Large Action Models: From Inception to Implementation

Anonymous authors
Paper under double-blind review

Abstract

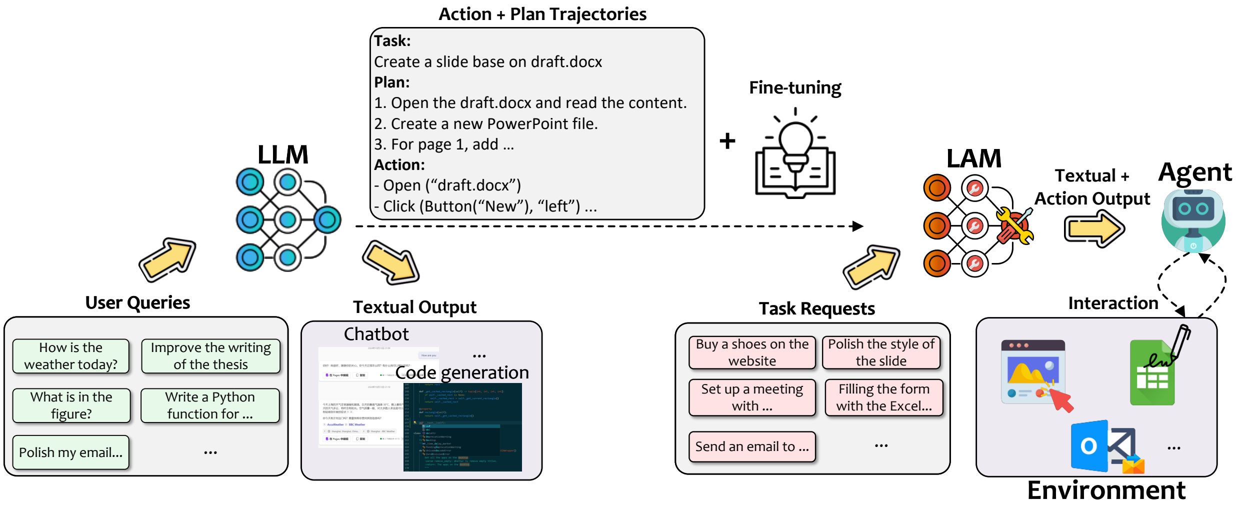
As AI continues to advance, there is a growing demand for systems that go beyond language-based assistance and move toward intelligent agents capable of performing real-world actions. This evolution requires the transition from traditional Large Language Models (LLMs), which excel at generating textual responses, to Large Action Models (LAMs), designed for action generation and execution within dynamic environments. Enabled by agent systems, LAMs hold the potential to transform AI from passive language understanding to active task completion, marking a significant milestone in the progression toward artificial general intelligence.

In this paper, we present a comprehensive framework for developing LAMs, offering a systematic approach to their creation, from inception to deployment. We begin with an overview of LAMs, highlighting their unique characteristics and delineating their differences from LLMs. Using a Windows OS-based agent as a case study, we provide a detailed, step-by-step guide on the key stages of LAM development, including data collection, model training, environment integration, grounding, and evaluation. This generalizable workflow can serve as a blueprint for creating functional LAMs in various application domains. We conclude by identifying the current limitations of LAMs and discussing directions for future research and industrial deployment, emphasizing the challenges and opportunities that lie ahead in realizing the full potential of LAMs in real-world applications.

1 Introduction

In recent years, large language models (LLMs) have demonstrated remarkable advancements across a range of natural language processing (NLP) tasks Wei et al. (2021); Brown (2020); Yang et al. (2023b). These models, often incorporating multiple modalities such as language, vision, and speech, have become foundational in numerous AI-driven applications Thirunavukarasu et al. (2023); Rubenstein et al. (2023); Wang et al. (2024c); Jiang et al. (2024). Their success is evident in systems like question answering in conversational agents Ma et al. (2023), code generation in GitHub Copilot Yetiştiren et al. (2023), and improved search capabilities in platforms like Bing Thomas et al. (2024). The key strengths of LLMs—namely their vast knowledge, ability to support multimodal inputs, and capacity for human-like responses—have propelled them to the forefront of AI research Minaee et al. (2024). Their capability to generalize via zero-shot learning has further expanded the horizons of what AI systems can achieve, making significant contributions to the productivity of both everyday tasks and specialized professional activities. These innovations mark an important milestone on the path toward artificial general intelligence (AGI) Feng et al..

However, while LLMs excel in generating intricate textual responses, they are often constrained by their inability to directly interact with or manipulate the physical world Wang et al. (2024b). In many real-world applications, intelligent systems need to perform tasks that go beyond conversational exchanges—tasks that involve tangible actions Gao et al. (2024). The maxim “actions speak louder than words” Pennycook (1985) underscores the limitations of purely text-based interactions, as users increasingly expect intelligent agents to go beyond passive responses and engage in real-world actions. For instance, a truly transformative AI assistant could automate tasks in software applications, manage household chores, or even engage with children in meaningful ways. The realization of such capabilities would mark a revolutionary shift in how we integrate AI into our daily lives, enabling widespread automation and augmenting human capabilities across diverse environments Ruan et al. (2023).


Action + Plan Trajectories
Fine-tuning
+
LLM
LAM
Textual +
Action Output
Agent
User Queries
How is the weather today?
Improve the writing of the thesis
What is in the figure?
Write a Python function for ...
Polish my email...
...
Textual Output
Chatbot
Code generation
...
Task Requests
Buy a shoes on the website
Polish the style of the slide
Set up a meeting with ...
Filling the form with the Excel...
Send an email to ...
...
Interaction
Environment
...

Task:
Create a slide base on draft.docx
Plan:
1. Open the draft.docx and read the content.
2. Create a new PowerPoint file.
3. For page 1, add ...
Action:
- Open ("draft.docx")
- Click (Button("New"), "left") ...

Figure 1: The transition from LLMs to LAMs.

Achieving this vision requires LLMs to extend their expertise from language processing to action generation. However, this transition is not straightforward. While leading LLMs from industry giants have demonstrated impressive performance in language-based tasks, they encounter substantial limitations when tasked with action generation Yao et al. (2020). Completing a task in the real world involves a sequence of complex steps: accurately understanding user intent, devising a plan, and executing the necessary actions Kalakonda et al. (2023). Current LLMs may excel at understanding and planning in textual form but often fall short when required to produce actionable outputs. This is particularly true in scenarios that demand precise task decomposition, long-term planning Ding et al. (2023); Zhang et al. (2024e), and the coordination of multi-step actions Valmeekam et al. (2022). Furthermore, LLMs are generally optimized for broad, general-purpose tasks rather than tailored for specific scenarios or environments. This lack of specialization can result in suboptimal performance, especially when interacting with unfamiliar or dynamic environments where adaptive and robust action sequences are essential Ling et al. (2023).

These limitations highlight a significant gap in the ability of LLMs to transition from passive understanding to active, real-world engagement. To address these challenges, the development of Large Action Models (LAMs) represents a transformative shift in AI capabilities He et al. (2024). Unlike traditional LLMs that primarily focus on text generation and response, LAMs are designed to perform actions in both physical and digital environments. These models are capable of interpreting user intentions from diverse data inputs, automating complex processes, planning for task completion, and interacting with the world via agents. This evolution marks a critical step toward a future where intelligent systems not only comprehend human language but can also translate that understanding into tangible, meaningful actions Zhang et al. (2024c).

LAMs are often built upon the foundation of LLMs, but the transition from LLMs to LAMs is neither straightforward nor seamless, as shown in Figure 1. The process of transforming an LLM into a functional LAM involves multiple intricate stages, each requiring substantial effort and expertise. First, it is essential to collect comprehensive datasets that capture user requests, environmental states, and corresponding actions Deng et al. (2024). These data serve as the basis for training or fine-tuning LLMs to perform actions rather than merely generate text. This stage involves the integration of advanced training techniques that enable the model to understand and execute actions within specific environments Hong et al. (2024). Once the LAM has been trained, it must be incorporated into an agent system that can effectively interact with its environment. This system typically includes components for gathering observations, utilizing tools, maintaining memory, and implementing feedback loops. These components are critical for ensuring that the LAM can not only execute actions but also adapt its behavior based on real-time feedback and evolving situations Zhang et al. (2024a). The integration of these elements enhances the LAM's capacity to perform tasks autonomously, interact meaningfully with its surroundings, and make decisions that are grounded in the context of its environment.

A final but crucial step in the development of LAMs is evaluation Xie et al. (2024). Before deploying a LAM for real-world applications, it is imperative to rigorously assess its reliability, robustness, and safety. Unlike LLMs, which may be limited to generating text-based outputs, LAMs have the capacity to directly affect their environment through actions. This introduces new risks, as incorrect or inappropriate actions could have significant consequences. Therefore, thorough evaluation processes are essential to ensure that both the LAM and its accompanying agent are capable of making reliable decisions while minimizing potential risks. These evaluations often involve testing the model in a variety of scenarios to ensure that it can generalize across different environments and tasks, as well as effectively handle unexpected situations.

Given the complexity involved in developing LAMs, the purpose of this paper is to provide a comprehensive understanding of LAMs and guide practitioners in transforming an LLM into a functional LAM for real-world applications. To this end, we first present an overview of LAMs, clarifying their distinctions from traditional LLMs and discussing their unique characteristics. By offering this foundational knowledge, we aim to give readers a clear conceptual understanding of LAMs, enabling them to grasp the broader implications of their development and use.

Next, we delve into the practical process of obtaining a LAM from scratch. Using a Graphical User Interface (GUI) agent on Windows OS as an example, we provide a detailed, step-by-step exploration of the entire pipeline—beginning with data collection and preparation, followed by model training, integration, and grounding. This includes how to prepare datasets that capture user requests, environmental states, and actions, as well as how to fine-tune LLMs to generate executable actions rather than text responses. We also demonstrate how to integrate a trained LAM into an agent system, equipping it with tools, memory, and feedback mechanisms to enable dynamic interaction with its environment. The final stages focus on rigorous evaluation, ensuring that the LAM is robust, safe, and capable of handling real-world tasks. While this paper uses the Windows OS as a case study, the methodology outlined can be adapted to other environments, providing a generalizable workflow for obtaining functional LAMs. Finally, we address several limitations and challenges faced by LAMs in both research and industry. While LAMs represent a significant advancement over traditional LLMs, they are still in an early stage of development and present substantial areas for improvement. Issues such as privacy concerns, latency, safety risks, scalability, and ethical considerations all pose challenges that must be addressed for LAMs to be fully realized as practical tools.

The emergence of LAMs represents not merely an incremental advancement over LLMs, but a fundamental shift from passive language processing to active, real-world engagement. By executing actions, LAMs can interact dynamically with both digital and physical environments, marking a transformative milestone in the broader pursuit of AGI. We envision this paper as a foundational guide to LAMs, offering both theoretical insights and practical, actionable steps for creating and deploying LAMs in real-world scenarios.

2 Large Action Models 101

Large Action Models (LAMs) represent a significant advancement in artificial intelligence, extending the capabilities of Large Language Models (LLMs) Zhang et al. (2024c). While LLMs are proficient at generating human-like text based on user inputs, LAMs go beyond text generation by performing actions in both physical and digital environments Zeng et al. (2023). These models interpret user intentions from various data forms, automate entire processes as per user requirements, plan for task completion, and interact with the world. This evolution signifies a shift from mere language interaction to action sequences that are grounded in real-world contexts.

2.1 Large Language Models

LLMs are neural networks with billions to hundreds of billions of parameters, trained on extensive text corpora to address general-purpose language tasks Kasneci et al. (2023); Xu et al. (2022); Zhang et al. (2024b); Zhu et al. (2023); Liu et al. (2024a). These models demonstrate exceptional capabilities in natural language understanding and generation, allowing them to perform complex tasks such as answering questions Jiang et al. (2021), generating code Zhang et al. (2023a), and providing human-like textual responses Dasgupta et al. (2022) with minimal task-specific training, known as zero-shot Wei et al. (2021) or few-shot Brown

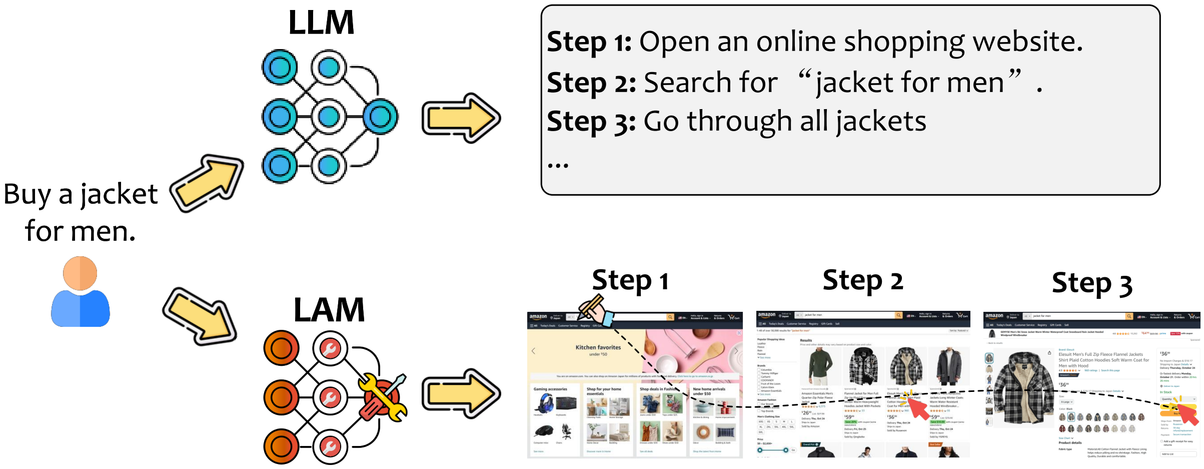

LLM
Buy a jacket for men.
Step 1: Open an online shopping website.
Step 2: Search for "jacket for men".
Step 3: Go through all jackets
...
LAM
Step 1
Step 2
Step 3

Figure 2: The objective difference between LLMs and LAMs.

(2020) learning. Unlike traditional language models, which required extensive task-specific data and training, LLMs leverage their vast knowledge base to generalize across diverse tasks with minimal supervision.

While LLMs possess significant language understanding and generation capabilities, they are primarily limited to generating text-based outputs. They excel at interacting with users and generating text, but they lack the ability to directly interface with environments to execute actions. This limitation restricts their applicability in scenarios that require tangible interaction with digital or physical environments.

To extend their utility, LLMs are often embedded within agent frameworks Wang et al. (2024b). These agent systems augment LLMs, enabling them to interact with dynamic environments by collecting data from various sources Xi et al. (2023), structuring it into meaningful inputs Kim et al. (2024), and prompting the LLM for inference Xi et al. (2023). The agent then interprets the model's output—whether in the form of code Wang et al. (2024d) or tool-based actions Ruan et al. (2023)—and grounds it within the environment by executing actions and collecting feedback Shinn et al. (2024). Agents equipped with LLMs typically function in a loop, continuously gathering environmental information, using LLM inference to form plans, executing those plans, and refining future actions based on feedback. This iterative process can incorporate external memory systems, enabling the agent to track historical actions and environmental states, further improving the decision-making process over time Zhang et al. (2024f); Hu & Lu (2024).

2.2 From LLMs to LAMs

LAMs build upon the foundational capabilities of LLMs but are specifically optimized for action-oriented tasks. They are designed to perform actions in both physical and digital environments, interpreting user intentions from various data forms, automating processes as per user requirements, planning for task completion, and interacting with the world Zeng et al. (2023). This evolution signifies a shift from passive language interaction to generating action sequences that are grounded in real-world contexts.

An illustrative example is shown in Figure 2. An LLM can comprehend a user's request to purchase a jacket and generate a detailed textual plan or recommendation, but it cannot autonomously complete the transaction on a website. In contrast, a LAM leverages this foundational understanding to generate action sequences that directly interact with the website, completing the request on the user's behalf. This ability to transition from understanding to execution bridges the gap between the model and real-world applications, moving beyond mere language output to tangible outcomes.

Furthermore, due to their specialization in specific domains or tasks, LAMs can be smaller in scale compared to general-purpose LLMs while achieving comparable or superior performance within their operational scope. By focusing on a narrower range of tasks, LAMs prioritize efficiency and effectiveness, leveraging targeted data and optimized architectures to reduce computational overhead without sacrificing capability. This specialization not only makes LAMs more practical for deployment in real-world applications but also opens opportunities for developing lightweight models that can operate in resource-constrained environments.

The evolution from LLMs to LAMs is achieved through specialized training and integration with agent systems. These systems enable LAMs to translate their inferences into real-world actions, bridging the gap between understanding and execution. Thus, LAMs not only enhance the functionality of LLMs but also redefine their applicability in real-world scenarios.

2.3 Key Characteristics of LAMs

LAMs are distinguished by advanced capabilities that enable them to perform complex tasks effectively. These characteristics include:

2.3.1 Interpretation of User Intentions

A fundamental capability of LAMs is the ability to accurately interpret user intentions from diverse forms of input. These inputs may include natural language requests, voice commands, images, or videos, such as device screenshots or instructional videos Cheng et al. (2024). User inputs are often abstract or implicit Chen et al. (2024a), requiring LAMs to leverage their internal knowledge and complementary information to discern the true intent behind the input. This process involves understanding nuances, disambiguating instructions, and inferring unstated objectives. LAMs must translate these user intentions into actionable plans and steps, facilitating subsequent interactions with the environment to fulfill the user's objectives. This requires a robust foundation in LLMs, particularly those with multi-round conversational capabilities Shah et al. (2023), enhancing LAMs' proficiency in engaging with users to accurately understand and execute their requests.

