LUCI: Lightweight UI Command Interface

Guna Lagudu
Arizona State University
Tempe, USA
glagudu@asu.edu

Vinayak Sharma
Arizona State University
Tempe, USA
Vinayak.Sharma@asu.edu

Aviral Shrivastava
Arizona State University
Tempe, USA
aviral.shrivastava@asu.edu

Abstract

Modern embedded systems are powered by increasingly powerful hardware and are increasingly reliant on Artificial Intelligence (AI) technologies for advanced capabilities. Large Language Models (LLMs) are now being widely used to enable the next generation of human-computer interaction. While LLMs have shown impressive task orchestration capabilities, their computation complexity has limited them to run on the cloud – which introduces internet dependency and additional latency. While smaller LLMs ($< 5B$ parameters) can run on modern embedded systems such as smartwatches and phones, their performance in UI-interaction and task orchestration remains poor. In this paper we introduce LUCI:Lightweight UI Command Interface. LUCI follows a separation of tasks structure by using a combination of LLM agents and algorithmic procedures to accomplish sub-tasks while using a high-level level LLM-Agent with rule-based checks to orchestrate the pipeline. LUCI addresses the limitations of previous In-Context learning approaches by incorporating a novel semantic information extraction mechanism that compresses the frontend code into a structured intermediate Information-Action-Field (IAF) representation. These IAF representations are then used by an Action Selection LLM. This compression allows LUCI to have a much larger effective context window along with better grounding due to the context information in IAF.

Pairing our multi-agent pipeline with our IAF representations allows LUCI to achieve similar task success rates as GPT-4Von the Mind2Web benchmark, while using 2.7B-parameter text-only PHI-2 model. When testing with GPT 3.5, LUCI shows a 20% improvement in task success rates over the state-of-the-art (SOTA) on the same benchmarks.

LCTES '25, Seoul, Republic of Korea

ACM ISBN 979-8-4007-1921-9/25/06
https://doi.org/10.1145/3735452.3735536

CCS Concepts: • **Human-centered computing** → **User interface management systems**; • **Theory of computation** → *Parsing*.

Keywords: LLM, Embedded Systems, System Automation

ACM Reference Format:
Guna Lagudu, Vinayak Sharma, and Aviral Shrivastava. 2025. LUCI: Lightweight UI Command Interface. In *Proceedings of the 26th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES '25), June 16–17, 2025, Seoul, Republic of Korea.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3735452.3735536

1 Introduction

In today's connected technology landscape, embedded systems play a critical role. Most Internet of Things(IoT) devices are powered by increasingly capable embedded systems. The prevalence of these smaller embedded devices, such as smartwatches and the newly burgeoning field of "smart glasses", requires a rethinking of what human-computer interaction looks like and what next-generation operating systems for embedded systems may look like. A key challenge here is the ability to efficiently bridge the gap between interfaces designed for larger devices that utilize touch or keyboard-/mouse input and the smaller devices that rely on voice or minimal gesture and button inputs.

Natural Language Processing (NLP) has been a key building block in these voice-first user interfaces, such as those found in smart speakers and smart watches, with digital assistants such as Google Assistant, Siri, and Alexa serving as the first interfaces for user interaction on voice-only devices. The advent of large language models (LLMs) has ushered a new era of voice command-focused devices, such as the Rabbit R1. We define '*voice command-focused devices*' as devices whose interface primarily relies on parsing and acting on voice commands from a user along with any additional context information such as images. However, while these devices have proved extremely powerful in their limited scope, they lack the ability to truly interact with the Web and serve as a true replacement for pure touch and keyboard/mouse interfaces.

