Explore, Select, Derive, and Recall: Augmenting LLM with Human-like Memory for Mobile Task Automation

Sunjae Lee
School of Computing, KAIST
Republic of Korea
sunjae1294@kaist.ac.kr

Junyoung Choi
School of Computing, KAIST
Republic of Korea
joonchoi518@kaist.ac.kr

Jungjae Lee
School of Computing, KAIST
Republic of Korea
dlwjdwo00701@kaist.ac.kr

Hojun Choi
School of Computing, KAIST
Republic of Korea
hchoi@kaist.ac.kr

Steven Y. Ko
Simon Fraser University
Canada
steveyko@sfu.ca

Sangeun Oh
Ajou University
Republic of Korea
sangeunoh@ajou.ac.kr

Insik Shin
School of Computing, KAIST
Republic of Korea
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 MemoDroid, an innovative LLM-based mobile task automator enhanced with a unique app memory. MemoDroid 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 components that can be re-used, re-arranged, and adapted for various objectives. We implement MemoDroid using online LLMs services (GPT-3.5 and GPT-4) and evaluate its performance on 50 unique mobile tasks across 5 widely used mobile apps. The results indicate that MemoDroid can adapt learned tasks to varying contexts with 100% accuracy and reduces their latency and cost by 69.22% and 77.36% compared to a GPT-4 powered baseline.

1 Introduction

As smartphones have grown from simple call-making bricks to pocket-sized powerhouses, they have become integral to our daily lives. Today, we rely on a whole universe of mobile apps to manage tasks ranging from communication to home security. Consequently, the need for efficient task automation in the mobile environment has become increasingly important, to alleviate digital fatigue from repetitive and complex digital interactions that often overwhelm many users today [9, 15].

Unsurprisingly, there have been a number of approaches to achieve task automation. API-based approaches like Siri [2] and Google Assistant [13] enable users to interact with specific functionality of a mobile app using natural language conversation. However, they require developers' substantial coding efforts, where they have to manually configure new logic for each new task. Demonstration-based methods [22, 26, 27, 29] empower end-users to custom-program automation. However, as it heavily relies on human demonstrations, it suffers from scalability issues. Similarly, learning-based approaches [17, 25, 30, 59, 60] require a large corpus of human-annotated datasets, hampering the widespread application.

Most recently, LLMs [1, 37, 41], equipped with high reasoning and generalization abilities, have been a game-changer. LLM-based task automators [39, 44, 56, 66] 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, due to the inherent non-deterministic nature of LLMs, the reliability of task automation can be compromised. LLMs might not only produce incorrect responses but can also yield different outputs for the same query. Such unreliability is critical in mobile environments as many mobile tasks nowadays involve sensitive and private actions. Second, leveraging LLMs for task automation can be 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 MemoDroid[1], an LLM-based mobile task automator that can learn and recall mobile tasks through its own app memory. Specifically, MemoDroid aims to achieve the 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 adjust its execution in response to varying contexts, rather than simply duplicating the previous runs. *iii) Low latency and cost*: MemoDroid should minimize the cost of both learning tasks and recalling the tasks.

In designing MemoDroid, we drew inspiration from how humans effectively learn and recall mobile tasks. Assume you have learned a new task *"Send a message to Bob."* Having learned this, you can not only reproduce similar tasks like *"Send a message to Alice"* but also effortlessly evoke past experiences and apply them to perform new tasks like *"Open chat with 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 [7, 8, 19, 35].

Specifically, consider how humans learn new tasks in 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 clicking, inputting text, or scrolling. Once we have completed the task by repeating these 3 steps, it becomes part of the memory, allowing for easy **4) recall** and repetition of not only the task itself but also its involved *sub-tasks.*

This human capacity to store and recall memories at the unit of sub-tasks is what we aim to replicate with MemoDroid. To this end, we tackle three key challenges: *i) Accurate and reliable task execution*: MemoDroid leverages LLMs to emulate the way humans learn new mobile tasks. To ensure that the task is correctly executed and stored in memory, we employ several prompting techniques to improve LLMs' accuracy and enable users to correct the execution in case LLMs make mistakes. *ii) Efficient memory storage:* To facilitate the reuse of sub-tasks across different tasks, MemoDroid efficiently stores task information in a hierarchical memory structure, in which tasks are decomposed into multiple sub-tasks and each sub-task is further decomposed into a sequence of primitive actions. *iii) Flexible memory recall:* To ensure robust

[1]See https://mobilegpt-cps.github.io/ for the demo video.

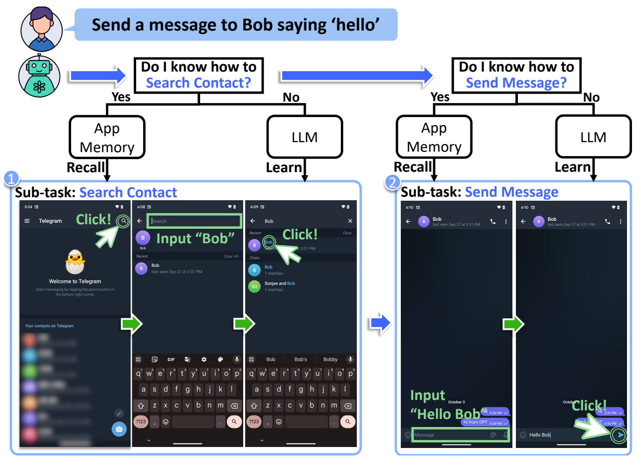


Figure 1: Illustration of MemoDroid.

and cost-effective task automation in dynamic mobile environments, MemoDroid flexibly adapts past executions to varying contexts and interface changes.

We have implemented a prototype of MemoDroid using off-the-shelf online LLM services (GPT-3.5 and GPT-4). Our experiments involved 50 unique mobile tasks across 5 widely-used mobile apps (i.e., Google Dialer, Telegram, Twitter, TripAdvisor, Gmail). The results show that MemoDroid successfully execute learned tasks with 100% accuracy, achieving a 69.22% reduction in task completion time and a 77.36% decrease in LLM query expenses. This significantly outperforms the baseline task automators without memory augmentation, while only experiencing an 8.05% rise in latency and a 0.5% increase in LLM query cost during the initial 'cold start' phase. Moreover, a usability study with 23 participants demonstrates that MemoDroid's human-in-the-loop memory repair system enables users to interact intuitively with the autonomous agent, allowing them to repair and collaboratively build upon the task automation process.

2 System Overview

Drawing inspiration from the way humans learn new tasks in mobile apps, we introduce MemoDroid—a pioneering task automation framework that harnesses the power of an LLM augmented with app memory (illustrated in Figure 1).

2.1 System Workflow

MemoDroid operates through offline and online stages to emulate humans' cognitive cycles of decomposing and learning new mobile tasks—Explore-Select-Derive process. MemoDroid achieves this by employing multiple LLM agents, which are software modules that leverage the LLM to perform a specific operation. This section gives a high-level overview of the system's workflow. Details of each agent, including their prompts are discussed in Section 3.

