MobileGPT: Augmenting LLM with Human-like App Memory for Mobile Task Automation

Sunjae Lee[1,*], Junyoung Choi[1], Jungjae Lee[1], Munim Hasan Wasi[1], Hojun Choi[1], Steven Y. Ko[2], Sangeun Oh[3,†], Insik Shin[1,4,*,†]
[1]KAIST, S. Korea [2]Simon Fraser University, Canada [3]Korea University, S. Korea [4]Fluiz, S. Korea
*{sunjae1294,ishin}@kaist.ac.kr

Abstract

The advent of large language models (LLMs) has opened up new opportunities in the field of mobile task automation. Their superior language understanding and reasoning capabilities allow users to automate complex and repetitive tasks. However, due to the inherent unreliability and high operational cost of LLMs, their practical applicability is quite limited. To address these issues, this paper introduces MobileGPT[1], an innovative LLM-based mobile task automator equipped with a human-like app memory. MobileGPT emulates the cognitive process of humans interacting with a mobile app—explore, select, derive, and recall. This approach allows for a more precise and efficient learning of a task's procedure by breaking it down into smaller, modular sub-tasks that can be re-used, re-arranged, and adapted for various objectives. We implement MobileGPT using online LLMs services (GPT-3.5 and GPT-4) and evaluate its performance on a dataset of 185 tasks across 18 mobile apps. The results indicate that MobileGPT can automate and learn *new* tasks with 82.7% accuracy, and is able to adapt them to different contexts with near perfect (98.75%) accuracy while reducing both latency and cost by 62.5% and 68.8%, respectively, compared to the GPT-4 powered baseline.

CCS Concepts

• **Computing methodologies** → *Artificial intelligence*; • **Human-centered computing** → **Ubiquitous and mobile computing**.

[†]Co-corresponding authors: Sangeun Oh, Insik Shin.
[1]The system is available at: https://mobile-gpt.github.io/

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivs International 4.0 License.
ACM MobiCom '24, November 18–22, 2024, Washington D.C., DC, USA
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0489-5/24/11
https://doi.org/10.1145/3636534.3690682

Keywords

AI Agent; Task Automation, Large Language Models

ACM Reference Format:
Sunjae Lee[1,*], Junyoung Choi[1], Jungjae Lee[1], Munim Hasan Wasi[1], Hojun Choi[1], Steven Y. Ko[2], Sangeun Oh[3,†], Insik Shin[1,4,*,†]. 2024. MobileGPT: Augmenting LLM with Human-like App Memory for Mobile Task Automation. In *The 30th Annual International Conference on Mobile Computing and Networking (ACM MobiCom '24), November 18–22, 2024, Washington D.C., DC, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3636534.3690682

1 Introduction

Today, we rely on a whole universe of mobile apps to handle tasks integral to our daily lives, ranging from communication to home security. Consequently, the demand for efficient task automation in the mobile environment has intensified, seeking to alleviate digital fatigue stemming from repetitive and complex digital interactions that often overwhelm many users today [8, 14].

Unsurprisingly, numerous methods have been explored to achieve task automation. API-based approaches like Siri [2] and Google Assistant [11] allow users to interact with a mobile app's specific functionality through natural language conversation. However, these approaches demand significant coding efforts from developers, who must manually create new logic for each task. Demonstration-based methods [22, 25, 26, 28] empower end-users to program custom automation scripts. Yet, due to their heavy reliance on human demonstrations, they face scalability challenges. Similarly, learning-based approaches [17, 24, 29, 52, 53] require extensive collections of human-annotated datasets, hampering the widespread application.

Recently, LLM-based task automators [39, 44, 49, 59, 62], equipped with high reasoning and generalization abilities [1, 35, 41], have been a game-changer. They enable task automation to be fully autonomous and generally applicable, sidestepping the labor-intensive manual development, demonstration, and training that previous methods required. However, this approach also comes with its own limitations. First, the inherent non-deterministic and unpredictable nature of LLMs can undermine the reliability and consistency of task

automation. This is critical in mobile environments as many mobile tasks nowadays involve sensitive and private information. Second, LLMs are costly, both in terms of budget and time. We observed that tasks that would take just over 30 seconds for a human could take more than two minutes for an LLM, and cost over a dollar each time they are performed.

To overcome the limitations of previous approaches, we introduce MobileGPT, an innovative LLM-based mobile task automator augmented with an app memory capable of learning and recalling mobile tasks. MobileGPT is designed to accomplish following design goals: *i) Accurate and Consistent*: it should perform tasks with high accuracy and consistency, ensuring that once a task is learned, it can be faithfully reproduced. *ii) Adaptability*: When revisiting a task, it should dynamically adjust its execution in response to varying contexts, rather than blindly replicating the previous executions. *iii) Efficiency*: MobileGPT seeks to significantly reduce the cost and time involved in task automation, especially for tasks that are performed repeatedly.

In designing MobileGPT, we drew inspiration from how humans decompose complex tasks into smaller sub-tasks to effectively learn and recall tasks [6, 7, 19, 33]. Specifically, consider how humans learn new tasks using mobile apps: given an app screen, we first **1) explore** the candidate sub-tasks by analyzing screen interfaces and identifying their functionalities. Then, we **2) select** the most promising sub-task that can bring us closer to the goal. Lastly, we **3) derive** and execute the primitive actions required to complete the chosen sub-task—low-level actions such as click, input, and scroll. Once we have completed the task by repeating these 3 steps, it becomes part of our memory, allowing for easy **4) recall** and repetition of not only the task itself but also its involved *sub-tasks.* For example, assume you have learned how to *"Send a message to Bob."* Having learned this, not only can you easily adapt this knowledge to similar instructions like "Send a message to Alice," but also can apply it to execute new tasks such as "Read messages from John." This inherent human ability stems from our tendency to break down tasks into smaller *sub-tasks* and encapsulate them in a modular, reusable format.

This human capacity to learn and recall memories at the unit of sub-tasks is what we aim to replicate with MobileGPT. To achieve this, we address three key challenges: *i) Accurate and reliable task execution*: To ensure accurate task execution and memory construction, we employ several prompting techniques to improve LLMs' accuracy and provide mechanisms for users to correct errors in case LLMs make mistakes. *ii) Efficient memory storage:* To facilitate the use of memory, MobileGPT efficiently parameterizes and stores task information in a hierarchical memory structure, in which tasks are decomposed into multiple function-call formatted sub-tasks and each sub-task is further decomposed into a sequence of primitive actions. *iii) Flexible memory retrieval:* To ensure robust and cost-effective task automation in the dynamic landscape of mobile interfaces, MobileGPT leverages pattern matching and in-context learning techniques to flexibly adapt actions involved in sub-tasks to varying contexts and interface changes.

We have implemented a prototype of MobileGPT using online LLM services (GPT-3.5 and GPT-4). Our comparison study with state-of-the-art mobile task automators AutoDroid [59] and AppAgent [62] demonstrates that MobileGPT achieves a task completion rate of 82.7% when executing a new task, outperforming these systems by 8% and 15.3% respectively. Furthermore, MobileGPT achieves a near-perfect (98.75%) success rate in adapting learned tasks to a new instruction with different task parameters. Our ablation study suggests that MobileGPT achieves a 62.5% reduction in task completion time and a 68.8% decrease in LLM query costs when recalling learned tasks. A usability study with 23 participants demonstrates that MobileGPT's human-in-the-loop task repair mechanism enables users to interact intuitively with the task automator, allowing them to repair and collaboratively build upon the task automation process.

2 System Overview

2.1 System Workflow

Akin to the human cognitive learning process, MobileGPT operates on a cycle of *Explore-Select-Derive* phases to execute and learn new mobile tasks. This section gives a high-level overview of the system's workflow (Figure 1).

Explore. Prior to performing any tasks, MobileGPT undertakes multiple *explore* operations offline to pre-emptively collect and organize the functionalities of the app. To achieve this, MobileGPT employs two software tools—random explorer [53] and user trace monitor [10]—to visit and analyze as many app screens as it can during the offline stage. For each screen it visits, MobileGPT asks the LLM to generate a list of sub-tasks available on the screen. This generated list is cached into memory to be used during the select phase upon receiving user instructions. App screens that have not been explored offline are analyzed on demand during the task execution.

Select. When a user issues a voice instruction, MobileGPT retrieves the list of sub-tasks associated with the current screen from the memory. Then, it asks LLM which sub-task to perform in order to complete the user instruction. The selected sub-task is carried on to the Derive phase to be translated into a sequence of low-level actions.