A key challenge in this domain is the ability of an LLM to convert user prompts into actions. Most tools and websites have been designed for humans and create challenges for LLMs to navigate. The first approaches attempted to extend the coding capabilities of LLMs to work with APIs to perform tasks [3]. They adopted a Planner, Actor and Reporter

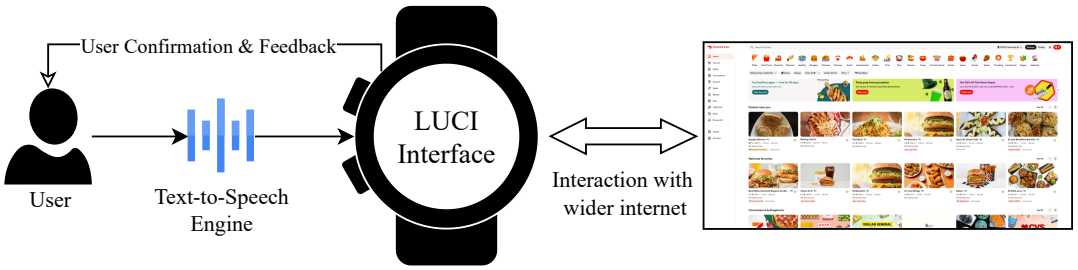

Figure 1. LUCI use-case diagram. LUCI would enable embedded devices to interact with a variety of applications and websites while migitating the limitations of their input mechanisms.

structure to provide the necessary context to the LLMs, i.e., grounding, allowing them to interact with the environment (computer systems). However, API-based methods required the creation of natural language instructions for the API of each application, which limited their generalization across multiple applications. *In-Context Learning*(ICL) was used to remedy this by providing the necessary context on relevant prompts rather than pre-training. However, this was limited by the context length of the LLMs.

Simultaneously, another avenue of research was LLM frameworks [4] to manipulate the graphic user interface (GUI). These focused on teaching LLMs to interact with the GUI front-end of applications, forgoing the need for specific APIs and leading to better generalization. Early Reinforcement Learning (RL) on Hypertext Markup Language(HTML) struggled with the selection of relevant UI elements that required supervised training. In this work, our aim is to remedy these challenges and create a lightweight LLM GUI control framework.

In order to remedy challenges with identifying UI elements from HTML, a multi-modal model was introduced that operated on both text and image inputs such as GPT 4V [23]. These models were able to achieve state-of-the-art performance on the Mind2Web benchmark. However, these models were limited by their large size and the need for fine-tuning on a large multimodal dataset. This large size makes it impossible to run these models locally on current and near-term embedded systems, leading to internet dependency. A direct consequence of this dependency are continuous server costs and latency issues.

We introduce a novel framework - Lightweight UI Command Interface (LUCI), that enables multi-application GUI-based orchestration via an ensemble of lightweight LLMs. The LUCI framework is designed to be modular, hierarchical, and OS-agnostic, enabling it to work seamlessly across both native and web interfaces. This is achieved by using a separation of responsibilities model to decompose tasks into sub-tasks and recursively solve them with a variety of LLM and rule-based components. Our approach is based on the insights of multi-agent based LLM pipelines [8] and the Mixture of Experts models [14]. LUCI uses a set of fine-tuned models with specialized prompts to handle different aspects of the system such as high-level planning, tool selection and action selection. The information flow between these models is shown in Figure 2. LUCI adopts an application centric approach to planning, where the system first identifies the most relevant application that will be needed to address the given task prompt. This both greatly simplifies the planning as well as serves as effective grounding for downstream tasks. A '*conversational model*' then generates sub-tasks for the prompt based on the selected applications. These subtasks are processed iteratively. Additionally, in order to efficiently parse the front-end code from applications, we developed a rule-based semantic parser which we refer to as '*UI Extractor*'. This parser compresses the code into an intermediate representation known as Information-Action-Field (IAF) format. This allows LUCI to greatly increase the effective attention window of the lightweight LLMs. Overall, the key contributions of LUCI can be summarized as:

1. An application-centric planning framework for fast and efficient model grounding for adaptable multi-application task planning.
2. A modular OS-agnostic framework capable of scaling across native and web interfaces.
3. A rule-based semantic parser for efficient compression of front-end code into structured IAF representations, allowing for larger effective attention windows and better task grounding.
4. A multi-agent framework for task orchestration, enabling LUCI to achieve state-of-the-art performance on the Mind2Web benchmark while using lightweight LLMs.

Our experiments show that LUCI achieves a 99 + % success rate on the MiniWoB++ benchmark. Additionally, LUCI achieves an upto 31% improvement in Step SR and 24% improvement in OP F1 Score over GPT-4V on the Mind2Web benchmark while using GPT 3.5. When using the 2.7B parameter PHI-2 model, LUCI demonstrates 10% increase in step SR and 7.9% increase in OP F1 score over GPT 4V while 10^3 times fewer parameters 1.8- Trillion (GPT 4V) vs 3-Billion parameters (PHI-2).

