LUCI: Multi-Application Orchestration Agent

Anonymous

***Abstract*—Research in building agents by employing Large Language Models (LLMs) for computer control is expanding, aiming to create agents that can efficiently automate complex or repetitive computational tasks. Prior works showcased the potential of Large Language Models (LLMs) with in-context learning (ICL). However, they suffered from limited context length and poor generalization of the underlying models, which led to poor performance in long-horizon tasks, handling multiple applications and working across multiple domains. While initial work focused on extending the coding capabilities of LLMs to work with APIs to accomplish tasks, a new body of work focused on Graphical User Interface (GUI) manipulation has shown strong success in web and mobile application automation. In this work, we introduce LUCI: Large Language Model-assisted User Control Interface, a hierarchical, modular, and efficient framework to extend the capabilities of LLMs to automate GUIs. LUCI utilizes the reasoning capabilities of LLMs to decompose tasks into sub-tasks and recursively solve them. A key innovation is the application-centric approach which creates sub-tasks by first selecting the applications needed to solve the prompt. The GUI application is decomposed into a novel compressed Information-Action-Field (IAF) representation based on the underlying syntax tree. Furthermore, LUCI follows a modular structure allowing it to be extended to new platforms without any additional training as the underlying reasoning works on our IAF representations. These innovations alongside the 'ensemble of LLMs' structure allow LUCI to outperform previous supervised learning (SL), reinforcement learning (RL), and LLM approaches on Miniwob++, overcoming challenges such as limited context length, exemplar memory requirements, and human intervention for task adaptability. LUCI shows a 20% improvement over the state-of-the-art (SOTA) in GUI automation on the Mind2Web benchmark. When tested in a realistic setting with over 22 commonly used applications, LUCI achieves an 80% success rate in undertaking tasks that use a subset of these applications. We also note an over 70% success rate on unseen applications, which is a less than 5% drop as compared to the fine-tuned applications.**

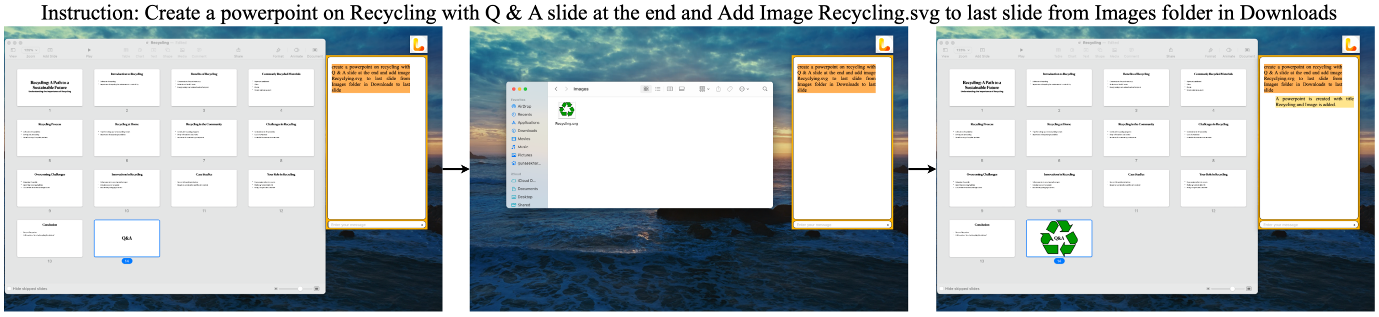
***Index Terms*—Computer Agent, Automation, GUI applications, Structured UI representation, Multi-application Orchestration.**

I. Introduction

Automation and assisted computer interaction have been a significant area of investment for researchers and industry professionals. Digital assistants such as Siri and Google Assistant [36] have enabled the creation of the Internet of Things (IoT) devices and the home automation space. They also play a critical role in the accessibility domain, enabling users with disabilities [20] to interact with computers and mobile devices. Today these virtual assistants are regularly used to automate a variety of simple tasks [24] such as setting alarms and reminders, controlling music playback, etc. However, they have always struggled with more complex tasks and require specific language based on keywords to function correctly.

A key breakthrough in this domain was the advent of Large Language Models (LLMs), which worked directly with natural language and displayed strong reasoning and planning capabilities [43]. This allowed them to create a more capable and user-friendly interface for human-computer interaction (HCI) [5]. Applications such as ChatGPT and Google Gemini proved the promise of the underlying models which are now set to become the backbone of the aforementioned assistants. While LLMs proved to be effective in interacting with the user, their ability to interact with computer systems and use tools was still limited and needed to be augmented. The first approaches attempted to extend the coding capabilities of LLMs to work with APIs to accomplish tasks [41, 31]. These systems adopted a Planner, Actor, and Reporter [3] to ground the LLMs (restrict the response to relevant domains not part of the LLM's trained knowledge) and allow them to interact with the environment(computer systems). While they were initially successful, API-based LLM automation systems struggled with generalization across multiple applications due to the need for creating natural language instructions for the API of each application. These methods [1, 9] tried to leverage *In-Context Learning* (ICL), to improve the gener-

Instruction: Create a powerpoint on Recycling with Q & A slide at the end and Add Image Recycling.svg to last slide from Images folder in Downloads


create a powerpoint on recycling with Q & A slide at the end and add image Recyclying.svg to last slide from Images folder in Downloads to last slide

A powerpoint is created with title Recycling and Image is added.

Fig. 1. An illustrative execution trace of LUCI creating a presentation to satisfy the given instruction: "Create a presentation on Recycling with Q & A slide at the end and Add Image "Recycling.svg" to last slide from Images folder in Downloads". First LUCI opens the Keynote Application and creates a presentation on Recycling. Then, open the Images folder from the Downloads directory and select "Recycling.svg" file. Finally, LUCI adds the image to the last slide of the presentation. **This showcases LUCI's ability to execute tasks involving multiple applications.**

alization of the underlying models by forgoing training and instead placing relevant information in the context for prompts. However, the limited context lengths of these models led to poor performance in long-horizon tasks and handling multiple applications.

This led to the development of Graphical User Interface (GUI) LLM frameworks[4, 19]. These systems focused on manipulating applications via the User Interface(UI) instead of relying on APIs which may or may not exist. The earliest versions used Reinforcement Learning (RL) to train an agent to mimic user clicks on inputs containing Hyper Text Markup Language(HTML) Document Object Model (DOM) elements. However, they struggled with with selection of relevant UI elements requiring supervised training. Gur et al [14] demonstrated the difficulty in training LLMs with purely HTML due to the low information density and high noise in RAW HTML. Zeng et al [45] incorporated the multi-modal learning capabilities of GPT4 to address the grounding of UI elements by proving the images of webpages to the model. This approach revolves around understanding the visual aspects of rendered web pages and generating precise plans in text format for various websites and tasks. However, this method did not scale beyond HTML and was hence limited to web applications. Additionally, these systems cannot utilize multiple applications or websites to accomplish a single task.

In this work, we introduce LLM-assisted User Control Interface (LUCI), a novel framework that extends the capabilities of LLMs to automate GUIs. LUCI is designed to enable LLMs to orchestrate multiple applications and execute complex tasks by interacting with the GUI. We accomplish this by adopting an application-centric approach to task planning where a *Tool Selector* element (Section. III-B) selects the most relevant applications from a given set to solve a task. These are then used as the central focus when decomposing a given instruction into sub-tasks. Each sub-task is then mapped to a novel intermediate compressed structured representation based on Information-Action-Field (IAF) pairs via a*"UI Extractor"* (Section. III-E) and a *'UI Selector"* (Section. III-F). This structured representation allows the LLM to effectively interact with the GUI applications and execute the sub-tasks. The LUCI framework is designed to be modular, hierarchical, and OS-agnostic, enabling it to work seamlessly across both native and web interfaces. Additionally, we augment the performance of our conversational model with a *Task Verifier* (Section. III-D) to filter redundant sub-tasks and improve efficiency. The limited scope of the Task Verifier allows it to focus on the relevance of a given sub-task, effectively reducing the generative task of the conversational model into a simpler decision task. The structure of these elements and how they interact with each other is shown in Figure **??**.