Derive. In the Derive phase, MobileGPT prompts the LLM with a pre-defined set of low-level action (e.g., click, input, or scroll), and asks it to pick an action needed to accomplish the chosen sub-task. The action is then dispatched to the

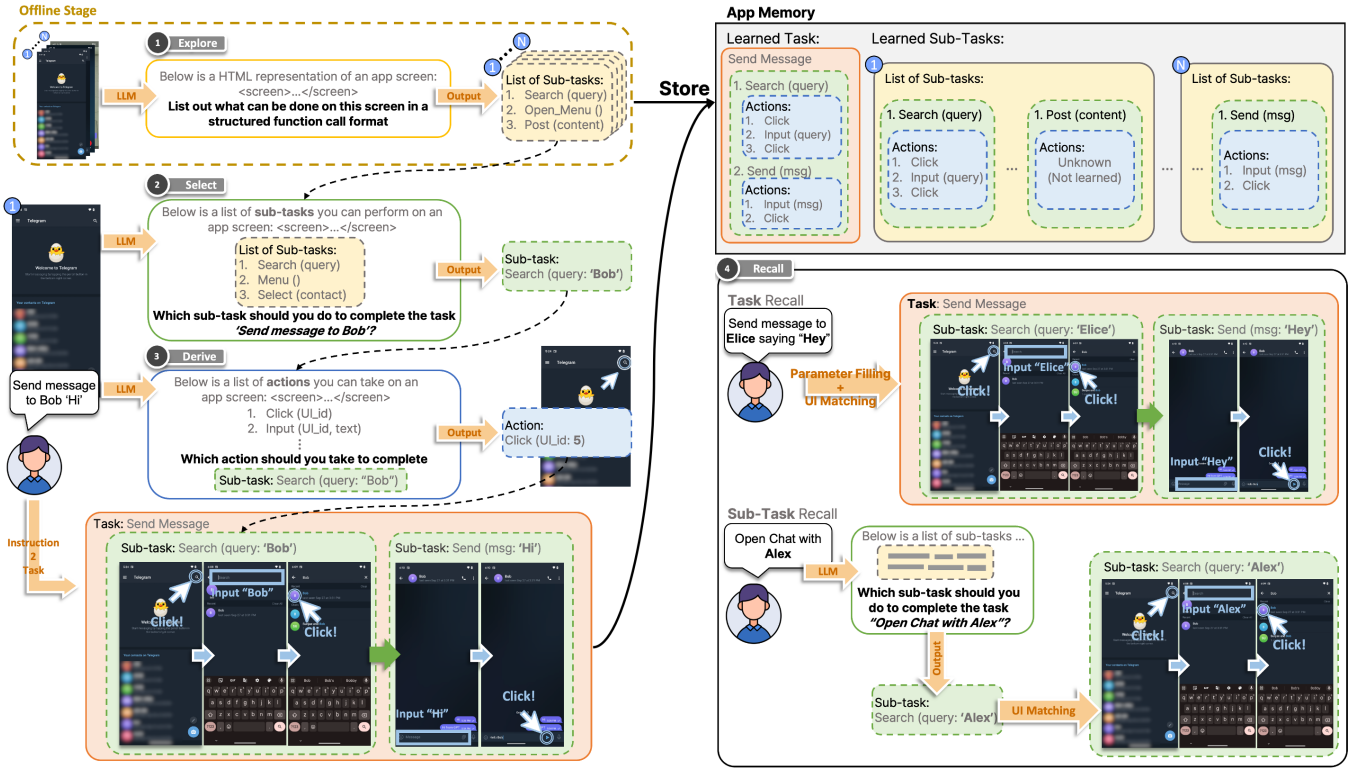


Figure 1: MobileGPT workflow and system overview

mobile device to be executed within the app. This phase is repeated until the app navigates to the next page, or the LLM explicitly indicates that the sub-task is completed.

Upon finishing a sub-task, the system returns to the Select phase and selects the next sub-task to execute. This iterative process is continued until the Select phase indicates that the user's instruction has been fully executed. Then, the instruction is translated into a high-level task (e.g., *"send a message to Bob" to "send message")* and saved in the memory in a hierarchical format, as illustrated in Figure 1.

Recall. When the system is given an instruction while its memory is not empty, it checks if the instruction matches any previously learned task by comparing the instruction's high-level task representation with those stored in its memory. If a match is found, it executes the instruction directly from the memory, If no match is found, MobileGPT cycles through Explore, Select, Derive to learn the new task. However, if the new task involves any known *sub-tasks*, MobileGPT can still reproduce them, akin to how humans transfer knowledge from one task to another. This facilitates faster execution and learning of new tasks.

2.2 System Design

To enable the workflow above, MobileGPT addresses the following challenges:

C1. How to *accurately and reliably execute* a task in the first try?
C2. How to *efficiently store* task executions?
C3. How to *flexibly recall* past task execution?

C1. The first step towards learning a task is to correctly execute it the first time. MobileGPT executes unknown tasks by leveraging multiple LLM queries to iterate through phases of Explore, Select, and Derive. However, given the complexity of mobile tasks and the unpredictability of LLM, we cannot guarantee the complete accuracy of these executions. For instance, when instructed to *"find 5-star hotels,"* LLMs may simply type "5-star hotel" in the search field, whereas the expected behavior is to click the '5-star' filter option. Therefore, to address this inherent non-determinism and the unreliability of LLMs, MobileGPT employs a dual-strategy correction mechanism that allows both the LLM and the user to fix errors in the execution process (more details in § 3).

C2. In many cases, tasks are not entirely independent of each other; they often share common sub-tasks. For instance, both 'Send a message to Bob' and 'Open chat with Alex' involve the sub-task of locating a contact. If the execution procedures of the two tasks are stored separately at the task level, the sub-task they perform in common should be trained redundantly. To minimize such inefficiency, MobileGPT employs a three-level hierarchical memory structure:

tasks, sub-tasks, and actions. This hierarchy enables MobileGPT to access memory at the sub-task level, facilitating the sharing of past execution experience across different tasks.

Moreover, each task often comes with different variations. The task of sending a message can be specified as "send message to Bob" or "send Elice a message 'Hey'." To address this range of variations, we need to properly parameterize the learned task. MobileGPT achieves this by representing each subtask in a function-call format and the task as a series of these function calls (see Figure 1). This approach allows for more granular parameterization, as MobileGPT identifies parameters incrementally while proceeding with the task, including implicit parameters that are hidden in the instruction, such as the message content parameter in the instruction "send message to Bob."

C3. Even when repeating the same task or sub-task, the specifics of each step may vary depending on the context of execution. For example, searching for a contact in the context of "Send a message to Bob" involves entering and clicking "Bob" in the search page, whereas "Send a message to John" requires searching for "John" instead. Therefore, instead of blindly replicating past actions, MobileGPT flexibly adapts its past executions to accommodate not only the intricate parameters of the task but also changes in screen content. In addition, in case direct adaptation falls short, MobileGPT leverages the few-shot learning capability of LLMs to guide them in generating responses that are both consistent and deterministic throughout multiple trials of the task (more details in § 5).

3 Accurate and Reliable Task Execution

LLMs have been reported to solve complex tasks more accurately by breaking them down into smaller sub-tasks [56, 58]. MobileGPT adopts this strategy by hierarchically decomposing tasks into sub-tasks according to the Explore-Select-Derive phases. This section outlines how MobileGPT effectively navigates each of these phases, while also addressing the challenges posed by the inherent unreliability of LLMs. For clarity, this section describes each phase under the assumption that MobileGPT's memory is empty.

3.1 Prompting Mobile Screen to LLM

The initial step in applying LLMs for mobile tasks involves converting mobile screens into text representation. Drawing upon previous research [54] showing that LLMs comprehend Graphical User Interfaces (GUIs) better when presented in HTML syntax, we convert mobile screens into a simplified HTML representation.

We begin by extracting the screen's layout information using Android Accessibility Service [10]. This information includes a hierarchical relationship between UI elements and various attributes (e.g., class_name, text, descriptions) and properties (e.g., clickable, editable) that describe the functionality and appearance of the UI. To make the layout file succinct, we prune UI elements that are neither interactive nor carry significant semantic attributes (e.g., empty layout container). Then, we map each remaining UI element into an HTML element, where the UI's text and description attributes serve as the content of the HTML element, and the UI's interactive property is translated into the corresponding HTML tag. For instance, the '<button>' tag is used for clickable UIs, '<input>' for editable UIs, '<scroll>' for scrollable UIs, and so on. Lastly, we assign each HTML element with a unique index number, which serves as a communication link between MobileGPT and LLMs to specifically refer to and interact with each element on the screen. This process not only ensures a cleaner, more relevant representation of app screens but also significantly reduces the number of tokens by an average of 84.6%. Throughout this paper, we will refer to this HTML representation of an app screen as the *"screen representation."*

3.2 Explore, Select, and Derive

We generate the screen representation every time there is a change in the screen. This screen representation is then used throughout the process of the Explore, Select, and Derive phases.

Explore. The goal of the *explore* phase is to generate a list of actionable sub-tasks for a given screen. Each sub-task represents an individual operation or functionality that the screen provides. To accomplish this, MobileGPT prompts the LLM with the screen representation and asks it to enumerate sub-tasks in a structured function call format with the following information included: 1) sub-task name and description, 2) parameter names and descriptions, and 3) index of relevant UI elements. For example, possible sub-tasks for Telegram's initial app screen (Figure 1) include:

```
1. { name: "Search", desc: "Search for a contact", params: {"query": "who
     are you looking for?" }, UI_index: 3 }
2. { name: "Open_Menu", desc: "Open menu", params: {}, UI_index: 7 }
```

Note that, unlike other phases, the explore phase operates independently of user instructions. Hence, we conduct this phase *offline*, before performing the user's designated task. We use tools like random explorer [53], which performs random UI actions to navigate through app screens, and user trace monitor [10], which tracks users' app usage in the background. These tools allow for the proactive collection of sub-tasks related to each app screen. While the current implementation of MobileGPT conducts this offline phase locally on the user's device, it could be offloaded to a server to enhance efficiency.

Select. During the *select* phase, MobileGPT prompts the LLM with *i)* the user instruction, *ii)* the current screen representation, and *iii)* a list of available sub-tasks. The list

includes both the sub-tasks identified during the *explore* phase and a predefined set of global sub-tasks common to all screens such as 'Read_Screen' for answering the user's question based on the current screen content and 'Finish' for indicating task completion. The LLM then picks the sub-task most pertinent to the instruction and fills in its required parameters. For instance, given the instruction "Send a message to Bob", the output of the first select phase would be:

```
{ name: "Search", description: "Search for a contact", parameters: {
    "query": "Bob" }, UI_index: 3 }
```

When parameter values are unknown, MobileGPT asks the user for the missing information. This allows for interactive communication between the user and the system, enabling task automation even when the user's instruction is not entirely clear.