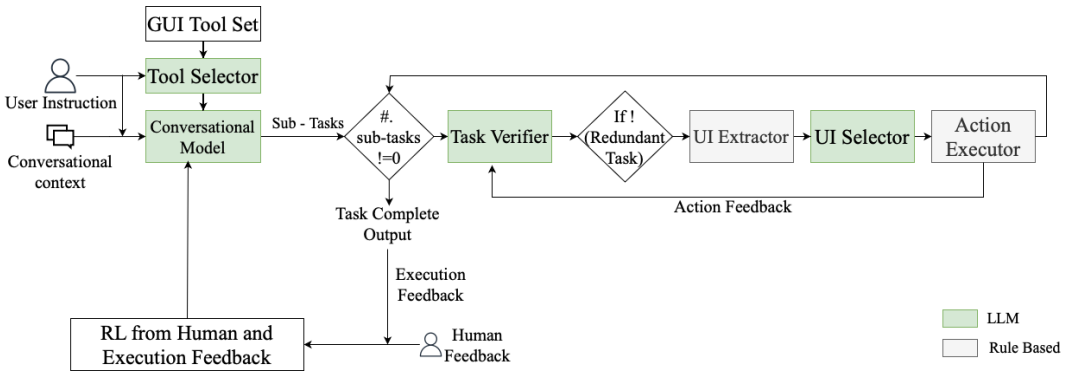

Figure 2. Architecture of LUCI. Given user instruction and the conversational context, the Tool Selector first selects a tool from the given GUI toolset and opens the applications. The conversational model then generates a solution outline, which is a text description of the list of sub-tasks needed 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.

2 Related Work

2.1 Building Agents With LLMs

Large language models (LLMs) have offered promising avenues for leveraging natural language in decision-making tasks.Huang et al. [11] 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 [20] and the action's value function as an indicator of suitability. Huang et al. [12] 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. [22] 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 [10] 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 [6].

2.2 Automated GUI Tasks

Recent research [9] proposes employing large language models (LLMs) to read HTML code and vision transformers 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.

Numerous ongoing initiatives are dedicated to the development of computer agents and benchmarks. Among the sophisticated benchmarks available, such as Mind2Web [4] and Webshop [21]. 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 [5] and Gpt-4V [23]. WebGUM [5] 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 requiring a lot of expert demonstrations for fine-tuning. Gpt-4V [23] uses LMMs to observe both HTML and captured screenshots to generate actions but still lacks sufficient visual perception abilities to serve as an effective agent.

3 LUCI Architecture

The LUCI framework is composed of 7 major components consisting of both rule-based and LLM components. The flow of information between these components is highlighted in Figure 2. The components are as follows:

1. **GUI Tool Set**, which contains a list of GUI tools authorized to be controlled by the LLM.
2. **Tool Selector**, which selects GUI tools from the GUI toolset required to accomplish given user instructions.
3. **Conversational Model**, which is responsible for interacting with users and generating a solution outline based on the GUI tool selected by the tool selector, user instruction along 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 an action on selected UI elements based on the type of UI element and returns the action feedback.

A more detailed exploration of each component is provided in the following sections.

3.1 Prompt Components

The prompt examples of the various components of LUCI consist of 5 entites- (1). User Prompt: These are the requests from the user, (2). System Prompt: these are system generated pre-formatted prompts used to ground and provide context to the LLMs, (3). System : algorithmic components used to call other components of LUCI, (4). System Response: The responses from other system components, (5). Response: The response of the LLM to the system prompt. Additionally, the when passing values to the LLMs, the variables (marked by $) are passed as values.

3.2 GUI Tool Set ($\mathcal{G}$)

The GUI tool set ($\mathcal{G}$) is the list of applications that LUCI can access. It is provided to the Tool selector as a set $\mathcal{G} = \{\mathcal{G}_1, \mathcal{G}_2, \mathcal{G}_3 ...\}$. Our tests consider a set of 22 diverse applications. Additionally, including web browsers expands the functionality via access to web apps.