2.1.1 Offline Preparation Stage. First, MemoDroid undertakes multiple *explore* operations offline in advance to initialize the necessary data for task learning. To do this, MemoDroid endeavors to identify app screens of a target mobile app by employing two software tools—the active random explorer [60], which performs random UI actions to navigate through app screens, and the passive activity monitor [12], which tracks a user's app usage in the background. For each visitied screen, MemoDroid's *explore agent* queries LLM to generate a list of potential sub-tasks by examining displayed UI elements and records them in MemoDroid's app memory. App screens that have not been explored offline will be analyzed on demand during the online task-learning stage.

2.1.2 Online Task Execution Stage. This is divided into two cases: learning a new task and recalling previously learned tasks. When a user issues a voice instruction, MemoDroid's *task agent* first identifies the high-level task (i.e., objective) of the instruction. For instance, the task for the instruction *"send a message to Bob saying hello"* is *"send message"*. Upon recognizing the task, MemoDroid initiates the relevant mobile app to execute the task.

Task learning. If the given task does not align with any known tasks, MemoDroid engages in two key operations, *select* and *derive*, to autonomously discover and learn the sequence of sub-tasks required to accomplish the given task. Specifically, upon app launch, MemoDroid's *select agent* asks LLM to determine the necessary single sub-task from the list of potential sub-tasks for the initial app screen (called a select operation). Subsequently, *the derive agent* prompts LLM to identify a series of low-level actions to perform the chosen sub-task, such as clicking or scrolling (called a derive operation). These actions are dispatched to the mobile app to navigate to the subsequent app screen, where MemoDroid repeats select and derive operations again. This iterative process continues until the target task is completed, and the resulting sequence of sub-tasks and actions is recorded in the app memory for future use.

Task recall. When the task agent recognizes that user instruction is similar to a previously learned task, MemoDroid retrieves the required sub-tasks and their actions directly from memory. This recall process incurs no additional task-learning costs, enabling MemoDroid to accomplish the given task more quickly compared to dealing with a new task.

2.2 System Design

To enable the workflow above, MemoDroid 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. MemoDroid executes unknown tasks by leveraging multiple LLM agents 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 assist LLMs in better understanding the context of its execution, we provide them with a context-rich textual representation of the app screens that captures both the structured and semantic information of the app's interfaces. Furthermore, to address the inherent non-determinism and the unreliability of LLMs, MemoDroid employs a dual-strategy correction mechanism (more details in Section 3).

C2. In many cases, tasks are not entirely independent of each other; many tasks share common sub-tasks. For instance, 'Send a message to Bob' and 'Open chat with Bob' both involve the sub-task of locating Bob's contact information. 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. This causes unnecessary costs in terms of training budget and latency. To minimize such inefficiency, MemoDroid employs a three-level hierarchical memory structure: *tasks, sub-tasks, and actions*. This hierarchy enables MemoDroid to access memory at the sub-task level, facilitating the sharing of past execution experience across different tasks (more details in Section 4.1).

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 "Bob" in the search field, whereas "Send a message to John" requires entering "John." Therefore, instead of blindly replicating past actions, MemoDroid 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, MemoDroid 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 Section 5).

3 Accurate and Reliable Task Execution

Similar to humans, LLMs have been reported to solve complex tasks more accurately by breaking them down into smaller sub-tasks and carrying out each through separate reasoning steps [63, 65]. MemoDroid adopts this strategy by hierarchically decomposing tasks into sub-tasks and low-level actions according to the Explore-Select-Derive phases. This section outlines how MemoDroid integrates LLMs to effectively navigate 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 MemoDroid's memory is empty.

3.1 Prompting Mobile Screen to LLM

As LLMs are only capable of processing plain text data, the first step to using LLMs for mobile tasks is converting mobile screens to text. On Android, a UI can be converted to XML using Accessibility Service [12]. These XMLs include a hierarchical relationship between UI elements and various attributes (e.g., class_name, text, descriptions, clickable, etc.) that describe the functionality and appearance of the UI elements. However, these attributes often include superfluous details such as text color, shadow color, and IME option, which makes it difficult for an LLM to interpret the screen.

Therefore, MemoDroid first simplifies the UI representation by pruning non-essential UI attributes, focusing on those crucial for understanding the UI. To determine which attributes are essential, we thoroughly examined the Android documentation [11], identifying semantically vital UI attributes like text, description, and id, and interaction properties like clickable, checkable, editable, and scrollable attributes. However, not all UI elements are annotated with semantic attributes. Some of them may miss annotations despite their visual importance, such as icons and images. To capture the semantics not covered by UI attributes, MemoDroid leverages *pix2struct* [21], a state-of-the-art image caption generator, to generate a short description of each interactable UI element that lacks semantic attributes. We use it as an off-the-shelf tool, as it was the top-ranked UI captioning model on Hugging Face [10], a popular online platform for open-source machine learning models, at the time of this research.

Then, we further simplify the screen's layout by eliminating UI elements that are neither interactive nor carry significant semantic value (e.g., empty layout container). Next, drawing on the findings of a prior research [61] that shows LLMs comprehend Graphical User Interfaces (GUIs) more efficiently when presented in HTML, we tag each remaining UI element with appropriate HTML tags based on its specific attributes. For instance, the '<button>' tag is used for a clickable UI attribute, '<input>' for an editable attribute, '<scroll>' for a scrollable attribute, and so on. Lastly, we assign each UI element with a unique index number, which serves as a communication link between MemoDroid and LLMs to specifically refer to and interact with each UI element on the screen. This process not only ensures a cleaner, more relevant representation of app screens but also significantly reduces its number of tokens, the basic units of text that an LLM uses to process language, by an average of 85.2%. For the purposes of this paper, we will refer to this HTML representation of an app screen as the *"screen representation."*

3.2 Explore, Select, and Derive

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, the Explore agent prompts the LLM with the screen representation and asks it to enumerate operations (sub-tasks) that can be performed on the screen. Each sub-task generated by the LLM includes the following information: 1) sub-task name, 2) sub-task description, 3) index of its relevant UI elements, 4) parameter names, and 5) questions required to obtain each parameter. For example, possible sub-tasks for Telegram's initial app screen (Figure 1) include:

```
1. { name: "Search", description: "Search for a contact", parameters: {
        "contact_name": "who are you looking for?" }, UI_index: 3 }
2. { name: "Menu", description: "Open menu", parameters: {}, UI_index: 7 }
```

This structured function call format enables MemoDroid to reuse the sub-task for different contexts (parameters) once it has learned how to execute it.

To reduce the end-to-end latency of the online task execution, we conduct this explore phase *offline*, using random explorer [60] and user activity monitor [12]. These tools enable us to proactively collect a dataset of actionable sub-tasks for each screen within the app. In addition, to explore as many screens as possible during the random exploration, we record the visited app screens and clicked UIs to minimize revisiting the same screens. Although the current implementation of the MemoDroid performs the offline stage locally on the user device, it can be offloaded to a server for efficiency.

