AppAgentX: Evolving GUI Agents as Proficient Smartphone Users

Wenjia Jiang[1,2]**, Yangyang Zhuang**[1]**, Chenxi Song**[1]**, Xu Yang**[3]**, Joey Tianyi Zhou**[4,5] **Chi Zhang**[1]**,**

[1]Westlake University, China, [2]Henan University, China, [3]Southeast University, China
[4]IHPC, Agency for Science, Technology and Research, Singapore
[5]CFAR, Agency for Science, Technology and Research, Singapore
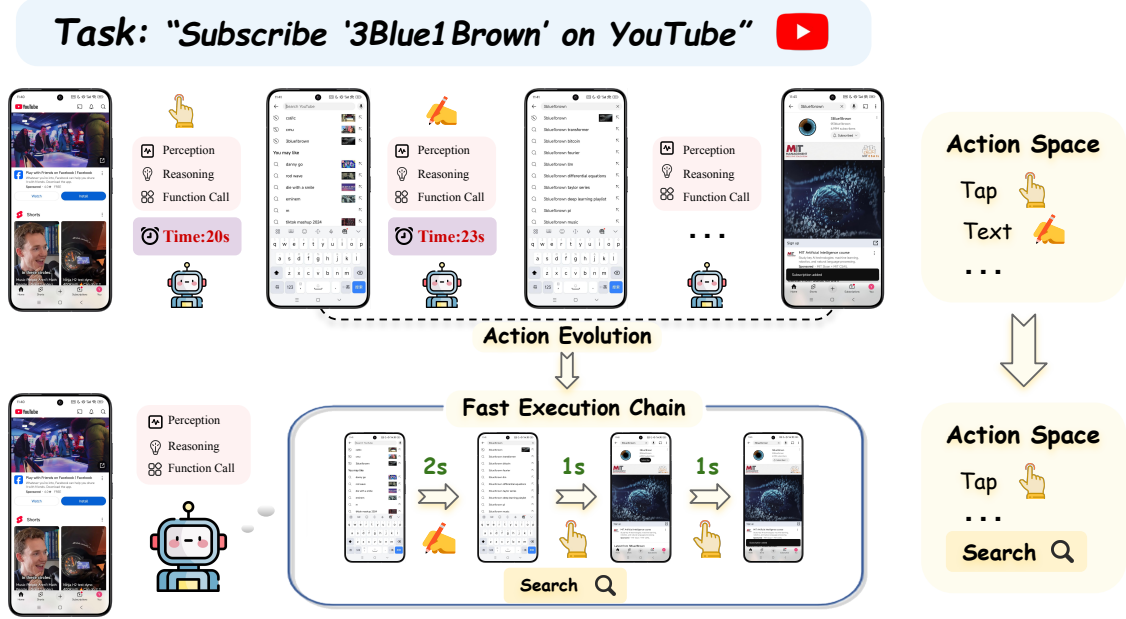{jiangwenjia, chizhang}@westlake.edu.cn,

Figure 1: **Illustration of the proposed evolutionary mechanism in GUI agents.** The agent evolves a high-level action, "Search", which replaces a sequence of inefficient low-level actions. This evolution eliminates the need for step-by-step reasoning, significantly enhancing the agent's efficiency.

Abstract

Recent advancements in Large Language Models (LLMs) have led to the development of intelligent LLM-based agents capable of interacting with graphical user interfaces (GUIs). These agents demonstrate strong reasoning and adaptability, enabling them to perform complex tasks that traditionally required predefined rules. However, the reliance on step-by-step reasoning in LLM-based agents often results in inefficiencies, particularly for routine tasks. In contrast, traditional rule-based systems excel in efficiency but lack the intelligence and flexibility to adapt to novel scenarios. To address this challenge, we propose a novel evolutionary framework for GUI agents that enhances operational efficiency while retaining intelligence and flexibility. Our approach incorporates a memory mechanism that records the agent's task execution history. By analyzing this history, the agent identifies repetitive action sequences and evolves high-level actions that act as shortcuts, replacing these low-level operations and improving efficiency. This allows the

agent to focus on tasks requiring more complex reasoning, while simplifying routine actions. Experimental results on multiple benchmark tasks demonstrate that our approach significantly outperforms existing methods in both efficiency and accuracy. The code will be open-sourced to support further research.

1 Introduction

Recent advancements in artificial intelligence have been significantly influenced by the development of LLMs such as GPT-4 (OpenAI, 2024) and DeepSeek-V3 (DeepSeek-AI, 2024). These models, renowned for their ability to process and generate human-like text, have catalyzed innovations across various domains, including natural language understanding, generation, and reasoning. One promising application is the creation of LLM-based agents, which use natural language to autonomously interact with users and perform a wide range of tasks. Unlike traditional rule-based systems, LLM-based agents exhibit the flexibility to

1

understand complex tasks and generalize to novel scenarios, enhancing human-computer interaction.

A notable development in this area is the emergence of LLM-based agents capable of operating graphical user interfaces (GUIs). Unlike Robotic Process Automation (RPA), which relies on predefined rules, these agents can engage with GUIs dynamically, mimicking human-like interactions by outputting low-level actions such as clicks, drags, and scrolls. This approach not only increases operational flexibility but also enables the execution of tasks that require adaptability and reasoning, without needing access to backend systems or APIs.

To achieve optimal performance, various mechanisms have been introduced to guide the LLMs in generating accurate and contextually appropriate actions. Techniques such as reflection(Huang et al., 2022) and chain-of-thought(Wei et al., 2023) reasoning have been employed to help the model carefully consider the implications of each action before execution. While these methods can significantly enhance the agent's decision-making abilities, they often come at the cost of efficiency, particularly for tasks that do not require advanced reasoning. For example, as illustrated in Figure 1, during simple tasks like search operations, the agent may need to reason each step, such as clicking the search box, typing text, and pressing submit, leading to unnecessary delays. While traditional RPA methods can execute these fixed steps rapidly, they lack the flexibility to handle tasks requiring intelligent judgment. This highlights the need for a more efficient, adaptable approach to automate routine tasks.

In response to this challenge, we propose a novel, evolving framework for GUI agents that aims to enhance both the efficiency and intelligence of the agent's behavior.