3.3 Tool selector (TS)

The Tool Selector is an LLM model that selects the set of most relevant applications from $\mathcal{G}$ based on the given user instruction. The model is inherently grounded due to providing $\mathcal{G}$. Additionally, the tool selector can return 2 special responses | (1). *"Tool Not Found"* when none of the applications can address the prompt and (2). *"No Tool Required"* for tasks such as text summarization which can be handled directly by the LLM. If duplicate tools exist, user preference is considered. An example is shown in Prompt Example. 1

Prompt Example: 1. Tool Selector functionality example using a valid and invalid example

System_Prompt:> GUI Tool Set (G): {Calculator, Notepad, Browser, Outlook}
User_Prompt:> Send an email to johndoe@gmail.com
System_Prompt:> Given toolset G, select tool to accomplish "Send an email to johndoe@gmail.com"
Response:> Tool Required: Outlook
User_Prompt:> Order a pizza from dominos
System_Prompt:> Given toolset G, select tool to accomplish "Order a pizza from dominos"
Response:> Tool Not Found

3.4 Conversational Model (CM)

The Conversational Model(CM) is the plan orchestrator of LUCI. It uses an LLM to generate a solution outline based on the selected GUI tool from the Tool Selector, user instruction, and conversational context. The CM utlizes Chain-of-Thought prompting to incorporate the relevant information in the solution outline.

3.4.1 CM Outline Structure. In order to maintain context across multiple applications, the CM outlines follow a strict structure. Each response contains 3 fields:

- **Tool Required** [True/False] : Used to identify if the sub-task requires a GUI tool. Tasks such as text summarization do not require an application and can be solved using the inherent capabilities of the CM LLM.
- **Tool** [Application Used] : The application used to solve the sub-task. If '*Tool Required*' flag is false, returns 'No tool required'
- **Task description** [Sub task description] : The instruction to be executed for the sub-task.

The solution outline is then processed my the LUCI system, where sub-tasks are divided into 3 types:

1. **Application Launching Tasks**: Tasks that open or switch applications.
2. **Application Interaction Tasks** : Tasks that manipulate the user interface of an application or website.
3. **Direct LLM Tasks**: Tasks that are directly performed by the LLM.

Prompt Example: 2. Coversational Model functionality

User_Prompt:> Calculate wages earned for 50 minutes of work at $10/hr
System:> Query Tool selector for relevant tools and store in $G
System_Response:> $G = {Calculator}
System_Prompt:> Given toolset G, generate solution outline for user prompt
Response:>
1. Tool Required:True, Tool: Calculator, Task description: Open Calculator

2. Tool Required:True, Tool: Calculator, Task description: Convert 50 minutes to hours by dividing by 60
3. Tool Required:True, Tool: Calculator, Task description: Press "=" key
4. Tool Required:True, Tool: Calculator, Task description: Multiply results with 10
5. Tool Required:True, Tool: Calculator, Task description: Press "=" key
6. Tool Required:True, Tool: Calculator, Task description: Copy results and return to user

3.5 Task Looper

The system component responsible for extracting each sub-task from the CM and looping over them. The task looper is an algorithmic component.

3.6 Task Verifier

The first step in the processing of every sub-task is the Task Verifier. The Task Verifier uses a fine-tuned LLM to evaluate if the current sub-task is necessary or redundant based on previously executed actions and future sub-tasks. Based on the the context the Task Verifier also emits an '*execution feedback*' to further refine the CM.

Prompt Example: 3. Task Verifier example for Example 2 task 2

System:> Fetches Action feedback for executed tasks into $A, fetches future tasks into $SF and marks current sub–task as $S

System_Response:> variables $S,$A,$SF and $i

System_Prompt:> Check whether the current sub–task $S is necessary for the objective based on previously executed tasks $A and future tasks $SF.

Response:> Answer: Yes, Reason: Converting 50 minutes to hours is necessary for the objective.

The task verifier is critical in errors caused due to ambiguity in certain user interfaces. For example, task 3 from Example 2 is redundant as the calculator might auto calculate and pressing the "=" might cause the operation to run again, leading to an error. The task verifier is responsible for catching such errors. The functionlity of the Task Verifier is contingent upon the '*action feedback*' is generated by the action executor and consists of information about the actions performed during that sub-task execution.