2.3.2 Action Generation

The hallmark feature of LAMs is their capacity for action generation grounded in the environment. LAMs translate user intentions into actionable steps that can be executed within specific contexts. These actions can take various forms: operations on graphical user interface (GUI) elements, API calls for software applications, physical manipulations performed by robots, invoking other AI agents or models, or autonomously generating code or combining meta-actions Carta et al. (2023). By incorporating detailed knowledge of the environment, including available actions, system states, and expected inputs, LAMs can select appropriate actions and apply them correctly to meet user requests. This involves not only executing predefined actions but also adapting to new situations by generating novel action sequences when necessary.

2.3.3 Dynamic Planning and Adaptation

LAMs exhibit a sophisticated capability for dynamic planning and adaptation, which is crucial for handling complex user requests that span multiple steps Guan et al. (2023). They can decompose a complex task into several subtasks, each further broken down into specific action steps. This hierarchical planning enables LAMs to approach task execution with a forward-looking perspective, anticipating future requirements and potential obstacles. Moreover, as the execution of each action alters the state of the environment, LAM will react to these changes, adapting and revising their plans and actions accordingly Shinn et al. (2024). This flexibility ensures robustness in dynamic scenarios where deviations from initial expectations are common. For instance, if an unexpected error occurs or a resource becomes unavailable, a LAM can replan and adjust its actions to still achieve the desired outcome.

2.3.4 Specialization and Efficiency

LAMs are fine-tuned for executing specialized sequences of actions within specific environments Cheng et al. (2024). By focusing on particular domains, LAMs achieve a high degree of accuracy and adaptability, outperforming general-purpose LLMs in targeted applications. This specialization allows LAMs to encode comprehensive knowledge about the environment deeply into their architecture, including available actions, system constraints, and contextual nuances. As a result, LAMs can operate more efficiently, reducing

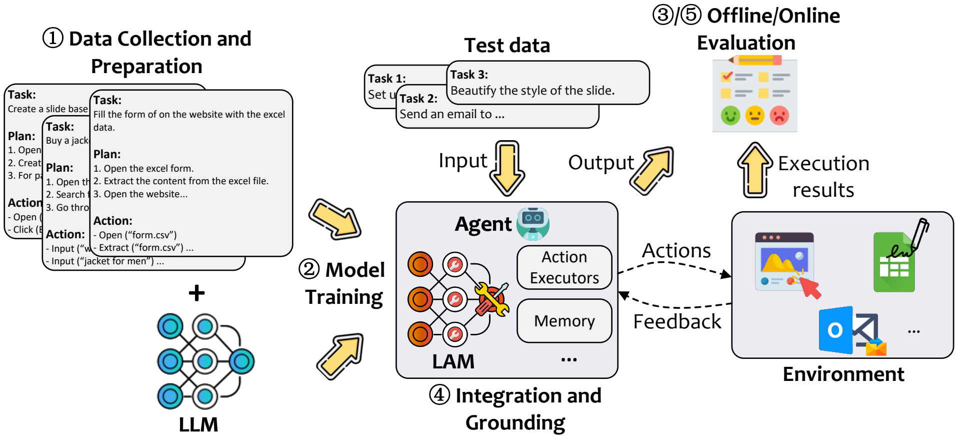


Figure 3: The process pipeline for LAM development and implementation.

computational overhead and improving response times. Furthermore, since LAMs are expected to complete actionable tasks within a more limited scope, their scale can be smaller compared to general-purpose LLMs while achieving a comparable level of performance within that specific domain. This makes LAMs more practical for deployment in real-world applications, including resource-constrained environments such as edge devices or local systems.

2.3.5 Summary

In summary, LAMs transcend the basic functionality of converting user requests into a series of steps by comprehending the underlying logic that interconnects and contextualizes these actions. They understand sequence dependencies—why certain steps must precede or follow others—and recognize when to adapt the plan to accommodate changing circumstances. LAMs extend AI systems into the realm of actionable intelligence. This significantly enhances their ability to autonomously perform complex, real-world tasks, making them invaluable in applications requiring precise interaction and manipulation within defined operational contexts.

2.4 From Inception to Implementation

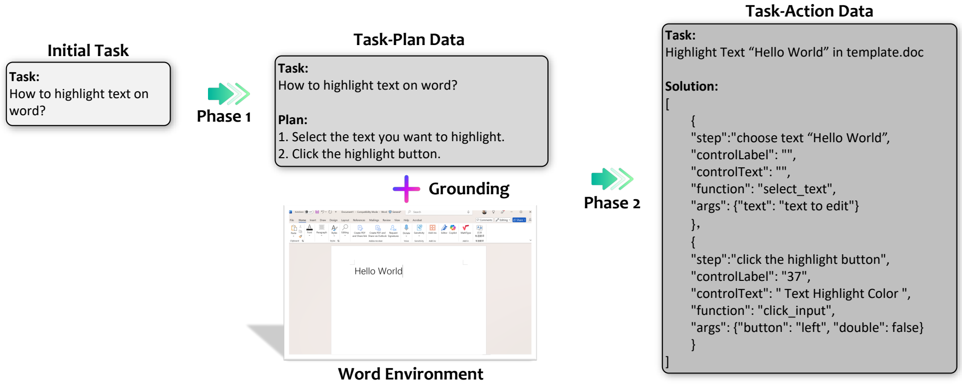
LAMs have the potential to significantly extend the impact of LLMs by enabling tangible interactions with real-world environments. To harness this potential, an LAM must be developed from the ground up and deployed within a real-world application, allowing it to operate effectively in a physical environment. This process involves 5 critical steps, as shown in Figure 3:

1. **Data Collection and Preparation (Section 3)**: The first step involves gathering and curating the necessary data for the specific use case. This includes not only user queries but also environmental context, potential actions, and any other relevant data required to train the LAM effectively. The data must undergo cleaning and pre-processing before it is used for training or fine-tuning a LAM.

2. **Model Training (Section 4)**: Using the prepared data, the next step is to train the LAM. This training process can involve various techniques such as supervised fine-tuning and reinforcement learning to ensure the model can perform the desired actions accurately and efficiently.

3. **Offline Evaluation (Section 5):** After obtaining the LAM, we evaluate its performance using an offline dataset to verify its reliability in a controlled, static environment.

4. **Integration and Grounding (Section 6)**: The LAM is integrated into an agent framework that serves as its operational platform. This involves grounding the model with the ability to interact with external tools, maintain memory, and interface with the environment. By equipping the LAM with these capabilities, it becomes capable of making meaningful impacts in the physical world.



Task:
How to highlight text on word?

Task:
How to highlight text on word?

Plan:
1. Select the text you want to highlight.
2. Click the highlight button.

Task:
Highlight Text “Hello World” in template.doc

Solution:

```
[
    {
    "step":"choose text “Hello World”,
    "controlLabel": "",
    "controlText": "",
    "function": "select_text",
    "args": {"text": "text to edit"}
    },
    {
    "step":"click the highlight button",
    "controlLabel": "37",
    "controlText": " Text Highlight Color ",
    "function": "click_input",
    "args": {"button": "left", "double": false}
    }
]
```

Figure 4: The two-phrase data collection and preparation process.

5. **Online Evaluation (Section 7)**: Finally, the performance of the LAM must be rigorously evaluated in the real environment from multiple perspectives, including accuracy, efficiency, and effectiveness in completing tasks. This step is crucial to ensure that the LAM functions as intended and meets the desired operational standards.

Through these steps, LAMs can be effectively developed and deployed to bring LLMs' capabilities into real-world applications, enabling them to interact with and manipulate the physical environment, thereby making a tangible impact.

In the following sections, we use the Windows GUI agent UFO Zhang et al. (2024a)[1] as a case study to illustrate the process of building a robust LAM from the ground up. This LAM will serve as the core inference engine for UFO, enabling it to autonomously fulfill user requests within the Windows OS environment. While this example focuses on a Windows GUI agent, the outlined steps can be adapted for developing LAMs in other scenarios or for different applications.

3 Data Collection and Preparation

Data is a cornerstone in training LLMs, where high-quality data significantly enhances their performance Wang et al. (2023); Li et al. (2024). Similarly, LAMs require well-prepared, high-quality action-oriented data during the supervised fine-tuning phase. Off-the-shelf LLMs often face challenges when interacting with real-world environments. These difficulties typically arise from either a lack of domain-specific knowledge or the generation of hallucinated outputs that fail to be actionable. To mitigate these issues, we adopt a two-phase data collection approach: *task-plan collection* and *task-action collection*, as shown in Figure 4. Specifically:

1. **Task-Plan Data Collection:** In this phase, we collect data consisting of tasks and their corresponding plans. Tasks are user requests expressed in natural language, while plans are detailed, step-by-step procedures designed to fulfill these requests. For example, a task such as *“How to change the font size in Word?”* would have a corresponding plan outlining the steps required to complete the task. This data is used to fine-tune the model to generate effective plans and improve its high-level reasoning and planning capabilities. However, task-plan data cannot be directly executed in the environment, requiring the following data conversion phase.

2. **Task-Action Data Collection:** In this phase, the task-plan data is converted into task-action data, which includes tasks, plans, and the associated action sequences needed to execute those plans. Tasks and plans are refined to become more concrete and grounded within a specific environment. Action sequences are generated at this stage, such as `select_text(`

[1] `https://github.com/microsoft/UFO`

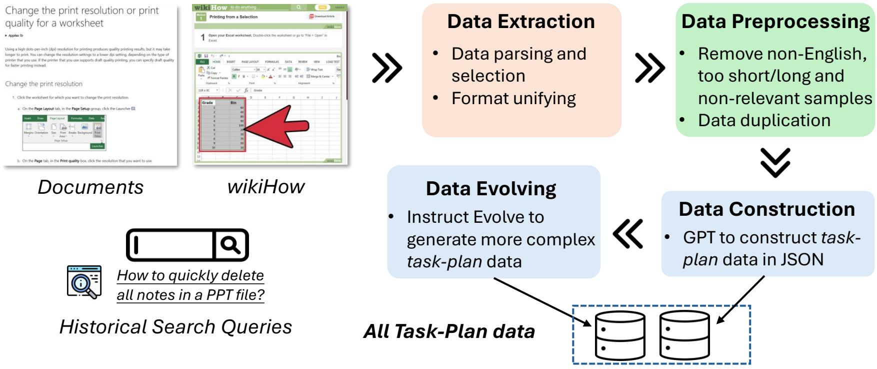

Change the print resolution or print quality for a worksheet
Documents
wikiHow
How to quickly delete all notes in a PPT file?
Historical Search Queries
Data Extraction
• Data parsing and selection
• Format unifying
Data Preprocessing
• Remove non-English, too short/long and non-relevant samples
• Data duplication
Data Construction
• GPT to construct task-plan data in JSON
Data Evolving
• Instruct Evolve to generate more complex task-plan data
All Task-Plan data

Figure 5: The pipeline to construct the task plan data.

text="hello") or click(on=Button("20"), how="left", double=False), which represent actionable instructions capable of directly interacting with the environment. This enriched data provides the necessary granularity for training an LAM to perform reliable and accurate task executions in real-world scenarios.

The task-plan data aims at enhancing the model's high-level planning capabilities, allowing it to generate detailed, step-by-step plans based on user requests. Meanwhile, the task-action data focuses on refining the model's ability to execute these plans by converting each planned step into a concrete, executable step or sequence while considering environmental feedback. The data collection and preparation pipeline ensures that the model is capable of both high-level planning and low-level action execution, thereby bridging the gap between natural language plans and executable actions.

In the following sections, we detail the methodologies employed for data collection, pre-processing, and integration of task-plan and task-action data. We illustrate how these steps enable the LLM to LAM transformation.

3.1 Task-Plan Data

Figure 5 outlines a multi-step pipeline for collecting and processing task-plan data, essential for training LAMs. The process begins with gathering raw data from diverse sources, including application documentation, WikiHow, and historical search queries. This is followed by structured pre-processing to ensure that the data is high-quality and relevant to specific tasks.

3.1.1 Data Sources

1. **Application Documentation:** Documentation and usage manuals for software applications provide authoritative task descriptions. These resources, maintained by product teams, are considered highly reliable. Relevant documentation, such as M365 documentation[2], is crawled, with outdated or inaccessible pages being filtered out. The HTML content is converted into markdown format, and GPT-4o is used to extract task-plan pairs in the desired structured format.

2. **WikiHow:** WikiHow[3] hosts a wide range of how-to articles, including application-specific operational guides. Webpages related to Windows platform applications are crawled, and GPT-4o extracts task and plan components, ensuring the resulting data aligns with the desired structured format.

3. **Historical Search Queries:** Search engine logs provide insight into real user demands, addressing gaps not covered by formal documentation. From Bing search logs, a 1% sample of queries mentioning application names (*e.g.*, Word, Excel, PowerPoint) from the past year was taken.

[2] https://learn.microsoft.com/en-us/microsoft-365/?view=o365-worldwide
[3] https://www.wikihow.com/Main-Page

3.1.2 Data Extraction and Pre-Processing

The initial step in processing raw data involves parsing to extract task-relevant content while filtering out unnecessary or irrelevant information. This includes removing non-English entries, samples that are excessively short or long based on predefined heuristics, and data unrelated to actionable tasks (*e.g.*, content focused on smartphone operations). The filtered data is then standardized into a unified format for further processing.

3.1.3 Data Construction

To create structured JSON samples, GPT-4o is employed to extract and format tasks along with their associated plans. For historical search queries, synthetic data is generated to enrich the raw input, addressing the common issue of insufficient context. GPT-4o reformulates these queries into complete, sentence-like user requests, ensuring consistency across all data sources and facilitating effective downstream processing.

The resulting dataset contains structured JSON samples, with each entry including a unique task identifier (task_id), the task description (task), and a step-by-step plan (plan). An example is shown below:

```
{"task_id": "word_032",
 "task": "Add a border to a page in Word",
 "plan": [
    1. Go to Design > Page Borders.
    2. Make selections for how you want the border to look.
    3. To adjust the distance between the border and the edge of the page,
       select Options. Make your changes and select OK.
    4. Select OK.
    ]
}
```

With the above process, we initially collected a total of 29,182 task-plan data samples.

3.1.4 Data Evolving

With the initial dataset processed, we employ data augmentation techniques to enhance its diversity and complexity. Inspired by WizardLM Xu et al. (2023) and AgentGen Hu et al. (2024), we use GPT-4o to evolve the raw task to generate new task-plan pairs, improving the model's ability to follow instructions and handle more complex tasks.

The data evolving process generates new tasks from the original ones by introducing additional complexity, constraints, or steps while preserving relevance. The guidelines for task evolution are as follows:

- The evolved task must be executable step-by-step on a Windows OS or application.
- The evolved task should include additional requirements, increasing its complexity without exceeding 20 extra words.
- The evolved task must remain concise and related to the original task.

For each evolved task, GPT-4o generates a corresponding plan adhering to the following guidelines:

- The plan must provide correct and actionable steps for Windows environments or applications.
- The plan should be concise and highlight critical action objects using bold emphasis.

This augmentation process results in a richer dataset where tasks become progressively more challenging, and plans incorporate domain-specific knowledge. For example:

Raw task: Create a drop-down list in Excel for Office 365.

Evolved Task: Create a dependent drop-down list in Excel for Office 365, where selecting an item from the first list filters options in the second list.

Evolved Plan:

- Prepare your data by organizing it into two columns. The first column contains items for the primary drop-down list, and the second column contains items for the dependent list.
- Name your ranges for the first and second lists.
- Create the primary drop-down list using Data Validation.
- Use the INDIRECT function to create the dependent drop-down list linked to the first selection.
- . . .

Using data augmentation, we increased the original task-plan dataset by 150%, generating a larger pool of samples. This augmentation significantly enhances the diversity and complexity of the dataset, allowing the model to learn from a broader range of scenarios and develop robust planning capabilities. The augmented data introduces more challenging tasks and detailed plans, further enriching the training process and enabling the LAM to handle complex real-world applications effectively.

3.2 Task-Action Data

The task-plan data collected in the previous stage provides high-level, step-by-step plans for resolving user-requested tasks, serving as general guidelines. However, these plans are textual and not directly executable in a real-world environment. For instance, a task-plan data sample for the task “Highlight text in document” outlines the necessary steps but does not translate into actionable instructions for interacting with the application’s GUI. This gap highlights the need for actionable task-action data to bridge the divide between planning and execution. To enable LAMs to produce actionable outputs, we generate task-action data derived from the previously collected task-plan data. Task-action data captures the granular interactions required to complete a task in the application environment, including GUI navigation, button clicks, and responding to environmental feedback.

Traditional approaches for action data collection often involve manual or agent-based annotation for each task, which is both costly and labor-intensive. To address these limitations, we propose an efficient, fully automated, and low-cost pipeline that leverages LLMs and real-world application interactions. This pipeline consists of four stages, as depicted in Figure 6: **Instantiation**, **Execution**, **Evaluation**, and **Post-Processing**. Specifically,

1. **Instantiation:** In this stage, the task-plan data is transformed into an executable trajectory. Using an LLM, each task is instantiated with specific operational objects, and related high-level plan is instantiated into a concrete sequence of actions that can be directly executed in the application environment.
2. **Execution:** The instantiated trajectory is then executed within the real-world application environment. During this stage, the system interacts with the application’s GUI to carry out the specified actions. For example, the instantiated trajectory for highlighting text would involve selecting the appropriate text, navigating to the highlight tool, and applying the highlight. The result of this execution is the captured executed trajectory, including any feedback or environmental changes observed during the process.
3. **Evaluation:** Once the execution is complete, the trajectory is evaluated for correctness using an LLM. The evaluation stage verifies whether the executed trajectory successfully accomplishes the intended task. This involves comparing the observed outcomes with the expected results outlined in the task-plan data. Tasks that fail to meet the criteria are flagged for review, while successful executions are retained for further processing.

