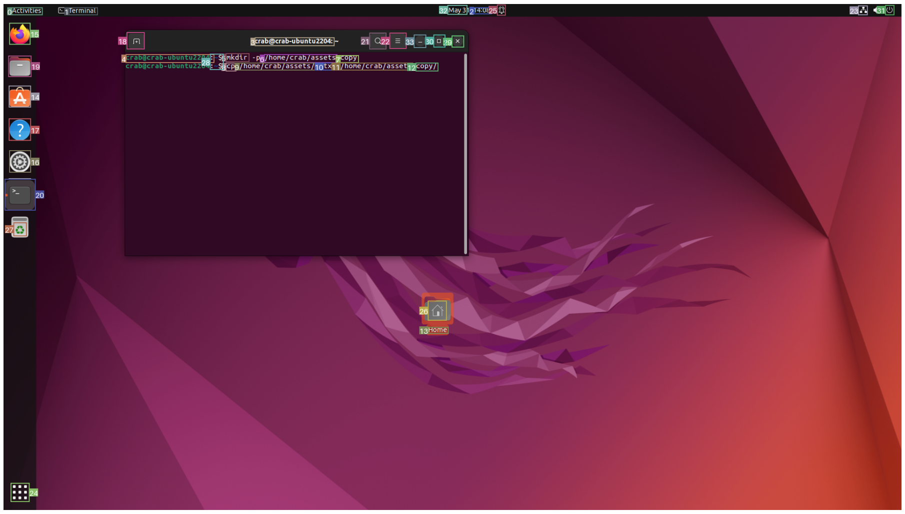
Continued on next page

Table 9 continued from previous page

3	 `click(6)`
4	 `write_text("mkdir -p /home/crab/assets_copy")`
5-6	

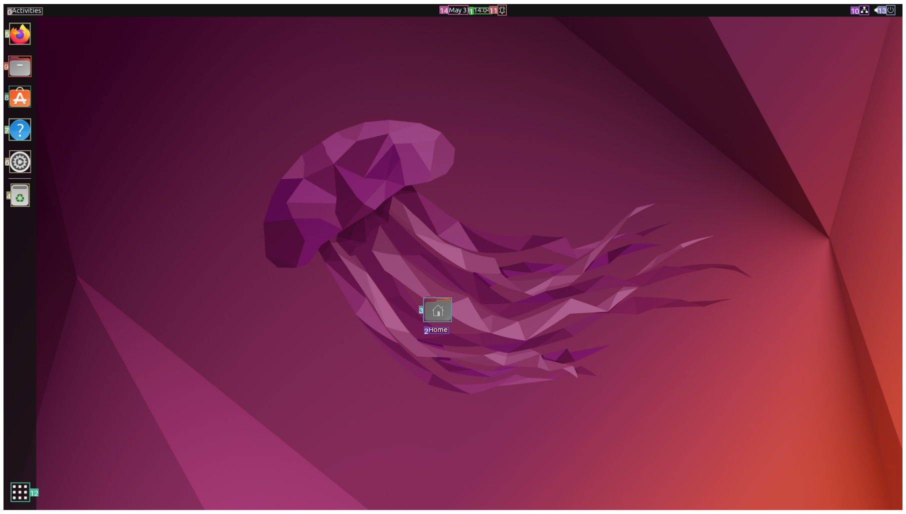
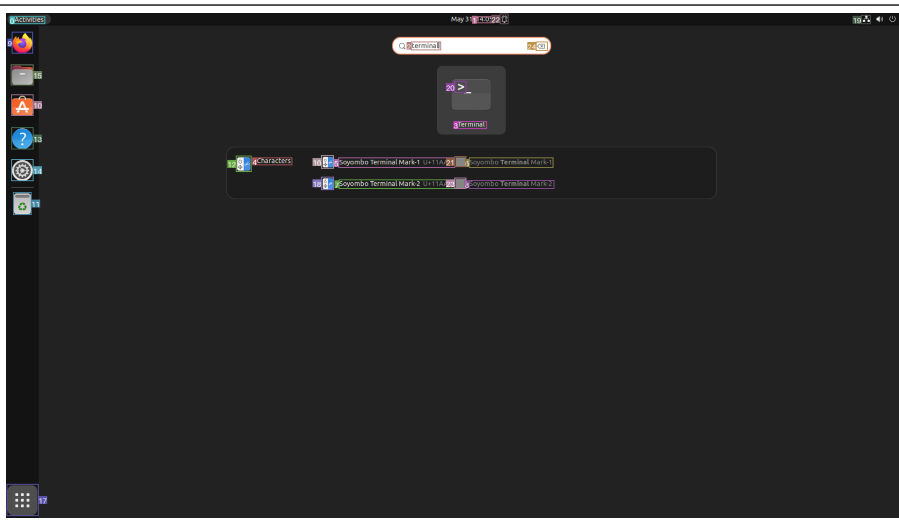
Continued on next page