Our approach enables the agent to learn from their past interactions and dynamically evolve more abstract, high-level actions, eliminating the need for repetitive low-level operations. Specifically, the agent will analyze its execution history to identify patterns in repetitive, low-intelligence actions, such as those involved in routine tasks. From this analysis, the agent can generate a high-level action, encapsulating a series of low-level actions, enabling it to perform tasks more efficiently. For example, in a search operation, the agent would automatically evolve a "search" action that directly executes the required sequence, improving both speed and accuracy.

Intuitively, our framework enables the agent to focus its reasoning capabilities on tasks requiring more intelligent judgments, while simplifying repetitive tasks into a more compact form. To support this approach, we design a knowledge base structured as a chain to record task execution history and facilitate the abstraction and evolution of behaviors. This knowledge base allows the agent to continuously improve its task execution strategies, further optimizing its performance and enabling the evolution of more intelligent, high-level actions over time.

Our approach relies entirely on visual information, eliminating the need for backend access or APIs. Extensive experiments show that it outperforms baseline and state-of-the-art (SOTA) methods in both efficiency and accuracy across several benchmark tasks.

The main contributions of this paper are summarized as follows:

- We propose an evolutionary mechanism that enables a GUI Agent to learn from its task execution history and improve its efficiency by abstracting repetitive operations.

- We design a chain-based framework for recording and optimizing the agent's execution behavior.

- Our code will be open-sourced to facilitate further research in this area.

2 Related Works

Large language models. Recent advancements in large language models (LLMs) have significantly expanded the scope of AI-driven automation. Models such as GPT-4 (OpenAI, 2024) and DeepSeek-V3 (DeepSeek-AI, 2024) exhibit strong natural language understanding and reasoning abilities. These capabilities allow LLMs to process complex UI structures and facilitate interactive decision-making, forming the foundation for LLM-driven GUI Agents (Naveed et al., 2024). Unlike traditional script-based or rule-based approaches (Tentarelli et al., 2022; Hellmann and Maurer, 2011), LLM-powered agents can generalize across diverse applications and dynamic interfaces without explicitly predefined rules. However, challenges remain in model efficiency, adaptability, and spatial reasoning, necessitating further optimization in both architectural design and training methodologies.

LLMs as Agents. LLMs have significantly advanced intelligent agents, enabling complex task execution. AutoGPT(Yang et al., 2023), AFlow(Zhang et al., 2024b), MetaGPT(Hong et al., 2024), and AutoAgent(Chen et al., 2024) exemplify autonomous task decomposition and execution, while Stanford Smallville(Park et al., 2023) and Agent Hospital(Li et al., 2025) showcase multi-agent simulations. LLM-driven multimodal agents also enhance perception and decision-making across domains. GPT-Driver(Mao et al., 2023) enables adaptive motion planning for autonomous driving, SmartPlay(Wu et al., 2024) improves agent intelligence in gaming, and MP5(Qin et al., 2024) integrates active perception for efficient task execution in robotics.

LLMs as GUI Agents. Beyond general agents, LLMs have also enhanced GUI automation, surpassing traditional script-based methods in flexibility and adaptability. WebVoyager(He et al., 2024), AppAgent(Zhang et al., 2023), and MobileAgent(Wang et al., 2024b) leverage multimodal perception for interactive interfaces, while UFO(Zhang et al., 2024a), AutoGLM(Liu et al., 2024), and MMAC-Copilot(Song et al., 2025) improve cross-platform adaptability and multi-agent collaboration. To refine UI understanding, OmniParser(Lu et al., 2024) and Ferret-UI(You et al., 2024a) enhance element recognition, while TinyClick(Pawlowski et al., 2024) and CogAgent(Hong et al., 2023) improve interaction precision and vision-based task execution. Additionally, WebGUM(Furuta et al., 2024), MobileVLMS(Chu et al., 2024), and DigiRL(Bai et al., 2024) optimize performance in dynamic web and mobile environments. However, most existing agents rely on static training data and lack continuous adaptation. To address this limitation, we propose AppAgentX, integrating task trajectory memory to enhance efficiency and adaptability in GUI automation.

3 Preliminary

Before delving into our proposed methodology, we first introduce a baseline method for LLM-based GUI agents. This baseline serves as a foundation for understanding the core components and tasks involved in enabling LLMs to control smartphones.

The process of utilizing LLMs for smartphone control involves two key stages: screen perception and action execution. The screen perception phase begins with capturing a screenshot of the device's current interface. In order to accurately interpret this screenshot, we employ OmniParser (Lu et al., 2024) to detect and label all interactive elements within the interface, such as buttons and text boxes. OmniParser annotates these elements with tagged bounding boxes, which are subsequently overlaid onto the original screenshot for clear visualization. Following this, the annotated screenshot is passed to the LLM for action planning. At this stage, the LLM interprets the UI components and generates corresponding actions based on its understanding of the interface.

In the second stage, action execution, we follow AppAgent (Zhang et al., 2023) to define a set of low-level actions that the agent can perform within the smartphone environment. These actions include common gestures such as tapping, long-pressing, swiping, text input, and navigating back. These actions collectively define a basic, app-agnostic action space to simulate typical human interactions with a smartphone interface. Formally, the low-level action space is defined as follows:

$$\mathcal{A}_{\text{basic}} = \{a_{\text{tap}}, a_{\text{long\_press}}, a_{\text{swipe}}, a_{\text{text}}, a_{\text{back}}\}, \quad (1)$$

where $\mathcal{A}_{\text{basic}}$ represents the set of atomic actions available to the agent.

The LLM employs a structured process of observation, reasoning, and function-calling to interact with the smartphone interface. In this process, the LLM iteratively analyzes the current UI, reasons about the appropriate next steps to achieve the desired task, and invokes the corresponding actions from the defined action space. This cycle continues until the task is completed successfully. An illustration of this process is shown in Figure 2.

4 Methodology

This section outlines the core methodology of the proposed evolutionary framework. The framework comprises three main components: a memory mechanism for recording the agent's operational history, an evolutionary mechanism for improving performance, and an execution strategy for utilizing the evolved agent. We will detail each of these elements in the following subsections.

4.1 Memory mechanism

To support the agent's evolution from inefficient to efficient operational modes, it is essential for the agent to retain a record of its past actions and the corresponding outcomes. This enables the agent

Figure 2: **Overview of the LLM-based GUI agent baseline.** At each step, the agent captures the current screen of the device and analyzes the interface to select an appropriate action from the predefined action space. The chosen action is then executed to interact with the GUI.

to learn from its experiences and improve future interactions. To achieve this, we propose a memory mechanism designed to capture and store the agent's trajectory during its interactions with the environment.