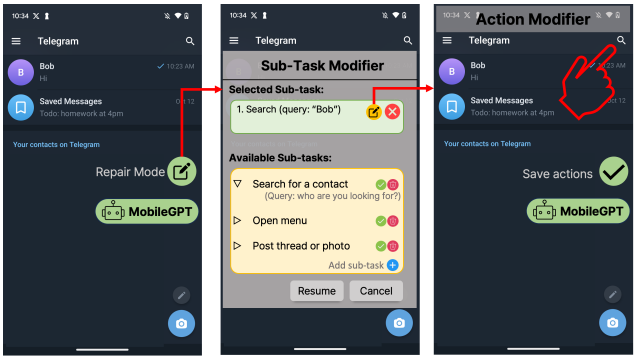
After a sub-task has been selected and its parameters filled, MobileGPT checks if the sub-task has already been learned (i.e., present in the memory) in the context of another task. If the sub-task is known, MobileGPT executes the sub-task directly from the memory, bypassing the Derive phase. Otherwise, it proceeds to the Derive phase.

Derive. During the *derive* phase, MobileGPT incrementally derives and executes low-level actions to accomplish the sub-task selected during the *select* phase. Specifically, MobileGPT prompts the LLM with *i)* the sub-task to execute, *ii)* the current screen representation, and *iii)* a pre-defined list of low-level actions—click, input, scroll, long-click—. Then, the LLM selects one of the actions from the list along with the index of the target UI element on which the action needs to be performed. For instance, if the index of the search button is 5, the output of the first *derive* action for the sub-task *'Search'* would be ***click**(ui_index=5)*. The action is then dispatched to the mobile device to be executed within the app. MobileGPT repeats this process until there is a transition in the app screen or the LLM explicitly indicates that the sub-task is completed. Afterward, MobileGPT returns to the Select phase to choose the next sub-task, and alternates between the Select and Derive phases until the LLM *selects* the subtask *'Finish'*, indicating that the user's instruction has been completed.

3.3 Dual Strategy Failure Handling

Despite their high reasoning abilities, LLMs sometimes show inconsistent and erroneous behavior. To address this, MobileGPT employs a dual-strategy correction mechanism.

Self-Correcting through feedback generation. Previous studies [13, 34] indicate that language models can self-correct when given appropriate feedback. To facilitate this in MobileGPT, we have heuristically identified two main types of commonly occurring errors and developed a rule-based self-feedback generation module. Upon detecting an error, MobileGPT generates appropriate feedback and appends it



(a) Enter repair mode (b) Modify Sub-task (Explore, Select) (c) Modify actions (Derive)

Figure 2: Illustration of HITL task repair mechanism

at the end of the next prompt. This guides LLMs in refining their approach and self-correcting the ongoing execution.

The first type of error occurs when the LLM incorrectly attempts to interact with the UIs. Feedback for this error includes:*"There is no UI with index i"* and *"The UI is not (clickable)."* These errors can be easily detected as they result in a failure when executing the action. The second type of error is when LLM gets stuck in a loop. For example, endlessly scrolling through a YouTube video list or trying to scroll when already at the bottom of a list. Feedback for this error includes: *"There is no change in the screen."* and *"You have looped the same screens X times."* These errors can be detected by monitoring the screen changes and tracking the visited app screens.

Human-in-the-loop (HITL) task repair. In case self-correction falls short, MobileGPT provides mechanisms for users to correct the LLM's mistakes themselves. In MobileGPT, there are three major points of potential LLM failure: Explore, Select, and Derive. *i) Explore* fails when sub-tasks are missing from the list of available sub-tasks, *ii) Select* fails when the LLM chooses incorrect sub-tasks or inputs wrong parameters, and *iii) Derive* fails when incorrect actions are derived and executed. MobileGPT provides specific repair mechanisms for each failure type[1].

Figure 2 is a simplified illustration of the repair mechanism. As illustrated, users can enter the repair mode anytime during the task execution by clicking on the MobileGPT's floating button. Upon entering the repair mode, MobileGPT pauses its execution and hands control over to the user. The user can then perform repairs directly on the current screen or navigate back to previous screens to re-do mistakes made by MobileGPT. Note that certain critical actions, such as sending a message or deleting a contact, cannot be undone. In such cases, users would have to manually restore the app to its original state. Upon identifying the screen needing repair, users can *i)* add or remove sub-tasks from the list of

[1]see https://mobile-gpt.github.io for demo video.

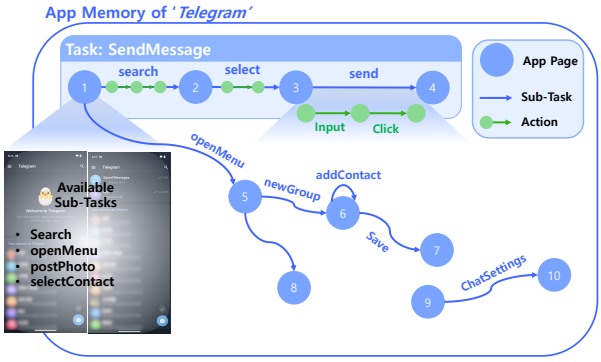


Figure 3: Transition graph of the app 'Telegram'

available sub-tasks (Explore), *ii)* change the selected sub-task and its parameters (Select), or *iii)* edit the actions involved in the sub-task by demonstrating a sequence of actions, similar to the programming by demonstration techniques (Derive). Additionally, MobileGPT assists users in locating the point of repair by providing a detailed visual summary of the task execution both in terms of sub-tasks and actions. This allows users to pinpoint the failure point even if they were not closely monitoring the execution.

A key advantage of MobileGPT's HITL repair is that users can return control back to the MobileGPT after fixing an error. This seamless transition requires a mutual understanding between the users and the LLMs, so that LLMs can recognize the corrections made and refine their approach accordingly. MobileGPT's design—organizing tasks as a sequence of sub-tasks rather than low-level actions—plays a pivotal role. The natural language descriptions of sub-tasks not only help in delivering the LLM's current progress to the user but also effectively capture the users' intentions behind their repair. For example, when the user corrects actions involved in the sub-task 'Search', the correction is encapsulated in a feedback *"User repaired how to: Search for a contact"* and appended to the next prompt. This feedback enables the LLM to grasp the intent behind the user's repair and proceed accordingly. Without such contextual information, LLMs could repeat the same error or actions already performed by the user.

4 Hierarchical App Memory

MobileGPT's memory architecture resembles how humans get familiar with the app—accumulate knowledge about the app as a whole, not just the tasks themselves. This section outlines how MobileGPT systematically archives the results of the Explore, Select, and Derive phases, and utilizes them for future task executions.

4.1 Memory Structure

MobileGPT organizes its memory in the form of a transition graph that encapsulates the following key information: i) available sub-tasks of each app screen *(Explore)*, ii) sequence of sub-tasks involved in each task *(Select)*, and iii) how to perform each sub-task *(Derive)*. Figure 3 illustrates an example of this graph for the app 'Telegram.' Note that these graphs exist per app, not per task, meaning that tasks within the same app all share the same graph.

Node. Each node in the transition graph symbolizes an app *page*—a particular state within an app that offers a unique set of functionalities (i.e., sub-tasks). Figure 4 illustrates examples of app pages; Note that screens with different visual appearances can belong to the same page (Figure 4 a), while screens that look similar (Figure 4 b) could belong to different pages, depending on their functionalities.

To this end, MobileGPT uses a list of sub-tasks to represent each page node (e.g., node 1 in Figure 3), categorizing screens that share the same list of sub-tasks as the same node. This facilitates the sharing of sub-task knowledge across a wider variation of app screens, ensuring that a sub-task learned on one screen can be applied to another, even if the two screens do not look exactly alike. For example, once MobileGPT learns how to 'Follow' in Elon Musk's profile page, it can use the same knowledge to 'Follow' Mark Zuckerberg.

Edge. Transitions between nodes are defined by sub-tasks, where each sub-task consists of a sequence of primitive actions that outline the steps for executing the sub-task. This hierarchical structure enables MobileGPT to efficiently reproduce learned sub-tasks by following the sequence of actions involved. MobileGPT establishes a new edge when it undergoes the Derive phase to execute previously unknown sub-tasks. Impotantly, these sub-task edges are not confined to individual tasks but are shared across tasks.

Task. After MobileGPT successfully completes the user instruction, it stores the task as a collection of node and sub-task pairs. This representation effectively indicates which sub-task needs to be executed on which page. Consequently, MobileGPT can bypass all three phases of learning—Explore, Select, and Derive when executing previously learned tasks. Additionally, it facilitates state-agnostic task execution; Regardless of which app page the app is currently on, MobileGPT can find which sub-task to execute.

4.2 Sub-task based Screen classification

Since an app page (i.e., a node) is a group of app screens that share the same list of sub-tasks, our memory architecture requires an accurate classification of app screens based on their functionalities. However, traditional screen classification methods [27, 61] are not suitable for this, as they focus on the screen's appearance, rather than its functionalities.