4. **Post-Processing:** In the final stage, successful task-action trajectories undergo post-processing to ensure consistency, completeness, and readiness for training. This includes refining the data format, ensuring compatibility with the training pipeline, and annotating the data with relevant metadata (*e.g.*, task IDs, execution time, and step-by-step feedback). The post-processed task-action data is then added to the training dataset, enabling the LAM to learn from real-world interactions.

The pipeline minimizes human intervention and reduces the number of LLM calls required, significantly improving scalability and efficiency.

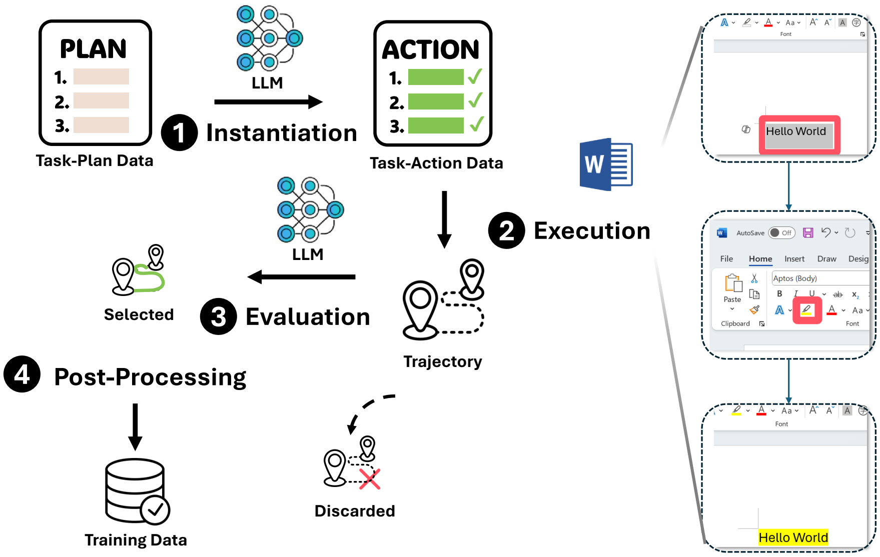


Figure 6: The pipeline of task-action data conversion and collection.

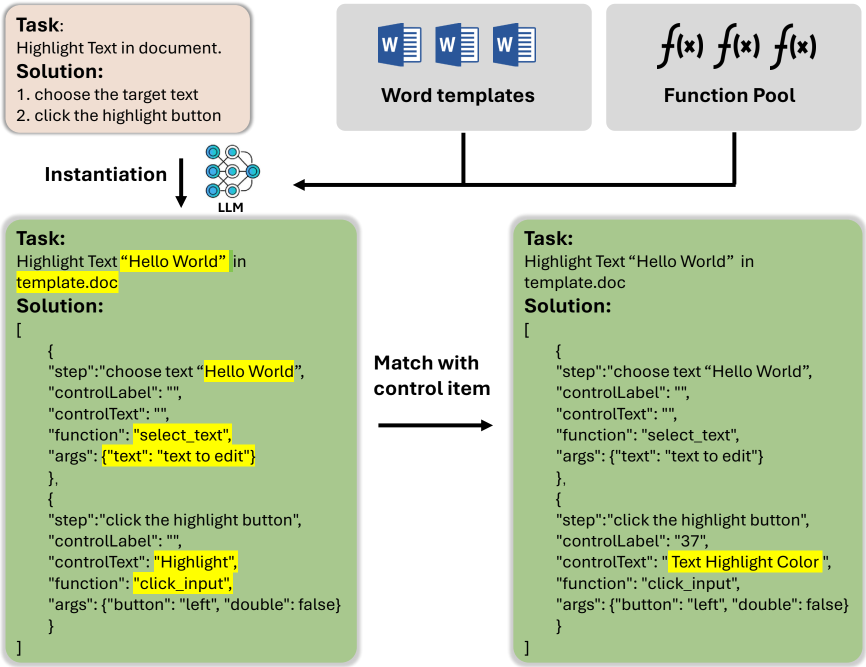
3.2.1 Instantiation

The task-plan data are primarily collected from help documents or public websites, creating a gap between the generalized task-plan data and the specific requirements needed for execution within a particular environment. A common issue is the lack of specificity. For instance, the task *"highlight text in document"* does not specify actionable objects, such as *"which text"* or *"which document"*. This lack of detail poses significant challenges in executing tasks within real-world applications.

To address this problem, we instantiate the task-plan data to impute target objects and related functions. First, we prepare template Word files to serve as specific targets for the actions. These template files include various Word components such as paragraphs, tables, and figures. Each template file is accompanied by a description indicating its content, providing context for grounding actions. Several sample template files can be found in Appendix A.

Given a task-plan data sample, the task description is matched with the template file descriptions to select an appropriate template file as the target for actions. GPT-4 is then prompted to instantiate the task-plan with target objects present in the selected template file (detailed prompts can be found in Appendix B.1). Simultaneously, we filter relevant functions from the available function pool using the task description, allowing the instantiation process to populate the task-action data with specific functions and their input parameters.

As a result of this process, the task description becomes more concrete and grounded in a specific environment, while the corresponding action sequences needed to complete the task are generated. Figure 7 provides an example of the instantiation process. Notably, the task-action data is not directly generated with GPT-4 due to the risk of hallucinations. Instead, instantiating grounded task-plan data ensures the generation of more reliable and faithful step-by-step actions.



Task:
Highlight Text in document.
Solution:
1. choose the target text
2. click the highlight button

Task:
Highlight Text “Hello World” in template.doc
Solution:

```
[
      {
      "step":"choose text “Hello World”,
      "controlLabel": "",
      "controlText": "",
      "function": "select_text",
      "args": {"text": "text to edit"}
      },
      {
      "step":"click the highlight button",
      "controlLabel": "",
      "controlText": "Highlight",
      "function": "click_input",
      "args": {"button": "left", "double": false}
      }
]
```

Task:
Highlight Text “Hello World” in template.doc
Solution:

```
[
      {
      "step":"choose text “Hello World”,
      "controlLabel": "",
      "controlText": "",
      "function": "select_text",
      "args": {"text": "text to edit"}
      },
      {
      "step":"click the highlight button",
      "controlLabel": "37",
      "controlText": " Text Highlight Color ",
      "function": "click_input",
      "args": {"button": "left", "double": false}
      }
]
```

Figure 7: An example of task instantiation.

3.2.2 Execution

To ensure that the steps in the instantiated task-plan data are accurate and truly actionable, the execution stage verifies the action sequence by matching control items with the real application environment and performing the specified actions. This process validates the task-action data, ensuring its correctness and compatibility with the application GUI.

For instance, as shown in Figure 7, the control item *“Text Highlight Color”* with its associated control label is retrieved using the action text *“Highlight”* from the control item pool. The corresponding task-action data is then executed in the application without further intervention from the LLM. During execution, if an error occurs (*e.g.*, a mismatch between the predicted control item and the actual environment), the instantiated task is discarded. Conversely, if all actions in the task execute successfully, the action-validated task is forwarded to the evaluation stage described in the following section. Additionally, screenshots of the application environment are captured after each step in the execution process, forming a detailed trajectory to assist in subsequent evaluation.

It is important to note that the instantiated task-action data is not guaranteed to be valid. Since the data is generated through a single GPT-4 call based on task-plan data, it lacks the step-by-step refinement that might be necessary for certain tasks. In some cases, execution results from previous steps are required to instantiate subsequent steps accurately. In such scenarios, the one-call instantiated task-action data may fail in validation and is removed from the dataset. This execution stage bridges the gap between planning and action, ensuring that task-action data is actionable, robust, and aligned with real-world application requirements.

3.2.3 Evaluation

Even if the task-action data is successfully executed in the real application without errors, further evaluation is required to ensure its validity. Some tasks may be incorrectly instantiated from the task-plan data, resulting in trajectories that, while executable, do not fulfill the original task description. Similarly, the executed results might fail to align with the intended task outcomes. For evaluation, we utilize the instantiated task along with its execution trajectory, which includes:

- Consecutive actions performed during execution.
- Screenshots captured before and after each action.
- Environmental changes observed between the initial and final states[4].

Using this comprehensive trajectory, we prompt GPT-4o to evaluate whether the executed task aligns with the original task description and achieves successful completion. The evaluation considers both the sequence of actions and the resulting application state. The process assigns a "task-complete" key to indicate the outcome as "yes," "no," or "unsure." If the task is evaluated as `"yes"`, the trajectory is deemed successful; otherwise, it is classified as a failure. The detailed prompt used for this evaluation is provided in Appendix B.2. This evaluation step ensures that only valid, accurate task-action data is included in the training dataset, contributing to the reliability and robustness of the LAM.

3.2.4 Post-Processing

As noted in Section 3.2.2, a trajectory was recorded during the execution process. This trajectory includes:

- Screenshots captured at each step.
- Environment states before and after each action.
- Plans and corresponding actions for every step.

During the post-processing stage, these trajectories are combined with the original task requests to generate synthetic step-wise training data. The resulting data format uses the task request as input and LAM's plan and actions as output. This structured format is critical for training LAMs to map task requests to actionable sequences effectively. The detailed template for the data format can be found in Appendix C.

4 Model Training

Our objective is to develop an LAM from scratch that can map user inputs to appropriate plans and executable actions, ultimately enabling complex task completion. To achieve this, we adopt a staged training strategy consisting of four phases, each building upon the previous one. As illustrated in Figure 8, these phases guide the model from learning structured task plans, to imitating expert demonstrations, to self-boosting from its own successes, and finally leveraging reward-based optimization. Throughout these stages, the model progressively evolves from LAM^1 to LAM^4.

At a high level, **Phase 1: Task-Plan Pretraining** provides a strong foundation by teaching the model to generate coherent, step-by-step plans for various tasks. **Phase 2: Learning from Experts** then introduces action trajectories labeled by GPT-4o, enabling LAM^2 to align its plan generation with actionable steps. However, relying solely on expert successes limits diversity and adaptability. To address this, **Phase 3: Self-Boosting Exploration** encourages the model to tackle tasks that even GPT-4o failed to solve, autonomously generating new success cases and evolving into LAM^3. Finally, **Phase 4: Learning from a Reward Model** incorporates reinforcement learning (RL) principles, allowing LAM^4 to learn from both successes and failures, refining its decision-making in complex, previously unseen scenarios. Table 1 summarizes the data used in each phase. Each phase uses different training objectives, namely *(i)* task-plan pretraining (phase 1) and *(ii)* decision-making training (phase 2-4), as detailed in Appendix E.

4.1 Phase 1: Task-Plan Pretraining

The initial stage focuses on imparting a broad understanding of how tasks can be decomposed into logical steps. We start with Mistral-7B Jiang et al. (2023) as the base model. A total of **76,672** task-plan pairs (t_i, P_i) are collected from various sources, including application help documentation, WikiHow, and historical

[4] More specifically, we compare the `.xml` files which is the underlying data representation of Microsoft Word.

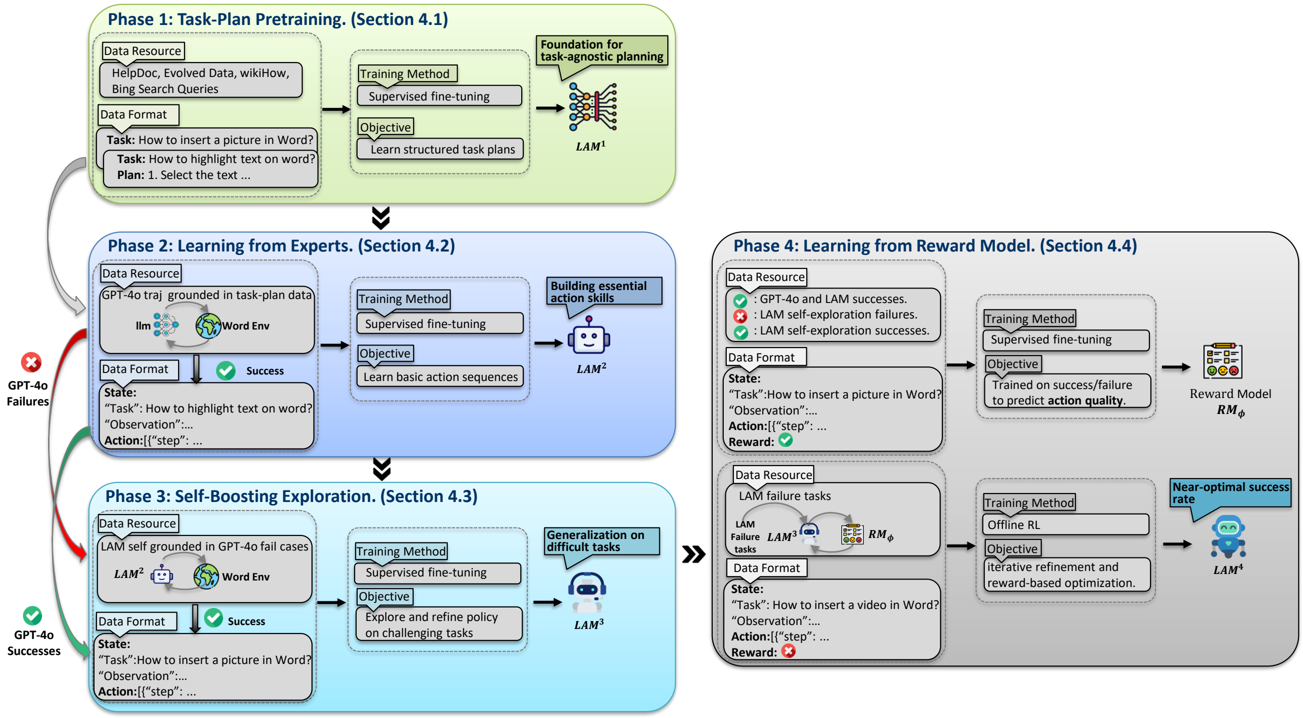

Phase 1: Task-Plan Pretraining. (Section 4.1)
Data Resource
HelpDoc, Evolved Data, wikiHow, Bing Search Queries
Data Format
Task: How to insert a picture in Word?
Task: How to highlight text on word?
Plan: 1. Select the text ...
Training Method
Supervised fine-tuning
Objective
Learn structured task plans
Foundation for task-agnostic planning
LAM¹
Phase 2: Learning from Experts. (Section 4.2)
Data Resource
GPT-4o traj grounded in task-plan data
llm
Word Env
Data Format
Success
State:
"Task": How to highlight text on word?
"Observation":...
Action:[{"step": ...
Training Method
Supervised fine-tuning
Objective
Learn basic action sequences
Building essential action skills
LAM²
GPT-4o Failures
GPT-4o Successes
Phase 3: Self-Boosting Exploration. (Section 4.3)
Data Resource
LAM self grounded in GPT-4o fail cases
LAM²
Word Env
Data Format
Success
State:
"Task":How to insert a picture in Word?
"Observation":...
Action:[{"step": ...
Training Method
Supervised fine-tuning
Objective
Explore and refine policy on challenging tasks
Generalization on difficult tasks
LAM³
Phase 4: Learning from Reward Model. (Section 4.4)
Data Resource
✓ : GPT-4o and LAM successes.
✗ : LAM self-exploration failures.
✓ : LAM self-exploration successes.
Data Format
State:
"Task":How to insert a picture in Word?
"Observation":...
Action:[{"step": ...
Reward: ✓
Training Method
Supervised fine-tuning
Objective
Trained on success/failure to predict action quality.
Reward Model
RMϕ
Data Resource
LAM failure tasks
LAM Failure tasks
LAM³
RMϕ
Data Format
State:
"Task": How to insert a video in Word?
"Observation":...
Action:[{"step": ...
Reward: ✗
Training Method
Offline RL
Objective
iterative refinement and reward-based optimization.
Near-optimal success rate
LAM⁴

Figure 8: The overview of LAM training pipeline.

Table 1: Training data summary for each phase of LAM training.

Model	Data Type	Data Source	Input → Output	Data Size
LAM^1	Task-Plan Pairs	Application documentation, WikiHow, evolved data historical search queries	$t_i \rightarrow P_i$	76,672 tasks
LAM^2	Task-Action Trajectories	GPT-4o	$s_t \rightarrow a_t$	2,192 trajectories
LAM^3	Task-Action Trajectories	LAM^2 + GPT-4o	$s_t \rightarrow a_t$	2,688 trajectories
LAM^4	Task-Action-Reward Trajectories	RM + LAM^3	$(s_t, r_t) \rightarrow a_t$	1,788 trajectories
Reward Model	Task-Action-Reward Trajectories	GPT-4o + LAM^3	$(s_t, a_t) \rightarrow r_t$	4,476 trajectories

search queries. Of these, 29,182 pairs are sourced directly, while 47,490 are generated via data evolution techniques (as described in Section 3.1.4), enriching the dataset with more complex and diverse tasks.

In this phase, LAM^1 is trained via supervised fine-tuning (SFT) to predict the correct plan sequence P_i for a given task t_i:

$$\mathcal{L}_{\text{SFT}}(\text{LAM}^1_\theta) = \frac{1}{N}\sum_{i=1}^{N}\mathcal{L}_{\text{CE}}(P_i^{\text{pred}}, P_i^{\text{true}}).$$

Here, $\mathcal{L}_{\text{CE}}$ denotes the cross-entropy loss, and N is the number of tasks. Although no actions are generated at this stage, LAM^1 gains a robust task-agnostic planning capability. This knowledge will prove critical in guiding the model's action execution in later phases, ensuring that the agent understands the logical structure of tasks before attempting to perform them.