When combined these components allow LUCI to solve complex multi-step and multi-application tasks across web and native interfaces without the need for additional multi-modal context. An example of this capability is shown in Figure 1, where LUCI creates a presentation on Recycling with a Q & A slide at the end and adds an image "Recycling.svg" to the last slide from the Images folder in Downloads. This showcases LUCI's ability to execute tasks involving multiple applications.

The main contributions of LUCI can be summarized as:

1) An application-centric approach to task planning, where the selection of relevant applications is the basis of sub-task generation.
2) A modular OS-agnostic agent capable of functioning seamlessly across both native and web interfaces.
3) A novel tool selection mechanism is implemented to identify relevant tools for tasks involving multiple applications, enhancing adaptability and effectiveness.
4) A novel UI parser is designed to extract the web and desktop interfaces into a compressed and structured representation based on Information-Action-Field (IAF) pairs, thereby facilitating efficient orchestration by large language models (LLMs).
5) A hierarchical control structure within LUCI, enabling effective management of tasks across multiple applications, thereby empowering LLMs with comprehensive control in diverse environments.

Our experimental results support our claim. We note that LUCI achieves a greater than 99% success rate on the MiniWoB++ benchmark. LUCI also achieves up to 31% improvement in Step SR and up to 24% improvement in OP. F1 Score over GPT4V on the Mind2Web benchmark. When we tested the generalization capability we noted a less than 5% drop in accuracy, indicating strong generalization. Finally, our experiments show that LUCI maintains a 75% average success rate when using 4 applications simultaneously. We believe that the application-centric design proposed in LUCI presents a promising direction for future research in GUI automation and multi-application orchestration.

II. Related Work

A. Building Agents with LLMs

Large language models (LLMs) have offered promising avenues for leveraging natural language in decision-making tasks. One approach involves enhancing LLMs with executable actions [27]. Huang et al. [18] showed LLMs can plan and perform basic domestic activities by mapping embeddings to a predefined list of actions. However, their study lacks specificity for contemporary activities. Ahn et al. [1] introduced SayCan, grounding actions by multiplying candidate action probabilities with FLAN [40] and the action's value function as an indicator of suitability. Huang et al. [17] extended SayCan with Inner Monologue, adding a feedback loop to select actions according to the current state. However, Inner Monologue relies on a pre-trained, robot policy conditioned on language with limited flexibility, impeding generalization across various domains. Zeng et al. [44] combined LLMs conditioned on a robot policy along with a vision-language model (VLM) for open vocabulary pick-and-place tasks. Dasgupta et al. [3] utilized Chinchilla [15] as a planner in PycoLab, requiring RL pre-training for their actor module to follow natural language instructions. Moreover, previous methods were restricted to the necessity for fine-tuning.

Recently, enhancing LLM effectiveness involves integrating them with APIs for utilizing external tools like information retrieval systems, code interpreters, and web browsers [10, 31].

Integrating APIs with LLMs is done using 4 main techniques :

1) **pre-training/fine-tuning** the model with API-enabled examples [35, 31]. This approach has limited API space.
2) **Few-shot in-context learning** of LLMs with APIs [1, 9]. This approach is limited by context length.
3) **Reinforcement learning with human feedback** to enhance API usage [28, 25].
4) **Creating natural language documents (NLD) or structured programs** to instruct the model [37, 30].

The NLD approach is extremely sensitive to the quality of the API documentation [25], requiring developers to maintain comprehensive and well-structured documentation. Alongside the issues with poor generalizability across different API documents for zero-shot usage and the limitations mentioned above, there has been a recent shift towards GUI-based methods.

B. Automated GUI tasks

P Pursuing the goal of interaction between humans and the computer, significant efforts have been dedicated to developing autonomous computer agents capable of understanding language instructions and efficiently carrying out tasks on a computer [13, 7, 25]. To evaluate the models for human-like computer interactions, MiniWoB++ extending the MiniWoB benchmark [33, 26], serves as a standard benchmark. Early researchers utilized reinforcement learning and imitation learning to solve MiniWoB++ challenges [26, 13, 21, 12], but reaching human-level performance necessitates a large amount of expert demonstration data (6,300 hours) [19].

Recent research [14, 7] proposes employing large language models (LLMs) to read HTML code and vision transformers [6, 16] for extracting screenshot features, using a few-shot in-context method yielding promising results without prolonged RL training. Nonetheless, large volumes of expert demonstration data are still needed to fine-tune LLMs. WebGPT [28] and WebShop [42] demonstrate LLMs automating web tasks with custom commands, but they are restricted in scope and don't address general computer tasks requiring keyboard and mouse inputs.

RCI [22] achieves a 90.6% success rate in 54 MiniWoB++ tasks using recursive self-correction, yet its reliance on task-specific examples restricts generalization to new scenarios. In contrast, our proposed method works well without relying on self-correction. AdaPlanner [34] achieved a 92.9% success rate in 53 tasks by leveraging environment feedback for self-correction, but faced similar generalization challenges as RCI. Pix2Act [32] addressed 59 MiniWoB++ tasks through tree search and BC, based on 1.3 million demonstrations. WebGUM [8] fine-tunes Language Multimodal Model (LMM) with a huge multimodal corpus for web agents, allowing web agents to observe both HTML and the captured screenshot but requires multiple samples of expert demonstrations for fine-tuning. Synapse [46] uses structured prompts with an LLM and achieves human-level performance. However, the performance is largely dependent on the quality of examples passed through prompts. On the other hand, our proposed method uses few-shot in-context learning with structured prompts and structured representation of UI elements on a browser making the agent independent of the quality of examples. Moreover, our approach excels in addressing open-domain tasks on a large scale.

Numerous ongoing initiatives are dedicated to the development of computer agents and benchmarks. Among the sophisticated benchmarks available, such as Mind2Web [4], Webshop [42], and WebArena [47]. WebArena [47] generates website simulations in a sandbox environment across four popular categories, replicating functionality and data from real-world equivalents. In contrast, Mind2Web [4] offers environments spanning diverse domains, websites, and various tasks extracted from live websites, complete with action trajectory annotations. So, we have chosen Mind2Web for our real-world evaluation. To solve the Mind2Web benchmark, MindAct utilized an element-ranking model to extract the top 50 relevant elements as clean observations. It employs MCQ to recursively query the LLM for action selection from five alternative candidates until either all options are wrong or just one action is selected. While MCQ-formatted exemplars perform better than direct generation, MindAct frequently has trouble selecting the right element [4]. Another approach involves the use of language and vision transformers. Recent works in this works include WebGUM [8] and Gpt-4v [45]. WebGUM [8] fine-tunes Language Multimodal Model (LMM) with a huge multimodal corpus for web agents, allowing web agents to observe both HTML and the captured screenshot but require lot of expert demonstrations for fine-tuning. Gpt-4v [45] uses LMMs to observe both HTML and captured screenshots to generate actions but still lack sufficient visual perception abilities to serve as an effective agent.

III. LUCI Architecture

In this work, our goal is to enable large language model to use various GUI tools to extend its capabilities. The primary architecture of LUCI comprises of 7 key components, namely

1) **GUI Tool Set**, which contains list of GUI tools authorized to be controlled by the LLM.
2) **Tool Selector**, which selects GUI tools from GUI tool set required to accomplish given user instruction.
3) **Conversational Model**, which is responsible for interacting with users and generating a solution outline based on GUI tool selected by the tool selector, user instruction along with its conversational context.
4) **Task Verifier**, which filters the redundant sub-tasks in the solution outline using action feedback.
5) **UI Extractor**, which extracts the information of UI elements from the selected GUI tool.
6) **UI Selector**, which selects appropriate UI elements from the list of UI elements generated by UI extractor for the given sub-task.
7) **Action Executor**, which performs action on selected UI elements based on type of UI element and return the action feedback.