A brute-force approach to classify screens by their functionalities would be to run the Explore phase (i.e., extract a list of available sub-tasks) each time we encounter a new screen and fuzzy match with those in the memory. However, this would incur significant overhead and cost. Therefore, MobileGPT introduces a lighter approach to a sub-task-based

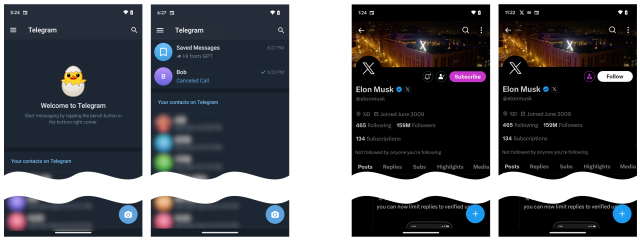


Figure 4: Examples of app pages: Screens in (a) belong to the same app page because they provide the same functionalities. Screens in (b) belong to different app pages because they provide different functionalities (the left is for 'Subscribe' but the right is for 'Follow').

screen classification: instead of checking if a screen *has* the same list of sub-tasks, it verifies if the screen can *perform* those sub-tasks.

Specifically, during the Explore phase, MobileGPT asks the LLM to return each subtask with relevant UI indexes (see § 3.2). These UI elements serve as requirements for performing a specific sub-task. For example, if the LLM returns the following 'Search' subtask during the Explore phase,

```
{ name: "Search", desc: "Search for a contact", params: {...}, UI_index: 3 }
```

the Search button, which corresponds to the UI index of 3 will be the key element for the subtask 'Search'.

```
<button UI_index=3 id="search_button" description="Search"/>
```

Then, each time MobileGPT arrives at a new screen, it verifies if the current screen includes key UI elements required by each sub-task. In specific, we check if the screen representation has a UI that matches the attributes (e.g., ui_id, description, and class_name) of the key UI elements. If the screen representation has matching UIs for all of the node's sub-tasks, we categorize the screen under that node.

If no matching node is found, MobileGPT undergoes the Explore phase. However, before creating a new node, MobileGPT calculates the cosine similarity between the screen's newly generated sub-tasks and those of existing nodes to ensure that no existing node has the same list of sub-tasks. If a match is found, we map the screen to the node and update its sub-task information without creating a new node. This double-check process minimizes the creation of redundant nodes, preventing already established sub-task edges from becoming obsolete.

In our evaluation, where MobileGPT encountered and classified 269 app screens, this method showed only 3 false positives (misclassifying different app pages as the same node) and no false negatives (failing to find the correct page node). The result is significantly better compared to existing SotA text-based [27] and vision-based [61] screen classification methods, which had 38, and 57 false positives, and 14, and 28 false negatives, respectively.

5 Flexible Task Recall

When recalling tasks or sub-tasks, the specific details of each action should be adjusted to the current context of its execution. To address this, MobileGPT leverages the attribute-based pattern matching and LLM's few-shot learning capability [5] to adjust and reproduce actions flexibly and accurately.

5.1 Attribute-based Action Adaptation.

There are two cases where we need to adjust the action. The first case is when a task parameter changes. For instance, if the user instruction changes from *"Send a message to Bob"* to *"Send a message to Alice"*, we need to search and click for *"Alice"* instead of *"Bob"*. The second case is when there is an alteration in the screen's content. For example, in a contact page, the hierarchical position of a specific contact (i.e., index of the UI) may change if contacts are added or removed. To effectively handle both scenarios, MobileGPT generalizes and adapts actions to both the parameters and the screen contents.

Generalizing Actions. When learning a new sub-task, MobileGPT stores actions in a generalized format to ensure that they are reusable across different contexts. This involves two steps: Screen Generalization and Parameter Generalization. For a given action (e.g., ***click*** *(ui_index=5)*), MobileGPT first generalizes it against the current screen representation by locating the target UI and recording its key attributes (e.g., id, text, description). Then, each key attributes are generalized further against the task parameters by comparing their values. If the value of the attribute matches that of the subtask parameter, the attribute's value is replaced with the corresponding parameter name. For example, given a subtask *'**select**(contact_name: "Bob")'* and a screen representation:

```
<!-- Contact list-->
<button index=5 id="contact" text="Bob"/>
<button index=6 id="contact" text="Alice"/>
```

the action '***click*** *(ui_index=5)*' gets generalized through following two steps:

```
click(ui_index=5) → 1. click(id:"contact", text: "Bob")
                  → 2. click(id:"contact", text:"[contact_name]")
```

Adapting Actions. When recalling the whole task, MobileGPT simply replays the given sequence of sub-tasks, bypassing all Explore, Select, and Derive phases. Yet, we still need to fill in the parameters for each sub-task and adjust the involved action accordingly. To do so, before executing each sub-task, MobileGPT queries LLM to slot-fill parameters based on the user instruction and the current screen representation. Similar to the Select phase, if parameter values

are unknown (e.g., missing from the instruction), MobileGPT asks the user for the information.

To adapt an action to a new context, we simply reverse the aforementioned two generalization steps based on the given sub-task parameters and the screen representation. Specifically, MobileGPT substitutes parameter names with the actual parameter values and identifies the UI element that matches these updated attributes. For example, given the sub-task *'**select**(contact_name: "Alice")'*, the generalized action for clicking a contact gets adjusted as follows:

```
click(id:"contact", text:"[contact_name]")
→ 1. click(id:"contact", text: "Alice") → 2. click(ui_index=6)
```

This two-step generalization and adaptation technique allows us to correctly locate the target UI in response to the change in the task parameters and screen.

Benefits. One advantage of identifying parameters at the sub-task level and using them for the action adaptation is its ability to handle ambiguities and incompleteness often found in real-world user instructions. A prior work [25], which generalizes actions based on the words in the instruction, fails to handle incomplete instructions like *"Send a message to Bob,"* where the message content is missing, and implicit instructions like *"Call the first contact from the recent call,"* where the contact name isn't explicitly stated.

On the other hand, MobileGPT effectively addresses this challenge by identifying and deriving information required to perform the task incrementally at the sub-task level. For example, when processing *"Send a message to Bob,"*, MobileGPT proactively asks for the missing message content before performing the subtask ***send**(message_content)*. Similarly, for *"Call the first contact from the recent call,"*, MobileGPT autonomously identifies the contact information by looking at the recent call list before executing the sub-task ***call**(contact_name)*. This allows MobileGPT to generalize actions against a more granular and precise set of parameters even if the user instruction is not entirely clear.

5.2 In-context Action Adaptation

The attribute-based adaptation method is not a one-size-fits-all solution, as several factors can lead to failure. First, the target UI may not include any key attributes for generalization. Second, multiple UIs could share the same attributes. Third, UIs may change unexpectedly with app updates.

Therefore, when the attribute-based adaptation fails, we resort to LLM to re-derive the action once again. However, LLMs are inherently non-deterministic. This means there is no guarantee they will reproduce a previously correct action. Worse yet, if the LLM had made a mistake in the past, it has a high probability of repeating the same error.

To address this issue, we leverage the few-shot learning capability of the LLM [5, 58] to produce more accurate and reliable responses. When querying the LLM, we present it with an example of how the action (i.e., one that rule-based adaptation failed) has been derived in the past. Specifically, the example includes a prior user instruction, an abbreviated version of the past screen representation, and its correct output. The output in the example could either be an action originally generated by the LLM during the Derive phase or one that has been modified by the user through the HITL task repair. In any case, the provided example always demonstrates a correct answer. By providing such examples, we effectively guide the LLM to produce consistent responses based on its memory. Moreover, if the example action is the one that has been corrected by the user, it prevents LLM from repeating the same mistakes.

6 Implementation

Our implementation of MobileGPT consists of an Android mobile app and a Python server, which communicates via Wi-Fi. The MobileGPT app captures screen representation and injects actions to the smartphone. The Python server manages app memory and processes the Explore, Select, Derive, and Recall phases.

LLM Agents. MobileGPT employs multiple LLM agents to cycle through the Explore, Select, Derive, and Recall phases. Each agent is equipped with a model tailored to its operational requirements. For Explore, Select, and Derive agents, which involve mobile screen understanding and multi-step reasoning, we used the most capable GPT-4-turbo language model. For translating user instructions to the high-level task representation and slot-filling subtask parameters, we opted for the faster and more economical GPT-3.5-turbo model.

App Launch. When an instruction is given, MobileGPT can recommend the most appropriate apps for the given task. It does so by crawling Google Play app descriptions for each app installed on the user's device and storing them in the vector database [45] using the text-embedding model [42]. Then, when a user gives an instruction, MobileGPT retrieves and presents the top three most relevant apps based on their descriptions. When the user makes a selection, MobileGPT launches the app and loads its corresponding app memory to proceed with the task.

Scrolling. When reproducing actions without definite target UI elements (i.e., scroll, sliding), MobileGPT checks for its subsequent action's target UIs. If the target is found, it skips all the scrolls in between and directly performs the subsequent action. Otherwise, the system reproduces the scroll actions until it can find the next action's target UI.

7 Evaluation

Our evaluation is divided into two sections: comparative study and ablation study. Throughout the evaluation, we used a Google Pixel 6 smartphone.

7.1 MobileGPT Dataset

The main contribution of MobileGPT lies in improving performance for repeated tasks and sub-tasks. However, existing datasets (e.g., AITW [48], DroidTask [59], PixelHelp [29]) do not capture this aspect, as they focus on task diversity with a highly independent set of user instructions.