Table 9 continued from previous page

5-6	`key_press("enter")` `write_text("cp /home/crab/assets/*.txt /home/crab/assets_copy/")`
7	 `key_press("enter")`

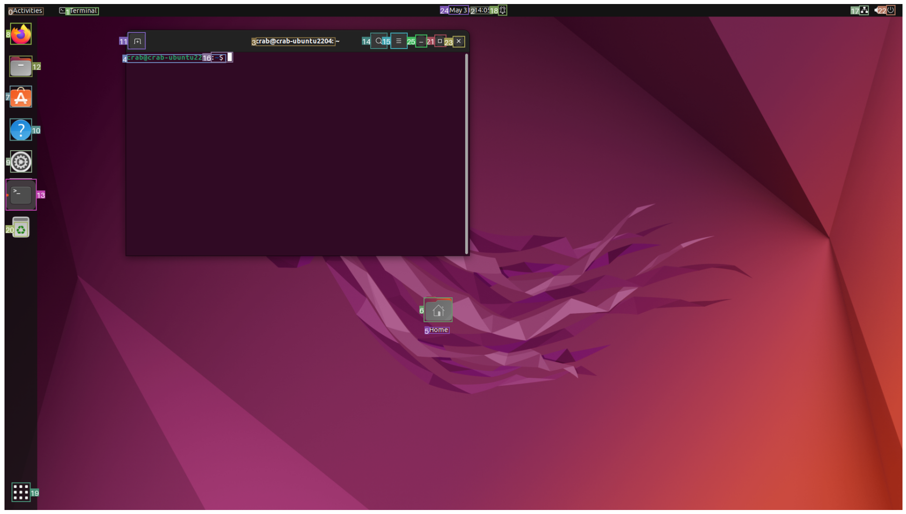
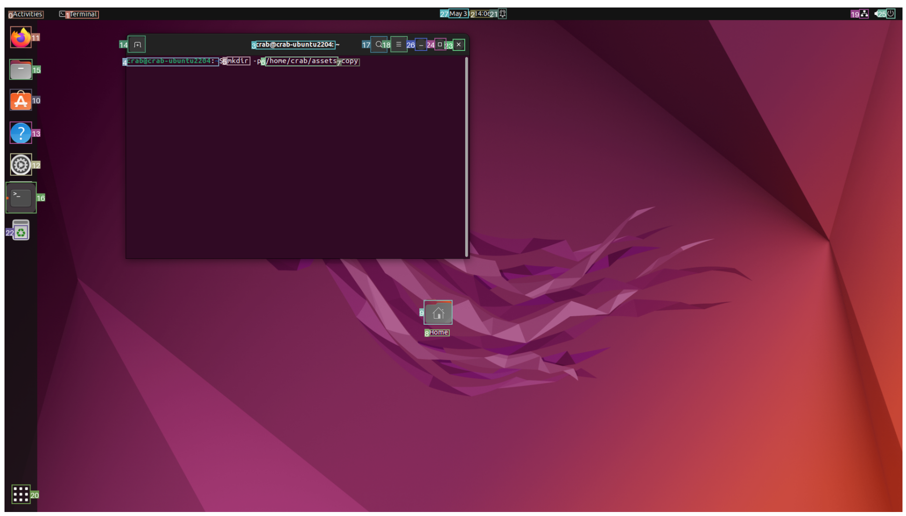
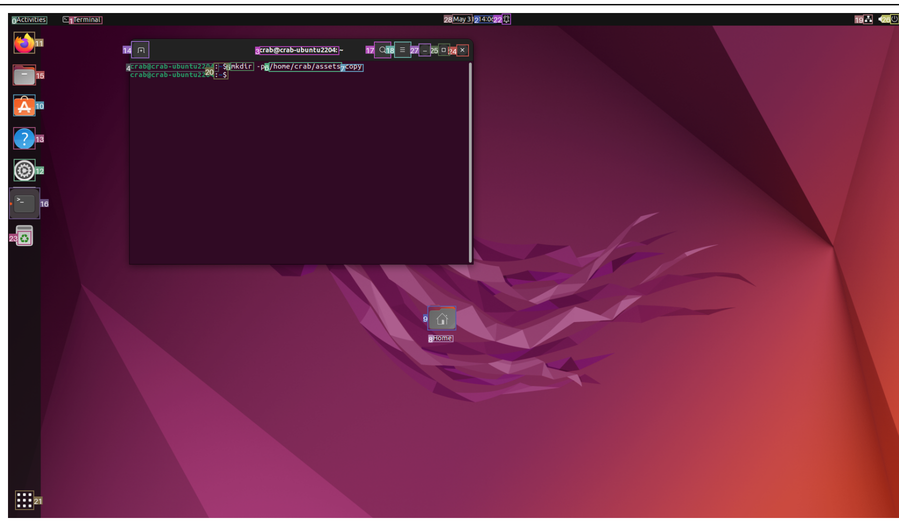
Completed Nodes / Total Nodes: 2 / 2 **Termination Reason**: Success