The overall architecture of LUCI is shown in Figure 2. The primary process within this architecture is the LLM's capability to execute action in response to user instructions.

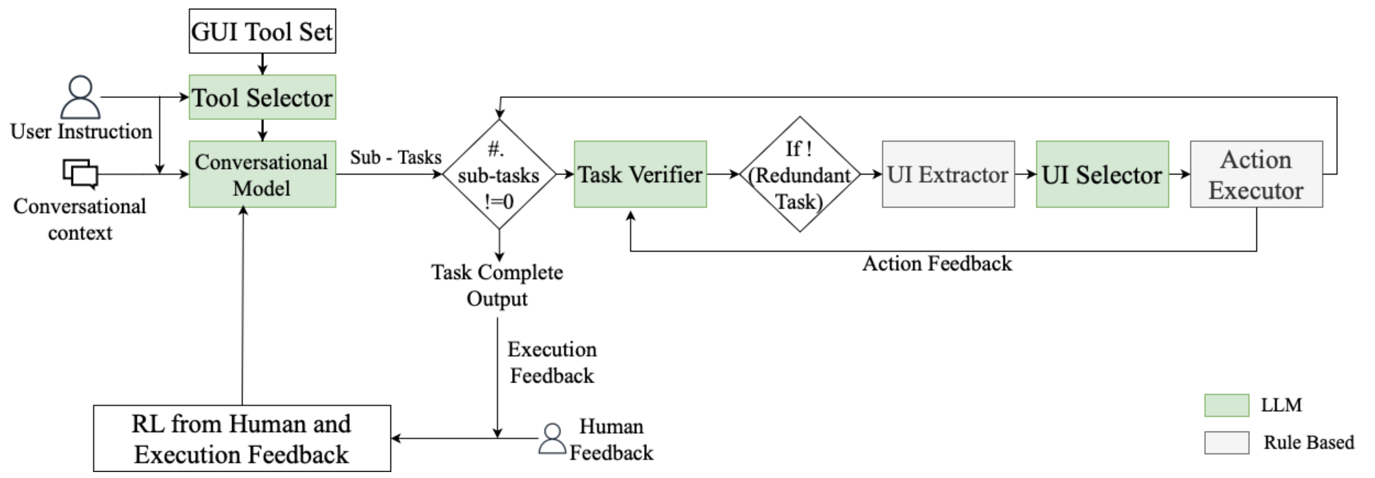


Fig. 2. **Architecture of LUCI**. Given user instruction and the conversational context, the Tool Selector first selects a tool from the given GUI toolset and opens the applications. The conversational model then generates a solution outline, which is a text description of the list of sub-tasks needed to solve the task. Next, the Task Verifier filters redundant tasks in the solution outline based on action feedback from previously executed tasks and future sub-tasks. Then, a rule-based UI Extractor extracts UI elements from the GUI application. UI Selector selects appropriate UI elements from the list of UI elements generated by the UI extractor for the given sub-task. Lastly, the Action Executor performs an action on the selected UI element based on the type of UI element and generates the action feedback.

This approach takes 4 inputs: large language model's parameters, represented as θ; GUI tool set, denoted as $\mathcal{G}$; the user instructions, designated as $\mathcal{I}$, along with the conversational context, referred to as $\mathcal{C}$. Utilizing these inputs, the LLM generates a set of actions, designated as $\mathcal{A}$, to execute and fulfill the user's instruction. This procedure can be defined as follows:

$$\mathcal{A} = LLM(\theta, \mathcal{G}, \mathcal{I}, \mathcal{C}) \tag{1}$$

Further, LUCI is employed with RLHF learning mechanism [29] to improve the task planning and task verification. Here, in addition to human feedback, we also employ execution feedback, this combined feedback is denoted by $\mathcal{F}$. The Loss function of the learning mechanism is parameterised by,

$$\mathcal{L} = \mathcal{L}(LLM(\theta, \mathcal{G}, \mathcal{I}, \mathcal{C}), \mathcal{F}) \tag{2}$$

A. *GUI Tool Set*

The GUI tool set is a primary element in this framework, the tool set G = $\mathcal{G} = \{\mathcal{G}_1, \mathcal{G}_2, \mathcal{G}_3.....\}$, which contains a collection of different GUI application names. The GUI tool set contains a list of all desktop application names (For instance, a weather application on your personal computer) that the agent is allowed to work on. In this work, we have used 22 GUI based desktop applications as shown in Figure 8. In this work web applications are accessed through a browser like Safari, Google Chrome or Microsoft Edge, which gives more flexibility for navigation and using multiple GUI applications.

B. *Tool selector*

The main objective of the tool selector is to select a most appropriate GUI tool from the GUI tool set that aligns with the given task requirements. The tool selector is a language model that uses in-context learning to select a GUI application $\mathcal{G}_s$ from the set of GUI tools $\mathcal{G} = \{\mathcal{G}_1, \mathcal{G}_2, \mathcal{G}_3.....\}$ based on given user instruction I, as shown below. The output of the language model is grounded to application names in the GUI tool set.

The tool selector in our study facilitates the use of multiple GUI tools based on user instructions as shown in Example Query 2 in Figure 3. If a specified GUI tool is incapable of performing a task, ”Tool Not Found” is chosen as shown in Example Query 3. For tasks that solely require language model capabilities, ”No tool required” is selected as shown in Example Query 4. In cases of duplicate functionality, the user's specified tool takes precedence, or the tool is selected alphabetically or based on previous user selections.

C. *Conversational Model*

The conversational model is a LLM, which acts as a “brain” for this framework by understanding user intentions from the current instructions and past conversations to generate a set of sub-tasks (solution outline) needed to use the selected GUI application. To generate a Solution outline, we employed the Chain-of-thought (CoT), as it has shown significant improvement in performance across various tasks including arithmetic reasoning, commonsense reasoning, and symbolic reasoning [39, 23]. An example of a solution outline generated for using a calculator is shown below.

To facilitate the utilization of multiple GUI tools, we have specified a field called “Tool Name” for each sub-task along with sub-task description within the generated solution outline. The generated solution outline, consists of 3 types of sub-tasks:

1) Sub-tasks that contain description of opening an application. In our work, the GUI application is opened right after the tool selector selects an application name from the tool set and is not considered as a sub-task.
2) Sub-tasks that can be directly performed by the conversational model, such as text summarization (Sub-task 9 in the Example Query 5), will not trigger any action on GUI application.

GUI Tool Set (G) : Calculator, Calendar, Keynote, Maps, Messages, Music, News, Numbers, Pages, Safari, Stocks, TextEdit, Weather, Script Editor, Terminal, Outlook.
Example Query 1: From the given list of GUI tools G, select a tool to solve the task " Send an email to johndoe@gmail.com".
Response: Outlook
Example Query 2: From the given list of GUI tools G, select a tool to solve the task "Weng earns $ 12 an hour for babysitting. Yesterday, she just did 50 minutes of babysitting. How much did she earn? And write the entire proof."
Response: Calculator, Text Edit
Example Query 3: From the given list of GUI tools G, select a tool to solve the task " Draw a 3D model of cube with dimensions 3m x 3m x 3m".
Response: Tool Not Found
Example Query 4: From the given list of GUI tools G, select a tool to solve the task " Write a summary of the given paragraph, ... ".
Response: No tool required

Fig. 3. Illustrative of examples of Tool Selector in selecting a GUI application from a given GUI Tool set. Example Query 1 shows the ability of Tool selector to select a GUI application. Example Query 2 shows the ability of Tool selector to select multiple GUI application. Example Query 3 shows the ability of Tool selector to notify the conversation model that given tasks cannot be performed with given GUI applications. Example Query 4 shows the ability of Tool selector to to notify the conversation model that given tasks can be solved without GUI application.