Therefore, we created a dataset[1] designed to assess how effectively a system can learn from one instruction and adapt to another. The dataset includes 80 tasks across eight widely used off-the-shelf mobile apps: Google Dialer, Telegram, Twitter, TripAdvisor, Gmail, Microsoft To-Do, Uber Eats, and YouTube Music. Each task is accompanied by two user instructions with different task parameters—totaling *160 user instructions*. For example, the task of "Post Reply" for Twitter comes with two instructions: *"Post reply to Elon Musk's new tweet"* and *"Post reply to Bill Gate's new tweet saying 'Reply from MobileGPT'."* The dataset's complexity is comparable to the existing dataset, with the following average and maximum step length statistics—**MobileGPT: 5.3(avg)/13(max)**; DroidTask: 4.5/17; AITW: 5.5/14.

7.2 Comparative Study

7.2.1 Experimental Setup.

Baselines. We evaluate MobileGPT's performance against two baseline systems: AppAgent [62] and AutoDroid [59]. AppAgent is a vision-language model (GPT-4-turbo) powered task automator that uses screenshots to execute tasks. AutoDroid is an LLM (GPT-4-turbo) powered task automator that uses text screen representation, similar to MobileGPT. The key distinction between MobileGPT and the two baseline systems is that MobileGPT leverages app memory to learn and recall past executions, thereby optimizing costs and latency. For MobileGPT, we used the GPT-4-turbo model for the Explore, Select, and Derive phases and the GPT-3.5 Turbo model for slot-filling subtask parameters. For each baseline, we used their open-source version available on GitHub [37, 38].

Dataset. We evaluate the performance of three systems using datasets from all three works. After excluding overlapping or deprecated apps and tasks, the total number of apps is 18, and the total number of tasks is 185. This breaks down as follows: AutoDroid dataset (7 apps, 91 tasks), AppAgent dataset (3 apps, 14 tasks), and MobileGPT dataset (8 apps, 80 tasks). The average task complexity, measured by the number of steps, is 4.0, 6.6, and 5.3, respectively, and screen diversity, measured by the number of unique app pages, totals 237 (AutoDroid: 82, AppAgent: 39, MobileGPT: 116).

Procedure. The evaluation of MobileGPT is divided into two stages: *cold-start* and *warm-start*. In the *cold-start*, MobileGPT executes the task for the first time and constructs its memory. Then, in the *warm-start*, it utilizes the memory to execute the same task again. Note that AutoDroid and AppAgent run each task only once, as they do not have distinctions between cold-start and warm-start.

We manually monitored each step to identify errors during task execution. When an error occurred, we allowed the system three additional attempts for self-correction. A task was considered successful if the system navigated to the final screen within these three attempts. To ensure a fair evaluation in the comparison study, we did not utilize MobileGPT's HITL task repair feature.

Dataset	AutoDroid	AppAgent	MobileGPT	
			Cold	Warm
AutoDroid	83.5%	80.2%	**86.8%**	85.7%
AppAgent	27.3%	21.4%	**50.0%**	**50.0%**
MobileGPT	68.8%	60.5%	**82.5%**	**82.5%**
Overall	74.7%	67.4%	**82.7%**	82.16%

Table 1: Task success rate across different datasets.

7.2.2 Accuracy.

Table 1 compares the task success rate of each system across different datasets. The success rate of AppAgent dataset is significantly lower because the dataset includes more complex apps (YouTube, Spotify, Google Maps).

Notably, MobileGPT's *cold-start* demonstrates the highest accuracy across all three datasets, which can be attributed to several factors. First, AppAgent exhibits the lowest task accuracy because it relies solely on screenshot images, which proves inadequate for text-heavy applications such as Gmail, Twitter, and Telegram. Second, AutoDroid represents the app screen in a list of interactable UI elements. However, this overly abbreviated representation often omits crucial information about the screen, such as the hierarchical relationship between UIs and layout UI elements that describe nearby UIs, potentially misleading the LLM to make incorrect decisions.

In contrast, MobileGPT retains hierarchical information and minimally omits UI elements when converting the screen to a text representation, preserving as much information as possible from the original Android layout file. Additionally, MobileGPT's method of splitting tasks into subtasks, which resembles the chain-of-thought prompting techniques [55, 58], contributes to its overall *cold-start* task accuracy.

MobileGPT's *warm-start* had one additional failure than its *cold-start* due to an error while recalling a task from memory. Yet, the fact that the *warm-start* accuracy is nearly identical to the *cold-start* demonstrates that MobileGPT can recall tasks very successfully using the memory built through cold-start (More details in § 7.3).

7.2.3 Cost & Latency.

Figure 5 illustrates the average task latency and cost[2]. Comparisons were made only for tasks successfully completed

[1]The dataset is available at https://github.com/mobile-gpt/MobileGPT

[2]At the time of evaluation, the LLM costs (per 1K tokens) were: GPT-4-Turbo $0.01, GPT-3.5 Turbo $0.003

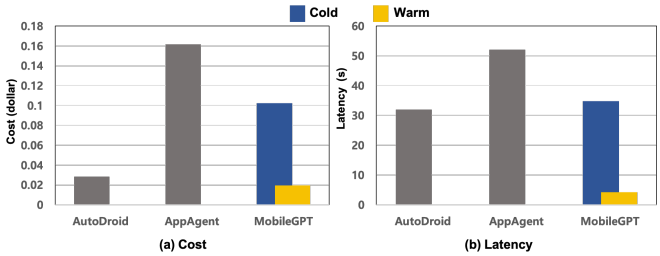


Figure 5: Average task completion cost and latency of AutoDroid, AppAgent and MobileGPT

by all three systems (77 tasks). The number of steps taken for each task was comparable across all systems.

AutoDroid achieves significantly lower costs through its concise screen representation and instruction prompts (i.e., system prompts). However, note that this cost-effectiveness may come at the expense of accuracy, as overly abbreviated screen representation could omit important details, and LLMs typically perform better with more comprehensive instructions. Conversely, AppAgent exhibits the highest cost and latency despite the lowest accuracy, demonstrating that relying solely on screenshots is highly inefficient both in terms of accuracy and cost.

MobileGPT's *cold-start* cost and latency fall between AutoDroid and AppAgent. MobileGPT's *warm-start*, however, exhibits the lowest cost and latency, with 87% and 90% reduction in latency compared to those of AutoDroid and AppAgent. This high efficiency is largely due to MobileGPT having learned the necessary sub-tasks and actions during the initial cold-start stage, leaving only simple tasks such as subtask slot-filling for the *warm-start* stage. We further analyze these benefits in § 7.3.

Overall, the comparison demonstrates that MobileGPT has slightly higher latency and cost during cold starts but becomes significantly faster and cheaper during warm starts. Given the recurrent nature of mobile tasks [15, 20, 32, 50], our approach is highly practical as it significantly reduces the burden of common warm-starts, with only a minor increase in that of cold-starts.

7.3 Ablation Study

In evaluating LLM-based systems, it is crucial to recognize that system performance heavily depends on the prompts used—a system could perform better simply from better prompts. Therefore, isolating the effect of prompts is essential to accurately measure the architectural benefits of such systems.

7.3.1 *Experimental Setup.*

Baseline. To further analyze MobileGPT's structural design, including its three-phase inference structure (Explore-Select-Derive) and memory, we compared MobileGPT against a custom baseline system. This baseline is designed to share the same prompts as MobileGPT while following the traditional Derive-only structure employed in prior approaches [54, 59, 60]. This comparison allows us to measure performance gains attributable solely to MobileGPT's unique structural design.

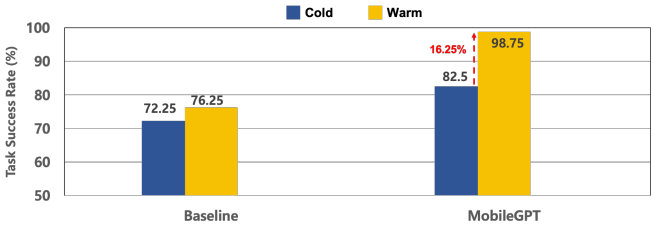


Figure 6: Task success rate of MobileGPT and the baseline.

Dataset. In the ablation study, we used the MobileGPT dataset, as datasets from other works cannot fully assess MobileGPT's unique ability to learn and recall tasks (see § 7.1).

Procedure. The evaluation is divided into *cold-start* and *warm-start* stages. The *cold-start* executes the first instruction set from each high-level task, and the *warm-start* executes the second instruction set. This procedure assesses how well MobileGPT learns from one instruction and adapts to another with different task parameters.

When the system deviates from the correct path, we allow three extra attempts for self-correction. If it fails to correct itself, we mark the task as a failure and manually repair the error using the HITL task repair. After the repair, we let the system continue the task. In the case of MobileGPT, The final path after the repair gets stored in the memory and utilized during the *warm-start*.

Offline exploration. For MobileGPT, we used a random explorer to *explore* each target app before beginning the tasks. we discovered 50 unique app pages for each app, which typically took between 10 to 15 minutes. These app pages accounted for 89.65% (104 out of 116) of the total app pages required during the evaluation. The remaining app pages, along with those unsuitable for random exploration (e.g., payment page), can be explored offline by monitoring user interactions with the app. The total cost for this random exploration was $10.78, which is deemed reasonable considering that this preparation is a one-time process.