The agent's interaction with the UI is modeled as a series of page transitions, where each UI page is represented as a "page node". Interactions performed on these pages, such as button clicks or text input, lead to transitions between these nodes. Each page node is characterized by several key attributes, including:

➢**Page Description:** This attribute stores a text description of the entire UI page, initially setting it to empty at the beginning.

➢**Element List:** This property holds a JSON list containing information of all elements detected by OmniParser(Lu et al., 2024), such as the relative screen position, OCR results, *etc.*

➢**Other Properties:** The page nodes also include other properties required for logging, such as page screenshots, ID, timestamps, *etc.*

For more detailed information, we introduce "element nodes". Each element node corresponds to a specific UI element, such as a button, text field, or icon. The interactions with these UI elements lead to the page transitions. Similarly, each element node encapsulates essential attributes:

➢**Element Description:** This attribute records a textual description of the element's functionality, providing a semantic understanding of the element's purpose within the UI. Similar to page

descriptions, this attribute is initialized as empty.

➢**Element Visual Embeddings:** This property stores the identifier of the element's screenshot in the vector database. The visual features are extracted using a pre-trained ResNet50 (Microsoft, 2024) model.

➢**Interaction Details:**This property includes information related to the basic action for the current element, *e.g.*, tapping, along with the corresponding arguments and other relevant interaction details.

This structure enables the agent to record and learn from both the high-level transitions (from one page to another) and the low-level interactions (with individual UI elements) that lead to those transitions. Figure 3 (c) illustrates this process.

Extracting Features and Descriptions. We next leverage the LLM to generate functional descriptions of pages and individual elements based on observed action sequences. Specifically, we follow the approach in (Zhang et al., 2023) by decomposing the agent's trajectory into multiple overlapping triples. Each triple consists of a source page, an action performed on an element, and a target page. These triples capture the changes in page states before and after an action is executed.

The resulting triples are then passed to the LLM for reasoning. The model generates detailed descriptions and functionalities for both the page and element nodes based on the context of the action. This process, illustrated in Figure 3 (a), enables the system to build accurate and contextually aware descriptions.

Merging Overlapping Descriptions. Since the descriptions of page nodes are generated from multiple overlapping action triples, it is likely that the descriptions for the same page node will be generated twice, based on different contexts. Therefore, it is necessary to merge them to generate a unified description for each page node. To achieve this, we instruct the LLM to combine the descriptions generated from different triples, taking into account both the specific context of each individual action and the broader global task the agent is performing. This allows the LLM to generate a more enriched and complete description of the page, considering its function in the overall task. The merged descriptions provide a detailed, unified record of the agent's interactions with the UI, contributing to a coherent chain of node attributes that document the agent's progress as it completes the task.
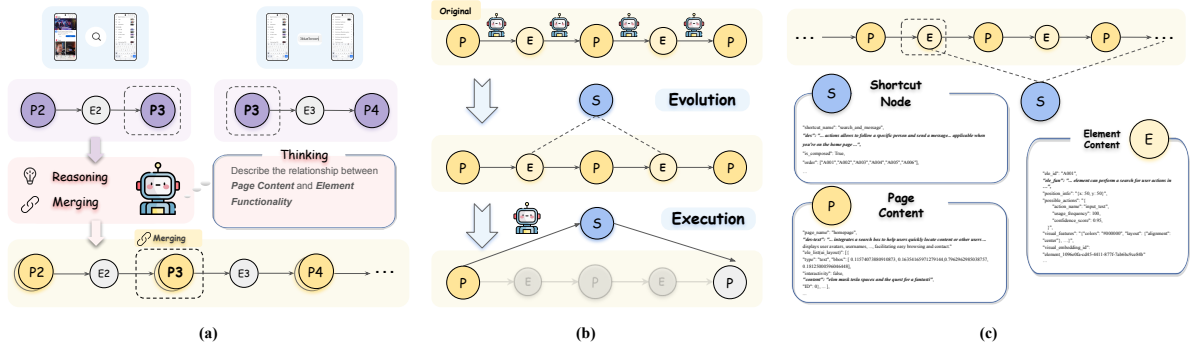
Figure 3: **Overview of the proposed framework.** (a) The trajectory of a task execution is decomposed into multiple overlapping triples. Based on these triples, the LLM generates functional descriptions of both pages and UI elements. Descriptions of pages that are repeatedly generated are then merged. The entire interaction history is recorded using a chain of nodes. (b) The proposed evolutionary mechanism and execution process. The evolutionary mechanism generates shortcut nodes, which allow the agent to execute a series of actions efficiently without reasoning step by step. (c) An example of the content of nodes within the chain. Each node records essential information, including descriptions of pages, UI elements, and high-level actions, to facilitate understanding of the agent's interactions.

4.2 Evolutionary Mechanism

The primary goal of the evolutionary mechanism is to improve the agent's accuracy and efficiency in executing similar tasks by learning from its previous execution histories. In typical task execution scenarios, the action sequence may contain repetitive patterns that do not require deep reasoning at each step. For example, to search for a content item, the agent may repeatedly perform actions such as tapping the search box, inputting text, and tapping the search button. By identifying and exploiting these repetitive patterns, the agent can replace inefficient action sequences with higher-level actions that streamline the process, thereby improving overall task efficiency and accuracy. To achieve this, we introduce the concept of a *shortcut node*, which represents a high-level action that bypasses previously inefficient action sequences. The shortcut node is designed to streamline its decision-making process, eliminating unnecessary reasoning process and accelerating task completion.

To identify whether an action sequence contains repetitive patterns that can be replaced by shortcut nodes, we design mechanisms that evaluate the trajectory of each task. Initially, LLM, drawing from prior knowledge, is tasked with determining whether a given task contains repetitive patterns that may be optimized through the introduction of shortcut nodes. If the task is deemed to contain such patterns, the next step involves inspecting the actual trajectory data. The trajectory data, which

includes the descriptions of all relevant pages and elements, is input into the LLM. Upon receiving this data, the model is prompted to generate the description of a shortcut node, specifying the scenarios in which the shortcut node can be invoked and the specific low-level action sequences that it is intended to replace. This process effectively constructs an evolved action space by integrating new, higher-level actions into the agent's operational repertoire, allowing it to perform tasks more efficiently.