Example Query 5: What are the tasks to perform on Calculator and Text Edit for the given task "Weng earns $12 an hour for babysitting. Yesterday, she just did 50 minutes of babysitting. How much did she earn? And write the entire proof". Generate the tasks in the format "Tool Required: True / False, Tool: tool Name, Task description: description of the task".
Response:

1) Tool Required: False, Tool: Calculator, Task description: Open Calculator on your Macbook.
2) Tool Required: True, Tool: Calculator, Task description: Set the calculator to the "Standard" mode".
3) Tool Required: True, Tool: Calculator, Task description: Convert 50 minutes to hour by dividing 50 by 60 in the Calculator.
4) Tool Required: True, Tool: Calculator, Task description: Press the equals (=) key to calculate the result.
5) Tool Required: True, Tool: Calculator, Task description: Multiply number of minutes with money earned per hour ($12) into the Calculator.
6) Tool Required: True, Tool: Calculator, Task description: Press the equals (=) key to calculate the result.
7) Tool Required: True, Tool: Calculator, Task description: Observe the calculated result displayed on the Calculator screen.
8) Tool Required: False, Tool: Text Edit, Task description: Open Text Edit.
9) Tool Required: False, Tool: Text Edit, Task description: Generate a summary of the working.
10) Tool Required: True, Tool: Text Edit, Task description: Write the summary to Text Edit.

Fig. 4. Illustrative example of solution outline from Conversation model to solve a task which involves integration of two applications, Calculator and Text Edit

3) Sub-tasks that contain action descriptions to be performed on GUI applications as seen in Sub-tasks 2-7 and 10 in the Example Query 5 in Figure 4.

From above categorization, to distinguish sub-tasks requiring GUI interaction, a boolean field "Tool Required" is introduced in the Solution outline, preventing unnecessary execution of subsequent steps for tasks not involving tool interaction. The final output format of each sub-task in the solution outline is shown below:

<**Tool Required**: True/False, **Tool**: GUI Application Name, **Task description**: Sub task description
>

When "No tool required" is chosen, the conversational model directly responds to user instructions, such as summarizing a text, and provides feedback in cases where "Not Found" is selected by the tool selector.

D. Task Verifier

Before executing each sub-task in the generated solution outline, it should be evaluated for 2 things to remove redundant tasks and improve efficiency:

1) Whether the sub-task is necessary or not based on future tasks.
2) Sub-task is already executed or not based on previous tasks.

In the response of Example Query 5, after converting 50 minutes to hours, there is no need to press "=" key, one can directly multiply this with the hourly earnings. Also, there might be a chance of pressing "=" key after dividing 50 by 60 in Sub-task 3 is valid, but doing so in Sub-task 4 leads to an error due to redundant division by 60. To prevent errors and redundant sub-tasks, we introduced Task verifier. The main aim of the Task verifier is to identify and remove the unnecessary sub-tasks in the solution outline.

Task verifier is a language model which takes current sub-task, future sub-tasks, and action feedback from previous sub-

Example Query 6 : Check whether the current task s3 is necessary for the objective I based on previously executed tasks {(s1; AF1), (s2; AF2)} and future tasks s4, s5, s6, s7. The answer format should be " Reason: Reason to execute sub-task , Answer: Yes/No".
Response: Reason: This task is required to convert 50 minutes to hours, Answer: Yes

Fig. 5. Illustrative example of Task verifier in deciding whether a task is redundant or not by reasoning.

UI elements (U): For Calculator available UI elements are,
window : {
1, 2, 3, 4, 5, 6, 7, 8, 9, 0, =, +, , ×, ÷, ., +/, %, all clear, close button, zoom button, minimize button, main display: {0}
}
Menu bar : {
File: {Close, Close All, Save Tape As, Page Setup, Print Tape},
Edit: {Undo, Redo, Cut, Copy, Paste, Clear, Select All, Start Dictation, Emoji Symbols},
View: {Basic, Scientific, Programmer, Show Thousands Separators, RPN Mode, Decimal Places, Enter Full Screen}
Convert: {Recent Conversions, Area, Currency, Energy or Work, Length, Power, Pressure, Speed, Temperature, Time, Volume, Weights and Masses}
}
Example Query 7 : From the given list of available UI elements : U, select list of UI elements required for task "Convert 50 minutes to hour by dividing 50 with 60 in the Calculator" . The answer should be format [UI element 1, UI element 2, ...]
Response: [5, 0, ÷, 6, 0, =]

Fig. 6. Illustrative example of UI selector selcting a list of UI elements.

tasks to decide whether to proceed with execution or not by reasoning as shown in Example Query of Figure 5. The action feedback is generated through the analysis of interactions with user interface (UI) elements and the resulting changes in the user interface. This feedback mechanism informs the task verifier about actions performed in the past and reduces the execution of redundant actions in subsequent sub-tasks. At the end of execution, action feedback is combined to generate the execution feedback used in RLHF for model improvement.

E. UI Extractor

We developed a rule-based UI element extractor to extract UI elements from desktop applications and its parameters like type of UI element, allowed actions, location, size, description, value (if any) from GUI tool. In any desktop application, the graphical interface is based on hierarchy / tree structure. We can use accessibility tools provided by Windows and MAC OS to extract this tree structure. In this tree based structure, we divide the UI elements into 3 categories :

- **Information Elements (I)**: UI elements that contain only Information. Example: $<p>, <h1> - <h6>, <span>, <div>$ e.t.c (in context of HTML).
- **Action Elements (A)**: UI elements which perform can post and get methods. Example: buttons, hyperlinks, submit buttons e.t.c.
- **Field Elements (F)**: UI elements that collect user input. Example: textbox, checkbox, radio buttons e.t.c.

Based on these 3 categorizations, we assume the following:

- User input is sent to server when a post method is called. It means every **F** is associated with a **A**. For instance the submit button comes after the search box. If there are multiple **F**'s while parsing through the tree it is associated with the same **A** within the branch. Every **F** has a corresponding **A** but **A** need not have a **F**.
- If there is **I**, it is linked with **A** either in its child nodes or child nodes to parent nodes of **I** to get the context.
- **I** can contain multiple **A**. For example, heading and list of links.
- **I** can have multiple **I**'s, each associated with '**A**'. For example, a webpage contains a heading, subheading and list of links for each subheading.

Based on the above assumptions, we parse through the tree structure using a bottom-top approach and the tree structure can be broken into:
$I[I1, <A1, F1>, I2, <A2, F2>, I3, <A3, F3>, ...]$
I1, I2, I3,...... contains all text information in a given node. $<An, Fn>$ is a set of A and its corresponding F pairs.

F. UI Selector

Recent studies [2, 11] propose training language models for discrimination rather than generation, as it enhances generalizability and sample efficiency for grounding tasks. We adopt this approach by transforming UI element selection into a multi-choice question-answering problem. The language model is trained to select from a list of options instead of generating the complete target element. Once the UI elements are extracted, the UI elements required for a sub-task are selected by using an LLM as shown in Example Query 7 of Figure 6. If a UI element is not found, a feedback signal is sent to the conversational model, triggering a revision of the current sub-task.

G. Action Executor

The action executor is formulated with the purpose of executing a finite set of actions, including clicking, right-clicking, text entry and selection. The action executor is a python program that utilizes information about UI elements, such as their path, location and size, to execute an appropriate action based on the type of UI element. In order to enhance accuracy and reliability, the action executor is incorporated with a feedback mechanism to ascertain whether an action is performed or not. Once an action is performed, it generates action feedback for the next sub-task. Following the execution of all the sub-tasks, the action executor will send a message to the conversation model to generate execution feedback and furnish results to users.

IV. Novel Contributions of LUCI

A. *Application-Centric Architecture*

As seen in Figure. 2, the tool selector is the first component in our architecture. The conversational model generates sub-tasks based on the selected applications, which simplifies the problem statement and enables high-quality sub-task generation. The application-centric approach also **enable multi-application orchestration by separating the selection and orchestration tasks** between the Tool Selector and the Conversational Model.