Select. The *select* phase selects a sub-task that will take us one step closer to accomplishing the given task. To do this, the select agent prompts the LLM with the user instruction (i.e., the original, verbal task description), a list of available sub-tasks identified during the *explore* phase, and the screen representation of the current screen. The LLM then picks the sub-task from the list and fills in its required parameters For instance, given the user instruction "Send a message to Bob," the output of the first select phase would be:

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

When parameter values are unknown, MemoDroid asks the user for the missing information, utilizing the questions formulated during the Explore phase. This process 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, MemoDroid 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, MemoDroid executes it directly from the memory, bypassing the Derive phase. Otherwise, we proceed to the Derive phase.

Derive. During the *derive* phase, the Derive agent incrementally selects and executes low-level actions to accomplish the sub-task selected in the *select* phase. Specifically, the Derive agent prompts the LLM with the sub-task to execute, a pre-defined list of low-level actions—click, input, scroll, long-click—, and the screen representation of the current screen. 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 3, the output of the first *derive* action for the sub-task *'Search'* would be ***click**(ui_index=3)*. The Derive agent repeats this process until there is a transition in the app screen, indicating that the sub-task has been finished. Afterward, MemoDroid returns to the Select phase to choose the next sub-task, and alternates between the Select and Derive phases until it fulfills the user's instruction.

3.3 Dual Strategy Failure Handling

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

Self-Correcting through feedback generation. Previous studies [14, 36] indicate that language models can self-correct when given appropriate feedback. To facilitate this in MemoDroid, we have identified two main types of errors commonly encountered in our experience with Large Language Models (LLMs) and developed a heuristic rule-based self-feedback generation module. Upon detecting an error, MemoDroid generates an appropriate feedback and appends it at the end of the next prompt to guide LLMs in refining its approach and self-correct the on-going 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 include: *"There is no change in the screen. Try another approach"* and *"You have looped the same screens X times. Consider trying another approach."* These errors can be detected by monitoring the screen changes and tracking the visited app screens.

Human-in-the-loop memory repair. MemoDroid enables users to rectify errors that are not addressed by its automated self-correction. Each agent that interacts with the LLM can potentially cause errors as follows. *i)* The Explore agent fails when sub-tasks are missing from the list of available sub-tasks.*ii)* The Select agent fails when the LLM chooses incorrect sub-tasks or inputs wrong parameters. *iii)* Derive failures happen when inaccurate actions are performed. MemoDroid provides specific repair mechanisms

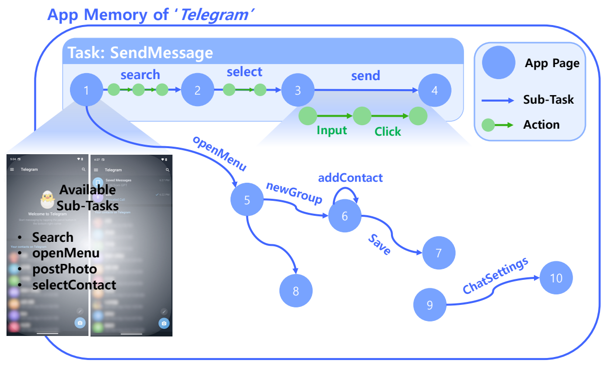


Figure 2: Transition graph for the app 'Telegram'

for each failure type[2]: *i)* The user can add or remove sub-tasks from the list of actionable sub-tasks for a given screen, *ii)* modify which sub-task to execute and its parameters, and *iii)* demonstrate a sequence of actions required to execute the sub-task. In addition, to aid users in detecting the failure point, MemoDroid provides a summary of task execution, where it visually represents the flow of the execution both in terms of sub-tasks and actions. Upon identifying the failure, users can navigate back to the failure point in the app, and fix errors using any one of the repair mechanisms.

To minimize user intervention, our human-in-the-loop memory repair is designed so that users only need to fix the specific sub-task where the error occurred. For instance, if the Derive agent makes a mistake while performing the sub-task *'search("Bob")'*, the user can pause the execution, return to the previous screen, and demonstrate how to perform the sub-task by clicking on the search button and inputting "Bob" in the search field. Then, when the user returns control to the MemoDroid, the user's demonstration is translated into natural language using the description of the given sub-task (*"User demonstrated how to: Search for a contact")* and delivered to the LLM. This helps the LLM understand the intent behind the user's intervention and refine their approach accordingly, enabling them to continue the execution with greater precision.

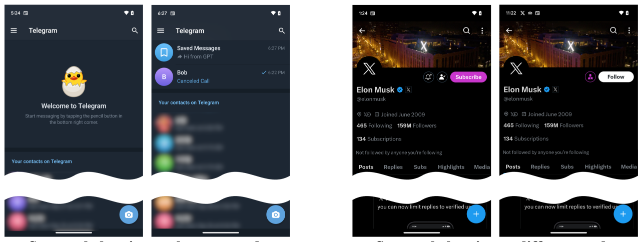
4 Hierarchical App Memory

MemoDroid's memory architecture is designed so that it can accumulate knowledge of the app as it executes mobile tasks. This resembles how humans get familiar with an app—the more a person uses the app, the more familiar they become with it. This section outlines how MemoDroid systematically archives the results from Explore, Select, and Derive phases in its memory and utilizes them for future task executions.

4.1 Memory Design

MemoDroid organizes its memory in the form of a transition graph that encapsulates the following key information: 1)

[2]see https://mobilegpt-cps.github.io/ for demo video.



Screens belonging to the same node
(a)

Screens belonging to different nodes
(b)

Figure 3: Examples of app pages: Screens in (a) are on the same app page because they provide same functionalities. Screens in (b) are on different app pages because they provide different functionalities (the left is for 'Subscribe' but the right is for 'Follow').

available sub-tasks for each app screen *(Explore)*, 2) sequence of sub-tasks involved in each task *(Select)*, and 3) how to perform each sub-task *(Derive)*. Figure 2 illustrates an example of this graph for the app 'Telegram'.

Node: App Page. Each node in the transition graph symbolizes an app page—a group of app screens that share the same functionalities (sub-tasks). Figure 3 illustrates examples of app pages. Each page node is represented by a list of sub-tasks that its app page provides. The rationale for defining nodes in this manner is twofold: First, it facilitates the sharing of sub-task execution information, as it ensures that screens within the same node are capable of performing the same set of sub-tasks. Second, it allows us to quickly check if a new app screen belongs to an existing node without going through the explore phase. As discussed in Section 3.2, a sub-task always corresponds to a set of UI elements, which means that if we identify the same set of UI elements, we can identify the corresponding sub-task as well. Thus, checking if a new screen belongs to an app page boils down to (i) getting the UI elements for each sub-task of an app page, (ii) checking if those UI elements exist on the new screen, and (iii) repeating it for all sub-tasks. This is a much more efficient and cost effective process than sending prompts to an LLM, i.e., the explore phase. As discussed in Section 4.2, we perform the explore phase only when a new app screen does not belong to any existing app page.