4.2 Phase 2: Learning from Experts

While LAM^1 can produce structured plans, it lacks the ability to execute them. In Phase 2, we introduce expert-labeled task-action trajectories from GPT-4o (Section 3.2) to teach the model how to perform actions. The illustrative application in this paper is the Microsoft Word environment, where we have **2,192** successful expert trajectories. Each trajectory consists of a sequence of state-action pairs (s_t, a_t), representing observed UI states and the corresponding actions to progress the task.

We split these 2,192 trajectories into a training set of 1,757 and a test set of 435 trajectories, providing a total of 3,959 steps for training. By imitation learning LAM^1 on these successful action sequences, we obtain LAM^2. The objective is to minimize:

$$\mathcal{L}_{\mathrm{SFT}}(\mathrm{LAM}^2_\theta) = \frac{1}{N}\sum_{i=1}^{N}\sum_{t=1}^{T_i}\mathcal{L}_{\mathrm{CE}}(\mathrm{LAM}^2_\theta(s_t), a_t),$$

where N is the number of trajectories and T_i is the number of steps in trajectory i. By imitating the expert's policy, LAM^2 transforms from a passive planner into a model capable of executing actions aligned with its plans, grounding its reasoning in the real application environment.

4.3 Phase 3: Self-Boosting Exploration

Up to Phase 2, LAM^2 only learns from successful trajectories provided by GPT-4o. This limits diversity and adaptability, as the model never sees how to handle situations that even GPT-4o could not deal with. To overcome this limitation, Phase 3 introduces self-boosting exploration.

Here, we revisit failed GPT-4o trajectories, *i.e.*, tasks that GPT-4o did not complete successfully, and let LAM^2 attempt them. Using the ReAct mechanism Yao et al. (2022); Shinn et al. (2024), LAM^2 interacts with the environment and tries alternative strategies for these challenging tasks. From these attempts, we sampled 2284 GPT-4o failed tasks and then collect **496** newly successful trajectories generated by LAM^2 itself. These self-labeled successes, combined with the original 2,192 GPT-4o successes, form an augmented dataset.

We then fine-tune LAM^2 on this enriched data, yielding LAM^3:

$$\mathcal{L}_{\mathrm{SFT}}(\mathrm{LAM}^3_\theta) = \frac{1}{N}\sum_{i=1}^{N}\sum_{t=1}^{T_i}\mathcal{L}_{\mathrm{CE}}(\mathrm{LAM}^3_\theta(s_t), a_t).$$

This self-boosting step allows the model to learn from its own newly discovered solutions, overcoming previous limitations and improving adaptability. By leveraging planning knowledge from Phase 1 and expert strategies from Phase 2, LAM^3 becomes more resourceful, even in scenarios with sparse or absent expert guidance.

4.4 Phase 4: Learning from a Reward Model

Despite the improvements, Phases 1–3 focus on successes or expert-like behavior. They offer limited insights into intermediate decision quality and fail to exploit learning opportunities presented by failed attempts. In Phase 4, we integrate reinforcement learning (RL) to address these shortcomings.

To this end, we design a two-stage approach, where we first The reward model (RM) is built using LAM^3 as the base model, with an additional output layer added to produce scalar values representing the quality of actions. Using the trained RM, we fine-tune LAM^4 in an offline RL setting. Here, the model refines its policy without additional environmental interactions, leveraging previously collected trajectories to learn from failures and improve action selection.

4.4.1 Reward Model Training

First, we train a reward model (RM) on both LAM^3's successful (496) and failed (1788) trajectories and GPT-4o's successful trajectories (2192) gathered in previous phases. All steps in successful trajectories are assigned a reward of $+1$, and all steps in failed trajectories a reward of -1. This uniform, binary labeling of outcomes ensures the RM consistently captures overall trajectory quality. Formally:

$$r_t = \mathrm{RM}(s_t, a_t; \phi),$$

where ϕ presents the RM parameters, and $r_t \in \{+1, -1\}$ is the assigned reward. The RM is trained via mean squared error (MSE) to approximate these ground-truth rewards.

The training dataset for the RM includes both failed and successful task-action trajectories generated by LAM^3, as well as the successful trajectories from the collected task-action data. All steps in successful trajectories receive a reward of +1, while every step in failed trajectories is assigned a reward of −1. This uniform labeling strategy ensures that the RM consistently reflects overall trajectory quality and effectively guides policy optimization.

4.4.2 Optimizing with Offline PPO

Armed with the RM to evaluate intermediate actions, we fine-tune LAM^4 via offline PPO Schulman et al. (2017). This stage focuses on the 1,788 failure trajectories collected during Phase 3, providing a unique opportunity to learn from mistakes. The training objective of PPO is:

$$\mathcal{L}_{\text{PPO}}(\text{LAM}^4_\theta) = \frac{1}{N}\sum_{i=1}^{N}\sum_{t=1}^{T_i} \min\left(\frac{\text{LAM}^4_\theta(a_t|s_t)}{\text{LAM}^4_{\theta_{\text{old}}}(a_t|s_t)}\hat{A}_t, \quad \text{clip}\Big(\frac{\text{LAM}^4_\theta(a_t|s_t)}{\text{LAM}^4_{\theta_{\text{old}}}(a_t|s_t)}, 1-\epsilon, 1+\epsilon\Big)\hat{A}_t \right),$$

where $\hat{A}_t$ denotes the advantage derived from RM-generated rewards, and ϵ is a clipping parameter to ensure stable updates.

By incorporating signals from both successes and failures, LAM^4 gains a deeper understanding of action quality. This RL-based fine-tuning helps the model generalize to complex, previously unseen scenarios, ensuring more robust and reliable decision-making.

4.5 Summary

The four-phase training pipeline incrementally builds a fully capable LAM. Phase 1 imparts a fundamental planning ability, Phase 2 incorporates expert knowledge for action execution, Phase 3 empowers the model to generate and learn from new successes, and Phase 4 leverages rewards from both successes and failures to optimize decision-making. By combining static knowledge with expert demonstrations, self-guided exploration, and reward-based refinement, we transform a general-purpose language model into a versatile LAM. This progressive training strategy ensures a robust, adaptive model ready to handle diverse and complex tasks.

5 Offline Evaluations

The offline evaluation results of **Task-Plan Pretraining Results (Phase 1)** and **Task-Action Results (Phases 2–4)** will be presented in this section. Offline evaluation allows us to systematically assess the performance of LAM^1 and subsequent phases (LAM^2, LAM^3, and LAM^4) without interacting with the environment. This setup effectively provides a controlled and reproducible framework for comparing task success rates, precision, and recall metrics across models.

5.1 Experiment Setup

5.1.1 SFT Training (Phase 1, 2, 3).

For supervised fine-tuning (SFT), the learning rate is set to 2×10^{-5} with cosine decay and 2 warmup steps. The batch size is 16, and the training is conducted for 3 epochs on the training data. Loss is calculated only for the target tokens rather than the full input sequence, optimizing the efficiency of the fine-tuning process. The training is performed on 8 × A100 80G NVIDIA GPUs.

5.1.2 Reward Training (Phase 4).

Reward scores are normalized to the range [0, 1] using sigmoid function. We employ the LoRA (Low-Rank Adaptation) method Hu et al. (2021) to train the reward model (RM). The LoRA parameters include rank of 8, LoRA alpha of 32, and LoRA dropout of 0.1. The task type is sequence classification. The training

Table 2: Performance (%) comparison of different models on planning.

Model	**TSR (%)**	**Step Precision (%)**	**Step Recall (%)**
LAM^1	82.2	**54.7**	55.7
GPT-4o	**84.5**	28.2	**66.1**
Mistral-7B	0.0	0.1	0.5

Table 3: Offline performance comparison across different models and metrics on decision making.

Metric	**LAM^1**	**LAM^2**	**LAM^3**	**LAM^4**	**GPT-4o**	**GPT-4o Mini**
Object Acc (%)	39.4	85.6	87.4	**87.8**	73.2	74.6
Operation Acc (%)	59.9	97.3	97.7	**97.7**	94.2	91.5
Status Acc (%)	32.7	97.8	98.2	**99.0**	52.1	67.4
Step Success Rate (SSR) (%)	33.0	83.6	85.9	**86.2**	68.8	73.4
Task Success Rate (TSR) (%)	35.6	76.8	79.3	**81.2**	67.2	62.3

process uses learning rate of 2×10^{-5} with linear decay, optimized with the AdamW optimizer, and spans 2 epochs. The training is conducted on 8 × A100 80G NVIDIA GPUs.

5.1.3 PPO Training (Phase 4).

For Proximal Policy Optimization (PPO) training, we use a learning rate of 1.4×10^{-5} and set the generated sample length to 256. The batch size is 8, and the mini-batch size is 1, with 4 PPO epochs and 1 gradient accumulation step per iteration. The target KL divergence is set to 0.1, and the initial KL coefficient is set to 0.2. To ensure robust training, reward values are normalized to the range [-0.5, 0.5]. The training is conducted on 8 NVIDIA A100 80G GPUs.

5.2 Task-Plan Pretraining Results (Phase 1)

5.2.1 Evaluation Metrics.

We evaluate LAM^1 on its ability to generate task plans. We use three metrics for this evaluation: *(i)* **Task Success Rate (TSR)**, measuring whether the predicted plan matches the ground truth at the task level; *(ii)* **Step Precision**, evaluating the proportion of predicted plan steps that appear in the ground truth; and *(iii)* **Step Recall**, assessing the proportion of ground truth plan steps that are correctly predicted.

To compute these metrics, we leverage GPT-4o to compare each step of the LAM^1 output with the corresponding ground truth steps. The counts of matched steps are then used to calculate the final evaluation metrics. Detailed prompt information for the evaluation can be found in Appendix D.

5.2.2 Performance of LAM on Planning

Table 2 presents the performance of LAM^1 in planning prediction across 15,334 tasks on Windows OS, utilizing the dataset detailed in Section 3.1. LAM^1 achieves a TSR of **82.2%**, which is comparable to GPT-4o's TSR of **84.5%**. While GPT-4o demonstrates a slightly higher TSR, it exhibits a lower Step Precision of **28.2%**, indicating inefficiencies in its planning by generating additional unnecessary steps. In contrast, LAM^1 achieves a higher Step Precision, reflecting its ability to produce more efficient and accurate plans. This superior precision is attributed to LAM^1's training regimen, which incorporates domain-specific knowledge through task-plan pretraining.

Additionally, the baseline Mistral-7B model, without any fine-tuning, performs inadequately with a TSR of **0.0%**, Step Precision of **0.1%**, and Step Recall of **0.5%**. These stark results underscore the critical importance of task-plan pretraining in transforming a general-purpose language model into a competent task planner.

Overall, the evaluation highlights that while general-purpose models like GPT-4o can achieve high success rates, their lower step precision suggests a propensity for overcomplicating plans. In contrast, specialized models like LAM^1 not only maintain competitive success rates but also generate more streamlined and accurate action sequences. This validates the effectiveness of targeted training approaches in enhancing planning capabilities and demonstrates the necessity of task-plan pretraining for developing reliable and efficient task planners.

5.3 Task-Action Results (Phases 2–4)

5.3.1 Evaluation Metrics

To assess the performance of agents in completing tasks, we employ five primary metrics: **Object Accuracy (Object Acc.)**, **Operation Accuracy (Operation Acc.)**, **Status Accuracy (Status Acc.)**, **Step Success Rate (SSR)**, and **Task Success Rate (TSR)**. The definitions and calculation methods for these metrics are detailed below:

1. **Object Accuracy (Object Acc.):** This metric measures the accuracy of selecting the correct control object for each task step. The predicted object is compared with the set of acceptable objects defined in the ground truth. It evaluates the agent's ability to correctly identify and interact with the appropriate UI elements.

2. **Operation Accuracy (Operation Acc.):** For operations such as `Click`, `Type`, or `Select Option`, this metric evaluates the correctness of the predicted action. It ensures that the agent performs the correct operation as specified in the ground truth.

3. **Status Accuracy (Status Acc.):** This metric assesses whether the agent correctly identifies the task's completion status based on its predictions. It evaluates the agent's understanding of the overall progression and whether the task is marked as finished appropriately.

4. **Step Success Rate (SSR):** A step is considered successful only if the selected object, predicted operation, and predicted status are all correct. This metric evaluates each step of the task independently by comparing the predicted outputs with the ground truth action history.

5. **Task Success Rate (TSR):** A task is considered successful only if all steps within the task are successful, making this a stringent evaluation metric. This metric provides a holistic measure of the agent's ability to complete complex, multi-step tasks accurately.

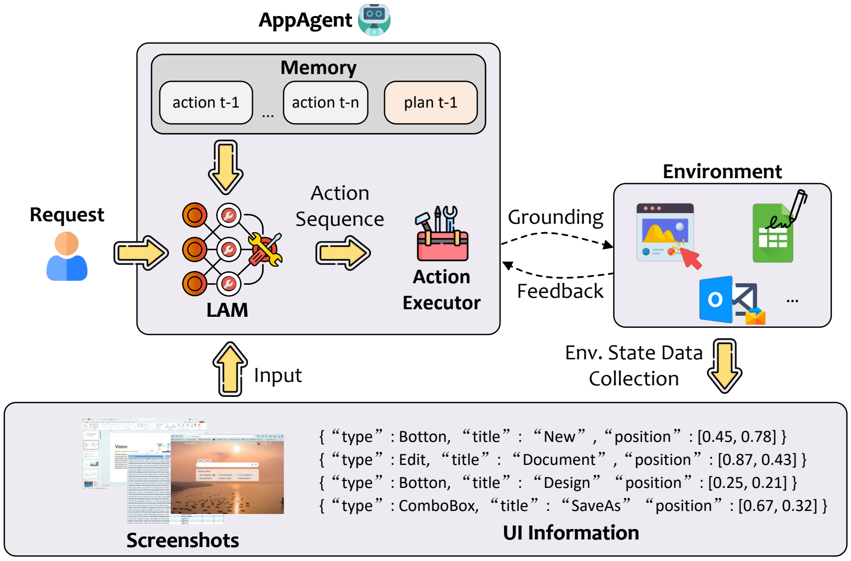
These metrics collectively cover various aspects of agent performance, including precision in object selection, operation execution, task understanding, and overall task completion. By combining step-level and task-level evaluations, they provide a comprehensive assessment of the agent's effectiveness in real-world task execution.

5.3.2 Performance on Decision Making

Table 3 summarizes the results on 435 tasks of the Word Application. The four-phase LAM training framework demonstrates incremental and cumulative improvements in task completion. Notably, LAM^4 achieves a TSR of **81.2**%, outperforming both GPT-4o (67.2%) and GPT-4o-mini (62.3%). This performance gap is substantial, considering that LAM's training process relies on progressively collected data and incremental refinements tailored to each phase.

The step-by-step training strategy explains these gains. In Phase 1 (LAM^1), task-plan pretraining establishes a foundational understanding of task structures, resulting in a modest increase in TSR. In Phase 2 (LAM^2), imitation learning on GPT-4o-labeled success trajectories imparts efficient execution strategies, driving a significant jump in TSR from 35.6% to 76.8%. Phase 3 (LAM^3) introduces self-boosting exploration, where LAM autonomously tackles cases previously failed by GPT-4o. This yields an additional increase in TSR to 79.3%. Finally, in Phase 4 (LAM^4), reward-guided fine-tuning refines decision-making based on sparse feedback, further elevating TSR to 81.2%.

An important outcome is that the LAM framework enables the model to surpass GPT-4o, despite GPT-4o providing initial annotations. Through targeted data collection and progressive refinement, LAM not only assimilates the strengths of GPT-4o, but also learns from its failures to develop more robust and adaptable



```
{ “type” : Botton, “title” : “New” , “position” : [0.45, 0.78] }
{ “type” : Edit, “title” : “Document” , “position” : [0.87, 0.43] }
{ “type” : Botton, “title” : “Design” “position” : [0.25, 0.21] }
{ “type” : ComboBox, “title” : “SaveAs” “position” : [0.67, 0.32] }
```

Figure 9: The overall architecture of the AppAgent employed in UFO.

policies. The ReAct mechanism plays a crucial role here, allowing LAM^2 and beyond to gather new success trajectories from challenging tasks, thereby enhancing its policy and overall performance.

In summary, the phased training approach and judicious data utilization enable LAM to excel where a state-of-the-art LLM (GPT-4o) falls short. This highlights the effectiveness of the LAM framework in crafting agents that are both data-efficient and capable of executing complex, multi-step tasks with high accuracy and reliability.

6 Integration and Grounding

Once the LAM is trained, we integrate it into the GUI agent UFO Zhang et al. (2024a), enabling the model's predicted actions to be grounded and executable within the Windows OS environment. The UFO agent accepts user requests in natural language and completes tasks by interacting with the UI controls of Windows applications.

6.1 LAM Agent In a Nutshell

In UFO, the LAM serves as the inference engine within the AppAgent, enabling efficient and accurate task completion. Figure 9 illustrates the architecture of the AppAgent. UFO, equipped with LAMs, is designed for interactive engagement with Windows applications. For simplicity, we focus on automating tasks within Microsoft Word, a widely used productivity tool with a sophisticated GUI and diverse functionalities, making it an ideal testbed for training and evaluating LAM.