To formally define the construction of high-level actions and the expansion of the action space, we introduce the following representations. A high-level action $\tilde{a}$ is composed by abstracting a sequence of low-level actions from the basic action set $\mathcal{A}_{\text{basic}}$. This abstraction allows the agent to perform more complex tasks with fewer steps, leveraging previously learned patterns of interaction. We then define the expanded action space $\mathcal{A}_{\text{evolved}}$, which incorporates both the original low-level actions and the newly introduced high-level actions:

$$\mathcal{A}_{\text{evolved}} = \mathcal{A}_{\text{basic}} \cup \{\tilde{a}\}. \qquad (2)$$

Here, $\mathcal{A}_{\text{evolved}}$ represents the enriched action space that includes both basic, low-level actions and the newly composed high-level actions. This expanded action space enables the agent to perform tasks more efficiently by utilizing higher-level abstractions that reduce the need for repetitive action sequences.

4.3 Dynamic Action Execution

Following the evolution of the action space, the agent can now select high-level actions from the expanded action space to efficiently execute tasks.

4.3.1 Execution Process

The high-level execution process begins after the agent captures a screenshot of the current page. At this stage, the system then matches the parsed elements on the page to the stored element nodes in memory, by comparing their visual embeddings. Next, we check whether these identified element nodes are associated with any shortcut nodes. If an association between the element nodes and shortcut nodes exists, the system leverages the LLM to determine whether the corresponding high-level actions can be executed. This decision is made by inspecting the description of the shortcut node in conjunction with the current task context. If the conditions for executing the high-level action are met, the LLM generates an action execution template. This template includes the sequence of low-level actions to be executed by the shortcut node, along with the corresponding function arguments necessary for each action. In cases where multiple elements on the page are associated with shortcut nodes, the system prioritizes executing the actions based on the order of matching elements. This order is determined by their execution sequence, ensuring that the actions are performed in a logical and efficient manner. As the sequence actions progresses, partially repetitive operations or even the entire task can be completed more efficiently through the use of high-level actions.

4.3.2 Fallback Strategy

To ensure robustness and task completion reliability, we introduce a fallback strategy that allows the agent to dynamically recover from failures. If the conditions for executing a high-level action are not met, the agent will default to selecting actions from the basic action space $\mathcal{A}$. Additionally, in cases where execution errors occur due to incorrect shortcut node matching or unexpected UI responses, the agent reverts to using actions from the basic action space as a fallback. This ensures that the agent can still operate effectively, even when high-level abstractions cannot be applied to the current task.

During the execution of high-level paths, operations that would traditionally require multiple reasoning steps by the LLM are transformed into a page-matching and retrieval-based process. This shift significantly enhances the overall execution efficiency, as the agent can bypass repeated reasoning processes and rely on pre-determined, efficient action sequences.

5 Experiments

In this section, we will present our evaluation of the evolutionary framework through a combination of various experiments on multiple benchmarks.

5.1 Experimental Setup

Evaluation metrics. To ensure a fair and accurate comparison of our proposed method with baseline models and existing works, we adopt several evaluation metrics, which have been commonly used in prior research (Zhang et al., 2023; Wang et al., 2024b; Li et al., 2024). The metrics we report and compare are as follows:

➢**Average Steps per Task (Steps):** This measures the average number of operations the agent performs to complete a task.

➢**Average Overall Success Rate (SR):** This metric evaluates the proportion of tasks completed successfully by the agent.

➢**Average Task Time (Task Time):** The total time taken to complete a task, starting from the initiation of the task execution process to the determination of task completion by the agent.

➢**Average Step Time (Step Time):** The average time consumed per operation (step) during task execution.

➢**Average LLM Token Consumption (Tokens):** The total number of tokens used by the language model (LLM) during the task execution, including both prompt tokens and completion tokens.

For time-related comparisons, we focus only on tasks that are successfully executed by all methods. This ensures that the comparisons are not skewed by failures due to early terminations or excessive retry attempts, which might distort the results. To calculate token consumption, we compute the total number of prompt and completion tokens used by the LLM for each task and report the average number across all tasks. To mitigate the impact of random fluctuations, each task is repeated five times by default, and we report the averaged results across these repetitions. This helps ensure that our findings are statistically robust and not affected by outlier performance from individual task executions.

Benchmarks. We evaluate the performance of our

Table 1: **Analysis of Different Components in AppAgentX.** This table compares the performance differences resulting from the different designs with the baseline. In that table for the GPT-4o approach, we use direct LLM invocation. Both our memory design and evolution mechanism can improve success rate and efficiency.

Method	Memory Type	Action Space	Steps↓	Step Time (s)↓	Tokens (k)↓	SR ↑
GPT-4o	None	Basic	10.8	26	6.72	16.9%
AppAgent	Element	Basic	9.3	24	8.46	69.7%
AppAgentX	Chain	Basic	9.1	23	9.26	70.8%
AppAgentX	Chain	Basic+Evolve	**5.7**	**16**	**4.94**	**71.4%**

method on several widely used benchmarks to validate its effectiveness and generalizability. These benchmarks cover a diverse range of tasks and applications, providing a comprehensive assessment of our framework's capabilities. The benchmarks used in our experiments include:

➢**AppAgent Benchmark (Zhang et al., 2023):** This benchmark consists of 50 tasks across 9 different applications, including 45 general tasks and 5 long-duration tasks.

➢**DroidTask (Wen et al., 2024):** Comprising 158 high-level tasks derived from 13 widely used mobile applications.

➢**AndroidWorld (Rawles et al., 2025):** AndroidWorld is a fully functional and dynamic Android benchmark that supports 116 programmatic tasks across 20 widely used real-world Android applications.

➢**A3 (Android Agent Arena) (Chai et al., 2025):** It consists of 201 tasks derived from 20 widely used third-party applications, covering common user scenarios. The benchmark provides a comprehensive evaluation mechanism, which significantly accelerates our experimental work. **Implementation details.** The implementation of our framework leverages several key platforms. For the foundational LLM, we selected GPT-4o (OpenAI, 2024) as the default model unless otherwise stated. The LangGraph framework (LangChain, 2024) is used as the agent platform, providing essential features for LLM input-output parsing and process control. To implement the memory mechanism, we integrated Neo4j (Neo4j, 2024) for graph-based storage and retrieval and Pinecone (Pinecone, 2024) for vector search. For feature matching, we employed cosine similarity with embeddings derived from ResNet-50 (He et al., 2015), which enables effective task representation and retrieval. All experiments were conducted using the Android Debug Bridge (ADB), enabling on-device evaluations for mobile applications.