Edge: Sub-Task and Actions. Transitions between nodes are defined by sub-tasks. Each edge (sub-task) is composed of a sequence of primitive actions that detail the execution steps for the sub-task, as shown in Figure 2. This structure enables MemoDroid to efficiently repeat learned sub-tasks by following the previously recorded sequence of actions. Furthermore, these sub-task edges are not unique to individual screens but are shared across all screens within an app page.

4.2 Memory Utilization

Node Generation and Mapping. As discussed above, our graph architecture relies on accurately classifying app screens based on their functionality, since an app page (i.e., a node) is a group of app screens that share the same sub-tasks. Traditional screen classification methods [28, 68] are not suitable for this as they focus primarily on the screen's appearance, rather than its specific functionality. Therefore, MemoDroid introduces a simplified *subtask-based screen classification method* that opts to verify whether the screen can *perform* the sub-tasks of an existing node.

To implement this, we structure each node so that it includes information about the necessary UI elements for each sub-task. Upon encountering an app screen that does not fit into any existing nodes, MemoDroid undergoes the Explore phase to generate a list of actionable sub-tasks for the screen. Each of these sub-tasks is then generalized by recording key UI attributes, such as *ui_id*, *description*, and *class_name*, of its relevant UI elements. For instance, given the screen representation with a UI element:

```
<!-- Search Button-->
<button index=3 id="search_button" description="Search"/>
```

Each sub-task generated by the Explore agent is modified as follows and stored inside a new node:

```
{ name: "Search", ..., UI_index: 3 } →
{ name: "Search", ..., requires: {id:"search_button", description:"Search"} }
```

These UI attributes indicate the specifics of the UI elements required to perform the sub-task.

Then, when mapping a screen to a node, MemoDroid verifies if the current screen representation has UI elements that match the attributes required by each sub-task. If the screen representation has matching UIs for all of the node's sub-tasks, we categorize the screen under that node. This process can significantly reduce the number of Explore steps that would otherwise have to be performed for every screen we encounter, during both the offline and the online stages (see Section 7.1). If no matching node is found, the screen undergoes the Explore phase. However, before creating a new node, MemoDroid calculate 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 data without creating a new node. This double-check process minimizes the creation of redundant nodes, preventing already established sub-task edges from becoming obsolete.

During the online stage of the evaluation, where MemoDroid encountered and classified 157 app screens, this method showed only 1 false positive (misclassifying different app pages as the same node) and no false negative (failing to find the correct page node). The result is significantly better compared to existing SotA text-based [28] and vision-based screen classification methods [68], which had 17, and 10 for false positives, and 17, and 26 for false negatives, respectively.

Reusing learned sub-tasks. Once we obtain the list of actionable sub-tasks, either through node mapping or through the Explore phase, we proceed to the Select phase to pick a sub-task for execution. Afterward, prior to forwarding the selected sub-task to the Derive agent, MemoDroid checks whether the sub-task has already been learned by examining if the current node has a corresponding edge. If the edge exists, MemoDroid executes the sequence of actions recorded in the edge without undergoing the Derive phase. This allows MemoDroid to reuse learned sub-tasks across varying task contexts, akin to how humans apply past task execution experience to new tasks.

Learning new sub-tasks. A new edge is established as MemoDroid iterates through the Derive phase to execute a previously unknown sub-tasks. Specifically, each action executed during this phase gets recorded within its corresponding edge. To ensure that these actions are reproducible, a generalization process is required. This process will be discussed in the subsequent section (Section 5). Once the sub-task has been completed, MemoDroid establishes the edge between the starting node and the destination node and labels it with the corresponding sub-task name.

Storing Tasks. After MemoDroid successfully completes the user instruction, it stores the task as a collection of page node and sub-task pairs. This representation effectively indicates which sub-task needs to be executed on which page. This enables MemoDroid to 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, MemoDroid can find its way to execute the task.

5 Flexible Task Recall

MemoDroid aims to minimize the cost of the task execution by utilizing past execution information stored in its memory. However, the specific details of each action can vary based on the context of its execution. To address this, MemoDroid employs the rule-based adaptation method and LLM's few-shot learning capability [5] to reproduce actions involved in learned tasks and sub-tasks in a fast and consistent manner.

5.1 Rule-based Action Adaptation.

There are two cases where we need to modify the learned 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 input *"Alice"* into the search bar 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, MemoDroid generalizes and adapts actions to both the sub-task parameters and the screen contents.

Generalizing Actions. When learning a new sub-task, MemoDroid 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)*), MemoDroid first generalizes it against the current screen representation by replacing the action's target 'ui_index' with other key attributes (e.g., id, text, description). This enables MemoDroid to locate the action's target UI even if its hierarchical position changes. Then, to generalize it further against the task parameters, these attributes are compared against the current sub-task's parameter values. If a match is found, the attribute is updated with the name of the corresponding parameter. For example, given a screen representation:

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

the action '***click*** *(ui_index=5)*' gets generalized against the screen by replacing its target UI's index with other key attributes of the UI:

```
click(ui_index=5) → click(id:"contact", text: "Bob")
```

Then, we compare these attributes against the current sub-task, '***select****(contact_name: "Bob")*', and replace the attribute "Bob" with the parameter name "contact_name."

```
click(id:"contact", text: "Bob") → click(id:"contact", text:"[contact_name]")
```

This two-step generalization allows us to correctly locate which UI to click regardless of who and where the contact is.

MemoDroid generalize the actions against sub-task parameters because these parameters most precisely represent the generalization scope of the task. In contrast to a prior work [26], which uses a similar generalization technique but generalizes actions based on the words in the instruction, our method addresses certain limitations. Specifically, the previous approach is incapable of processing incomplete instructions like *"Send a message to Bob,"* which lack information about the message content, and implicit instructions like *"Call the first contact from the recent call,"* which don't explicitly mention the values of the contact name. These instructions pose challenges for the previous approach, as it fails to find parameters and their values from the instruction. On the other hand, MemoDroid effectively addresses this challenge by deriving and populating parameters *incrementally* at the sub-task level. Given these instructions, MemoDroid can successfully generalize the actions against the sub-tasks: ***send****(message_content)* and ***call****(contact_name)*. This allows

MemoDroid to not only identify hidden parameters but also update their values in real-time, enabling it to generalize and adapt learned tasks to various contexts.

Adapting Actions. When repeating a learned task, MemoDroid bypasses all Explore, Select, and Derive phases, as the sequence of sub-tasks for the task is already known. However, we still need to fill in the parameters for each sub-task. To address this, MemoDroid utilizes a new agent called the Parameter Filler to populate the sub-task parameters before executing each sub-task. The Parameter Filler agent queries LLM to determine the parameters based on the user instruction and screen representation provided. Similar to the Select agent, if parameter values are missing from the instruction, MemoDroid asks the user for more information.