3.7 UI Extractor

The UI extractor is a rule-based parser that extracts UI elements from web and desktop applications into a '*Information-Action-Field(IAF)*' representation. A UI extractor can be created for any front-end language such as XML, HTML, etc. LUCI has 2 existsing extractors for HTML and SwiftUI. The tree structure of frond-end languages is leveraged to extract categorize and extact elements based on the following 3 types:

- **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. Examples: buttons, hyperlinks, submit buttons e.t.c.
- **Field Elements (F)**: UI elements that collect user input. Example: textbox, checkbox, radio buttons e.t.c.

Using these 3 categorizations, we make the following assumptions to further reduce the complexity of the tree structure:

- User input is sent to the 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.

The front-end is then recursively passed based on these assumptions to create the IAF representation seen below-

$I[I1, < A1, F1 >, I2, < A2, F2 >, I3, < A3, F3 >, ...]$

I1, I2, I3,..... contains all text information in a given node. $< An, Fn >$ is a set of A and its corresponding F pairs.

3.8 UI Selector

The UI selector is an LLM based component that sections the appropriate UI element from a set of IAF representations based on the subtask description. The design of the UI selector is motivated by recent studies [2, 7] that show training language models for discrimination rather than generation is more effective for generalizability and sample efficiency for grounding tasks.

The use of a discriminative selector grounded to a specific tasks allows LUCI to improve performance compared to the one-shot generative approaches of prior models. If a UI element is not found, a feedback signal is sent to the conversational model, triggering a revision of the current sub-task.

Prompt Example: 4. UI Selector example for sub task 2 in Example 2

System:> Fetches UI element IAFs into $E, fetches the sub–task description into $T

System_Response:> variables $E and $T
System_Prompt:> From the given list of UI elements $E, select the list of elements required for task $T
Response:> Answer: [5,0,/,6,0,=]

3.9 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.

4 Features of LUCI

4.1 Application Centric Architecture

As seen in Figure. 2, the tool selector is the first component in our architecture. The conversational model generates subtasks 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.

4.2 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.

4.3 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.

4.4 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.

4.5 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.

5 Experiments and Results

We explore the performance of LUCI using both the GPT-3.5 and PHI-2 model backends. The performance of GPT-3.5 variant showcases LUCI's ability to scale and achieve SOTA performance, while the PHI-2 results demonstrate LUCI's efficiency and suitability for embedded systems.

5.1 Testing Methodology

All the tests are conducted on a mix of established benchmarks (MiniWoB++ [17], Mind2Web [4]) and real world experiments. The benchmarks are used to demonstrate the efficiency and performance of LUCI compared to other approaches. In contrast real world tests are used to showcase the adaptability and robustness of LUCI in real world scenarios comparing fine-tuned and zero-shot results.

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. Two test sets are used - Cross-Task, and Cross-Domain to evaluate generalizability over tasks from the same, and completely unseen domains, respectively.

5.2 LUCI With PHI-2 Competes With Much Larger Models

The optimizations and inference pipeline of LUCI allows it to achieve comparable results to GPT-4V [23] while using the PHI-2 model that only uses 2.7B parameters. This is demonstrated by the performance on the MiniWoB++ Benchmark (shown in Figure. 3) and the Mind2Web Benchmark (shown in 1). These results showcase the effectiveness of the LUCI framework in optimally harnessing the capabilities of LLMs. An important note from our results is that while Synapse outperforms LUCI w/ PHI-2 on the highly saturated MiniWoB++ environment, LUCI w/ PHI-2 shows 2× the performance of Synapse on the Mind2Web. We can attribute this to the older and simpler nature of MiniWoB++ tasks where most models achieve high levels of performance compared to the more complex Mind2Web tasks.

LUCI w/ PHI-2 also outperforms the nearly 100× larger GPT-4V model on the Mind2Web benchmark. These results clearly show that LUCI /w PHI-2 can be used as a powerful on device task orchestrator with performance comparable to much larger cloud based models. Therefore, LUCI w/ PHI-2 can unlock new possibilities for input constrained embedded systems such as smartwatches.