During each inference step, the agent collects critical contextual information from the application environment, which is then passed to the LAM for decision-making. The LAM performs planning, orchestrates actions, and infers the necessary steps to fulfill the user request. These inferred actions are grounded in the environment by mapping them to predefined tools and function calls used by the agent, such as mouse clicks, keyboard inputs, or API calls. This process iterates, with LAM continuously adjusting its plan based on real-time feedback from the environment, until the task is completed. Additionally, the agent maintains a memory that logs historical actions and plans, providing essential context for the LAM to make more informed and adaptive decisions as the task progresses. This integration ensures that UFO can efficiently manage and complete complex, real-world tasks in Windows environments.

6.2 Environment

The UFO agent leverages the LAM to interact with applications in the Windows environment. At each decision step, UFO employs the UI Automation (UIA) API Dinh et al. (2018) to inspect all actionable controls within the target Windows application, retrieving contextual information for each control[5]. This information is passed to the LAM for control selection and action inference. The control data is structured as a list of dictionaries, where each control is assigned a numerical index (as a label), along with its title and control type, allowing the LAM to make informed decisions regarding control selection and the corresponding action. This input format mirrors the structure used during offline data collection for consistency in training and execution.

6.3 LAM Inference

Using the environmental observations of application control information, UFO constructs prompts in the same format as the offline training data, using planning and thought generation techniques Wei et al. (2022); Ding et al. (2023) to enable LAM to make reliable inferences about the appropriate controls and operations to invoke. These inferences target the controls detected by the UIA, where each control is selected from a predefined list. The function calls inferred by LAM are limited to pre-defined operations, such as mouse and keyboard actions, as well as APIs specific to Word-related tasks. Once inferred, these operations are parsed and executed within the environment.

6.4 Action Execution

UFO employs a control interactor to ground the action strings generated by LAMs, translating them into tangible impacts within the target application. Each action typically consists of two key components:

1. **Control Element:** This refers to the specific UI control within the application that will receive the action, such as a button, text box, or scroll bar.
2. **Function Call:** This represents the operation to be performed on the control element, such as a mouse click, keyboard input, or invocation of native APIs.

By combining the control element and its associated function call, UFO executes the inferred actions within the application.

6.5 Memory

UFO maintains additional information in its memory to assist LAMs in making more informed and accurate decisions. This memory includes:

1. **Historical Actions:** A log of action trajectories and their execution results from the initial step onwards. This helps LAM understand the current system state and aids in exploring the next steps based on prior actions.
2. **Previous Plan:** The textual planning for future actions, generated by LAM in the previous step. This serves as a reference for guiding the current and future actions, ensuring consistency across steps.

This memory is fed into LAM at each decision point, allowing for more effective decision-making. By maintaining a comprehensive record of past actions and plans, LAMs can better understand what has been accomplished, what remains to be done, and the outcomes of previous actions. This situational awareness enhances LAMs' ability to complete user requests more effectively and efficiently.

[5] UIA is the native Windows OS APIs used to detect actionable controls and provide their metadata, such as names and locations. For other platforms, UIA can be replaced by vision-based detectors that analyze screenshots or by utilizing alternative accessibility APIs.

7 Online Evaluations

With the integration of the Windows GUI agent UFO, we evaluate the performance of the LAM in real-world environments. The evaluation process and results are detailed in the following subsections.

7.1 Testing Dataset

The online performance of LAM is evaluated on the same set of 435 test requests used during LAM training. The testing environments, specifically the Word document templates corresponding to each task, are also maintained as identical to the training setup to ensure consistency and comparability.

Table 4: Performance comparison of LAM and baseline models across metrics.

Metric	**Text-only**			**Text + Visual**	
	LAM	**GPT-4o**	**GPT-4o Mini**	**GPT-4o**	**GPT-4o Mini**
Task Success Rate (%)	71.0	63.0	57.8	75.5	66.7
Task Completion Time (s)	30.42	86.42	35.24	96.48	46.21
Task Completion Steps	5.62	6.73	5.99	4.98	6.34
Average Step Latency (s)	5.41	12.84	5.88	19.36	7.29

7.2 Implementation

Our LAM was deployed on a virtual machine (VM) configured as NC24s v3. The VM is equipped with 24 virtual cores (vCPUs), 448 GB of memory, and two NVIDIA Tesla V100 GPUs, each with 16 GB of memory, to support efficient inference. This computational setup was designed to meet the demanding requirements of LAM's inference processes effectively.

The UFO agent operates on six VMs running in parallel using Azure Dedicated Host[6] to accelerate the testing process. Each VM is equipped with a 15-core Intel(R) Xeon(R) Platinum 8370C CPU @ 2.80GHz, 64GB of RAM, and runs Windows 11 Enterprise version 23H2. Microsoft applications, such as Word and Excel, are installed on version 2410. GUI control is facilitated through the MSTSC tool[7]. This setup ensures a consistent and controlled environment for evaluating the LAM's performance.

7.3 Baselines

To benchmark the performance of LAM, we compared it against two baseline models: GPT-4o and GPT-4o Mini. These models are widely recognized for their robust natural language processing and reasoning capabilities, making them popular choices in the development of GUI agents. To ensure consistency in evaluation, the `top_p` and `temperature` hyperparameters were set to 0 for both baseline models.

To further examine the impact of input modalities, we conducted an ablation study comparing performance with and without the inclusion of screenshots. Notably, LAM processes only textual inputs, excluding screenshots, while the baseline models were evaluated using both textual and visual modalities.

7.4 Evaluation Metrics

We employ the following metrics to comprehensively evaluate the performance of LAM:

- **Task Success Rate (TSR):** The percentage of tasks successfully completed out of the total tasks attempted. Task success is determined by an evaluation agent using GPT-4o, which assesses the full task completion trajectory, including plans, action sequences, and screenshots, to verify task completion.

[6] `https://azure.microsoft.com/en-us/products/virtual-machines/dedicated-host`

[7] `https://learn.microsoft.com/en-us/windows-server/administration/windows-commands/mstsc`

- **Task Completion Time:** The total time taken to complete each task, measured from the initial request to the final action.
- **Task Completion Steps:** The total number of action steps performed by the agent to successfully complete each task.
- **Average Step Latency:** The average time taken per action step, reflecting the model's efficiency in generating and executing each action.

These metrics collectively evaluate both the accuracy and efficiency of task completion, providing a comprehensive assessment of the LAM's capabilities in real-world scenarios.

7.5 Experimental Analysis

The experimental results are presented in Table 4. LAM achieves a TSR of 71.0%, demonstrating competitive performance compared to the GPT-4o models. While GPT-4o with visual inputs attains the highest TSR of 76.5%, slightly outperforming LAM, its reliance on visual data introduces significant trade-offs in efficiency. Notably, when visual inputs are excluded, GPT-4o's TSR drops to 63.0%, an 8.0 percentage point decrease compared to LAM. Similarly, GPT-4o Mini exhibits lower TSRs for both visual and non-visual settings (66.7% and 57.8%, respectively). These results underscore LAM's capability as a text-only model to maintain high task success rates, outperforming the text-only variants of the baseline models.

Efficiency is assessed through Task Completion Time and Average Step Latency, where LAM demonstrates clear superiority. LAM achieves the shortest Task Completion Time of **30.42 seconds**, substantially outperforming all baseline models. In comparison, GPT-4o without visual inputs records a completion time of 86.42 seconds, more than 2.84 times longer than LAM. GPT-4o with visual inputs fares even worse, with a completion time of 96.48 seconds. Although GPT-4o Mini models show slightly better efficiency than their larger counterparts, they remain less efficient than LAM, with completion times of 35.24 seconds (without visual inputs) and 46.21 seconds (with visual inputs).

LAM also excels in Average Step Latency, achieving the shortest time per action step at **5.41 seconds**. Without visual inputs, GPT-4o reduces its step latency to 12.84 seconds but still remains more than twice as slow as LAM. In comparison, GPT-4o with visual inputs exhibits the highest step latency at 19.36 seconds per step, more than triple LAM's latency. GPT-4o Mini models show moderate improvements but still fall short, with step latencies of 7.29 seconds (with visual inputs) and 5.88 seconds (without visual inputs).

These findings highlight LAM's strengths as a text-only model, offering a compelling balance of competitive accuracy and superior efficiency. It achieves rapid task completion and low latency without sacrificing performance, making it an effective solution for real-world applications. Its specialized training enables precise action inference and execution, underscoring the potential of LAMs to enhance automation and productivity in agent-based systems.

8 Limitation and Future Research

While significant strides have been made in the development of LAMs, their current state is still in its infancy. Many technical challenges and limitations prevent LAMs from being fully productized and integrated into commercial use for real-world applications. Below, we outline key limitations and areas for future research to address these challenges.

8.1 Safety Risk

The ability of LAMs to perform real-world actions in physical or digital environments introduces significant safety risks. Unlike traditional LLMs, which primarily generate text, LAMs have the potential to manipulate external systems, control hardware, or make changes within software environments. While this capability is a key strength, it also presents a double-edged sword: errors in inference or execution can lead to unintended or harmful consequences Liu et al. (2024b); Zhou et al. (2023).

For instance, a LAM controlling a robotic system could misinterpret a command and cause physical damage. Similarly, a LAM operating within a financial or healthcare application could execute erroneous actions with substantial real-world repercussions. Therefore, safety mechanisms such as formal verification, action validation, and fallback strategies must be integrated into LAM systems. Future research must focus on developing robust error detection, rollback mechanisms, and fail-safe systems that prevent actions from being executed until they have been thoroughly vetted for correctness and safety Gehring et al. (1993); Zhang et al. (2023b); Koo & Toueg (1987).

8.2 Ethical and Regulatory Concerns

The deployment of LAMs raises significant ethical and regulatory challenges Biswas & Talukdar (2023); Yan et al. (2024); Meskó & Topol (2023); Minssen et al. (2023); Piñeiro-Martín et al. (2023). As these models gain the ability to interact with real-world environments, questions about accountability, transparency, and fairness come to the forefront Ferrara (2024); Liesenfeld et al. (2023); Li et al. (2023a). For instance, who is held accountable if a LAM causes harm or damage due to a misinterpretation of a user's command? How do we ensure that these systems are making decisions in a fair and unbiased manner? These concerns are amplified by the fact that LAMs are often trained on large datasets that may contain biases, which can influence the model's decision-making processes Navigli et al. (2023).

Moreover, there are regulatory concerns regarding the deployment of LAMs in critical sectors such as healthcare, finance, and transportation, where strict guidelines govern the use of automated systems Karabacak & Margetis (2023); Li et al. (2023b); Cui et al. (2024). Future research must address these concerns by developing transparent model architectures that allow for interpretability and explainability of actions taken by LAMs. Additionally, establishing clear regulatory frameworks and ethical guidelines will be crucial for ensuring that LAMs are deployed in a manner that prioritizes safety, fairness, and accountability.

8.3 Scalability, Generalizability and Adaptability

LAMs are often tailored to specific environments or scenarios, making their scalability, generalizability, and adaptability significant limitations. Most LAMs are designed to operate within a narrowly defined context, such as a specific operating system, application, or interface. These environments, however, are subject to frequent updates, changes in APIs, and the introduction of new applications or functionalities. A LAM trained on a specific version of an environment may fail when confronted with changes it has not encountered before, leading to poor performance or outright failures Grosse et al. (2023); Zhang et al. (2024d); Kong et al. (2020).

In addition, scaling LAMs to new environments or applications is challenging due to the high cost of collecting domain-specific data Muennighoff et al. (2024); Minaee et al. (2024). Gathering sufficient training data for each new context is time-consuming and resource-intensive. Furthermore, the model's ability to generalize across different environments is often limited, as it may not be familiar with the nuances of new systems or tasks.

Future work should focus on improving the adaptability and generalizability of LAMs through techniques like transfer learning, multi-task learning, and few-shot learning. These approaches allow a model to generalize from one environment to another with minimal retraining. Moreover, developing automated data collection methods and self-supervised learning techniques could significantly reduce the effort required to scale LAMs to new domains.

Summary. While LAMs represent a promising advancement in the evolution of AI systems, they are still constrained by several technical, ethical, and practical limitations. Addressing these challenges will be essential for enabling the widespread adoption and commercialization of LAMs in real-world applications. By ensuring safety, addressing ethical concerns, and improving scalability and adaptability, future research can help unlock the full potential of LAMs.

9 Related work

The emergence of LAMs has led to significant impact across various agentic domains. In the following sections, we will review related research and practices at three levels: *(i)* data of LAMs, *(ii)* training LAMs, and *(iii)* agents with LAMs.

9.1 Data of LAMs

The emergence of LLM-based agents has spurred the development of numerous datasets specifically tailored to LAM applications and their corresponding agent systems. These datasets can be divided into two main categories, namely *(i)* **Datasets for Training LAMs:** These datasets provide the necessary input for training LAMs, including diverse user commands, environmental contexts, and action sequences. *(ii)* **Evaluation Benchmarks:** These benchmarks are curated for testing and evaluating the capabilities of LAMs and agent systems.

Mind2Web Deng et al. (2024) is the first dataset developed for web agents that follow natural language instructions to complete complex tasks across diverse websites. It includes task descriptions, action sequences, and webpage snapshots, offering rich data for training and testing models in various web-based scenarios. Rawles *et al.*, introduced a large dataset called Android in the Wild (AITW) Rawles et al. (2024b), which is designed specifically for training models to control Android devices. SeeClick Cheng et al. (2024) combines web, mobile, and general GUI tasks, creating a dataset of over 1 million samples for training LAMs. Similarly, GUICourse Chen et al. (2024b) and OmniACT Kapoor et al. (2024) provide datasets across web, smartphone, and desktop platforms, containing detailed user requests, environmental states, and action sequences. These datasets are invaluable resources for training LAMs in specific domains and evaluating their task execution abilities.

Several benchmarks have also been developed to evaluate the capabilities of LAMs and their associated agents in different environments. WebCanvas provides 542 tasks with dynamic environments, designed to assess the task completion ability of web agents. AndroidWorld Rawles et al. (2024a) offers a fully functional Android environment, featuring 116 programmatic tasks across 20 real-world Android apps with reward signals for performance evaluation. WindowsArena Bonatti et al. (2024) focuses on benchmarking LAMs within the Windows GUI, while OSWorld Xie et al. (2024) extends this to a more diverse environment, encompassing Windows, macOS, and Ubuntu. These benchmarks provide standardized settings to measure and compare the effectiveness of LAMs and their agents in various real-world environments, enabling a unified evaluation framework for agentic models.

9.2 Training LAMs

Using both open and private domain-specific datasets, significant research efforts have been directed toward training LAMs for specialized purposes, enhancing the action inference abilities of traditional LLMs to enable automation and tangible real-world impact. For example, SeeClick Cheng et al. (2024) and GUICourse Chen et al. (2024b), in addition to releasing their own datasets, leverage these resources to train LAMs, grounding real-world data into models that effectively interact with their environments.

Hong *et al.*, trained an 18-billion-parameter visual language LAM, named CogAgent Hong et al. (2024), which specializes in GUI understanding and navigation tasks across both PC and Android interfaces. By utilizing datasets like Mind2Web and AITW, CogAgent has been optimized for complex navigation and action execution tasks in diverse GUI environments. ScreenAI Baechler et al. (2024) introduced a textual representation for user interfaces (UIs) to teach models how to understand and interact with UIs. This approach also facilitates automatic generation of large-scale training data, which is then used to pretrain and fine-tune models for a wide spectrum of tasks, including UI and infographic understanding and navigation. Additionally, Zhang *et al.*, released a series of large action models (xLAM) tailored for AI agent tasks Zhang et al. (2024c), including five models with both dense and mixture-of-expert architectures. By unifying datasets from diverse environments, xLAM ensures consistency in data format, simplifying model training and enhancing generalization across multiple benchmarks. These models have achieved outstanding performance in diverse scenarios, demonstrating the capability of LAMs to extend beyond traditional LLMs and perform complex real-world tasks.

These pioneering works have laid the foundation for advancing the action-oriented capabilities of LLMs, making LAMs a critical component in achieving robust automation and impactful real-world applications.

9.3 Agents with LAMs

With the development of LAMs, researchers have integrated these models into real-world agent systems, which provide the necessary components and workflows to ensure effective interaction between LAMs and their environments, enabling them to fulfill user requests efficiently. As a pioneer, Zhang *et al.*, demonstrated that GPT-V can serve as a capable LAM for web navigation when coupled with appropriate agent techniques and tools, revealing the potential of LAMs in complex web interactions. In the mobile domain, MobileAgent Wang et al. (2024a) and AppAgent Yang et al. (2023a) focus on automating tasks within Android applications by leveraging GUI agents. These systems demonstrate how LAMs can power task automation on mobile platforms, transforming how users interact with applications.

One of the most advanced systems, UFO Zhang et al. (2024a), is a UI-focused agent designed for automating tasks on the Windows OS, further enhanced with APIs Lu et al. (2024). UFO is composed of two key components: a HostAgent that decomposes user requests into subtasks and an AppAgent that executes these subtasks within individual applications. This architecture significantly enhances UFO's capability to handle cross-application tasks seamlessly, providing robust task automation across diverse software environments. In parallel, ScreenAgent Niu et al. (2024), Cradle Tan et al. (2024), OS-Copilot Wu et al. (2024), and MMAC-Copilot Song et al. (2024) also focus on automating UI tasks in desktop environments. Notably, Cradle and OS-Copilot push the boundaries by enabling agents to learn from their experiences and self-evolve over time, further enhancing their effectiveness and autonomy.

By integrating LAMs into agents to handle complex tasks in these various scenarios, These pioneering efforts are opening new possibilities for the future of human-computer interaction, revolutionizing traditional methods of interacting with GUIs and paving the way for more intelligent, automated, and user-friendly systems.