Once the sub-task is filled in, adapting its actions to a new context is straightforward. Given its parameter values, and a screen representation, MemoDroid first maps any parameter names recorded in the action's attributes to the corresponding parameter values. Next, it locates the UI element in the current screen that aligns with these updated attributes and executes the action on this identified UI. For example, assuming the screen remained unchanged and the sub-task is *'**select**(contact_name: "Alice")'*, MemoDroid performs the click action on the UI element with index 6, which is identified by the attributes (id:"contact", text="Alice").

5.2 Few-Shot Learning Adaptation

The rule-based approach 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, there could be multiple UIs that meet the specified attributes. Third, UIs may change unexpectedly with app updates.

If the rule-based adaptation fails, we resort to re-deriving the action using the LLM. However, this time, we leverage the few-shot learning capability of the LLM [5, 65] to produce more accurate and reliable responses. When querying the LLM, we present it with an example showcasing how the desired 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. Notably, the output in the example could either be an action originally generated by the Derive agent or one that has been modified by the user through the correction mechanism. In any case, the provided example always demonstrates a successful result. By providing such examples, we effectively guide the LLM to produce consistent responses based on its memory. Moreover, if the example action is one that has been corrected by the user, it helps LLM to not repeat the same mistakes it has committed previously.

6 Implementation

LLM Agents. In MemoDroid, each agent is equipped with a language model tailored to its specific operational requirements. For Explore, Select, and Derive agents, which involve mobile screen understanding and multi-step reasoning, we used the most capable GPT-4 language model. For the Task agent and Parameter Filler agent whose roles are less demanding, we assigned a faster and more economical GPT-3.5 Turbo model. In our evaluation, GPT-3.5-based agents achieved a 100% accuracy rate for their designated functions.

App Launch. In mobile tasks, it's important to start with the correct app. MemoDroid assists in this by recommending the most appropriate apps. 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 [47] using the text-embedding model [42]. Then, when a user gives an instruction, MemoDroid retrieves and presents the top three most relevant apps based on their descriptions. After the user makes a selection, MemoDroid launches the app and loads its corresponding app memory to proceed with the task.

Merging similar sub-tasks. MemoDroid terminates the sub-task and selects a new sub-task every time the app page changes. However, this could cause an over-splitting of tasks, undermining the intent of using sub-tasks as an abstraction of actions. Therefore, to curb excessive fragmentation, after completing the user instruction, each sub-task description is converted to a fixed-dimension embedding. Then, we merge consecutive sub-tasks with high cosine similarity into a single edge before storing them in the graph.

Reproducing only the effective actions. In task recall, not every action needs to be repeated in the exact same order as originally performed. The need for the action 'scroll' depends on the specific task parameters or the contents displayed on the screen. For example, while selecting 'Zane' from a contact list might require multiple scrolls, choosing 'Alex' might not require any. For this reason, MemoDroid doesn't blindly reproduce scroll actions. Instead, it checks whether its subsequent action can be executed on the current screen. If it can, the system skips all the scrolls in between and directly performs the subsequent action. Only if it can't, the system reproduces the stored scroll actions. When more scrolls are needed than what is stored, MemoDroid transitions to the few-shot learning adaptation, where the Derive agent can perform additional scrolls. This adaptive approach ensures efficient task execution while maintaining accuracy.

7 Evaluation

7.1 Experimental Setup

Across the evaluation, we used a Google Pixel 6 smartphone.

Dataset. Our evaluation involved five popular off-the-shelf

App Name (avg # of actions)	Tasks (# of actions with MemoDroid)									
Google Dialer (4.5)	**1.** Add To Favorites	(6)	**2.** Call Contact	(4)	**3.** Remove From Favorites	(3)	**4.** Open Contact Info	(4)	**5.** Get Call History	(2)
	6. Call Recent Contact	(2)	**7.** Create Contact	(8)	**8.** Modify Contact Number	(6)	**9.** Delete Contact	(6)	**10.** Manage Favorites	(4)
Telegram (5)	**1.** Call Contact	(5)	**2.** Clear History	(6)	**3.** Create Group	(7)	**4.** Delete Chat	(6)	**5.** Find People Nearby	(3)
	6. Open Chat	(3)	**7.** Read Last Message	(4)	**8.** Read Saved Message	(3)	**9.** Send Message	(7)	**10.** Video Call Contact	(6)
Twitter (4.7)	**1.** Follow User	(6)	**2.** Post Photo	(5)	**3.** Post Tweet	(5)	**4.** Post Reply	(8)	**5.** Quote Reply	(5)
	6. Read New Tweet	(6)	**7.** Read Last Notification	(2)	**8.** Read Replies	(2)	**9.** Get Trending Tweets	(2)	**10.** Unfollow User	(6)
TripAdvisor (5.5)	**1.** Find Restaurants	(6)	**2.** Find Hotels	(8)	**3.** Find Day Trip Package	(5)	**4.** Find and Save Hotels	(12)	**5.** Find Hotels Nearby	(3)
	6. Find Restaurants Nearby	(3)	**7.** Find Restaurants Reviews	(8)	**8.** Find Hotels Rating	(5)	**9.** View Cart Items	(2)	**10.** View Trip Plan	(3)
Gmail (4.8)	**1.** Archive Email	(2)	**2.** Change Email Settings	(10)	**3.** Delete Email	(2)	**4.** Mark Email Important	(3)	**5.** Read Last Email	(2)
	6. Recover Deleted Email	(6)	**7.** Reply To Email	(6)	**8.** Schedule Email	(9)	**9.** Send Email	(6)	**10.** Get Promotional Emails	(2)

Table 1: Mobile Applications and tasks used in the evaluation.

mobile apps across various categories: Google Dialer, Telegram, Twitter, TripAdvisor, and Gmail. We designed 10 high-level tasks for each app (as detailed in Table 1), and for every task, we developed two sets of user instructions with varying task parameters. For example, in Twitter, the task 4 *'Post Reply'* splits to instructions: *"Post reply to Elon Musk's new tweet"* and *"Post reply to Bill Gate's new tweet saying 'Reply from MemoDroid'*. In total, our dataset consisted of 100 user instructions: two for each task and 20 for each app.

We created our own dataset of user instructions because existing datasets (e.g., MoTiF [6] and PixelHelp [30]) tend to be overly formalized and consist of low-level commands (e.g.,*"open device's settings app and then tap network & internet"*), whereas we aim to evaluate on more natural, conversational prompts. Our dataset includes both simple (e.g., *"Get Call History")* and complex instructions (e.g., *"Find me a five-star hotel in Las Vegas from October 15 to October 20 and save it"*) that are more likely to be used in real life.

Baseline. For comparison, we developed a baseline task automator that has no memory augmentation. This baseline system derives primitive actions directly from the user instruction without breaking them down into tasks and sub-tasks, akin to other existing LLM-based mobile task automators [61, 66, 67]. To ensure a balanced comparison, both systems used the same prompting structures and techniques (e.g., screen representation and self-feedback generation).