5.2 Experimental Analysis

In this part, the experiments are designed to validate AppAgentX's advantages in terms of efficiency and accuracy. We conducted five experiments for the baseline comparison and two experiments for the large dataset to mitigate the effects of randomness in LLMs.

Comparative Analysis. A comparison of our experimental results with model variants is shown in Table 1. We report the results on the AppAgent benchmark. We begin with a baseline model that does not incorporate any memory mechanism and progressively introduce our proposed enhancements. The first model variant integrates a module that records only information about elements, providing a basic form of memory. Subsequently, we introduce a more advanced memory design that maintains a structured memory chain, enabling the model to retain a more comprehensive representation of past interactions. Finally, we augment the system with an evolutionary mechanism that expands the action space to include high-level actions, further optimizing task execution.

As the result shows, incorporating the memory mechanism significantly improves the task success rate. The baseline model, which lacks memory, achieves an SR of only 16.9%. Introducing the chain-structured memory significantly enhances the success rate, reaching 70.8%, highlighting the clear advantage of maintaining a more structured history of interactions. Furthermore, it proves to be more effective than element memory in facilitating task completion. Moreover, the integration of the evolutionary mechanism leads to substantial efficiency gains. Expanding the action space to include high-level actions decreases the average required number of steps from 9.1 to 5.7, while the step execution time is reduced from 23 to 16 seconds. Additionally, average token consumption is significantly minimized, dropping from 9.26k to 4.94k, indicating a more efficient decision-making
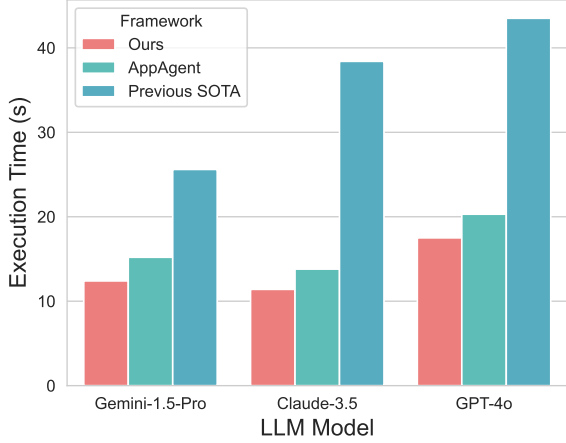
Figure 4: **Comparison of Average Execution Time per Step.** This figure presents the average execution time per steps across different LLMs and frameworks.
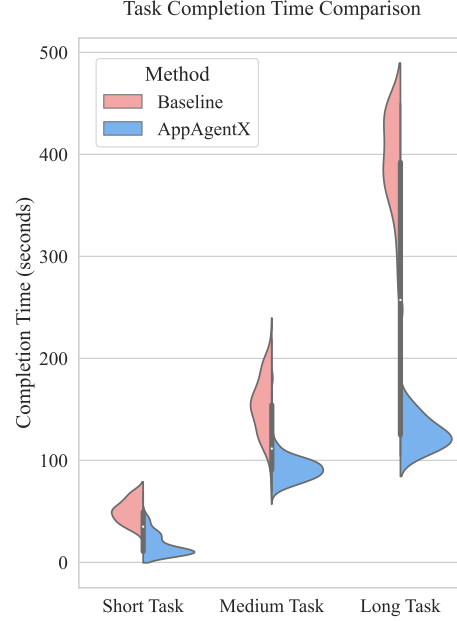

Figure 5: **Task Completion Times Across Different Task Lengths.** This figure shows the distribution of task completion times for short, medium, and long tasks. Each violin plot represents the density of completion times, with wider sections indicating higher data concentration. AppAgentX consistently outperforms the baseline, particularly for longer tasks.

process. Notably, this enhancement reduces computational overhead and improves the average success rate to 71.4%. These results show that our approach is effective. The memory mechanisms significantly improve task performance, while the evolutionary strategy boosts efficiency by reducing steps, execution time, and token use.

To further highlight the advantages of the AppAgentX framework, we compare its performance to other state-of-the-art frameworks on additional benchmarks, including DroidTask (Wen et al., 2024), Android Agent Arena(Chai et al., 2025) and AndroidWorld(Rawles et al., 2025). As shown in Figure 2, AppAgentX outperforms AppAgent, in both task execution time and accuracy. Specifically, AppAgentX achieves significantly higher efficiency while maintaining higher task success rates across a broader range of applications.

Additionally, we compare the execution efficiency of AppAgentX against two other frameworks (Wang et al., 2024a; Zhang et al., 2023) across several prominent foundational LLMs, including GPT-4o (OpenAI, 2024), Claude 3.5 Sonnet (Anthropic., 2024), and Gemini 1.5 Pro (Gemini, 2024). While acknowledging that certain discrepancies in the experimental setup may exist, we focus our evaluation primarily on execution time, as it serves as a practical and reproducible metric for assessing efficiency. As shown in Table 4, AppAgentX consistently demonstrates faster per-step completion times compared to the previous state-of-the-art (Wang et al., 2024a) and AppAgent, across multiple LLM backends. These results suggest improved efficiency and better alignment with

real-world deployment constraints.

Task difficulties analysis. To investigate the performance of AppAgentX across tasks of varying complexity, we categorize the tasks based on their ground-truth operation steps, annotated through expert demonstrations. Tasks with up to 5 steps are considered short, tasks with 6 to 10 steps are classified as medium-length, and tasks requiring more than 10 steps are categorized as long tasks. As shown in Figure 5, the violin plot provides an intuitive visualization of how task complexity affects performance. Notably, our method demonstrates a clear advantage in terms of task execution time as the task complexity increases. To account for random fluctuations, we conducted one-tailed paired t-tests (Ross and Willson, 2017) on each data group to verify that AppAgentX significantly outperforms the baseline in efficiency, with all p-values falling below 0.05. Our model consistently outperforms the baseline across all task lengths, resulting in reduced task completion times. This finding reinforces the robustness and efficiency of our approach, confirming its stability under varying task difficulties.

Ablation Study. To further evaluate the performance gains of our agent, we substitute different screen perceptrons to assess the effectiveness of

Table 2: **Comparison with AppAgent on Large Benchmarks.** This table evaluates the efficiency and accuracy of different frameworks on benchmarks containing a large number of tasks.