10 Conclusion

"Actions speak louder than words." The transition from generating language responses to executing tangible actions marks the evolution of large language models into large action models, enabling them to make real-world impacts, a critical step towards achieving AGI. This technical report provides a comprehensive introduction to LAMs, covering their conceptual foundations, system architecture, and the step-by-step process of developing a LAM—from data collection to model training and deployment in real-world agent systems. We use the Windows OS environment and its GUI agent UFO, as a case study to demonstrate how to build a LAM from the ground up. Detailed implementation strategies and evaluation results are presented to offer practical insights into this process.

However, despite progress, the development of high-quality LAMs is still in its early stages, with several limitations remaining. These include the extensive need for training data and computational resources, inference latency, and the risk of errors during real-world execution. While current LAMs have shown potential, there is substantial room for improvement. We anticipate that as these challenges are addressed, more sophisticated and reliable LAM applications will emerge, bringing us closer to fully autonomous systems capable of meaningful action in complex environments.

References

Gilles Baechler, Srinivas Sunkara, Maria Wang, Fedir Zubach, Hassan Mansoor, Vincent Etter, Victor Cărbune, Jason Lin, Jindong Chen, and Abhanshu Sharma. Screenai: A vision-language model for ui and infographics understanding. *arXiv preprint arXiv:2402.04615*, 2024.

Anjanava Biswas and Wrick Talukdar. Guardrails for trust, safety, and ethical development and deployment of large language models (llm). *Journal of Science & Technology*, 4(6):55–82, 2023.

Rogerio Bonatti, Dan Zhao, Francesco Bonacci, Dillon Dupont, Sara Abdali, Yinheng Li, Justin Wagle, Kazuhito Koishida, Arthur Bucker, Lawrence Jang, and Zack Hui. Windows agent arena: Evaluating multi-modal os agents at scale. *arXiv preprint arXiv:2409.08264*, 2024.

Tom B Brown. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*, 2020.

Thomas Carta, Clément Romac, Thomas Wolf, Sylvain Lamprier, Olivier Sigaud, and Pierre-Yves Oudeyer. Grounding large language models in interactive environments with online reinforcement learning. In *International Conference on Machine Learning*, pp. 3676–3713. PMLR, 2023.

Jin Chen, Zheng Liu, Xu Huang, Chenwang Wu, Qi Liu, Gangwei Jiang, Yuanhao Pu, Yuxuan Lei, Xiaolong Chen, Xingmei Wang, et al. When large language models meet personalization: Perspectives of challenges and opportunities. *World Wide Web*, 27(4):42, 2024a.

Wentong Chen, Junbo Cui, Jinyi Hu, Yujia Qin, Junjie Fang, Yue Zhao, Chongyi Wang, Jun Liu, Guirong Chen, Yupeng Huo, et al. Guicourse: From general vision language models to versatile gui agents. *arXiv preprint arXiv:2406.11317*, 2024b.

Kanzhi Cheng, Qiushi Sun, Yougang Chu, Fangzhi Xu, Yantao Li, Jianbing Zhang, and Zhiyong Wu. Seeclick: Harnessing gui grounding for advanced visual gui agents. *arXiv preprint arXiv:2401.10935*, 2024.

Can Cui, Yunsheng Ma, Xu Cao, Wenqian Ye, and Ziran Wang. Receive, reason, and react: Drive as you say, with large language models in autonomous vehicles. *IEEE Intelligent Transportation Systems Magazine*, 2024.

Ishita Dasgupta, Andrew K Lampinen, Stephanie CY Chan, Antonia Creswell, Dharshan Kumaran, James L McClelland, and Felix Hill. Language models show human-like content effects on reasoning. *arXiv preprint arXiv:2207.07051*, 2022.

Xiang Deng, Yu Gu, Boyuan Zheng, Shijie Chen, Sam Stevens, Boshi Wang, Huan Sun, and Yu Su. Mind2web: Towards a generalist agent for the web. *Advances in Neural Information Processing Systems*, 36, 2024.

Ruomeng Ding, Chaoyun Zhang, Lu Wang, Yong Xu, Minghua Ma, Wei Zhang, Si Qin, Saravan Rajmohan, Qingwei Lin, and Dongmei Zhang. Everything of thoughts: Defying the law of penrose triangle for thought generation. *arXiv preprint arXiv:2311.04254*, 2023.

Duong Tran Dinh, Pham Ngoc Hung, and Tung Nguyen Duy. A method for automated user interface testing of windows-based applications. In *Proceedings of the 9th International Symposium on Information and Communication Technology*, pp. 337–343, 2018.

Tao Feng, Chuanyang Jin, Jingyu Liu, Kunlun Zhu, Haoqin Tu, Zirui Cheng, Guanyu Lin, and Jiaxuan You. How far are we from agi: Are llms all we need? *Transactions on Machine Learning Research*.

Emilio Ferrara. Genai against humanity: Nefarious applications of generative artificial intelligence and large language models. *Journal of Computational Social Science*, pp. 1–21, 2024.

Jensen Gao, Bidipta Sarkar, Fei Xia, Ted Xiao, Jiajun Wu, Brian Ichter, Anirudha Majumdar, and Dorsa Sadigh. Physically grounded vision-language models for robotic manipulation. In *2024 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 12462–12469. IEEE, 2024.

William J Gehring, Brian Goss, Michael GH Coles, David E Meyer, and Emanuel Donchin. A neural system for error detection and compensation. *Psychological science*, 4(6):385–390, 1993.

Roger Grosse, Juhan Bae, Cem Anil, Nelson Elhage, Alex Tamkin, Amirhossein Tajdini, Benoit Steiner, Dustin Li, Esin Durmus, Ethan Perez, et al. Studying large language model generalization with influence functions. *arXiv preprint arXiv:2308.03296*, 2023.

Lin Guan, Karthik Valmeekam, Sarath Sreedharan, and Subbarao Kambhampati. Leveraging pre-trained large language models to construct and utilize world models for model-based task planning. *Advances in Neural Information Processing Systems*, 36:79081–79094, 2023.

Jianliang He, Siyu Chen, Fengzhuo Zhang, and Zhuoran Yang. From words to actions: Unveiling the theoretical underpinnings of llm-driven autonomous systems. *arXiv preprint arXiv:2405.19883*, 2024.

Wenyi Hong, Weihan Wang, Qingsong Lv, Jiazheng Xu, Wenmeng Yu, Junhui Ji, Yan Wang, Zihan Wang, Yuxiao Dong, Ming Ding, et al. Cogagent: A visual language model for GUI agents. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 14281–14290, 2024.

Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. Lora: Low-rank adaptation of large language models. *arXiv preprint arXiv:2106.09685*, 2021.

Mengkang Hu, Pu Zhao, Can Xu, Qingfeng Sun, Jianguang Lou, Qingwei Lin, Ping Luo, Saravan Rajmohan, and Dongmei Zhang. Agentgen: Enhancing planning abilities for large language model based agent via environment and task generation. *CoRR*, 2024.

Yucheng Hu and Yuxing Lu. Rag and rau: A survey on retrieval-augmented language model in natural language processing. *arXiv preprint arXiv:2404.19543*, 2024.

Albert Q Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, et al. Mistral 7b. *arXiv preprint arXiv:2310.06825*, 2023.

Yuxuan Jiang, Chaoyun Zhang, Shilin He, Zhihao Yang, Minghua Ma, Si Qin, Yu Kang, Yingnong Dang, Saravan Rajmohan, Qingwei Lin, et al. Xpert: Empowering incident management with query recommendations via large language models. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pp. 1–13, 2024.

Zhengbao Jiang, Jun Araki, Haibo Ding, and Graham Neubig. How can we know when language models know? on the calibration of language models for question answering. *Transactions of the Association for Computational Linguistics*, 9:962–977, 2021.

Sai Shashank Kalakonda, Shubh Maheshwari, and Ravi Kiran Sarvadevabhatla. Action-gpt: Leveraging large-scale language models for improved and generalized action generation. In *2023 IEEE International Conference on Multimedia and Expo (ICME)*, pp. 31–36. IEEE, 2023.

Raghav Kapoor, Yash Parag Butala, Melisa Russak, Jing Yu Koh, Kiran Kamble, Waseem Alshikh, and Ruslan Salakhutdinov. Omniact: A dataset and benchmark for enabling multimodal generalist autonomous agents for desktop and web. *arXiv preprint arXiv:2402.17553*, 2024.

Mert Karabacak and Konstantinos Margetis. Embracing large language models for medical applications: opportunities and challenges. *Cureus*, 15(5), 2023.

Enkelejda Kasneci, Kathrin Seßler, Stefan Küchemann, Maria Bannert, Daryna Dementieva, Frank Fischer, Urs Gasser, Georg Groh, Stephan Günnemann, Eyke Hüllermeier, et al. Chatgpt for good? on opportunities and challenges of large language models for education. *Learning and individual differences*, 103:102274, 2023.

Geunwoo Kim, Pierre Baldi, and Stephen McAleer. Language models can solve computer tasks. *Advances in Neural Information Processing Systems*, 36, 2024.

Lingkai Kong, Haoming Jiang, Yuchen Zhuang, Jie Lyu, Tuo Zhao, and Chao Zhang. Calibrated language model fine-tuning for in-and out-of-distribution data. *arXiv preprint arXiv:2010.11506*, 2020.

Richard Koo and Sam Toueg. Checkpointing and rollback-recovery for distributed systems. *IEEE Transactions on software Engineering*, (1):23–31, 1987.

Wei Li, William Bishop, Alice Li, Chris Rawles, Folawiyo Campbell-Ajala, Divya Tyamagundlu, and Oriana Riva. On the effects of data scale on computer control agents. *arXiv preprint arXiv:2406.03679*, 2024.

Yingji Li, Mengnan Du, Rui Song, Xin Wang, and Ying Wang. A survey on fairness in large language models. *arXiv preprint arXiv:2308.10149*, 2023a.

Yinheng Li, Shaofei Wang, Han Ding, and Hang Chen. Large language models in finance: A survey. In *Proceedings of the fourth ACM international conference on AI in finance*, pp. 374–382, 2023b.

Andreas Liesenfeld, Alianda Lopez, and Mark Dingemanse. Opening up chatgpt: Tracking openness, transparency, and accountability in instruction-tuned text generators. In *Proceedings of the 5th international conference on conversational user interfaces*, pp. 1–6, 2023.

Chen Ling, Xujiang Zhao, Jiaying Lu, Chengyuan Deng, Can Zheng, Junxiang Wang, Tanmoy Chowdhury, Yun Li, Hejie Cui, Xuchao Zhang, et al. Domain specialization as the key to make large language models disruptive: A comprehensive survey. *arXiv preprint arXiv:2305.18703*, 2023.

Jun Liu, Chaoyun Zhang, Jiaxu Qian, Minghua Ma, Si Qin, Chetan Bansal, Qingwei Lin, Saravan Rajmohan, and Dongmei Zhang. Large language models can deliver accurate and interpretable time series anomaly detection. *arXiv preprint arXiv:2405.15370*, 2024a.

Zheyuan Liu, Guangyao Dou, Zhaoxuan Tan, Yijun Tian, and Meng Jiang. Towards safer large language models through machine unlearning. *arXiv preprint arXiv:2402.10058*, 2024b.

Junting Lu, Zhiyang Zhang, Fangkai Yang, Jue Zhang, Lu Wang, Chao Du, Qingwei Lin, Saravan Rajmohan, Dongmei Zhang, and Qi Zhang. Turn every application into an agent: Towards efficient human-agent-computer interaction with api-first llm-based agents. *arXiv preprint arXiv:2409.17140*, 2024.

Zilin Ma, Yiyang Mei, and Zhaoyuan Su. Understanding the benefits and challenges of using large language model-based conversational agents for mental well-being support. In *AMIA Annual Symposium Proceedings*, volume 2023, pp. 1105. American Medical Informatics Association, 2023.

Bertalan Meskó and Eric J Topol. The imperative for regulatory oversight of large language models (or generative ai) in healthcare. *NPJ digital medicine*, 6(1):120, 2023.

Shervin Minaee, Tomas Mikolov, Narjes Nikzad, Meysam Chenaghlu, Richard Socher, Xavier Amatriain, and Jianfeng Gao. Large language models: A survey. *arXiv preprint arXiv:2402.06196*, 2024.

Timo Minssen, Effy Vayena, and I Glenn Cohen. The challenges for regulating medical use of chatgpt and other large language models. *Jama*, 2023.

Niklas Muennighoff, Alexander Rush, Boaz Barak, Teven Le Scao, Nouamane Tazi, Aleksandra Piktus, Sampo Pyysalo, Thomas Wolf, and Colin A Raffel. Scaling data-constrained language models. *Advances in Neural Information Processing Systems*, 36, 2024.

Roberto Navigli, Simone Conia, and Björn Ross. Biases in large language models: origins, inventory, and discussion. *ACM Journal of Data and Information Quality*, 15(2):1–21, 2023.

Runliang Niu, Jindong Li, Shiqi Wang, Yali Fu, Xiyu Hu, Xueyuan Leng, He Kong, Yi Chang, and Qi Wang. Screenagent: A vision language model-driven computer control agent. *arXiv preprint arXiv:2402.07945*, 2024.

Alastair Pennycook. Actions speak louder than words: Paralanguage, communication, and education. *Tesol Quarterly*, 19(2):259–282, 1985.

Andrés Piñeiro-Martín, Carmen García-Mateo, Laura Docío-Fernández, and Maria Del Carmen Lopez-Perez. Ethical challenges in the development of virtual assistants powered by large language models. *Electronics*, 12(14):3170, 2023.

Christopher Rawles, Sarah Clinckemaillie, Yifan Chang, Jonathan Waltz, Gabrielle Lau, Marybeth Fair, Alice Li, William Bishop, Wei Li, Folawiyo Campbell-Ajala, et al. Androidworld: A dynamic benchmarking environment for autonomous agents. *arXiv preprint arXiv:2405.14573*, 2024a.

Christopher Rawles, Alice Li, Daniel Rodriguez, Oriana Riva, and Timothy Lillicrap. Androidinthewild: A large-scale dataset for android device control. *Advances in Neural Information Processing Systems*, 36, 2024b.

Jingqing Ruan, Yihong Chen, Bin Zhang, Zhiwei Xu, Tianpeng Bao, Hangyu Mao, Ziyue Li, Xingyu Zeng, Rui Zhao, et al. Tptu: Task planning and tool usage of large language model-based ai agents. In *NeurIPS 2023 Foundation Models for Decision Making Workshop*, 2023.

Paul K Rubenstein, Chulayuth Asawaroengchai, Duc Dung Nguyen, Ankur Bapna, Zalán Borsos, Félix de Chaumont Quitry, Peter Chen, Dalia El Badawy, Wei Han, Eugene Kharitonov, et al. Audiopalm: A large language model that can speak and listen. *arXiv preprint arXiv:2306.12925*, 2023.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

Chirag Shah, Ryen W White, Reid Andersen, Georg Buscher, Scott Counts, Sarkar Snigdha Sarathi Das, Ali Montazer, Sathish Manivannan, Jennifer Neville, Xiaochuan Ni, et al. Using large language models to generate, validate, and apply user intent taxonomies. *arXiv preprint arXiv:2309.13063*, 2023.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. *Advances in Neural Information Processing Systems*, 36, 2024.

Zirui Song, Yaohang Li, Meng Fang, Zhenhao Chen, Zecheng Shi, and Yuan Huang. Mmac-copilot: Multi-modal agent collaboration operating system copilot. *arXiv preprint arXiv:2404.18074*, 2024.

Weihao Tan, Ziluo Ding, Wentao Zhang, Boyu Li, Bohan Zhou, Junpeng Yue, Haochong Xia, Jiechuan Jiang, Longtao Zheng, Xinrun Xu, et al. Towards general computer control: A multimodal agent for red dead redemption ii as a case study. *arXiv preprint arXiv:2403.03186*, 2024.

Arun James Thirunavukarasu, Darren Shu Jeng Ting, Kabilan Elangovan, Laura Gutierrez, Ting Fang Tan, and Daniel Shu Wei Ting. Large language models in medicine. *Nature medicine*, 29(8):1930–1940, 2023.

Paul Thomas, Seth Spielman, Nick Craswell, and Bhaskar Mitra. Large language models can accurately predict searcher preferences. In *Proceedings of the 47th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pp. 1930–1940, 2024.

Karthik Valmeekam, Alberto Olmo, Sarath Sreedharan, and Subbarao Kambhampati. Large language models still can't plan (a benchmark for llms on planning and reasoning about change). In *NeurIPS 2022 Foundation Models for Decision Making Workshop*, 2022.

Junyang Wang, Haiyang Xu, Jiabo Ye, Ming Yan, Weizhou Shen, Ji Zhang, Fei Huang, and Jitao Sang. Mobile-Agent: Autonomous multi-modal mobile device agent with visual perception. *arXiv preprint arXiv:2401.16158*, 2024a.

Lei Wang, Chen Ma, Xueyang Feng, Zeyu Zhang, Hao Yang, Jingsen Zhang, Zhiyuan Chen, Jiakai Tang, Xu Chen, Yankai Lin, et al. A survey on large language model based autonomous agents. *Frontiers of Computer Science*, 18(6):186345, 2024b.

Wenhai Wang, Zhe Chen, Xiaokang Chen, Jiannan Wu, Xizhou Zhu, Gang Zeng, Ping Luo, Tong Lu, Jie Zhou, Yu Qiao, et al. Visionllm: Large language model is also an open-ended decoder for vision-centric tasks. *Advances in Neural Information Processing Systems*, 36, 2024c.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, et al. Opendevin: An open platform for ai software developers as generalist agents. *arXiv preprint arXiv:2407.16741*, 2024d.