B. *Modular OS-Agnostic Agent*

The Modular OS-Agnostic Agent represents a foundational aspect of LUCI's architecture. The decomposition of the entire architecture into independent and interchangeable components, each fulfilling specific functions within the framework, makes LUCI modular. Components such as the Tool Selector, Conversational Model, Task verifier and UI Selector are based on LLM's and are engineered to seamlessly operate across both native and web interfaces. Employing platform-independent techniques to traverse the UI hierarchy and extract relevant information, the UI Extractor within LUCI is designed to retrieve UI elements from desktop applications in a specific format, thus contributing to its operating system (OS) agnosticism. This design approach ensures LUCI's adaptability to diverse environments, facilitating its effectiveness across a range of operating systems and interface types.

C. *Novel tool selection mechanism*

The Tool Selector embodies a novel tool selection mechanism aimed at identifying relevant tools for multi-application tasks. At its core, the Tool Selector incorporates a sophisticated mechanism driven by in-context learning, which enables it to discern the most relevant GUI tool from the available toolset. Unlike traditional selection methods, which may rely solely on predefined rules or heuristics, the Tool Selector dynamically adapts its decision-making process based on the context provided by the user's instructions and task-specific requirements. This adaptive approach allows the Tool Selector to consider various factors when identifying the optimal GUI tool for a given task. For instance, it may take into account the nature of the user's instructions, such as the specific actions or functionalities requested, as well as any contextual information provided, such as the current state of the application or the user's preferences. By leveraging this contextual awareness, the Tool Selector can make more informed decisions, ultimately leading to better tool selections and improved task performance.

D. *Novel UI Parser*

The contribution pertaining to the novel UI parser is exemplified by the UI Extractor component within LUCI. This component employs a novel UI parsing technique to extract structured IAF representations of both web and desktop interfaces. At its core, the UI Extractor employs a sophisticated UI parsing technique that allows it to traverse the complex hierarchy of UI elements present in web and desktop applications. By systematically categorizing these elements and extracting pertinent information such as type, location, size, and description, the UI Extractor generates structured representations that can be easily interpreted by the LLM. These IAF representations **sovle the limited context length issue** seen in previous methods.

E. *Hierarchical Control Structure*

This contribution is manifested in the hierarchical control structure implemented within LUCI, facilitating the control of tasks across multiple applications. At the heart of this hierarchical control structure lies the collaborative effort of key components such as the Conversational Model, Task Verifier, and Action Executor. The conversational model within LUCI employs in-context learning to understand user instructions and adapt its behavior accordingly continuously refining its understanding of tasks through feedback. Additionally, the Task Verifier filters redundant sub-tasks based on future tasks and past actions, minimizing unnecessary actions without human intervention. These features collectively allow LUCI to autonomously adapt to varying user needs and preferences without frequent human intervention while contributing to the overall effectiveness of the framework.

V. Experiments and Results

In this section, we evaluate the performance of LUCI and demonstrate that: (1) **LUCI outperforms previous approaches on executing complex tasks** (Section V-A), **LUCI enables cross-application adaptability** (Section V-B) and **LUCI can utilize multiple applications for executing complex tasks** (Section V-C).

A. *LUCI outperforms previous approaches on executing complex tasks*

The performance of LUCI in executing complex tasks stands out prominently, especially when evaluated against other methodologies. Our comprehensive examination involved testing LUCI across two benchmarks Miniwob++ [33] and Mind2Web [4], to allow for fair comparisons with baselines. The MiniWoB++ task suite, designed to simulate real-world human-computer interactions [33, 26], poses simple tasks like click-checkboxes, and text-complete for computer agents. In our MiniWoB++ experiments, we performed experiments on

TABLE I

AVERAGE PERFORMANCE OF DIFFERENT METHODS ON MIND2WEB BENCHMARK. LUCI W/ GPT-3.5 ACHIEVES STATE-OF-THE-ART PERFORMANCE.

Baseline	Cross-Task		Cross-Website		Cross-Domain	
	Op. F1	Step SR	Op. F1	Step SR	Op. F1	Step SR
MINDACT	56.6	17.4	48.8	16.2	52.8	18.6
Synapse	-	30.6	-	29.1	-	26.4
WebGUM	75.9	64.9	75.3	62.5	77.7	66.7
GPT-4v	80.9	65.7	83.7	70	73.6	62.1
LUCI						
w/ GPT-3.5	93.8	86.7	96.3	89.1	91.7	84.2
w/ Phi2	82.3	72.8	84.9	77.3	79.4	69.1

two LLM's GPT-3.5-turbo and Phi-2, running 50 episodes to generate results for each task. In MiniWoB++ setup, we adopt RCI [22] configurations with action space comprising of click-xpath, type, press, and click-options. The primary evaluation criterion is the success rate, reflecting the agent's effectiveness in completing the assigned task [22]. The success rate is determined by the proportion of successful episodes, wherein the agent receives a positive reward.

Mind2Web [4] is a realistic dataset with human demonstrations of open-domain tasks from diverse 137 real-world websites like Airbnb and Twitter, for assessing generalization across tasks, websites, and domains. For Mind2Web, we utilized GPT-3.5-turbo and Phi-2 with greedy decoding (i.e, temperature set to 0). Metrics include Operation F1 (Op. F1) for token-level F1 score for predicted operation comprised of action and input value, and Step SR for success rate per task step. This dataset is divided into three test sets: Cross-Task, Cross-Website, and Cross-Domain, evaluating generalizability over tasks from the same, similar and completely unseen domains, respectively. We include a set of examples in the prompts.

Further, to evaluate our work on day-to-day tasks we performed our experiments on 22 GUI applications (including both desktop and web applications) shown in Figure 8. To evaluate the performance of GUI applications, our key evaluation criterion is the success rate, reflecting the agent's effectiveness in completing the assigned task. Here, we've categorized three types of failure: tool selection, selection of unwanted UI elements and task failure. Additionally, an episode is deemed failed if the agent successfully carries out the created plan but is unable to complete the assignment and thus not rewarded. Most of the applications lack an appropriate dataset for comprehensive evaluation. To solve this problem, we employed an approach similar to [38]. We used a set of hand-written tasks serving as seed examples and then, utilize ChatGPT to generate more tasks. Unless explicitly stated otherwise, for these manually curated test sets, human evaluators assess and determine whether the task is considered to be accurately accomplished.

*1) **Performance on MiniWoB++ Task Suite**:* We performed comprehensive experiments to assess LUCI's performance in comparison to state-of-the-art (SOTA) approaches on MiniWoB++. For baseline comparisons using Behavior Cloning (BC) and Reinforcement Learning (RL), we employed CC-Net [19] and Pix2Act [32], which leverage large-scale BC and RL techniques. Regarding fine-tuning baselines, we evaluated against WebGUM [8] and WebN-T5 [14], two language models fine-tuned on a substantial number of demonstrations. In the realm of in-context learning (ICL) methods, our baselines comprised Synapse [46], RCI [22] and AdaPlanner [34], both incorporating self-correction mechanisms. Additionally, we included human scores from Humphreys et al. [19] for supplementary benchmarking.

Figure 7 illustrates the average performance of different methods across Miniwob++ benchmark. LUCI, utilizing PHI2 and gpt-3.5-turbo, achieves human-level performance with mean success rates of 94% and 98.6%, respectively. Notably, LUCI using gpt-3.5-turbo outperforms all baselines on MiniWoB++. LUCI surpasses previous ICL methods by addressing issues associated with context length, the need for exemplar memory and human intervention for task adaptability.