Benchmarks	Task Num.	Framework	Task Time↓	Tokens (k)↓	SR↑
DroidTask(Wen et al., 2024)	158	AppAgent	106.24	11.5	46.3%
		AppAgentX	**56.29**	**5.1**	**88.2%**
AndroidWorld(Rawles et al., 2025)	116	AppAgent	147.17	18.9	41.7%
		AppAgentX	**59.74**	**6.2**	**62.5%**
A3 (Android Agent Arena)(Chai et al., 2025)	201	AppAgent	134.67	19.2	10.3%
		AppAgentX	**48.12**	**4.7**	**39.3%**

Figure 6: **Qualitative Results.** This figure presents the path execution utilizing shortcuts on Gmail. It demonstrates the efficiency of AppAgentX in decreasing the utilization of LLM for per-step reasoning.

Table 3: **Ablation Study of Different Screen Perceptrons and Action Spaces.** This table compares task success rates and times for different screen parsers and two action space variants: **BAS** (Basic Action Space) and **FAS** (Full Action Space).

Methods	Task Time↓	SR↑
Omniparser + BAS	209.3s	70.8%
Omniparser + FAS	91.2s	71.4%
Ferret-UI + BAS	201.7s	68.1%
Ferret-UI + FAS	92.5s	70.6%

our evolutionary approach. Specifically, Omni-Parser(Lu et al., 2024) is employed as the default visual element extractor to locate UI components on the screen. In this study, we replace it with Ferret-UI(You et al., 2024b), a component with equivalent functionality, and compare the baseline action space with the evolved action space to examine their respective impacts.

As shown in Table 3, the Full Action Space significantly improves task performance compared to the Basic Action Space, significantly reducing execution time while simultaneously improving success rates. These results validate the effectiveness

of the evolved action space.

Qualitative Analysis. In Figures 6, we show a qualitative analysis of some of the actions that used high-level actions to operate the screen. On the left side we label the various processes in the relevant execution operations, from the retrieval of tasks to the matching of shortcut nodes. Despite the different user interfaces and actions performed by these applications, AppAgentX successfully completed the given task.

6 Conclusion

We propose an evolving GUI agent framework that enhances efficiency and intelligence by abstracting high-level actions from execution history. Unlike rule-based automation, our approach generalizes task execution by compressing repetitive operations, balancing intelligent reasoning with efficient execution. A chain-based knowledge framework enables continuous behavior refinement, improving adaptability and reducing redundancy. Experiments show our framework outperforms existing methods in accuracy and efficiency.

References

Anthropic. 2024. Claude 3.5 sonnet.

Hao Bai, Yifei Zhou, Mert Cemri, Jiayi Pan, Alane Suhr, Sergey Levine, and Aviral Kumar. 2024. Digirl: Training in-the-wild device-control agents with autonomous reinforcement learning. *Preprint*, arXiv:2406.11896.

Yuxiang Chai, Hanhao Li, Jiayu Zhang, Liang Liu, Guangyi Liu, Guozhi Wang, Shuai Ren, Siyuan Huang, and Hongsheng Li. 2025. A3: Android agent arena for mobile gui agents. *Preprint*, arXiv:2501.01149.

Guangyao Chen, Siwei Dong, Yu Shu, Ge Zhang, Jaward Sesay, Börje F. Karlsson, Jie Fu, and Yemin Shi. 2024. Autoagents: A framework for automatic agent generation. *Preprint*, arXiv:2309.17288.

Xiangxiang Chu, Limeng Qiao, Xinyu Zhang, Shuang Xu, Fei Wei, Yang Yang, Xiaofei Sun, Yiming Hu, Xinyang Lin, Bo Zhang, and Chunhua Shen. 2024. Mobilevlm v2: Faster and stronger baseline for vision language model. *Preprint*, arXiv:2402.03766.

DeepSeek-AI. 2024. Deepseek-v3 technical report. *Preprint*, arXiv:2412.19437.

Hiroki Furuta, Kuang-Huei Lee, Ofir Nachum, Yutaka Matsuo, Aleksandra Faust, Shixiang Shane Gu, and Izzeddin Gur. 2024. Multimodal web navigation with instruction-finetuned foundation models. *Preprint*, arXiv:2305.11854.

Gemini. 2024. Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context. *Preprint*, arXiv:2403.05530.

Hongliang He, Wenlin Yao, Kaixin Ma, Wenhao Yu, Yong Dai, Hongming Zhang, Zhenzhong Lan, and Dong Yu. 2024. Webvoyager: Building an end-to-end web agent with large multimodal models. *Preprint*, arXiv:2401.13919.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2015. Deep residual learning for image recognition. *Preprint*, arXiv:1512.03385.

Theodore D. Hellmann and Frank Maurer. 2011. Rule-Based Exploratory Testing of Graphical User Interfaces . In *2011 Agile Conference*, pages 107–116, Los Alamitos, CA, USA. IEEE Computer Society.

Sirui Hong, Mingchen Zhuge, Jiaqi Chen, Xiawu Zheng, Yuheng Cheng, Ceyao Zhang, Jinlin Wang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, Chenyu Ran, Lingfeng Xiao, Chenglin Wu, and Jürgen Schmidhuber. 2024. Metagpt: Meta programming for a multi-agent collaborative framework. *Preprint*, arXiv:2308.00352.

Wenyi Hong, Weihan Wang, Qingsong Lv, Jiazheng Xu, Wenmeng Yu, Junhui Ji, Yan Wang, Zihan Wang, Yuxiao Dong, Ming Ding, and Jie Tang. 2023. Cogagent: A visual language model for gui agents. *Preprint*, arXiv:2312.08914.

Jiaxin Huang, Shixiang Shane Gu, Le Hou, Yuexin Wu, Xuezhi Wang, Hongkun Yu, and Jiawei Han. 2022. Large language models can self-improve. *Preprint*, arXiv:2210.11610.

LangChain. 2024. Langgraph: A framework for agent-based workflows. Accessed: 2024-06-01.

Junkai Li, Yunghwei Lai, Weitao Li, Jingyi Ren, Meng Zhang, Xinhui Kang, Siyu Wang, Peng Li, Ya-Qin Zhang, Weizhi Ma, and Yang Liu. 2025. Agent hospital: A simulacrum of hospital with evolvable medical agents. *Preprint*, arXiv:2405.02957.