7.3.2 *Task Success Rate*

Figure 6 illustrates the task success rate of MobileGPT compared to the baseline. Impressively, MobileGPT achieves near-perfect accuracy (98.75%) during the warm-start phase, with a single exception observed in Uber Eats. This improvement over the cold-start accuracy is due to the user's ability to correct task executions before saving them to memory, ensuring reliability and correctness of the task recall even if LLMs made mistakes during the cold-start. For instance, in Gmail's *"Recover Deleted Email"* task, MobileGPT initially failed to open the More Options Menu because there were two seemingly identical More Options buttons on the screen. However,

Accuracy (%)	Task Learning (cold-start)			Task Recall (warm-start)	
	Explore	Select	Derive	Slot Filling	In-context Adaptation
Phase	96.4%	96.2%	99.1%	99.5%	100%
Step	95.5%			99.8%	

Table 2: MobileGPT's step and phase accuracy.

after "teaching" it which button to click through HITL task repair, MobileGPT successfully clicked the correct button during the warm start. Furthermore, MobileGPT's combination of Attribute-based and In-context action adaptation enables precise and consistent recall of learned tasks, even when task parameters and resulting screen content change.

MobileGPT consistently outperforms the baseline, even during the cold-start stages. This advantage arises from two factors: MobileGPT's capability to decompose complex tasks into multiple subtasks and its ability to share learned subtasks across different tasks. Notably, when MobileGPT makes an error during a subtask and receives a correction, either through self-feedback generation or HTIL task repair, it avoids repeating the same mistake in other tasks involving overlapping subtasks. For example, finding a contact in Telegram requires using the search button instead of scrolling through the contact list. In its initial task, MobileGPT attempts to scroll to locate the contact. However, after correcting this approach through self-feedback generation and learning to use the search button, MobileGPT subsequently goes directly to the search button for similar tasks, bypassing the scrolling.

In contrast, the baseline system frequently resorts to scrolling through contacts, consistently repeating the same errors across tasks. Furthermore, the baseline sometimes fails or performs inefficiently during the warm start phase, even if it has successfully completed the task during the cold start. This inconsistency highlights the non-deterministic nature of LLMs, which can generate varying outputs based on factors such as task parameters, screen representation, and even the timing of the query.

7.3.3 Step Accuracy.

To further analyze accuracy, we examine LLM queries at each step of the tasks. In MobileGPT, each step is broken down into multiple phases: during the cold-start, MobileGPT goes through Explore, Select, and Derive phases to learn new tasks. During the warm-start, it employs sub-task slot-filling and in-context action adaptation for recalling learned tasks. A failure in any single phase results in the entire step being marked as a failure. The step accuracy of the baseline is 92.4%

Table 2 presents MobileGPT's accuracy across its different phases. MobileGPT exhibits a higher step accuracy than the baseline, even during cold-starts. This improvement can be traced to MobileGPT's ability to distribute the reasoning load for each step across multiple phases, similar to how the Chain-of-thought prompting [58] improves LLM accuracy by decomposing problems into a series of intermediate steps before reaching the final solution.

App Name	Average Memory Hit Rate	
	Cold-Start	Warm-Start
Google Dialer	35.9%	95.8%
Telegram	50.1%	98.6%
Twitter	31.7%	80.1%
TripAdvisor	29.1%	89.3%
Gmail	6.7%	70.6%
Microsoft To-Do	27.8%	92.6%
Uber Eats	26.1%	78.0%
YouTube Music	18.0%	87.2%

Table 3: MobileGPT Average Memory Hit Rate per App.

When recalling previously learned tasks (*warm-start*), MobileGPT demonstrates near-perfect accuracy, with only one slot-filling query failing. Remarkably, MobileGPT achieves 100% accuracy for in-context adaptation. In our evaluation, out of 327 primitive actions, 53 actions (16.2%) were not adaptable through attribute-based adaptation and therefore required in-context adaptation via LLM. As a result, LLM was able to adapt all 53 actions to new instructions and screens.

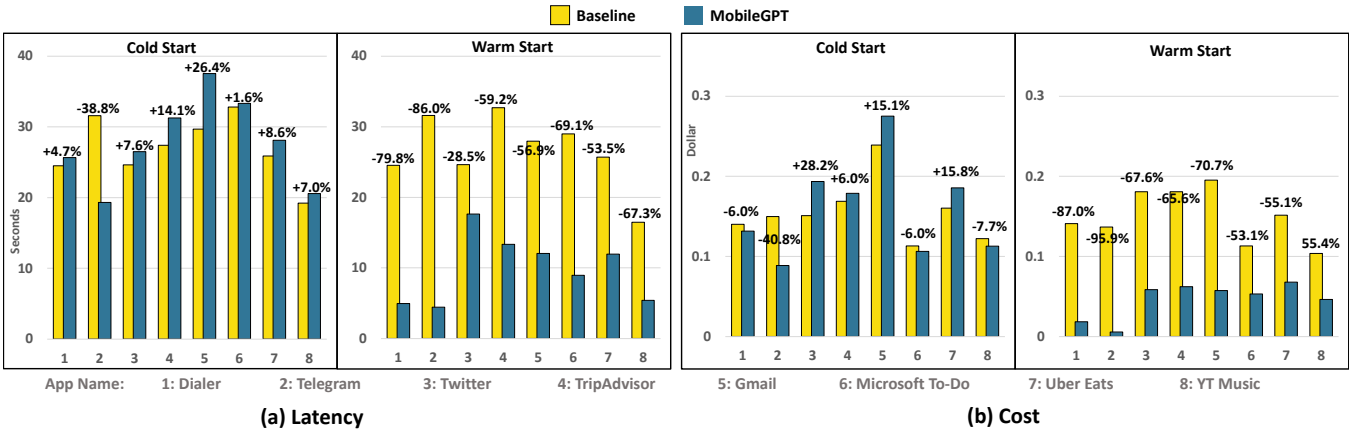
7.3.4 Cold-start latency & cost.

Figure 7 compares the latency and cost between MobileGPT and the baseline. During the initial cold start, even with the additional Select phase that MobileGPT employs to break down tasks into subtasks, the increase in latency and cost is minimal, averaging only 3.9% and 0.6%, respectively.

Moreover, MobileGPT even outperforms the baseline in `Telegram` due to its ability to reuse learned subtasks to be reusable across different tasks. Table 3 displays the average memory hit rate for each app, indicating the percentage of actions retrieved directly from memory without engaging the LLM. Since LLM is the primary source of latency, a higher memory hit rate results in faster task execution. `Telegram` serves as an excellent example; since many of its tasks involve the common subtask of searching for a specific contact, MobileGPT significantly outperforms the baseline, even during the cold-starts. Conversely, `Gmail` shows a low memory hit rate (6.7%), as its tasks have few overlapping sub-tasks. While current results suggest that `Gmail` does not fully leverage our design when learning new tasks (cold-start), we anticipate that both latency and cost will decrease exponentially as more subtasks are accumulated over time.

7.3.5 Warm start latency & cost.

In the warm start stage, where the same high-level task is repeated but with different parameters (i.e., different user instructions), MobileGPT dramatically reduces latency and cost, achieving improvements of 62.5% and 68.8% over the baseline, respectively. The effectiveness of MobileGPT's memory varies depending on the memory hit rate. The memory hit rate for *warm start* is not always 100% because some actions still require LLM for its adaptation (i.e., In-context



(a) Latency (b) Cost

Figure 7: Average task latency and cost of MobileGPT and the baseline

adaptation). Nevertheless, even in the worst case where no actions are adaptable (e.g., Twitter), our evaluation indicates that MobileGPT consistently outperforms the baseline.

7.4 Resource Consumption

We measured the resource consumption of the MobileGPT on mobile devices using the Android Studio profiler tool [12]. The analysis indicated no significant overhead, as the majority of computations are offloaded to the server. The average power consumption of the MobileGPT app during task execution was 330.38 mW, comprising CPU at 237.05 mW, memory at 67.15 mW, and WLAN at 26.18 mW, representing 11.78% of the total power consumption. he average memory consumption was 213 MB, which is only 0.04% of a typical modern smartphone's memory capacity (6GB).

7.5 User Study

Participants and Study Procedure. We recruited 23 participants (16 male, 7 female, mean age 23.2, SD=3.5, range=19-32) through an online community posting, with each participant receiving a compensation of 12 USD. The study consisted of two sessions: the first session evaluated the overall usability of MobileGPT in comparison to Samsung Bixby Macro (Programming by Demonstration) and the custom baseline. The second session assessed the usability of MobileGPT's human-in-the-loop task repair mechanisms. Each session lasted 30 minutes. All recruitment and experiments complied with our institution's IRB policies.

In the first session, participants were asked to perform the following three tasks using the three task automation tools: *1) "Find me available hotels in Las Vegas from November 10 to November 15," 2) "Find me hotels in New York," and 3) "Find me restaurants in Las Vegas."* These tasks were designed to assess the usability of the automation tools across three scenarios: *i)* executing a completely new task, *ii)* repeating the same task with different parameters, and *iii)* executing a new but similar task. After each task, participants rated the usability of each tool on a 7-point Likert scale. Upon completing all tasks, participants ranked the tools based on their preferences and indicated their willingness to use each tool in real-world scenarios.

	Overall Usability (7-point-scale)		
Scenario	**PbD**	**Baseline**	**MobileGPT**
New Task	3.2	4.4	4.7
Repeat Task	3.0	4.3	6.1
Similar Task	3.0	4.3	5.7

Table 4: Usability score for different task scenarios.

In the second session, participants were introduced to MobileGPT's repair mechanisms, which allow users to correct mistakes made by the LLM. After a brief tutorial, participants were tasked with executing the instruction *"Modify Tom's phone number to 123-456-789"* using MobileGPT. During the study, MobileGPT was configured to make predefined errors covering all three areas of repair: Explore, Select, and Derive. Participants were not informed in advance about the specific mistakes the system would make. Following the task, they rated the usability of the repair mechanism using the System Usability Scale (SUS) [4]. Additionally, they evaluated the necessity and effectiveness of the repair mechanism, along with their accuracy tolerance for a task automator with and without the repair mechanism.