First, the UI extractor in LUCI compresses the UI into a structured IAF representation, which solves the limited context length issue seen in previous methods enabling it to solve tasks such as booking flights that require more context and were unsolvable by previous methods. Second, LUCI's hierarchical control structure and continuous user interaction enable dynamic task execution and adaptation without relying on exemplar memory, unlike Synapse [46]. **While Synapse and LUCI have around 99% success rate on the simpler Miniwob++ benchmark, LUCI shows nearly 3 times higher performance on Mind2WEB benchmark**. Third, the conversational model within LUCI employs in-context learning to understand user instructions and adapt its behavior accordingly

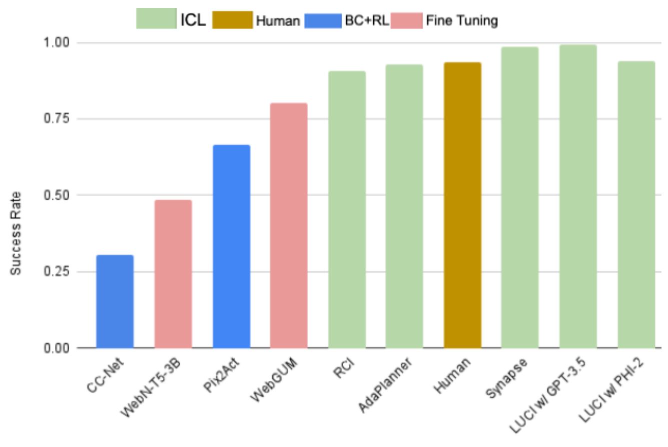


Fig. 7. Average performance comparison with baselines in MiniWoB++ environment. LUCI w/ GPT-3.5 achieves state-of-the-art performance and LUCI w/ PHI-2 is the first model to achieve human-level performance with LLM less than 3B parameters.

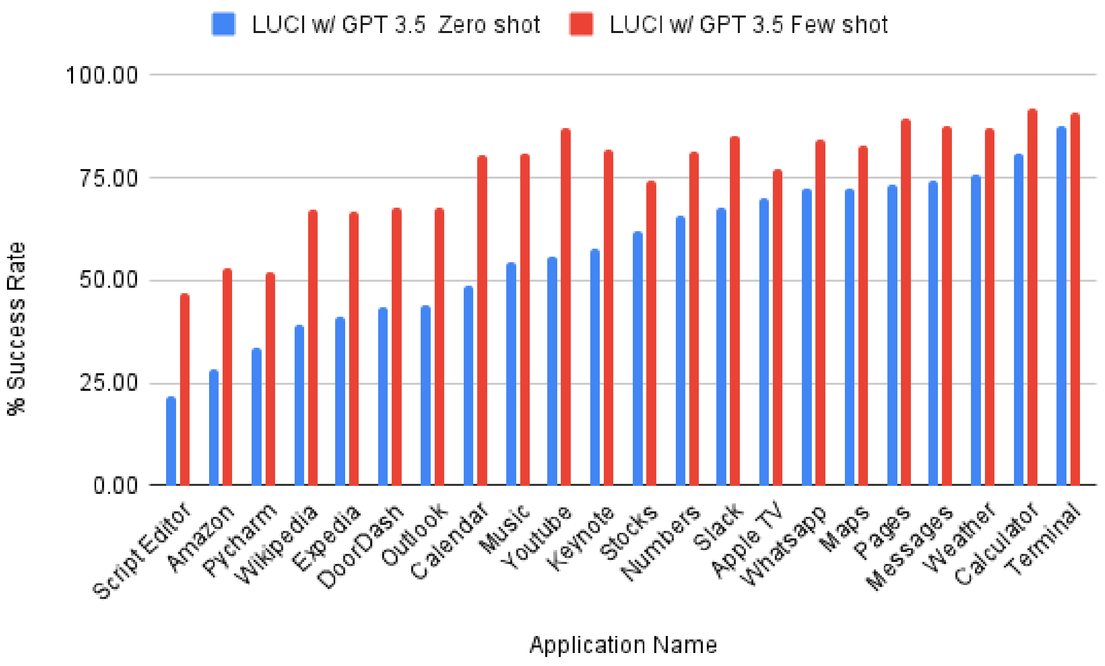


Fig. 8. Average success rate of LUCI in using GUI Applications with GPT-3.5 under zero-shot setting and Few Shot Setting.

to continuously refine its understanding of tasks through feedback. Additionally, the Task Verifier filters redundant sub-tasks based on future tasks and past actions, minimizing unnecessary actions without human intervention. These features collectively allow LUCI to autonomously adapt to varying user needs and preferences without frequent human intervention. The instances of failure in LUCI are primarily attributed to the inherent challenges of tasks that cannot be planned ahead. For instance, tasks like tic-tac-toe, where the LUCI has to make dynamic decision-making at each turn, and the outcome of the game is contingent on the opponent's moves. Unlike other tasks that have deterministic or predictable outcomes, tic-tac-toe requires adaptability and the ability to react to the changing state of the game. LUCI cannot accurately plan ahead because it cannot foresee the opponent's moves beyond the current turn, making the traditional pre-planning approach less effective.

*2) **Performance on Mind2Web**:* We showcase LUCI's applicability to real-world scenarios by testing it on Mind2Web [4]. For baseline comparisons we used MindACT with GPT-3.5, WebGUM [8], Gpt-4v [45]. The current SOTA in this benchmark is Gpt-4v [45] with oracle grounding but requires human annotations. It requires LMM to generate an action and then select the UI element based on the action. In our experiments, we directly select the UI element instead of generating action. LUCI with GPT-3.5-turbo achieves a Step success rate of 86.7 %, 89.1% and 84.2% across three test splits, respectively. As demonstrated in Table I, our approach performs significantly better than other methods across three test splits over every metric. Notably, it achieves at least 19% more in step success rate improvement over GPT-4(v) in all three settings using GPT-3.5. LUCI with Phi2 still performs admirably, demonstrating solid performance across various scenarios. It outperforms other models in most categories, showcasing the efficiency of LUCI with smaller LLMs in handling cross-task, cross-website, and cross-domain challenges

*3) **Performance of LUCI on GUI Applications**:* In this section, we assess the effectiveness of our approach in empowering the model to autonomously leverage GUI applications, without the need for additional supervision. The results of our experiments, depicted in Figure 8, showcase the performance of LUCI when integrated with GPT-3.5 under both the zero-shot and few-shot in-context settings. Specifically, under the zero-shot setting, where the language model relies solely on its pre-existing knowledge to generate a solution outline, LUCI achieves an average success rate of 58%. In contrast, under the few-shot setting with limited context, the average success rate significantly increases to 76.5%, with over 60% of applications achieving a success rate of at least 80%. LUCI exhibits a comparatively lower performance in PyCharm, primarily attributed to the language model's limitations in generating accurate and effective code. LUCI demonstrates good performance on desktop applications even under the zero-shot setting when compared to web applications. However, a significant decrease is observed in the performance of LUCI with web applications under the few-shot setting. This disparity may stem from the language model's training data, which potentially contains information on how to navigate and interact with desktop applications but lacks comprehensive guidance on web applications. These findings highlight the LUCI's adaptability to scenarios where the model encounters unfamiliar domains with just a few prompts.