Yanda Li, Chi Zhang, Wanqi Yang, Bin Fu, Pei Cheng, Xin Chen, Ling Chen, and Yunchao Wei. 2024. Appagent v2: Advanced agent for flexible mobile interactions. *Preprint*, arXiv:2408.11824.

Xiao Liu, Bo Qin, Dongzhu Liang, Guang Dong, Hanyu Lai, Hanchen Zhang, Hanlin Zhao, Iat Long Iong, Jiadai Sun, Jiaqi Wang, Junjie Gao, Junjun Shan, Kangning Liu, Shudan Zhang, Shuntian Yao, Siyi Cheng, Wentao Yao, Wenyi Zhao, Xinghan Liu, Xinyi Liu, Xinying Chen, Xinyue Yang, Yang Yang, Yifan Xu, Yu Yang, Yujia Wang, Yulin Xu, Zehan Qi, Yuxiao Dong, and Jie Tang. 2024. Autoglm: Autonomous foundation agents for guis. *Preprint*, arXiv:2411.00820.

Yadong Lu, Jianwei Yang, Yelong Shen, and Ahmed Awadallah. 2024. Omniparser for pure vision based gui agent. *Preprint*, arXiv:2408.00203.

Jiageng Mao, Yuxi Qian, Junjie Ye, Hang Zhao, and Yue Wang. 2023. Gpt-driver: Learning to drive with gpt. *Preprint*, arXiv:2310.01415.

Microsoft. 2024. Resnet-50 pre-trained model.

Humza Naveed, Asad Ullah Khan, Shi Qiu, Muhammad Saqib, Saeed Anwar, Muhammad Usman, Naveed Akhtar, Nick Barnes, and Ajmal Mian. 2024. A comprehensive overview of large language models. *Preprint*, arXiv:2307.06435.

Neo4j. 2024. Neo4j: The graph database platform.

OpenAI. 2024. Gpt-4 technical report. *Preprint*, arXiv:2303.08774.

Joon Sung Park, Joseph C. O'Brien, Carrie J. Cai, Meredith Ringel Morris, Percy Liang, and Michael S. Bernstein. 2023. Generative agents: Interactive simulacra of human behavior. *Preprint*, arXiv:2304.03442.

Pawel Pawlowski, Krystian Zawistowski, Wojciech Lapacz, Marcin Skorupa, Adam Wiacek, Sebastien Postansque, and Jakub Hoscilowicz. 2024. Tinyclick: Single-turn agent for empowering gui automation. *Preprint*, arXiv:2410.11871.

Pinecone. 2024. Pinecone: Scalable vector search for ai.

Yiran Qin, Enshen Zhou, Qichang Liu, Zhenfei Yin, Lu Sheng, Ruimao Zhang, Yu Qiao, and Jing Shao. 2024. Mp5: A multi-modal open-ended embodied system in minecraft via active perception. *Preprint*, arXiv:2312.07472.

Christopher Rawles, Sarah Clinckemaillie, Yifan Chang, Jonathan Waltz, Gabrielle Lau, Marybeth Fair, Alice Li, William Bishop, Wei Li, Folawiyo Campbell-Ajala, Daniel Toyama, Robert Berry, Divya Tyamagundlu, Timothy Lillicrap, and Oriana Riva. 2025. Androidworld: A dynamic benchmarking environment for autonomous agents. *Preprint*, arXiv:2405.14573.

Amanda Ross and Victor L. Willson. 2017. *Paired Samples T-Test*, pages 17–19. SensePublishers, Rotterdam.

Zirui Song, Yaohang Li, Meng Fang, Yanda Li, Zhenhao Chen, Zecheng Shi, Yuan Huang, Xiuying Chen, and Ling Chen. 2025. Mmac-copilot: Multi-modal agent collaboration operating copilot. *Preprint*, arXiv:2404.18074.

Sharon Tentarelli, Rómulo Romero, and Michelle L. Lamb. 2022. Script-based automation of analytical instrument software tasks. *SLAS Technology*, 27(3):209–213.

Junyang Wang, Haiyang Xu, Haitao Jia, Xi Zhang, Ming Yan, Weizhou Shen, Ji Zhang, Fei Huang, and Jitao Sang. 2024a. Mobile-agent-v2: Mobile device operation assistant with effective navigation via multi-agent collaboration. *Preprint*, arXiv:2406.01014.

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

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. 2023. Chain-of-thought prompting elicits reasoning in large language models. *Preprint*, arXiv:2201.11903.

Hao Wen, Yuanchun Li, Guohong Liu, Shanhui Zhao, Tao Yu, Toby Jia-Jun Li, Shiqi Jiang, Yunhao Liu, Yaqin Zhang, and Yunxin Liu. 2024. Autodroid: Llm-powered task automation in android. *Preprint*, arXiv:2308.15272.

Yue Wu, Xuan Tang, Tom M. Mitchell, and Yuanzhi Li. 2024. Smartplay: A benchmark for llms as intelligent agents. *Preprint*, arXiv:2310.01557.

Hui Yang, Sifu Yue, and Yunzhong He. 2023. Auto-gpt for online decision making: Benchmarks and additional opinions. *Preprint*, arXiv:2306.02224.

Keen You, Haotian Zhang, Eldon Schoop, Floris Weers, Amanda Swearngin, Jeffrey Nichols, Yinfei Yang, and Zhe Gan. 2024a. Ferret-ui: Grounded mobile ui understanding with multimodal llms. *Preprint*, arXiv:2404.05719.

Keen You, Haotian Zhang, Eldon Schoop, Floris Weers, Amanda Swearngin, Jeffrey Nichols, Yinfei Yang, and Zhe Gan. 2024b. Ferret-ui: Grounded mobile ui understanding with multimodal llms. *Preprint*, arXiv:2404.05719.

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. 2024a. Ufo: A ui-focused agent for windows os interaction. *Preprint*, arXiv:2402.07939.

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

Jiayi Zhang, Jinyu Xiang, Zhaoyang Yu, Fengwei Teng, Xionghui Chen, Jiaqi Chen, Mingchen Zhuge, Xin Cheng, Sirui Hong, Jinlin Wang, Bingnan Zheng, Bang Liu, Yuyu Luo, and Chenglin Wu. 2024b. Aflow: Automating agentic workflow generation. *Preprint*, arXiv:2410.10762.

A Design and Execution Details

This section presents the detailed design of the proposed chain structure and the overall task execution workflow in our system.