5.3 LUCI Scales Up to Achieve SOTA Performance on Complex Tasks

The efficiency of LUCI also allows it to scale when using a heavier LLM backend. This can be seen in the results showing that LUCI /w GPT-3.5-turbo can achieve state of the art (SOTA) performance on both the MiniWoB++ [17] and Mind2Web [4] benchmarks.

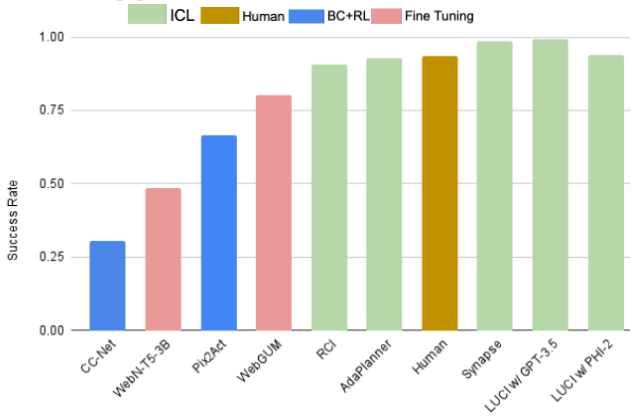

Figure 3. Average performance comparison with baselines in MiniWoB++ environment. LUCI w/ PHI-2 is competitive with all other models while using only 2.7B parameters. When scaled up to use GPT3.5-turbo LUCI achieves state-of-the-art performance.

5.3.1 Performance on MiniWoB++ Task Suite. The MiniWoB++ benchmark consists of simple tasks like click-checkboxes, and text-complete for computer agents to simulate simple human-computer interactions. In our MiniWoB++ experiments, we performed experiments on 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 [15] 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 [15]. The success rate is determined by the proportion of successful episodes, wherein the agent receives a positive reward.

To comprehensively demonstrate LUCI's performance, we compared it against a variety of baselines across multiple methodologies. To evaluate against Behavior Cloning (BC) and Reinforcement Learning (RL) baselines, we used CC-Net [13] and Pix2Act [16]. WebGUM [5] and WebN-T5 [9] were used to evaluate the effectiveness against fine-tuned models trained originally on a large general corpus on text. Synapse [24], RCI [15] and AdaPlanner [18], were used to evaluate the effectiveness against in-context learning (ICL) methods. Additionally, we included human scores from Humphreys et al. [13] for supplementary benchmarking.

Figure 3 illustrates the average performance of different methods across Miniwob++ benchmark. LUCI, utilizing PHI-2 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 [24]. **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 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.

5.3.2 Performance on Mind2Web. 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, and domains.

Table 1. Average performance of different methods on Mind2Web Benchmark. LUCI w/ GPT-3.5 achieves state-of-the-art performance.

Baseline	Cross-Task		Cross-Domain	
	Op. F1	Step SR	Op. F1	Step SR
MINDACT	56.6	17.4	52.8	18.6
Synapse	-	30.6	-	26.4
WebGUM	75.9	64.9	77.7	66.7
GPT-4V	80.9	65.7	73.6	62.1
LUCI				
w/ GPT-3.5	93.8	86.7	91.7	84.2
w/ PHI-2	82.3	72.8	79.4	69.1

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 [5], Gpt-4V [23]. The current SOTA in this benchmark is Gpt-4V [23] 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 1, our approach performs significantly better than other methods across the 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 PHI-2 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

5.3.3 Performance of LUCI on GUI Applications. 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 4. 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 [19]. 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.

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 4, 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