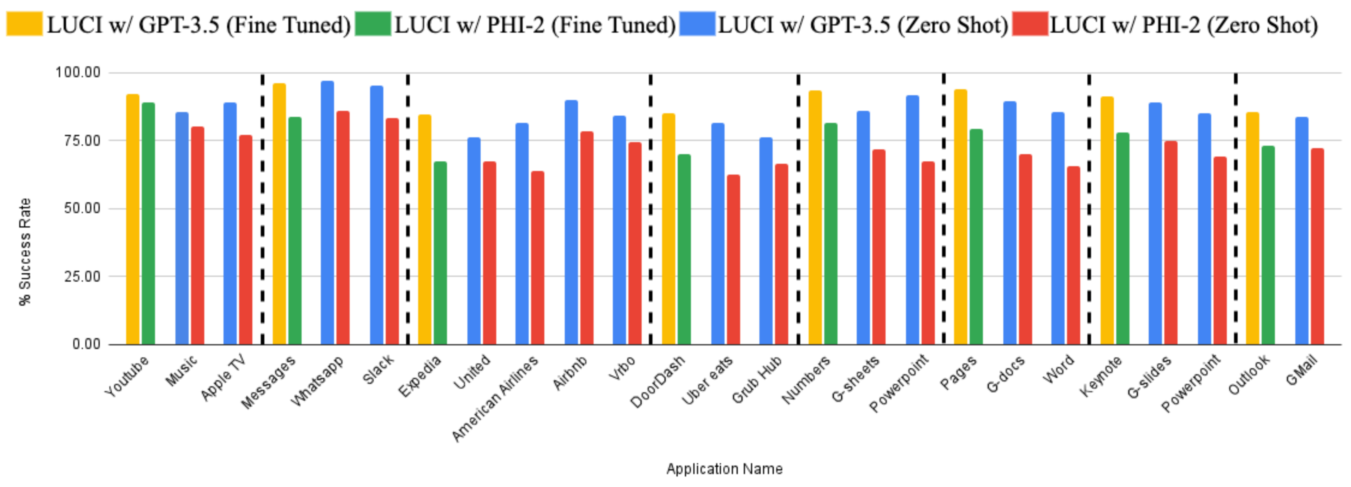


Fig. 9. Cross application performance of LUCI with GPT-3.5 and PHI-2. LUCI fine-tuned on an application that exhibits comparable performance on similar unseen applications. **LUCI can generalise to unseen environment.**

B. LUCI Enables Cross-Application Adaptability

In this section, we closely examine the cross-application performance of language models with LUCI. Here we fine-tune language models on a single application and subsequently evaluate their success rate on analogous applications within the same domains and task contexts. The models subjected to experimentation include GPT-3.5 Turbo and Phi-2. The objective of this investigation was to discern the adaptability of these agents when confronted with entirely new desktop or web applications, albeit within the familiarity of domains and task contexts they were originally fine-tuned for.

From Figure 9, it is observed that the models fine-tuned for a particular application exhibit a comparable success rate when tested on applications from the same domain. Quantitatively, the average deviation in performance is measured at 3.3 % for the GPT-3.5 Turbo setting and 4.5 % for the Phi-2 setting. This means that fine-tuning for a certain type of application helps the models do well on other applications in the same category.

C. LUCI can Utilize Multiple Applications for Executing Complex Tasks

Another noteworthy aspect of LUCI is its ability to carry out tasks that require the integration of multiple applications. In this section, we evaluate LUCI's proficiency in seamlessly orchestrating various applications to efficiently execute multi-faceted tasks, showcasing its potential for enhanced productivity and versatility in diverse user scenarios. To evaluate LUCI's capability to manage multiple applications, we created a set of hand-written tasks serving as seed examples and then, utilize ChatGPT to generate more tasks that require the utilization of one or more GUI applications listed in Figure 8. Then, these tasks with a number of GUI applications required to complete each task range from 1 to 6. In each case, at least 21 tasks are evaluated and 30 episodes to produce the results. Our key evaluation criterion is the success rate discussed in Section V-A, reflecting the agent's effectiveness in completing the assigned task.

Figure 10, delineates a trend wherein the success rate exhibits a gradual decline from 93.17% for tasks involving a single application to 79.36% when four applications are concurrently utilized. This trend underscores LUCI's commendable performance in handling tasks comprising up to four applications. However, surpassing this threshold, the success rates sharply decrease to 58.73%, indicating a substantial challenge for LUCI in managing tasks necessitating the simultaneous usage of more than five applications. These findings underscore the diminishing efficacy of LUCI with an increasing number of applications, implying complexities in its multitasking capabilities beyond a certain threshold.

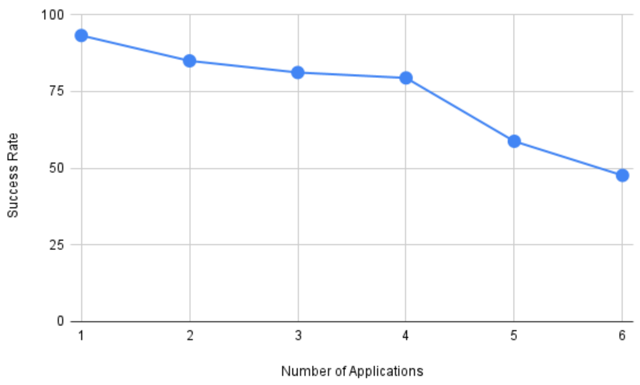


Fig. 10. Average success rate of LUCI across tasks involving the use of multiple applications. The trend shows LUCI's ability to use at least four applications without losing efficacy.

1) Task Ordering: A key observation from our tests was that the ordering of applications had a tangible impact on the success rate of the task. When complex applications such as Keynote were called later in the execution order, we noted an up to 20% decrease in the success rate. This effect can be attributed to the long-term attention limitations in LLMs, a fact that is further supported by the fact that the PHI2 variant showed a much starker decline in performance compared to GPT-3.5 Turbo. This implies that improvements in context length would directly impact the number of applications LUCI can orchestrate.

VI. Future work

LUCI is designed to simplify the creation and evaluation of versatile agents optimized for GUI tools. These agents show great potential in improving the accessibility and usability of GUI tools, especially for those who are unfamiliar with information technology or have impairments that may make it difficult to navigate complex tools or applications. Despite its potential benefits, there remain significant concerns and limits regarding present data gathering approaches, system design, and the necessary safety precautions for deployment in real-world circumstances.

Representation in Data: Our data and methodology have undergone evaluation for English instructions and user interfaces containing English text. In future, we would expand to different languages.

Use of Multimodal Information: LUCI, focuses on modeling the GUI environment into textual context from underlying hierarchy, neglecting other information such as images, videos e.t.c. This makes LUCI vulnerable to performs actions based on information other than text. Leveraging this multimodal information holds promise for enhancing model performance.

Tolerance to Noise: In LUCI, a solution outline is generate ahead of execution based on previous knowledge. Deviations of desktop or web application from the original user interfaces, often triggered by Pop-ups and Ads, result in errors as LUCI struggles to adjust to unexpected scenarios.

Safety Concerns: The development of general-purpose action agents holds the potential to enhance efficiency and user experiences but requires careful consideration of safety concerns. Key issues include managing sensitive actions, privacy-related activities, and the risk of breaching security measures. Amid these challenges, action agents pose a significant risk of breaching security involving authentication and authorization processes, including CAPTCHA, and may be exploited for malicious activities. A comprehensive approach is needed for responsible deployment, urging proactive cybersecurity research to develop preemptive protective measures.

VII. Conclusion

In this work, we introduced LLM assisted User Control Interface (LUCI), a computer agent that leverages the reasoning capabilities of LLMs, such as GPT-3.5 and PHI-2, to interact and control wide range of desktop and web applications to execute repetitive actions and solve complex tasks. LUCI addresses context-length issues as seen in previous methods by using IAF representations for UI elements across both native and web interfaces. This extends the capabilities of previous single platform approaches. Additionally, LUCI leverages a hierarchical structure enabling multi-application control. LUCI accomplishes all this while maintaining similar or up to 20% better performance on the benchmarks like MiniWoB++, Mind2Web.

References

[1] Michael Ahn et al. *Do As I Can, Not As I Say: Grounding Language in Robotic Affordances*. 2022. arXiv: 2204.01691 `[cs.RO]`.

[2] Hyung Won Chung et al. *Scaling Instruction-Finetuned Language Models*. 2022. arXiv: 2210.11416 `[cs.LG]`.

[3] Ishita Dasgupta et al. *Collaborating with language models for embodied reasoning*. 2023. arXiv: 2302.00763 `[cs.LG]`.

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

[5] Smit Desai, Tanusree Sharma, and Pratyasha Saha. "Using ChatGPT in HCI Research—A Trioethnography". In: *Proceedings of the 5th International Conference on Conversational User Interfaces*. CUI '23. ACM, July 2023. DOI: 10.1145/3571884.3603755. URL: http://dx.doi.org/10.1145/3571884.3603755.