Zige Wang, Wanjun Zhong, Yufei Wang, Qi Zhu, Fei Mi, Baojun Wang, Lifeng Shang, Xin Jiang, and Qun Liu. Data management for large language models: A survey. *arXiv e-prints*, pp. arXiv–2312, 2023.

Jason Wei, Maarten Bosma, Vincent Y Zhao, Kelvin Guu, Adams Wei Yu, Brian Lester, Nan Du, Andrew M Dai, and Quoc V Le. Finetuned language models are zero-shot learners. *arXiv preprint arXiv:2109.01652*, 2021.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837, 2022.

Zhiyong Wu, Chengcheng Han, Zichen Ding, Zhenmin Weng, Zhoumianze Liu, Shunyu Yao, Tao Yu, and Lingpeng Kong. Os-copilot: Towards generalist computer agents with self-improvement. *arXiv preprint arXiv:2402.07456*, 2024.

Zhiheng Xi, Wenxiang Chen, Xin Guo, Wei He, Yiwen Ding, Boyang Hong, Ming Zhang, Junzhe Wang, Senjie Jin, Enyu Zhou, et al. The rise and potential of large language model based agents: A survey. *arXiv preprint arXiv:2309.07864*, 2023.

Tianbao Xie, Danyang Zhang, Jixuan Chen, Xiaochuan Li, Siheng Zhao, Ruisheng Cao, Toh Jing Hua, Zhoujun Cheng, Dongchan Shin, Fangyu Lei, et al. Osworld: Benchmarking multimodal agents for open-ended tasks in real computer environments. *arXiv preprint arXiv:2404.07972*, 2024.

Can Xu, Qingfeng Sun, Kai Zheng, Xiubo Geng, Pu Zhao, Jiazhan Feng, Chongyang Tao, and Daxin Jiang. Wizardlm: Empowering large language models to follow complex instructions. *arXiv preprint arXiv:2304.12244*, 2023.

Frank F Xu, Uri Alon, Graham Neubig, and Vincent Josua Hellendoorn. A systematic evaluation of large language models of code. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*, pp. 1–10, 2022.

Lixiang Yan, Lele Sha, Linxuan Zhao, Yuheng Li, Roberto Martinez-Maldonado, Guanliang Chen, Xinyu Li, Yueqiao Jin, and Dragan Gašević. Practical and ethical challenges of large language models in education: A systematic scoping review. *British Journal of Educational Technology*, 55(1):90–112, 2024.

Zhao Yang, Jiaxuan Liu, Yucheng Han, Xin Chen, Zebiao Huang, Bin Fu, and Gang Yu. Appagent: Multimodal agents as smartphone users. *arXiv preprint arXiv:2312.13771*, 2023a.

Zhengyuan Yang, Linjie Li, Kevin Lin, Jianfeng Wang, Chung-Ching Lin, Zicheng Liu, and Lijuan Wang. The dawn of lmms: Preliminary explorations with gpt-4v (ision). *arXiv preprint arXiv:2309.17421*, 9(1):1, 2023b.

Shunyu Yao, Rohan Rao, Matthew Hausknecht, and Karthik Narasimhan. Keep calm and explore: Language models for action generation in text-based games. *arXiv preprint arXiv:2010.02903*, 2020.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629*, 2022.

Burak Yetiştiren, Işık Özsoy, Miray Ayerdem, and Eray Tüzün. Evaluating the code quality of ai-assisted code generation tools: An empirical study on github copilot, amazon codewhisperer, and chatgpt. *arXiv preprint arXiv:2304.10778*, 2023.

Fanlong Zeng, Wensheng Gan, Yongheng Wang, Ning Liu, and Philip S Yu. Large language models for robotics: A survey. *arXiv preprint arXiv:2311.07226*, 2023.

Chaoyun Zhang, Liqun Li, Shilin He, Xu Zhang, Bo Qiao, Si Qin, Minghua Ma, Yu Kang, Qingwei Lin, Saravan Rajmohan, Dongmei Zhang, and Qi Zhang. UFO: A UI-Focused Agent for Windows OS Interaction. *arXiv preprint arXiv:2402.07939*, 2024a.

Chaoyun Zhang, Zicheng Ma, Yuhao Wu, Shilin He, Si Qin, Minghua Ma, Xiaoting Qin, Yu Kang, Yuyi Liang, Xiaoyu Gou, et al. Allhands: Ask me anything on large-scale verbatim feedback via large language models. *arXiv preprint arXiv:2403.15157*, 2024b.

Jianguo Zhang, Tian Lan, Ming Zhu, Zuxin Liu, Thai Hoang, Shirley Kokane, Weiran Yao, Juntao Tan, Akshara Prabhakar, Haolin Chen, et al. xlam: A family of large action models to empower ai agent systems. *arXiv preprint arXiv:2409.03215*, 2024c.

Shun Zhang, Zhenfang Chen, Yikang Shen, Mingyu Ding, Joshua B Tenenbaum, and Chuang Gan. Planning with large language models for code generation. *arXiv preprint arXiv:2303.05510*, 2023a.

Xingxuan Zhang, Jiansheng Li, Wenjing Chu, Junjia Hai, Renzhe Xu, Yuqing Yang, Shikai Guan, Jiazheng Xu, and Peng Cui. On the out-of-distribution generalization of multimodal large language models. *arXiv preprint arXiv:2402.06599*, 2024d.

Yudi Zhang, Pei Xiao, Lu Wang, Chaoyun Zhang, Meng Fang, Yali Du, Yevgeniy Puzyrev, Randolph Yao, Si Qin, Qingwei Lin, et al. Ruag: Learned-rule-augmented generation for large language models. *arXiv preprint arXiv:2411.03349*, 2024e.

Zeyu Zhang, Xiaohe Bo, Chen Ma, Rui Li, Xu Chen, Quanyu Dai, Jieming Zhu, Zhenhua Dong, and Ji-Rong Wen. A survey on the memory mechanism of large language model based agents. *arXiv preprint arXiv:2404.13501*, 2024f.

Zhexin Zhang, Leqi Lei, Lindong Wu, Rui Sun, Yongkang Huang, Chong Long, Xiao Liu, Xuanyu Lei, Jie Tang, and Minlie Huang. Safetybench: Evaluating the safety of large language models with multiple choice questions. *arXiv preprint arXiv:2309.07045*, 2023b.

Xin Zhou, Yi Lu, Ruotian Ma, Tao Gui, Qi Zhang, and Xuanjing Huang. Making harmful behaviors unlearnable for large language models. *arXiv preprint arXiv:2311.02105*, 2023.

Yutao Zhu, Huaying Yuan, Shuting Wang, Jiongnan Liu, Wenhan Liu, Chenlong Deng, Haonan Chen, Zhicheng Dou, and Ji-Rong Wen. Large language models for information retrieval: A survey. *arXiv preprint arXiv:2308.07107*, 2023.

A Template Word files

Figure 10, 11, and 12 show three template word file examples used in the instantiation phase when converting task-plan data to task-action data.

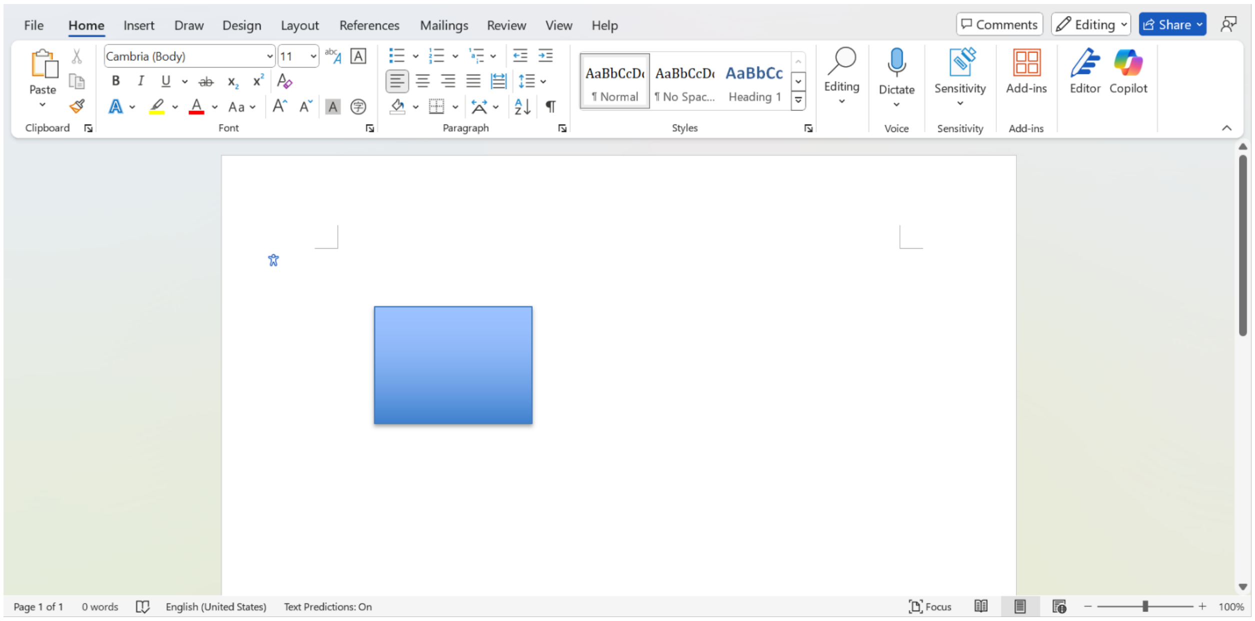
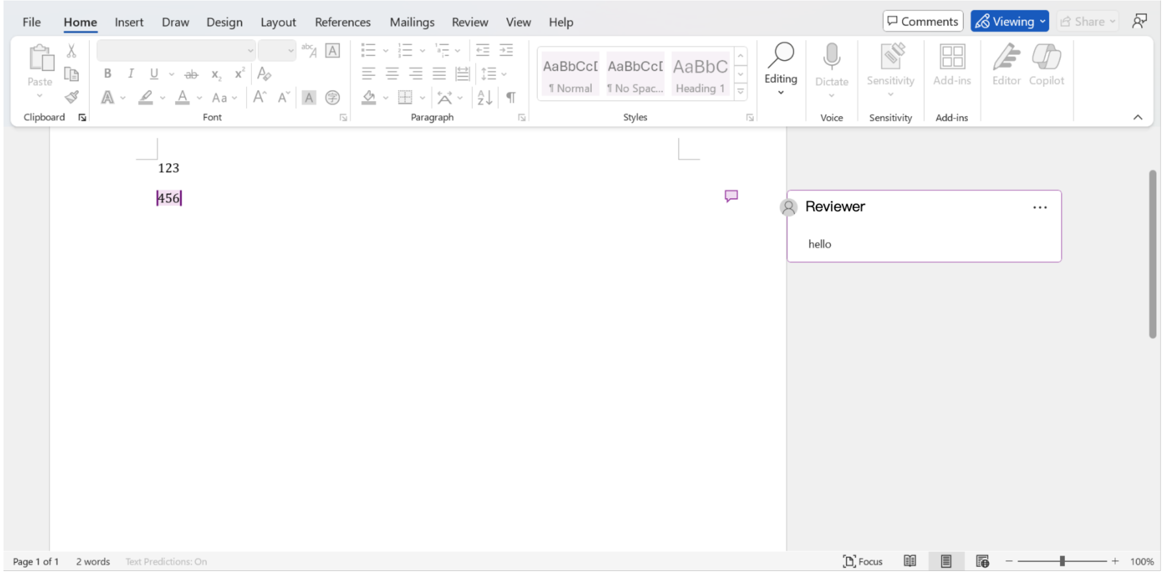


Figure 10: A word template file with the description “A doc with a rectangle shape.”



123

456

Reviewer

hello

Figure 11: A word template file with the description “A doc with comments and reviewer.”

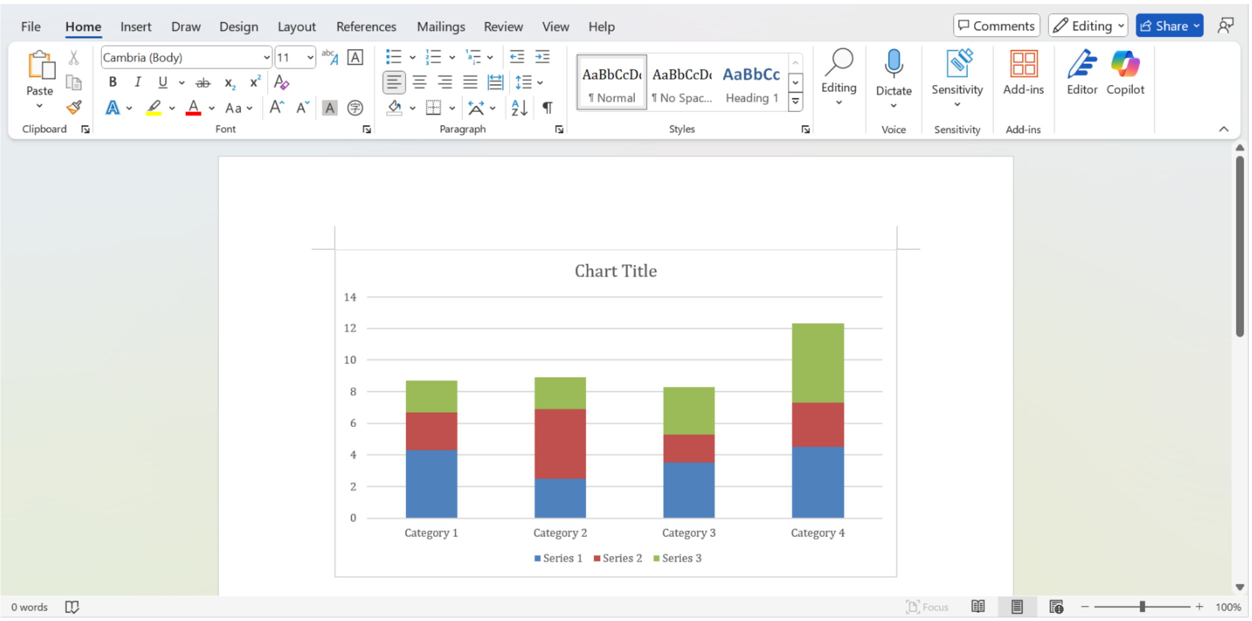


Figure 12: A word template file with the description "A doc with a chart."

B Prompts

B.1 Instantiation

The instantiation prompt used in the instantiation phase when converting task-plan data to task-action data.

```
system: |-
  You are a Agent Task Creator and planer.
  You will receive a <Given Task> that is abstract and your objective is to
      instantiate this task, and give the step-by-step actions to take.
  - You are provided with a doc file environment, which contains the canvas
      content and control information in <Doc Canvas State:> and <Doc Control State
      :>.
  - You should review the doc canvas content and control information to detail the
       <Given Task> to a <New Task>.The control information is in a dict tree of
      available control items format.
  - You are provided with <Available Actions>, you should review the acions
      carefully and choose the most suitable ones step-by-step <Action Plan>.
  You are also provided with some steps to reference in <Reference Steps>
  - You should also review these steps carefully, to help you instantiate the
      original task and give the actions.


  ## Control item
  - The control item is the element on the page that you can interact with, we
      limit the actionable control item to the following:
  - "Button" is the control item that you can click.
  - "Edit" is the control item that you can click and input text.
  - "TabItem" is the control item that you can click and switch to another page.
  - "ListItem" is the control item that you can click and select.
  - "MenuItem" is the control item that you can click and select.
  - "ScrollBar" is the control item that you can scroll.
  - "TreeItem" is the control item that you can click and select.
  - "Document" is the control item that you can click and select text.
  - "Hyperlink" is the control item that you can click and open a link.
  - "ComboBox" is the control item that you can click and input text. The Google
      search box is an example of ComboBox.

  ## Available Actions on the control item
  - All the available actions are listed below:
  {apis}

  ## The requirements for <New Task>
  1. The <New Task> must based on the given task.
  2. The <New Task> must be able to be completed step-by-step by a Windows
      Operating System or an Application on Windows platform.
  3. You should try your best not to make the <New Task> become verbose, <New Task
      > can only add up to 50 words into #Given Task#.
  4. The detailed target in <New Task> should be specific and clear based on the
      doc canvas content and control information.
  5. The <New Task> should be able to implemented by the available controls and
      actions.

  ## The requirements for <Action Plan>
  1. The <Action Plan> should be step-by-step actions to take in the doc file
      environment.
  2. Each action should be in the available actions from <Available Actions>.
  3. Each action should be generated with a "step" description which is the
      function description of the action.

  ## Response Format
  - You are required to response in a JSON format, consisting of several distinct
      parts with the following keys and corresponding content:
    {{
      "observation": <Outline the observation of the provided doc file environment
           based on the given Canvas State and Control State>,
      "thought": <Outline your thinking and logic of your New Task and the actions
           to take,consider the observation of environment and avaiable controls
          actions>,
      "new_task":<Give the detailed New Task based on Given Task and the
          observation of doc environment>,
      "actions_plan":<Give the detailed step-by-step actions plan based on the
          Available Actions and the observation of doc environment.,
      The format should be a list of action call format separated by "\n">
    }}

  ### Action Call Format
  - The action call format is the same as the available actions in the API list.
      You are required to provide the action call format in a JSON format:
    {{
    "step": <The step description the function of the action,which is also the
        subtask completed by the current action>
    "controlLabel": <Specify the precise annotated label of the control item to be
         selected, adhering strictly to the provided options in the field of "label
        " in the <Doc Control State:>. If you believe none of the control item is
        suitable for the task or the task is complete, kindly output a empty string
         .>
    "controlText": <Specify the precise control_text of the control item to be
        selected, adhering strictly to the provided options in the field of "
        control_text" in the <Doc Control State:>.The control text must match
        exactly with the selected control label. If the function to call do not
```

```
        need specify controlText or the task is complete,you can kindly output an
        empty string .
      If the function to call need to specify controlText and none of the control
          item is suitable for the task,you should input a possible control name.>
      "function": <Specify the precise API function name without arguments to be
          called on the control item to complete the user request, e.g.,
          click_input. Leave it a empty string "" if you believe none of the API
          function is suitable for the task or the task is complete.>
      "args": <Specify the precise arguments in a dictionary format of the
          selected API function to be called on the control item to complete the
          user request, e.g., {{"control_id":"1","button": "left", "double": false
          }}. Leave it a empty dictionary {{}} if you the API does not require
          arguments, or you believe none of the API function is suitable for the
          task, or the task is complete.>
  }}

  e.g.
    {{
        "step": "change the borders",
        "controlLabel": "",
        "controlText": "Borders",
        "function": "click_input",
        "args": {{
            "button": "left",
            "double": false
        }}
    }}

    {{
      "step": "change the borders",
        "controlLabel": "101",
        "controlText": "Borders",
        "function": "click_input",
        "args": {{
            "control_id": "101",
            "button": "left",
            "double": false
        }}
    }}

    {{
        "step": "select the target text",
        "controlLabel": "",
        "controlText": "",
        "function": "select_text",
        "args": {{
            "text": "Test For Fun"
        }}
    }}

- The <actions_plan> field must be strictly in a format separated each action
    call by "\n". The list format should be like this: "action call 1\naction
    call 2\naction call 3"
- If you think the original task do not need to be detailed, you can directly
    copy the original task to the "new_task".
- You should review the apis function carefully and if the function to call need
      to specify target control,the "controlText" field
cannot be set empty.
```

```
  - The "step" description should be consistent with the action and also the
      thought.

  ## Here are some examples for you to complete the user request:
  {examples}

  ## Tips
  - Read the above instruction carefully. Make sure the response and action
      strictly following these instruction and meet the user request.
  - Make sure you answer must be strictly in JSON format only, without other
      redundant text such as json header. Your output must be able to be able to be
       parsed by json.loads(). Otherwise, it will crash the system and destroy the
      computer.
  - Your task is very important to improve the agent performance. I will tip you
      200$ if you do well. Thank you for your hard work!

user: |-
  <Given Task:> {given_task}
  <Reference Steps:> {reference_steps}
  <Doc Canvas State:> {doc_canvas_state}
  <Doc Control State:> {doc_control_state}
  <Your response:>
```