Table 10: **Ubuntu task case with GPT-4 Turbo (Single):** Create a new directory "/home/crab/assets_copy" and copy all files with the specified "txt" extension from "/home/crab/assets" to the directory "/home/crab/assets_copy".

Step	Agent Observation and Action
0	 `search_application("terminal")`
1	 `click(20)`

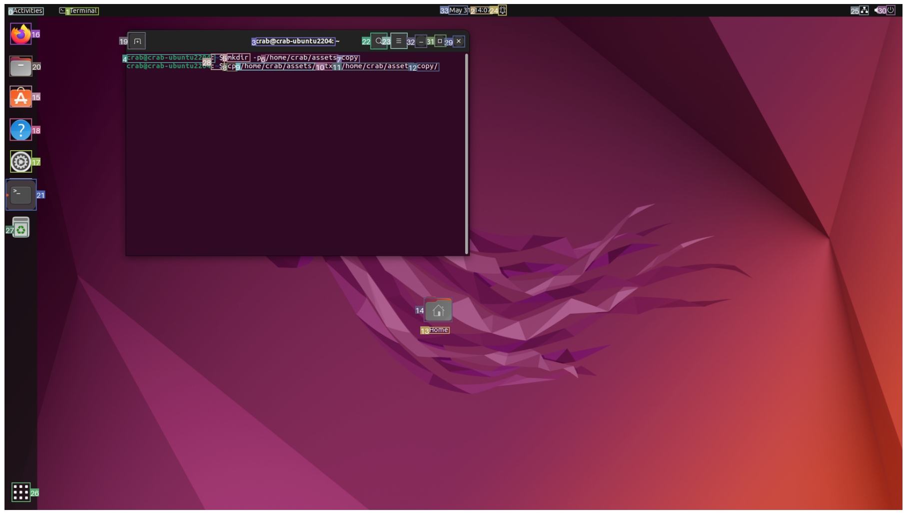
Continued on next page

Table 10 continued from previous page

2	 `write_text("mkdir -p /home/crab/assets_copy")`
3	 `key_press("enter")`
4	 `write_text("cp /home/crab/assets/*.txt /home/crab/assets_copy")`

Continued on next page

Table 10 continued from previous page

5	 key_press("enter")

Completed Nodes / Total Nodes: 2 / 2 **Termination Reason**: Success