Procedure. The evaluation is divided into two stages. The first stage (*cold-start*), executes the first set of instructions for each task. In the second stage (*warm-start*), we execute the second set of instructions. For MemoDroid, we start the evaluation with an empty memory. This procedure is designed to evaluate how well MemoDroid learns the task from the initial set of instructions and reuses them for the subsequent instruction set. For both systems, if an error occurs during the execution (e.g., deviates from the expected path), we repair it so that the system can continue with the execution.

Offline Preparation. Before beginning the tasks, we employed a random explorer to identify the app pages of each target app. For each app, we conducted random exploration until it discovered 50 unique app pages, which typically took between 10 to 15 minutes. These app pages accounted for 91.5% (76 out of 83) of the app pages required by our task. The remaining app pages, along with those unsuitable for random exploration (such as those involving payments or sensitive information), can be explored offline by monitoring user interactions with the app. The total cost for this random exploration was $21.04. Given that this preparation is a one-time process, the expense is deemed reasonable.

7.2 Efficiency

Figure 4 plots the latency and cost[3] for cold start and warm start executions. The baseline system shows different results between cold and warm starts because their number of steps is different due to the LLMs' non-determinism. We will further discuss this phenomenon in the accuracy evaluation.

Cold Start. For the initial *cold start*—i.e., the first trial of the task—, despite the additional Select phase that MemoDroid employs to decompose the tasks into sub-tasks, latency and cost only increases by an average of 8% and 0.5%, respectively, compared to the baseline. Moreover, MemoDroid even outperforms the baseline for the `Telegram`. This is attributed to our design choice that enables learned sub-tasks to be reusable across different tasks.

Specifically, MemoDroid can replace a portion of its action derivations with previously learned sub-tasks. Table 2 shows the average memory hit rate for each app. The memory hit rate represents the percentage of actions retrieved directly from the memory without involving the LLM. Since

[3]At the time of evaluation, the cost of using LLM is as follows: for GPT-4, $0.03 / 1K tokens and for GPT-3.5 Turbo, $0.003 / 1K tokens

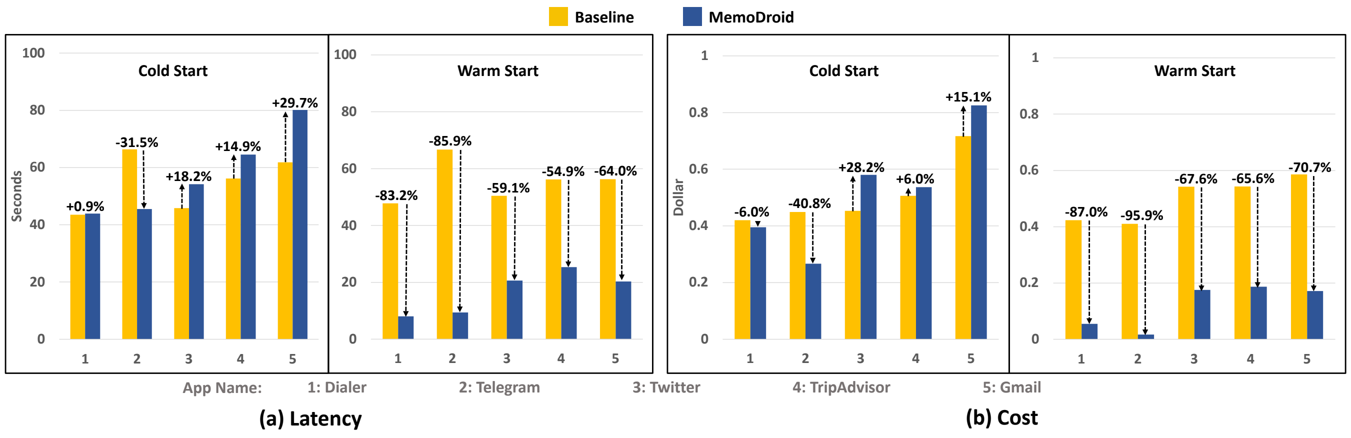


Figure 4: Average Task Latency and Cost.

LLM is the primary contributor to the latency, a higher hit rate translates to less time spent learning and executing the task. `Telegram` well exemplifies this; since many of its tasks involve a common sub-task of searching a specific contact (50% memory hit-rate), MemoDroid outperforms the baseline by 30%, even when executing new tasks. Conversely, `Gmail` exhibits low memory hit-rate (6.7%) as its tasks have few overlapping sub-tasks. While our current results indicate that `Gmail` does not fully benefit from our design, theoretically for any application, the latency and cost associated with MemoDroid learning new tasks can decrease exponentially as it accumulates more sub-tasks.

Warm Start. In the case of *warm start*, where we repeat the same task, but with different user instructions, MemoDroid significantly reduces the latency and cost, achieving 68.22% and 77.36% improvements over the baseline, respectively. This high level of efficiency is largely due to MemoDroid having already learned the necessary sub-tasks and actions during the initial cold-start stage. The effectiveness of MemoDroid can vary depending on the memory hit-rate. The memory hit-rate for *warm start* does not always reach 100% because some actions require LLM for its adaptation (i.e. when rule-based adaptation fails). Nevertheless, even in the worst-case scenario where no actions are adaptable (e.g., Twitter Task 8), our evaluation results suggest that MemoDroid maintains comparable performance to the baseline.

Given the recurrent nature of mobile tasks [16, 20, 34, 58], it is reasonable to assume that the cold-start is less common compared to the warm-starts. Therefore our approach, which substantially reduces the cost of these more common warm starts, at the expense of a relatively small increase in the cold-start cost, is highly practical in real-world scenarios.

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%

Table 2: MemoDroid Average Memory Hit Rate per App.

7.3 Accuracy

We evaluated the accuracy of our system as follows. During the task execution, if the system (both MemoDroid and baseline) deviates from the expected path, we gave the system a chance to refine itself by going through three more rounds of action derivation. If it did not return to the correct path, we manually corrected it using the human-in-the-loop repair before giving the system control back. Note that the repair process for MemoDroid can involve any one of the Explore, Select, and Derive phases, while the baseline only requires repairs in the Derive phase, the only phase it employs.

Task Completion Accuracy. Table 3 illustrates the task completion rate for MemoDroid and the baseline. A task is considered complete if the system has completed the task without any user intervention. Most notably, MemoDroid achieves 100% accuracy for the warm start, demonstrating its ability to adapt and reproduce learned tasks precisely and consistently, even when task parameters and the resulting screen content changes. Furthermore, our human-in-the-loop repair mechanism ensures the reliability of the memory even if LLMs had made several errors during the cold start. We verified that our few-shot learning adaptation successfully guides LLM in generating correct answers, even when it failed to do so in the cold start. For example, in the Gmail's *Recover Deleted Email* task, MemoDroid initially failed to open

App Name	Task Completion Accuracy			
	MemoDroid		Baseline	
	Cold	**Warm**	**Cold**	**Warm**
Google Dialer	90%	100%	90%	90%
Telegram	90%	100%	80%	90%
Twitter	100%	100%	80%	90%
TripAdvisor	70%	100%	60%	60%
Gmail	80%	100%	70%	70%

Table 3: Task Completion Accuracy.