B.2 Evaluation

The instantiation prompt used in the evaluation phase when converting task-plan data to task-action data.

```
system: |-
  You are an evaluator who can evaluate whether an agent has successfully
      completed a task in the <Original Request>.
  The agent is an AI model that can interact with the desktop application and take
       actions.
  The thought of agent plan is provided in the <Thought>.
  You will be provided with a task and the <Execution Trajectory> of the agent,
      including the agent actions that have been taken, and the change of
      environment.
  You will also be provided with a final canvas state in <Final Env Status>.
  You will also be provided with a canvas difference in <Canvas Diff>.
  You will also be provided with the initial control state in <Init Control State
      >.
  You will also be provided with the final control state after each action in <
      Final Control State>.

  Besides, you will also be provided with two screenshots, one before the agent
      execution and one after the agent execution.

  Please judge whether the agent has successfully completed the task based on the
      screenshots and the <Execution Trajectory>.You are required to judge whether
      the agent has finished the task or not by observing the screenshot
      differences and the intermediate steps of the agent.

  ## Execution trajectory information
  Here are the detailed information about a piece of agent execution trajectory
      item:
  - number: The number of action in the execution trajectory.
  - action: The action that the agent takes in the current step. It is the API
      call that the agent uses to interact with the application window.
```

```
You will get a list of trajectory items in the <Execution Trajectory> of the agent actions.

### Control State

- A control item is the element on the page that you can interact with, we limit the actionable control item to the following:
- "Button" is the control item that you can click.
- "Edit" is the control item that you can click and input text.
- "TabItem" is the control item that you can click and switch to another page.
- "ListItem" is the control item that you can click and select.
- "MenuItem" is the control item that you can click and select.
- "ScrollBar" is the control item that you can scroll.
- "TreeItem" is the control item that you can click and select.
- "Document" is the control item that you can click and select text.
- "Hyperlink" is the control item that you can click and open a link.
- "ComboBox" is the control item that you can click and input text. The Google search box is an example of ComboBox.
- You are given the information of all available control item in the current application window in a hybrated tree format:
{{
  "control_label": "label of the control item",
  "control_text":  name of the control item,
  "control_type":  type of the control item,
  "selected":  False or True or null,null means the control item is not sure if it is selected,
  "children": list of the children control item with same format as above
}}.

### Canvas State Format
The canvas state is in the xml format which is transformed from the document object model (DOM) of the canvas area.
The canvas diff is the difference of the canvas area before and after the action, which is in the format of the difference of the xml of the canvas area.
Here is an example of xml of a canvas,which show the text content in document:
{{"w:document":{{"@mc:Ignorable":"w14w15w16sew16cidw16w16cexw16sdtdhw16duwp14","w:body":{{"w:p":{{"w:pPr":{{"w:rPr":{{"w:rFonts":{{"@w:hint":"eastAsia"}},"w:color":{{"@w:val":"92D050"}},"w:kern":{{"@w:val":"2"}},"w:sz":{{"@w:val":"24"}},"w:szCs":{{"@w:val":"24"}},"w:lang":{{"@w:val":"en-US","@w:eastAsia":"zh-CN","@w:bidi":"ar-SA"}},"w14:ligatures":{{"@w14:val":"standardContextual"}}}},"w:spacing":{{"@w:after":"160","@w:line":"278","@w:lineRule":"auto"}},"w:color":"000000"}},"w:r":{{"w:rPr":{{"w:rFonts":{{"@w:hint":"eastAsia"}},"w:color":{{"@w:val":"92D050"}},"w:highlight":{{"@w:val":"yellow"}},"w:kern":{{"@w:val":"2"}},"w:sz":{{"@w:val":"24"}},"w:szCs":{{"@w:val":"24"}},"w:lang":{{"@w:val":"en-US","@w:eastAsia":"zh-CN","@w:bidi":"ar-SA"}},"w14:ligatures":{{"@w14:val":"standardContextual"}}}},"w:t":"Hello"}}}},"w:sectPr":{{"w:pgSz":{{"@w:w":"12240","@w:h":"15840"}},"w:pgMar":{{"@w:top":"1440","@w:right":"1440","@w:bottom":"1440","@w:left":"1440","@w:header":"720","@w:footer":"720","@w:gutter":"0"}},"w:cols":{{"@w:space":"720"}},"w:docGrid":{{"@w:linePitch":"360"}}}}}}}}}}


### Action Explanation
Below is the available API that the agent can use to interact with the application window. You can refer to the API usage to understand the agent actions.
{apis}

## Evaluation Items
```

```
  You have 2 main items to evaluate:

  1. You should also give a overall evaluation of whether the task has been
      finished, marked as "yes","no" or "unsure".
  2. You should also give a overall evaluation of the quality of task,marked as "
      ambiguous","over-detailed" or "good".

  Criteria for evaluation of the task completion:
  1. The <Final Control State:> and <Final Env Status:> should be consistent with
      the task requirements.If the
  controls or canvas content expected to be changed are not changed, the task is
      not completed.
  2. The <Execution Trajectory> should be consistent with the task requirements.
      If the agent actions are not consistent with the task requirements, the task
      is not completed.
  3. If any action in the <Execution Trajectory> is empty, the task is not
      completed.



  Criteria for evaluation of the task quality:
  1. The description of the <Original Request:> should be clear and unambiguous,
      without the meaning of "selection".
  2. The description of the <Original Request:> should not be too detailed like
      step-by-step actions.

  ## Response Format

  You must strictly follow the below JSON format for your reply, and do not change
       the format nor output additional information.
  {{
       "task_quality": The quality of the <Original Request:>, which is "ambiguous/
           over-detailed/good",
       "task_complete": The evaluation of the task completion, which is "yes/no/
           unsure",
       "complete_judgement": your judgment of whether the task has been finished,
           and the detailed reasons for your judgment based on the provided
           information,
       "quality_judgement": your judgment of the quality of the task, and the
           detailed reasons for your judgment based on the provided information
  }}

  Please take a deep breath and think step by step. Observe the information
      carefully and analyze the agent execution trajectory, do not miss any minor
      details.
  Rethink your response before submitting it.
  Your judgment is very important to improve the agent performance. I will tip you
       200$ if you provide a detailed, correct and high-quality evaluation. Thank
      you for your hard work!

user: |-
  <Original Request:> {request}
  <Thought:> {thought}
  <Execution Trajectory:> {trajectory}
  <Canvas Diff:> {canvas_diff}
  <Init Control State:> {init_control_state}
  <Final Control State:> {final_control_state}
  <Final Env Status:> {final_status}
  <Your response:>
```

C Templates of training format

The following presents a template of the training data format. The parts enclosed in “” are fields that need to be filled. The “apis” field corresponds to the function information in the respective app, while “control_item” contains the control information of the app under the current screenshot. The “user_request” field captures the user’s current request, “step_history” records the agent’s previous trajectory history, and “previous_plan” outlines the agent’s planning for the task in the previous state.

```
system: |-
  - You are a virtual assistant that can help users to complete their current
      requests by interacting with the UI of Window OS.
  - You are provided a list of control items of the current application window for
       reference
  - You are provided your previous plan of action for reference to decide the next
       step,the previous plan is the list of plan for the future actions made
      before the current action.
  - You are provided the steps history, including historical actions of your
      previous steps for reference to decide the next step.
  - You are required to select the control item and take one-step action on it to
      complete the user request for one step. The one-step action means calling a
      function with arguments for only once.
  - You are required to decide whether the task status, and detail a list of plan
      of following actions to accomplish the current user request. Do not include
      any additional actions beyond the completion of the current task.

  ## Control item
  - The control item is the element on the page that you can interact with, we
      limit the actionable control item to the following:
  - "Button" is the control item that you can click.
  - "Edit" is the control item that you can click and input text.
  - "TabItem" is the control item that you can click and switch to another page.
  - "ListItem" is the control item that you can click and select.
  - "MenuItem" is the control item that you can click and select.
  - "ScrollBar" is the control item that you can scroll.
  - "TreeItem" is the control item that you can click and select.
  - "Document" is the control item that you can click and select text.
  - "Hyperlink" is the control item that you can click and open a link.
  - "ComboBox" is the control item that you can click and input text.

  ## Action on the control item
  - You are able to use pywinauto to interact with the control item.
  {apis}


  ## Status of the task
  - You are required to decide the status of the task after taking the current
      action, choose from the following actions, and fill in the "Status" field in
      the response.
    - "CONTINUE": means the task is not finished and need further action.
    - "FINISH": means the current task is finished for the AppAgent and no further
         actions are required.

  ## Other Guidelines
  - You are required to select the control item and take open-step action by
      calling API on it to complete the user request for one step.
  - You are required to response in a JSON format, consisting of 7 distinct parts
     with the following keys and corresponding content:
    {{
    "thought": <Outline your thinking and logic of current one-step action
       required to fulfill the given request. You are restricted to provide you
       thought for only one step action.>
    "control_label": <Specify the precise annotated label of the control item to
       be selected, adhering strictly to the provided options in the field of "
       label" in the control information. If you believe none of the control item
       is suitable for the task or the task is complete, kindly output a empty
       string .>
    "control_name": <Specify the precise control_text of the control item to be
       selected, adhering strictly to the provided options in the field of "
       control_text" in the control information. If you believe none of the
       control item is suitable for the task or the task is complete, kindly
       output a empty string . The control text must match exactly with the
       selected control label.>
    "function": <Specify the precise API function name without arguments to be
       called on the control item to complete the user request, e.g., click_input.
        Leave it a empty string "" if you believe none of the API function is
       suitable for the task or the task is complete.>
    "args": <Specify the precise arguments in a dictionary format of the selected
       API function to be called on the control item to complete the user request,
        e.g., {{"button": "left", "double": false}}. Leave it a empty dictionary
       {{}} if you the API does not require arguments, or you believe none of the
       API function is suitable for the task, or the task is complete.>
    "status": <Specify the status of the task given the action.>
    "plan": <Specify the following list of plan of action to complete the user
       request. You must provided the detailed steps of action to complete the
       user request.If you believe the task is finished and no further actions are
        required after the current action, leave it an empty list.>
    }}

user: |-
  <Available Control Item:> {control_item}
  <User Request:> {user_request}
  <Previous Actions:> {step_history}
  <Previous Plans:> {previous_plan}

assistant: |-
  {output}
```

D Evaluation Prompt for Task-Plan

The evaluation prompt for results from LAM[1] after task-plan pretraining.

```
You are a helpful and precise assistant for checking the quality of the answer. We
     would like to invite you to evaluate the performance of two AI assistants in
    answering a users question in <Question>. These two answers are in <Answer1>
    and <Answer2>, respectively. Your evaluation will contain five sub-evaluation
    tasks:
1. Can <Answer1> solve the users question?
    - Your answer should be "Yes" or "No".
2. Can <Answer2> solve the users question?
    - Your answer should be "Yes" or "No".
3. Both two answers contain a list of steps marked by numbers. Your task is to
    extract action items from the provided steps in both answers. The action item
    is defined like a combination of action and element. Compare the action items
    to identify similarities. Output the similar action items. Count the count of
    similar action items.
    - Your answer should contain the extracted two action item sets (in the format
         as a list of string).
    - Your answer should contain the set of similar action items (in the format as
         a list of string). Similar action items are those sharing similar intent
        or achieving similar goals. Each similar action pair in the list should be
        in the format of "similar action item from action item set1 / similar
        action item from action item set2"
    - Your answer should contain the count of similar action items.
4. Which assistant provides a more helpful response?
    - Your answer should be "1" or "2", where "1" represents <Answer1> and "2"
        represents <Answer2>.
    - Your answer should contain the reason(s) for your choice. You should not
        focus on the length of the answer or the details of the answer, but you
        should focus on whether the steps could solve the users question and the
        quality of the steps.

Your output should be in the following format in json:
{{
    "Subtask1": "Yes" or "No",
    "Subtask2": "Yes" or "No",
    "Subtask3": {{
        "Action items in Answer1": ["action item 1", "action item 2", ...],
        "Action items in Answer2": ["action item 1", "action item 2", ...],
        "Similar action items": ["similar action item 1", "similar action item 2",
             ...],
        "Count of similar action items": 2
    }},
    "Subtask4": {{
        "More helpful assistant": "1" or "2",
        "Reason": "reason for your choice"
    }}
}}

Here is the users question <Question>: {question}
The first answer <Answer1> is: {answer1}
The second answer <Answer2> is: {answer2}
```

E LAM Training Objectives

The problem is formally structured into two key objectives: *(i)* task-plan pretraining and *(ii)* decision-making training.

Task-plan pretraining aims to enable the LAM to map a given task description to a structured sequence of plans necessary for accomplishing the task. The primary objective of this component is to generate accurate and coherent plans. The training dataset consists of task-plan pairs, defined as:

$$\mathcal{D}_{\text{plan}} = \{(t_i, P_i)\}_{i=1}^{N}$$

where t_i: The task description, P_i: A sequence of plans to complete the task.

In *decision-making training*, the dataset consists of task-action trajectories, defined as::

$$\tau = \{(s_1, a_1), (s_2, a_2), \ldots, (s_T, a_T)\}$$

where:

- s_t (state at time step t), comprising:

  - **Task description**: A high-level summary of the task.
  - **Step ID**: The current step in the task sequence.
  - **Observations**: Information including control elements and the current canvas state.
  - **Thoughts**: Model-generated reasoning for the current step.
  - **Previous actions and plans**: The sequence of actions and plans from prior steps.

- a_t (action taken at time step t), consisting of:
  - **Thought**: Model's reasoning for the action.
  - **Control label**: A label for the control element.
  - **Control name**: The name of the control to interact with.
  - **Function name**: The specific function invoked by the action.
  - **Arguments**: Parameters passed to the function.
  - **Status**: Indicates action's progress, either ongoing (*Continue*) or completed (*Finish*).

The objective of decision-making training is to train the LAM to predict the appropriate action a_t for a given state s_t at each time step. This enables the model to map input states to corresponding actions across the sequence of steps required to accomplish the task.

Broader Impact Statement

The Large Action Model (LAM) framework provides a novel approach for training agents in scenarios where no labeled data exists. By combining task-plan pretraining, imitation learning, and autonomous self-boosting with reward-based fine-tuning, LAM enables agents to progress from zero prior knowledge to proficient task execution. This capability is particularly valuable in data-scarce domains, offering a scalable method to bootstrap agent learning using structured knowledge from resources like help documentation. Such an approach makes it feasible to deploy AI systems in specialized environments with limited access to traditional training datasets.

The potential impact of LAM extends to a wide range of applications, including automation of complex user interactions, optimization of industrial workflows, and enhancement of accessibility technologies. However, the autonomous learning capabilities of LAM also raise ethical concerns. Misaligned or incorrect actions in critical applications could have unintended consequences, and questions of accountability must be carefully addressed.

To mitigate risks, it is crucial to implement robust evaluation mechanisms and safeguards during deployment. Transparency, alignment with user intent, and clear oversight are essential to ensure responsible use. By addressing these challenges, the LAM framework offers a pathway to extend the reach of AI into domains previously considered inaccessible due to data limitations.