A.1 Chain Design Details

In chain design, there are three types of relationships between nodes:

Page-HAS_ELEMENT→Element

Type: HAS_ELEMENT

Direction: (Page) -[:HAS_ELEMENT]-> (Element)

Purpose: Indicates that a page contains a specific UI element.

Shortcut-COMPOSED_OF→Element

Type: COMPOSED_OF

Direction: (Shortcut) -[:COMPOSED_OF]-> (Element)

Relationship Attributes:

- **order**: Integer, representing the execution sequence.

- **atomic_action**: Type of basic action (e.g., click, text input, etc.).

- **action_params**: JSON format, potentially including input text, click parameters, etc.

Purpose: Defines how a composite action consists of certain elements and their corresponding steps, which must be executed in a specific order.

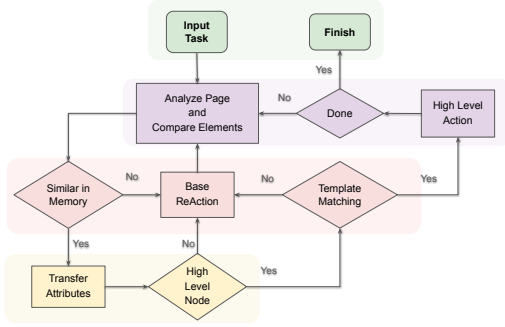**Element-LEADS_TO→Page**

Type: LEADS_TO

Figure 7: **Overall Execution Process.** This figure illustrates the flowchart from the input of the task to the final execution result after we then add the shortcut route.

Direction: (Element) -[:LEADS_TO]-> (Page)
Purpose: Represents the linkage of a composite action.

Together, these types of edges, along with the nodes described in the main text, constitute the structure of the complete memory store. This modeling approach aligns with the relationships between page jumps and containment, and offers a solid foundation for analyzing the evolution of actions.

A.2 Execution Process Details

Figure 7 presents a detailed flowchart that delineates the task processing workflow. The primary components of this workflow include page analysis, element comparison, matching, and memory storage. As depicted in Figure 7, the system utilizes a hierarchical action framework comprising high-level nodes and basic action spaces. During the operation, if the matching retrieval process of a high-level node fails, the system seamlessly transitions to the corresponding basic action spaces.

B Detailed Numerical Results

This section provides the numerical analysis results of various shortcut designs, including evaluations of robustness and statistical tests to validate the effectiveness of the proposed system design.

To supplement the grouped bar chart presented in Figure 4, this subsection provides the exact per-step execution time (in seconds) for each evaluated framework (MobileAgent2, AppAgent, and AppAgentX) across different LLM backends. These numerical results are intended to enhance reproducibility, facilitate further analysis, and offer a precise basis for comparing execution efficiency across systems.

Table 4: **Comparison of Average Execution Time per Step.** This table presents the average execution time (seconds ↓) across different LLMs and frameworks.

LLM	Previous SOTA	AppAgent	Ours
Gemini-1.5-Pro	25.6	15.2	**12.4**
Claude-3.5	38.4	13.8	**11.4**
GPT-4o	43.5	20.3	**17.5**

Table 4 lists the detailed values corresponding to each bar in the figure.

Table 5: **Statistical Test Results for Task Completion Times.** Results from one-tailed paired t-tests.

Task Length	Mean Difference (s)	t-value	p-value
Short Task	-12.4	-3.45	<0.01
Medium Task	-30.7	-5.12	<0.001
Long Task	-75.3	-6.89	<0.001

To reinforce the observed differences in task completion times, we complemented the analysis with detailed results from one-tailed t-tests, as summarized in Table 5. These results address data not fully elaborated upon in the main text. The negative t-values reported in this table indicate the presence of a directional effect, where AppAgentX consistently outperforms the baseline in reducing task completion times. Specifically, the negative sign reflects that the mean completion time for AppAgentX is significantly lower than that of the baseline across all task lengths. This directional result aligns with our hypothesis and further validates the consistent efficiency advantage of AppAgentX.

C Additional Quantitative Analysis

To further illustrate the behavior of our system in real-world settings, we include a qualitative example in Figure 8. This example showcases the execution path for a task in Apple Music, highlighting how AppAgentX leverages existing shortcuts to complete the task more efficiently. By using predefined shortcuts, the system reduces the reliance on large language model (LLM) reasoning at each step, thereby improving both speed and resource usage.
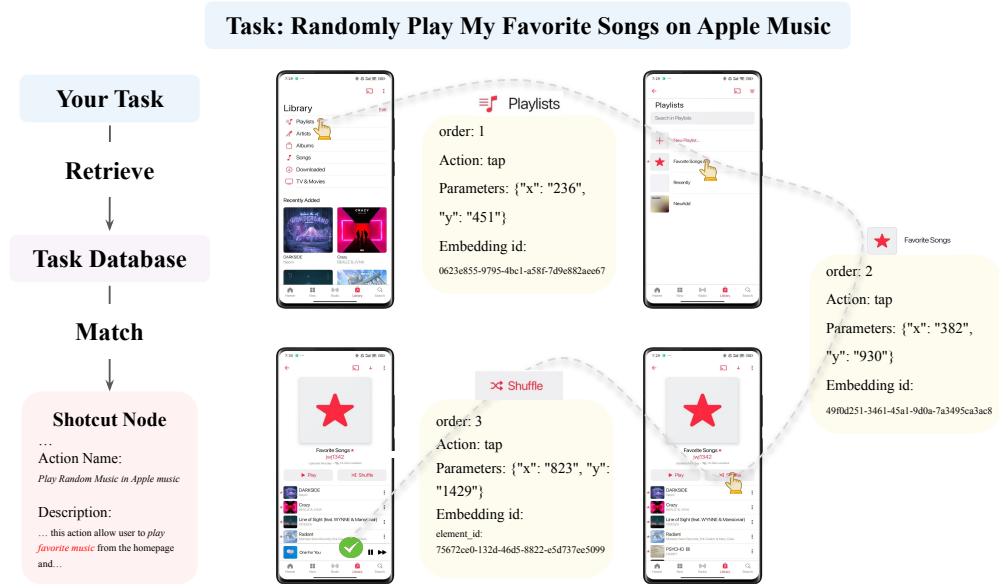
Figure 8: **Qualitative Results.** This figure presents the path execution utilizing shortcuts on Apple Music. It demonstrates the efficiency of AppAgentX in decreasing the utilization of LLM for per-step reasoning.