Session 1: Overall usability. Table 4 shows the usability scores for each task automation tool across scenarios. Throughout the scenarios, participants generally found PbD inefficient because they needed to re-create the macro script even at the slightest change in the screen or instructions, and the baseline as convenient but too slow. In contrast, MobileGPT initially had a usability score similar to the baseline in the first scenario, but showed significant improvement in the subsequent tasks, with Mann-Whitney U Test results of (U=216.5, p=0.26), (U=24.0, p=$4.8e^{-8}$), and (U=67.5, p=$6.5e^{-6}$), respectively—*"(MobileGPT) performs the task accurately at a fairly high speed. Possibly faster than doing it by hand. Seems to have good generalizability" (P4), "It was a little sluggish for*

things that I hadn't done before, but it was still faster than the baseline and I figured it would get faster with more training" (P7). This trend suggests that MobileGPT's ability to quickly reproduce tasks and adapt learned sub-tasks to related tasks greatly enhances its usability. This is corroborated by the post-survey results, where all but one participant favored MobileGPT the most, and all participants expressed willingness to use MobileGPT in real-world applications, compared to 35% and 30% for the baseline and PbD tools, respectively.

Session 2: Human-in-the-loop Task Repair. MobileGPT's task repair mechanism received an average score of 64.6 (std=16.9) on the System Usability Scale, indicating it as *"ok"* user-friendliness [3]. While there is room for improvement, the score is considered acceptable given the high complexity of the repair mission (repairing all three phases within a single task) and the participants' general unfamiliarity with the concept of autonomous agents making errors.

In the post-survey, participants rated the necessity and effectiveness of the repair mechanism highly, with scores of 4.7 and 4.8 out of 5, respectively. Participants also indicated that the acceptable accuracy threshold for a task automator without a repair mechanism is 96% on average, whereas, with a repair mechanism, this threshold drops to 84%. This indicates that participants are more lenient with accuracy expectations when a repair mechanism is available, underscoring its value in task automation.

8 Related Work

Using LLM in UI Task automation. Recently, there have been several efforts to leverage LLMs for task automation [40, 54, 59, 60, 62], capitalizing on their ability to comprehend and perform tasks without requiring prior training or user demonstrations. Noteworthy examples include AutoDroid [59] and AppAgent [62], which enhance LLMs with app-specific knowledge to improve their understanding of mobile applications and boost task performance accuracy. Despite these advancements, the practical deployment of these approaches in real-world task automation remains uncertain due to the inherent unreliability and high costs associated with LLMs. MobileGPT tackles these challenges by integrating human-like app memory with LLMs, allowing for the effective reuse of learned subtasks, which significantly reduces the number of LLM queries, thereby optimizing latency and cost.

Macro Mining. Another line of work closely related to MobileGPT is macro mining [16, 18, 29, 30, 53], which aims to generate macro scripts without human intervention. For instance, Li et al. [16] utilize LLM to discover sub-tasks within app traces and combine them to create task-oriented macros. However, a common challenge in using macros is the generalization of actions. Existing approaches tackle this challenge through embedding-based screen matching and attribute-based UI matching. However, these methods often struggle when screen similarity is ambiguous or when UIs lack critical attributes. In contrast, MobileGPT effectively addresses these issues by using sub-task-based screen classification and in-context action adaptation, enabling more robust and reliable task automation.

Caching LLM responses. Fundamentally, MobileGPT is built upon the principles of caching LLMs' responses. Similar research efforts include caching frequently occurring token states for faster inference [9], caching LLM responses to handle similar queries [64], and training smaller models to replicate cached responses [47, 51]. However, these existing studies primarily focus on the use of LLMs in chatbot applications. To the best of our knowledge, MobileGPT is the first to explore caching within the context of an LLM-based task automator.

9 Discussion

Security & Privacy. Screen representations can contain personal information, such as names and phone numbers, posing privacy risks when transmitted to the LLM. To mitigate this, we can employ a Personal Identifier Information (PII) Scanner [36, 46] to detect personal data within prompts and replace it with non-private placeholders. Additionally, certain actions—e.g., agreeing to terms of service or confirming a payment—should be performed with user supervision. To address this, we provide the LLM with the option to select a *'get user confirm'* action during the Derive phase, ensuring that any potentially risky actions are identified and user confirmation is requested before proceeding.

Unsupported Apps. The current implementation of MobileGPT does not support mobile apps that lack text representation of their screen layouts, such as those using third-party UI engines (e.g., Flutter, Web Apps) or screens dominated by images (e.g., maps, camera views). To address this limitation, we can leverage screen-to-text translation models [21, 23, 61, 63] or vision-language models (VLMs) [31, 43, 57] to process both the screenshot and the text-based screen representation, enabling MobileGPT to function effectively across a wider range of applications.

10 Conclusion

We introduced MobileGPT, a novel LLM-powered mobile task automator that enhances the efficiency and reliability of task automation by emulating human cognitive processes. We anticipate that MobileGPT will strengthen the integration of intelligent automation into everyday technology use.

Acknowledgments

This work was supported by IITP (RS-2023-00232728), TIPA (TIPS 00262147, RS-2024-00447529), K-Startup (20144069), and the National Research Foundation of Korea(NRF) grant (RS-2024-00347516).

References

[1] anthropic. 2023. *Talk to Claude.* anthropic. Retrieved Nov 11, 2023 from https://claude.ai/

[2] Apple. 2023. *Use Siri on all your Apple devices.* Meta. Retrieved Nov 11, 2023 from https://support.apple.com/en-us/HT204389

[3] Aaron Bangor, Philip Kortum, and James Miller. 2009. Determining what individual SUS scores mean: Adding an adjective rating scale. *Journal of usability studies* 4, 3 (2009), 114–123.

[4] John Brooke. 1996. Sus: a "quick and dirty'usability. *Usability evaluation in industry* 189, 3 (1996), 189–194.

[5] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems* 33 (2020), 1877–1901.

[6] Carlos G Correa, Mark K Ho, Frederick Callaway, Nathaniel D Daw, and Thomas L Griffiths. 2023. Humans decompose tasks by trading off utility and computational cost. *PLOS Computational Biology* 19, 6 (2023), e1011087.

[7] Carlos G Correa, Mark K Ho, Fred Callaway, and Thomas L Griffiths. 2020. Resource-rational task decomposition to minimize planning costs. *arXiv preprint arXiv:2007.13862* (2020).

[8] Pınar Erten and Oguzhan Ozdemir. 2020. The Digital Burnout Scale Development Study. *İnönü Üniversitesi Eğitim Fakültesi Dergisi* 21 (10 2020). https://doi.org/10.17679/inuefd.597890

[9] In Gim, Guojun Chen, Seung-seob Lee, Nikhil Sarda, Anurag Khandelwal, and Lin Zhong. 2023. Prompt cache: Modular attention reuse for low-latency inference. *arXiv preprint arXiv:2311.04934* (2023).

[10] Google. 2023. *Create your own accessibility service.* Google. Retrieved Nov 11, 2023 from https://developer.android.com/guide/topics/ui/accessibility/service

[11] Google. 2023. *Hey Google.* Google. Retrieved Nov 11, 2023 from https://assistant.google.com/

[12] Google. 2024. *Profile your app performance.* Google. Retrieved July 25, 2024 from https://developer.android.com/studio/profile/

[13] Zhibin Gou, Zhihong Shao, Yeyun Gong, Yelong Shen, Yujiu Yang, Nan Duan, and Weizhu Chen. 2023. Critic: Large language models can self-correct with tool-interactive critiquing. *arXiv preprint arXiv:2305.11738* (2023).

[14] Emilie Munch et al Gregersen. 2023. Digital dependence: Online fatigue and coping strategies during the COVID-19 lockdown. *Media, Culture, and Society* (2023).

[15] Emitza Guzman and Walid Maalej. 2014. How do users like this feature? a fine grained sentiment analysis of app reviews. In *2014 IEEE 22nd international requirements engineering conference (RE).* Ieee, 153–162.

[16] Forrest Huang, Gang Li, Tao Li, and Yang Li. 2023. Automatic Macro Mining from Interaction Traces at Scale. *arXiv preprint arXiv:2310.07023* (2023).

[17] Peter C Humphreys, David Raposo, Tobias Pohlen, Gregory Thornton, Rachita Chhaparia, Alistair Muldal, Josh Abramson, Petko Georgiev, Adam Santoro, and Timothy Lillicrap. 2022. A data-driven approach for learning to control computers. In *International Conference on Machine Learning.* PMLR, 9466–9482.

[18] Peter C Humphreys, David Raposo, Tobias Pohlen, Gregory Thornton, Rachita Chhaparia, Alistair Muldal, Josh Abramson, Petko Georgiev, Adam Santoro, and Timothy Lillicrap. 2022. A data-driven approach for learning to control computers. In *International Conference on Machine Learning.* PMLR, 9466–9482.

[19] Akshay Kumar Jagadish, Marcel Binz, Tankred Saanum, Jane X Wang, and Eric Schulz. 2023. Zero-shot compositional reinforcement learning in humans. (2023).

[20] Chakajkla Jesdabodi and Walid Maalej. 2015. Understanding usage states on mobile devices. In *Proceedings of the 2015 ACM international joint conference on pervasive and ubiquitous computing.* 1221–1225.