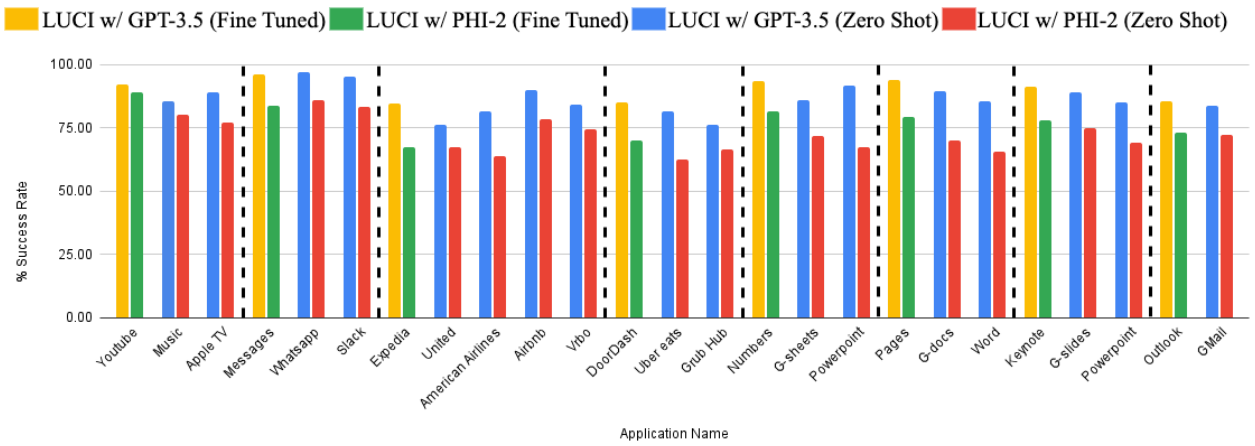

Figure 4. Cross application performance of LUCI with GPT-3.5 and PHI-2. LUCI fine-tuned on an application that exhibits comparable performance on similar unseen applications. **LUCI can generalise to unseen environment.**

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.

5.4 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 4, 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.

5.5 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 multifaceted 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 4. 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 5.3, reflecting the agent's effectiveness in completing the assigned task.

Figure 5, 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.

5.5.1 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 PHI-2 variant showed a much starker decline in performance compared to GPT-3.5 Turbo. This implies that

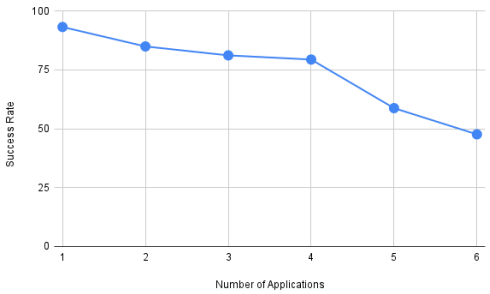

Figure 5. Average success rate of LUCI across tasks involving the use of multiple applications. The trend shows LUCI's ability to use at least four applications without losing efficacy.

improvements in context length would directly impact the number of applications LUCI can orchestrate.

6 Conclusion

In this work, we introduce LUCI, a multi-agent framework for task orchestration across web and native user interfaces using lightweight LLMs. LUCI presents a new direction for embedded system interaction and control using on-device lightweight PHI-2 models over internet connected heavy models. This would allow embedded systems to achieve a higher level of security and privacy while brining down latency and the dependency on ongoing server compute.

We demonstrated LUCIs abilities to match and surpass SOTA techniques such as GPT4-V despite only using a much lighter weight text only model like PHI-2. We additionally showcased LUCI's ability to adapt to unknown interfaces in both cross task and cross application cases.

Acknowledgments

This work was partially supported by funding from National Science Foundation grants CPS 1645578 and CCF 2436016, Semiconductor Research Corporation (SRC) project 3154.

References

[1] Michael Ahn et al. 2022. 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. 2022. Scaling instruction-finetuned language models. (2022). arXiv: 2210.11416 [cs.LG].

[3] Ishita Dasgupta, Christine Kaeser-Chen, Kenneth Marino, Arun Ahuja, Sheila Babayan, Felix Hill, and Rob Fergus. 2023. Collaborating with language models for embodied reasoning. (2023). arXiv: 2302.00763 [cs.LG].

[4] Xiang Deng, Yu Gu, Boyuan Zheng, Shijie Chen, Samuel Stevens, Boshi Wang, Huan Sun, and Yu Su. 2023. Mind2web: towards a generalist agent for the web. (2023). arXiv: 2306.06070 [cs.CL].

[5] Hiroki Furuta, Kuang-Huei Lee, Ofir Nachum, Yutaka Matsuo, Aleksandra Faust, Shixiang Shane Gu, and Izzeddin Gur. 2024. Multimodal web navigation with instruction-finetuned foundation models. (2024). arXiv: 2305.11854 [cs.LG].

[6] Amelia Glaese et al. 2022. Improving alignment of dialogue agents via targeted human judgements. (2022). arXiv: 2209.14375 [cs.LG].