App Name	Average Step Accuracy (%)				
	MemoDroid (Cold)			Baseline	
	Explore	**Select**	**Drive**	**Cold**	**Warm**
Google Dialer	97.1	100	100	96.0	98.0
Telegram	100	97.2	100	97.0	98.4
Twitter	100	100	100	94.0	97.9
TripAdvisor	97.8	95.6	100	92.6	88.1
Gmail	100	97.9	97.7	94.9	90.9

Table 4: Average Step Accuracy.

the More Options Menu because there were two seemingly identical More Options buttons on the screen. However, during the warm start, the Derive agent successfully derived an action to click the correct button when given an example that conveys which button to click among the two.

This advantage of MemoDroid extends beyond individual tasks and applies across multiple tasks. If an LLM makes a mistake during a sub-task and gets its correction, MemoDroid does not repeat the same mistake in the future tasks with overlapping sub-task. For instance, finding a contact in Telegram requires searching rather than scrolling through the contact list. In its first attempt to search for the contact (i.e., Task *'Call contact'*), MemoDroid initially tries to scroll. However, after correcting itself through self-feedback generation, it learns to directly search in subsequent tasks, bypassing the scrolling steps. This is similar to how humans learn from past mistakes.

On the other hand, the baseline system almost always tries to scroll through the contacts, repeating the same mistakes throughout the tasks. Furthermore, we have observed that the baseline occasionally fails or operates inefficiently during the warm start stage, even if it has successfully completed the task in the cold start (e.g., Gmail Task 8 and TripAdvisor Task 3). This inconsistency underscores the non-deterministic nature of the LLMs, which can produce varying outputs depending on factors including task parameters, screen representation, and even the timing of the query.

We have also observed that when a user corrects an LLM's mistakes and then returns control to the baseline system, the system often repeats the same incorrect response or becomes lost, failing to recognize the correction (e.g., Gmail Task 2, TripAdvisor Task 4). We hypothesize that this occurs because LLMs select and refine their approach based on the context of past events. For the baseline, since user interventions are represented only via low-level actions (click UI 5, click UI 4), it is difficult to convey the intention behind each correction.

Step Accuracy. We further analyze the accuracy of the LLM queries involved in each step of the systems. Table 4 presents the average step accuracy for MemoDroid and the baseline. In MemoDroid, each step is divided into three phases: *Explore*, where it identifies a list of actionable sub-tasks for the given screen; *Select*, in which it selects one of the sub-tasks that aligns most closely with the given user instruction; and Derive, in which it derives a primitive action to carry out the chosen sub-task. The baseline system only includes data for the Derive phase, as it directly derives the actions from the instruction, without breaking it down into sub-tasks. Note that the table omits data for MemoDroid's warm start because all its involved LLM queries had 100% accuracy, as shown in the Table 3.

Although MemoDroid processes a significantly higher number of LLM queries (Explore: 203, Select: 203, Derive: 217) compared to the baseline system (Derive: 280), it not only demonstrates higher accuracy but also incurs fewer failures (MemoDroid: 7, baseline: 14). This improved performance is attributed to MemoDroid's ability to distribute the reasoning required for each step across multiple phases, similar to how the Chain-of-thought prompting [65] improves LLM accuracy by allowing LLMs to solve a problem as a series of intermediate steps before giving a final answer. For instance, the action 'ask'—requesting additional information from the user—involves complex reasoning steps of identifying the necessary parameter and recognizing that the parameter is empty. In the baseline system, such reasoning steps are confined to a single Derive query, resulting in a low accuracy of 33% for the 'ask' action. For example, when given an incomplete instruction like *"Send a message to Bob"*, the baseline system tends to invent the message content and send "Hi, Bob" arbitrarily. Conversely, MemoDroid splits these reasoning steps between the Explore and Select phases: identifying the parameters in *Explore* and filling them in during *Select*. This distribution of reasoning loads enables MemoDroid to perform 'ask' action with 100% accuracy.

7.4 User Study

Participants and Study Procedure. We recruited 23 participants (16 male, 7 female, mean age 23.2, stdev=3.5, max=32, min=19) through an online community posting. Participants received a compensation of 1.2 USD. The first session evaluated the overall usability of MemoDroid compared to the Samsung Bixby Macro (Programming by Demonstration) and a baseline system (an LLM-based task automator without memory). The second session assessed the usability of MemoDroid's human-in-the-loop memory repair mechanisms. Each session lasted 30 minutes.

In the first session, participants were asked to perform the following three instructions using the three aforementioned

	Overall Usability (7-point-scale)		
Scenario	**PbD**	**Baseline**	**MemoDroid**
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 5: Usability score for different task scenarios.

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 instructions were designed to evaluate the usability of automation tools in the three scenarios: *i)* executing a new task, *ii)* repeating the same task but with different parameters, and *iii)* executing a new albeit similar task. After each instruction, participants rated each tool's usability on a 7-point Likert scale. After completing all instructions, participants ranked the tools based on their preferences and indicated whether they were willing to use the tool in the real world.

In the second session, participants learned how to use MemoDroid's repair mechanisms to correct the LLM's mistakes. After the tutorial, they executed the instruction *"Modify Tom's phone number to 123-456-789"* using the system preset to make mistakes during the execution. These mistakes covered all three areas of repair—Explore, Select, Derive. Participants had no prior information about the specific mistake the system would make. Following the task, they assessed 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 5 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 and task, and the baseline as convenient but too slow. In contrast, MemoDroid initially had a usability score similar to the baseline in the first scenario but showed significant improvement in the subsequent scenarios—*"(MemoDroid) 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 MemoDroid'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 MemoDroid, and all participants expressed willingness to use MemoDroid in real-world applications, compared to 35% and 30% for the baseline and PbD tools, respectively.

Session 2: Human-in-the-loop Memory Repair. MemoDroid's memory repair mechanism achieved 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 regarded as acceptable considering the high complexity of the given repair mission (repairing all three phases within a single task) and the fact that many participants were unfamiliar with the concept of autonomous agents making mistakes.

In the post-survey, participants highly rated both the necessity and effectiveness of the repair mechanism, with scores of 4.7 and 4.8 out of 5, respectively. Moreover, their acceptable accuracy threshold for an automator *without* a repair mechanism is 96% (average), whereas with repair mechanism, it drops to 84%, which is lower than MemoDroid's average task completion rate (86%). This suggests that participants are more lenient with accuracy expectations when a repair mechanism is present, recognizing its value in task automation.