[6] Alexey Dosovitskiy et al. *An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale*. 2021. arXiv: 2010.11929 `[cs.CV]`.

[7] Hiroki Furuta et al. *Multimodal Web Navigation with Instruction-Finetuned Foundation Models*. 2024. arXiv: 2305.11854 `[cs.LG]`.

[8] Hiroki Furuta et al. *Multimodal Web Navigation with Instruction-Finetuned Foundation Models*. 2024. arXiv: 2305.11854 `[cs.LG]`.

[9] Luyu Gao et al. *PAL: Program-aided Language Models*. 2023. arXiv: 2211.10435 `[cs.CL]`.

[10] Amelia Glaese et al. *Improving alignment of dialogue agents via targeted human judgements*. 2022. arXiv: 2209.14375 `[cs.LG]`.

[11] Yu Gu, Xiang Deng, and Yu Su. *Don't Generate, Discriminate: A Proposal for Grounding Language Models to Real-World Environments*. 2023. arXiv: 2212.09736 `[cs.CL]`.

[12] Izzeddin Gur et al. *Environment Generation for Zero-Shot Compositional Reinforcement Learning*. 2022. arXiv: 2201.08896 `[cs.LG]`.

[13] Izzeddin Gur et al. *Learning to Navigate the Web*. 2018. arXiv: 1812.09195 `[cs.LG]`.

[14] Izzeddin Gur et al. *Understanding HTML with Large Language Models*. 2023. arXiv: 2210.03945 `[cs.LG]`.

[15] Jordan Hoffmann et al. *Training Compute-Optimal Large Language Models*. 2022. arXiv: 2203.15556 `[cs.CL]`.

[16] Wenyi Hong et al. *CogAgent: A Visual Language Model for GUI Agents*. 2023. arXiv: 2312.08914 `[cs.CV]`.

[17] Wenlong Huang et al. *Inner Monologue: Embodied Reasoning through Planning with Language Models*. 2022. arXiv: 2207.05608 `[cs.RO]`.

[18] Wenlong Huang et al. *Language Models as Zero-Shot Planners: Extracting Actionable Knowledge for Embodied Agents*. 2022. arXiv: 2201.07207 `[cs.LG]`.

[19] Peter C Humphreys et al. *A data-driven approach for learning to control computers*. 2022. arXiv: 2202.08137 [cs.LG].

[20] Haris Isyanto, Ajib Setyo Arifin, and Muhammad Suryanegara. "Design and Implementation of IoT-Based Smart Home Voice Commands for disabled people using Google Assistant". In: *2020 International Conference on Smart Technology and Applications (ICoSTA)*. 2020, pp. 1–6. DOI: 10.1109/ICoSTA48221.2020.1570613925.

[21] Sheng Jia, Jamie Kiros, and Jimmy Ba. *DOM-Q-NET: Grounded RL on Structured Language*. 2019. arXiv: 1902.07257 [cs.LG].

[22] Geunwoo Kim, Pierre Baldi, and Stephen McAleer. *Language Models can Solve Computer Tasks*. 2023. arXiv: 2303.17491 [cs.CL].

[23] Takeshi Kojima et al. *Large Language Models are Zero-Shot Reasoners*. 2023. arXiv: 2205.11916 [cs.CL].

[24] Toby Jia-Jun Li, Amos Azaria, and Brad A. Myers. "SUGILITE: Creating Multimodal Smartphone Automation by Demonstration". In: *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*. CHI '17. Denver, Colorado, USA: Association for Computing Machinery, 2017, pp. 6038–6049. ISBN: 9781450346559. DOI: 10.1145/3025453.3025483. URL: https://doi.org/10.1145/3025453.3025483.

[25] Yaobo Liang et al. *TaskMatrix.AI: Completing Tasks by Connecting Foundation Models with Millions of APIs*. 2023. arXiv: 2303.16434 [cs.AI].

[26] Evan Zheran Liu et al. *Reinforcement Learning on Web Interfaces Using Workflow-Guided Exploration*. 2018. arXiv: 1802.08802 [cs.AI].

[27] Grégoire Mialon et al. *Augmented Language Models: a Survey*. 2023. arXiv: 2302.07842 [cs.CL].

[28] Reiichiro Nakano et al. *WebGPT: Browser-assisted question-answering with human feedback*. 2022. arXiv: 2112.09332 [cs.CL].

[29] Long Ouyang et al. *Training language models to follow instructions with human feedback*. 2022. arXiv: 2203.02155 [cs.CL].

[30] Bhargavi Paranjape et al. *ART: Automatic multi-step reasoning and tool-use for large language models*. 2023. arXiv: 2303.09014 [cs.CL].

[31] Timo Schick et al. *Toolformer: Language Models Can Teach Themselves to Use Tools*. 2023. arXiv: 2302.04761 [cs.CL].

[32] Peter Shaw et al. *From Pixels to UI Actions: Learning to Follow Instructions via Graphical User Interfaces*. 2023. arXiv: 2306.00245 [cs.LG].

[33] Tianlin Shi et al. "World of Bits: An Open-Domain Platform for Web-Based Agents". In: *International Conference on Machine Learning*. 2017. URL: https://api.semanticscholar.org/CorpusID:34953552.

[34] Haotian Sun et al. *AdaPlanner: Adaptive Planning from Feedback with Language Models*. 2023. arXiv: 2305.16653 [cs.CL].

[35] Ross Taylor et al. *Galactica: A Large Language Model for Science*. 2022. arXiv: 2211.09085 [cs.CL].

[36] Amrita S. Tulshan and Sudhir Namdeorao Dhage. "Survey on Virtual Assistant: Google Assistant, Siri, Cortana, Alexa". In: *Advances in Signal Processing and Intelligent Recognition Systems*. Ed. by Sabu M. Thampi et al. Singapore: Springer Singapore, 2019, pp. 190–201. ISBN: 978-981-13-5758-9.

[37] Sai Vemprala et al. *ChatGPT for Robotics: Design Principles and Model Abilities*. 2023. arXiv: 2306.17582 [cs.AI].

[38] Yizhong Wang et al. *Self-Instruct: Aligning Language Models with Self-Generated Instructions*. 2023. arXiv: 2212.10560 [cs.CL].

[39] Jason Wei et al. *Chain-of-Thought Prompting Elicits Reasoning in Large Language Models*. 2023. arXiv: 2201.11903 [cs.CL].

[40] Jason Wei et al. *Finetuned Language Models Are Zero-Shot Learners*. 2022. arXiv: 2109.01652 [cs.CL].

[41] Rui Yang et al. *GPT4Tools: Teaching Large Language Model to Use Tools via Self-instruction*. 2023. arXiv: 2305.18752 [cs.CV].

[42] Shunyu Yao et al. *WebShop: Towards Scalable Real-World Web Interaction with Grounded Language Agents*. 2023. arXiv: 2207.01206 [cs.CL].

[43] Junjie Ye et al. *A Comprehensive Capability Analysis of GPT-3 and GPT-3.5 Series Models*. 2023. arXiv: 2303.10420 [cs.CL].

[44] Andy Zeng et al. *Socratic Models: Composing Zero-Shot Multimodal Reasoning with Language*. 2022. arXiv: 2204.00598 [cs.CV].

[45] Boyuan Zheng et al. *GPT-4V(ision) is a Generalist Web Agent, if Grounded*. 2024. arXiv: 2401.01614 [cs.IR].

[46] Longtao Zheng et al. *Synapse: Trajectory-as-Exemplar Prompting with Memory for Computer Control*. 2024. arXiv: 2306.07863 [cs.AI].

[47] Shuyan Zhou et al. *WebArena: A Realistic Web Environment for Building Autonomous Agents*. 2023. arXiv: 2307.13854 [cs.AI].