[21] Kenton Lee, Mandar Joshi, Iulia Turc, Hexiang Hu, Fangyu Liu, Julian Eisenschlos, Urvashi Khandelwal, Peter Shaw, Ming-Wei Chang, and Kristina Toutanova. 2023. Pix2Struct: Screenshot Parsing as Pretraining for Visual Language Understanding. arXiv:2210.03347 [cs.CL]

[22] Sunjae Lee, Hoyoung Kim, Sijung Kim, Sangwook Lee, Hyosu Kim, Jean Young Song, Steven Y Ko, Sangeun Oh, and Insik Shin. 2022. A-mash: providing single-app illusion for multi-app use through user-centric UI mashup. In *Proceedings of the 28th Annual International Conference on Mobile Computing And Networking.* 690–702.

[23] Gang Li and Yang Li. 2023. Spotlight: Mobile UI understanding using vision-language models with a focus. (2023).

[24] Tao Li, Gang Li, Jingjie Zheng, Purple Wang, and Yang Li. 2022. MUG: Interactive Multimodal Grounding on User Interfaces. *arXiv preprint arXiv:2209.15099* (2022).

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

[26] Toby Jia-Jun Li, Yuanchun Li, Fanglin Chen, and Brad A Myers. 2017. Programming IoT devices by demonstration using mobile apps. In *End-User Development: 6th International Symposium, IS-EUD 2017, Eindhoven, The Netherlands, June 13-15, 2017, Proceedings 6.* Springer, 3–17.

[27] Toby Jia-Jun Li, Lindsay Popowski, Tom Mitchell, and Brad A Myers. 2021. Screen2vec: Semantic embedding of gui screens and gui components. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems.* 1–15.

[28] Toby Jia-Jun Li and Oriana Riva. 2018. KITE: Building conversational bots from mobile apps. In *Proceedings of the 16th Annual International Conference on Mobile Systems, Applications, and Services.* 96–109.

[29] Yang Li, Jiacong He, Xin Zhou, Yuan Zhang, and Jason Baldridge. 2020. Mapping natural language instructions to mobile UI action sequences. *arXiv preprint arXiv:2005.03776* (2020).

[30] Yuanchun Li and Oriana Riva. 2021. Glider: A reinforcement learning approach to extract UI scripts from websites. In *Proceedings of the 44th International ACM SIGIR Conference on Research and Development in Information Retrieval.* 1420–1430.

[31] Haotian Liu, Chunyuan Li, Qingyang Wu, and Yong Jae Lee. 2023. Visual instruction tuning. *arXiv preprint arXiv:2304.08485* (2023).

[32] Huaxiao Liu, Xinglong Yin, Shanshan Song, Shanquan Gao, and Mengxi Zhang. 2022. Mining detailed information from the description for App functions comparison. *IET Software* 16, 1 (2022), 94–110.

[33] Martin Lövdén, Benjamín Garzón, and Ulman Lindenberger. 2020. Human skill learning: expansion, exploration, selection, and refinement. *Current Opinion in Behavioral Sciences* 36 (2020), 163–168.

[34] Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, et al. 2023. Self-refine: Iterative refinement with self-feedback. *arXiv preprint arXiv:2303.17651* (2023).

[35] meta. 2023. *Introducing Llama 2.* meta. Retrieved Nov 11, 2023 from https://ai.meta.com/llama/

[36] Microsoft. 2023. *How to detect and redact Personally Identifying Information (PII).* Microsoft. Retrieved Nov 11, 2023 from https://learn.microsoft.com/en-us/azure/ai-services/language-service/personally-identifiable-information/how-to-call

[37] mnotgod96. 2024. *AppAgent-TencentQQGYLab.* Retrieved July 25, 2024 from https://github.com/mnotgod96/AppAgent

[38] MobileLLM. 2024. *AutoDroid.* Retrieved July 25, 2024 from https://github.com/MobileLLM/AutoDroid

[39] MultiOn. 2023. *The world's first Personal AI Agent.* MultiOn. Retrieved Nov 11, 2023 from https://www.multion.ai/

[40] Reiichiro Nakano, Jacob Hilton, Suchir Balaji, Jeff Wu, Long Ouyang, Christina Kim, Christopher Hesse, Shantanu Jain, Vineet Kosaraju, William Saunders, et al. 2021. Webgpt: Browser-assisted question-answering with human feedback. *arXiv preprint arXiv:2112.09332* (2021).

[41] openai. 2023. *Creating safe AGI that benefits all of humanity.* openai. Retrieved Nov 11, 2023 from https://openai.com/

[42] OpenAI. 2023. *New and improved embedding model.* OpenAI. Retrieved Nov 11, 2023 from https://openai.com/blog/new-and-improved-embedding-model

[43] OpenAI. 2023. *Vision.* OpenAI. Retrieved Nov 11, 2023 from https://platform.openai.com/docs/guides/vision

[44] OthersideAI. 2023. *Your AI assistant for everyday tasks.* OthersideAI. Retrieved Nov 11, 2023 from https://www.hyperwriteai.com/personal-assistant

[45] Pinecone. 2023. *Long-Term Memory for AI.* Pinecone Systems. Retrieved Nov 11, 2023 from https://www.pinecone.io/

[46] Endpoint Protector. 2023. *Cutting-Edge PII Scanner.* Endpoint Protector. Retrieved Nov 11, 2023 from https://www.endpointprotector.com/solutions/ediscovery/pii-scanner

[47] Guillem Ramírez, Matthias Lindemann, Alexandra Birch, and Ivan Titov. 2023. Cache & distil: Optimising API calls to large language models. *arXiv preprint arXiv:2310.13561* (2023).

[48] Christopher Rawles, Alice Li, Daniel Rodriguez, Oriana Riva, and Timothy Lillicrap. 2023. Android in the wild: A large-scale dataset for android device control. *arXiv preprint arXiv:2307.10088* (2023).

[49] Significant-Gravitas. 2023. *AutoGPT: the heart of the open-source agent ecosystem.* github. Retrieved Nov 11, 2023 from https://github.com/Significant-Gravitas/AutoGPT

[50] Christoph Stanik, Marlo Haering, Chakajkla Jesdabodi, and Walid Maalej. 2020. Which app features are being used? Learning app feature usages from interaction data. In *2020 IEEE 28th International Requirements Engineering Conference (RE).* IEEE, 66–77.

[51] Ilias Stogiannidis, Stavros Vassos, Prodromos Malakasiotis, and Ion Androutsopoulos. 2023. Cache me if you Can: an Online Cost-aware Teacher-Student framework to Reduce the Calls to Large Language Models. *arXiv preprint arXiv:2310.13395* (2023).

[52] Liangtai Sun, Xingyu Chen, Lu Chen, Tianle Dai, Zichen Zhu, and Kai Yu. 2022. META-GUI: Towards Multi-modal Conversational Agents on Mobile GUI. *arXiv preprint arXiv:2205.11029* (2022).

[53] Daniel Toyama, Philippe Hamel, Anita Gergely, Gheorghe Comanici, Amelia Glaese, Zafarali Ahmed, Tyler Jackson, Shibl Mourad, and Doina Precup. 2021. Androidenv: A reinforcement learning platform for android. *arXiv preprint arXiv:2105.13231* (2021).

[54] Bryan Wang, Gang Li, and Yang Li. 2023. Enabling conversational interaction with mobile ui using large language models. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems.* 1–17.

[55] Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. 2023. Voyager: An open-ended embodied agent with large language models. *arXiv preprint arXiv:2305.16291* (2023).

[56] Lei Wang, Wanyu Xu, Yihuai Lan, Zhiqiang Hu, Yunshi Lan, Roy Ka-Wei Lee, and Ee-Peng Lim. 2023. Plan-and-solve prompting: Improving zero-shot chain-of-thought reasoning by large language models. *arXiv preprint arXiv:2305.04091* (2023).

[57] Wenhai Wang, Zhe Chen, Xiaokang Chen, Jiannan Wu, Xizhou Zhu, Gang Zeng, Ping Luo, Tong Lu, Jie Zhou, Yu Qiao, et al. 2023. Visionllm: Large language model is also an open-ended decoder for vision-centric tasks. *arXiv preprint arXiv:2305.11175* (2023).

[58] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in Neural Information Processing Systems* 35 (2022), 24824–24837.

[59] Hao Wen, Yuanchun Li, Guohong Liu, Shanhui Zhao, Tao Yu, Toby Jia-Jun Li, Shiqi Jiang, Yunhao Liu, Yaqin Zhang, and Yunxin Liu. 2023. Empowering llm to use smartphone for intelligent task automation. *arXiv preprint arXiv:2308.15272* (2023).

[60] Hao Wen, Hongming Wang, Jiaxuan Liu, and Yuanchun Li. 2023. DroidBot-GPT: GPT-powered UI Automation for Android. *arXiv preprint arXiv:2304.07061* (2023).

[61] Jason Wu, Siyan Wang, Siman Shen, Yi-Hao Peng, Jeffrey Nichols, and Jeffrey P Bigham. 2023. WebUI: A Dataset for Enhancing Visual UI Understanding with Web Semantics. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems.* 1–14.

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

[63] Xiaoyi Zhang, Lilian de Greef, Amanda Swearngin, Samuel White, Kyle Murray, Lisa Yu, Qi Shan, Jeffrey Nichols, Jason Wu, Chris Fleizach, et al. 2021. Screen recognition: Creating accessibility metadata for mobile applications from pixels. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems.* 1–15.

[64] Hanlin Zhu, Banghua Zhu, and Jiantao Jiao. 2024. Efficient Prompt Caching via Embedding Similarity. *arXiv preprint arXiv:2402.01173* (2024).