[7] Yu Gu, Xiang Deng, and Yu Su. 2023. Don't generate, discriminate: a proposal for grounding language models to real-world environments. (2023). arXiv: 2212.09736 [cs.CL].

[8] Taicheng Guo, Xiuying Chen, Yaqi Wang, Ruidi Chang, Shichao Pei, Nitesh V. Chawla, Olaf Wiest, and Xiangliang Zhang. 2024. Large language model based multi-agents: a survey of progress and challenges. (2024). https://arxiv.org/abs/2402.01680 arXiv: 2402.01680 [cs.CL].

[9] Izzeddin Gur, Ofir Nachum, Yingjie Miao, Mustafa Safdari, Austin Huang, Aakanksha Chowdhery, Sharan Narang, Noah Fiedel, and Aleksandra Faust. 2023. Understanding html with large language models. (2023). arXiv: 2210.03945 [cs.LG].

[10] Jordan Hoffmann et al. 2022. Training compute-optimal large language models. (2022). arXiv: 2203.15556 [cs.CL].

[11] Wenlong Huang, Pieter Abbeel, Deepak Pathak, and Igor Mordatch. 2022. Language models as zero-shot planners: extracting actionable knowledge for embodied agents. (2022). arXiv: 2201.07207 [cs.LG].

[12] Wenlong Huang et al. 2022. Inner monologue: embodied reasoning through planning with language models. (2022). arXiv: 2207.05608 [cs.RO].

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

[14] Robert A Jacobs, Michael I Jordan, Steven J Nowlan, and Geoffrey E Hinton. 1991. Adaptive mixtures of local experts. *Neural computation*, 3, 1, 79–87.

[15] Geunwoo Kim, Pierre Baldi, and Stephen McAleer. 2023. Language models can solve computer tasks. (2023). arXiv: 2303.17491 [cs.CL].

[16] Peter Shaw, Mandar Joshi, James Cohan, Jonathan Berant, Panupong Pasupat, Hexiang Hu, Urvashi Khandelwal, Kenton Lee, and Kristina Toutanova. 2023. From pixels to ui actions: learning to follow instructions via graphical user interfaces. (2023). arXiv: 2306.00245 [cs.LG].

[17] Tianlin Shi, Andrej Karpathy, Linxi (Jim) Fan, Josefa Z. Hernández, and Percy Liang. 2017. World of bits: an open-domain platform for web-based agents. In *International Conference on Machine Learning*. https://api.semanticscholar.org/CorpusID:34953552.

[18] Haotian Sun, Yuchen Zhuang, Lingkai Kong, Bo Dai, and Chao Zhang. 2023. Adaplanner: adaptive planning from feedback with language models. (2023). arXiv: 2305.16653 [cs.CL].

[19] Yizhong Wang, Yeganeh Kordi, Swaroop Mishra, Alisa Liu, Noah A. Smith, Daniel Khashabi, and Hannaneh Hajishirzi. 2023. Self-instruct: aligning language models with self-generated instructions. (2023). arXiv: 2212.10560 [cs.CL].

[20] Jason Wei, Maarten Bosma, Vincent Y. Zhao, Kelvin Guu, Adams Wei Yu, Brian Lester, Nan Du, Andrew M. Dai, and Quoc V. Le. 2022. Finetuned language models are zero-shot learners. (2022). arXiv: 2109.01652 [cs.CL].

[21] Shunyu Yao, Howard Chen, John Yang, and Karthik Narasimhan. 2023. Webshop: towards scalable real-world web interaction with grounded language agents. (2023). arXiv: 2207.01206 [cs.CL].

[22] Andy Zeng et al. 2022. Socratic models: composing zero-shot multimodal reasoning with language. (2022). arXiv: 2204.00598 [cs.CV].

[23] Boyuan Zheng, Boyu Gou, Jihyung Kil, Huan Sun, and Yu Su. 2024. Gpt-4v(ision) is a generalist web agent, if grounded. (2024). arXiv: 2401.01614 [cs.IR].

[24] Longtao Zheng, Rundong Wang, Xinrun Wang, and Bo An. 2024. Synapse: trajectory-as-exemplar prompting with memory for computer control. (2024). arXiv: 2306.07863 [cs.AI].

Received 2025-03-13; accepted 2025-04-21