8 Related Work

Other approaches in UI Task automation. There are various approaches to develop a UI-based task automator. The traditional Programming by Demonstration approach [25–27, 53, 69] enables users to create 'Action Macros' themselves. Yet they suffer from a lack of scalability as it requires extensive user demonstrations for each new task. Vision-based approaches [6, 54, 59] follow user instructions by analyzing the pixel-level representation of the GUIs. Extraction-based approaches [18, 30, 31, 60] automatically generate macro scripts by labeling the action sequences. While these approaches sidestep the need for explicit user demonstrations, they rely on extensive human-annotated datasets and lack the ability to perform untrained tasks on demand.

Recently, the use of LLMs has gained attention [40, 50, 61, 66, 67], capitalizing on their ability to understand and execute tasks without prior training or user demonstrations. Most notably, AutoDroid [66] augments LLMs with app-specific domain knowledge using automatically generated UI transition memory. This memory enables LLMs to better understand specific apps and perform their tasks with greater accuracy. However, the practicality of LLMs in real-world task automation remains uncertain due to their inherent unreliability and high costs. MemoDroid addresses these limitations by augmenting LLMs with app memory that it can use to store and recall successful task executions.

LLM-based Autonomous Agent in other fields. LLMs demonstrate significant potential in automating human tasks. Specifically, it can iteratively derive the most appropriate actions for specific tasks when provided with a proper action space—a set of possible actions that can be performed by the agents (e.g., device interaction, API calls). Consequently, researchers across various fields have attempted to expand the capabilities of LLMs by integrating UI interactions [40, 50, 66, 67], programming tools [51, 55], APIs [24, 32, 46, 49, 52, 57], and games [45, 62, 71]. MemoDroid is unique in its ability to generate a higher-level action space (i.e., sub-tasks) on its own and utilize them for different goals (i.e., user instruction). This approach can be extended to other fields, as many digital tasks share common and recurring sub-tasks.

9 Discussion

Security & Privacy. The screen representation could include personal information such as name and phone number. This could pose a privacy risk when sent to the LLM. To address this, we could use the Personal Identifier Information (PII) Scanner [38, 48] to identify personal information from the prompt and mask it with non-private placeholders.

In addition, certain actions may pose risks if performed without supervision. For instance, agreeing to terms of service or confirming a payment, should be subject to user approval. To address this concern, we add a *'get user confirm'* action to our derive agent. This ensures that we identify any risky action and request the user's confirmation.

Sharing App Memory. MemoDroid currently stores memory on a local basis, meaning each device has its own version of app memory. To enhance this, the memory can be shared or crowd-sourced to create a large-scale app memory. This approach can effectively eliminate the need for each user to learn tasks individually, making the cold-start learning stage even less common. To accommodate the varying devices, app memory can be categorized based on the device form factor, as the app interface ㅍ varies depending on the form factor (e.g., smartphone or tablet device). The diversity of the device resolution is not a concern, as MemoDroid identifies UI elements using their hierarchical structure (i.e., index), not their pixel coordinates.

Cross App Task execution. Users may give instructions that involve multiple apps. For example, the instruction "Check my bank account, and if it has over 10 dollars, order me a pizza" requires interaction with both a banking app and a food delivery app. To accommodate such scenarios, MemoDroid can be extended to maintain a global dataset of known tasks across apps. MemoDroid can then use this to plan out which task to execute in which order to fulfill the user instruction.

Unsupported Apps. Our current implementation does not support mobile apps with third-party UI engines (e.g., Flutter, React Native, etc). This is due to its reliance on the Android Accessibility Service, which can only extract screen representation from native Android UIs. To overcome this, we can use screen-to-text translation models [21, 23, 68, 70] or vision-enhanced large language models [33, 43, 64]. These approaches would enable MemoDroid to process app screenshots along with the text screen representation.

10 Conclusion

We have presented MemoDroid, a novel LLM-based mobile task automator that significantly enhances the efficiency and reliability of task automation on mobile devices. By innovatively emulating human cognitive processes in learning new tasks and integrating a unique hierarchical app memory, MemoDroid effectively addresses the challenges associated with using LLMs in the mobile task automation. We expect MemoDroid to drive innovation and efficiency in numerous applications, further solidifying the role of intelligent automation in everyday technology use.

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] Andrea Burns, Deniz Arsan, Sanjna Agrawal, Ranjitha Kumar, Kate Saenko, and Bryan A Plummer. 2022. A dataset for interactive vision-language navigation with unknown command feasibility. In *European Conference on Computer Vision*. Springer, 312–328.

[7] 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.

[8] 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).

[9] 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

[10] Hugging Face. 2023. *Hugginf Face - The AI community building future.* Hugginf Face. Retrieved Nov 11, 2023 from https://huggingface.co/

[11] Google. 2023. *AccessibilityNodeInfo.* Google. Retrieved Nov 11, 2023 from https://developer.android.com/reference/android/view/accessibility/AccessibilityNodeInfo

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

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

[14] 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).

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

[16] 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.

[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] Minghao Li, Feifan Song, Bowen Yu, Haiyang Yu, Zhoujun Li, Fei Huang, and Yongbin Li. 2023. Api-bank: A benchmark for tool-augmented llms. *arXiv preprint arXiv:2304.08244* (2023).

[25] 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).

[26] 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

[27] 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.

[28] 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.

[29] 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.

[30] 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).

[31] 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.

[32] Yaobo Liang, Chenfei Wu, Ting Song, Wenshan Wu, Yan Xia, Yu Liu, Yang Ou, Shuai Lu, Lei Ji, Shaoguang Mao, et al. 2023. Taskmatrix. ai: Completing tasks by connecting foundation models with millions of apis. *arXiv preprint arXiv:2303.16434* (2023).

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

[34] 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.

[35] 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.

[36] 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).

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

[38] 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

[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] 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. *arXiv preprint arXiv:2304.03442* (2023).

[46] Shishir G Patil, Tianjun Zhang, Xin Wang, and Joseph E Gonzalez. 2023. Gorilla: Large language model connected with massive apis. *arXiv preprint arXiv:2305.15334* (2023).

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

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

[49] Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, et al. 2023. Toolllm: Facilitating large language models to master 16000+ real-world apis. *arXiv preprint arXiv:2307.16789* (2023).

[50] 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).

[51] Jingqing Ruan, Yihong Chen, Bin Zhang, Zhiwei Xu, Tianpeng Bao, Guoqing Du, Shiwei Shi, Hangyu Mao, Xingyu Zeng, and Rui Zhao.

2023. Tptu: Task planning and tool usage of large language model-based ai agents. *arXiv preprint arXiv:2308.03427* (2023).

[52] Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. 2023. Toolformer: Language models can teach themselves to use tools. *arXiv preprint arXiv:2302.04761* (2023).

[53] Alborz Rezazadeh Sereshkeh, Gary Leung, Krish Perumal, Caleb Phillips, Minfan Zhang, Afsaneh Fazly, and Iqbal Mohomed. 2020. VASTA: a vision and language-assisted smartphone task automation system. In *Proceedings of the 25th international conference on intelligent user interfaces*. 22–32.

[54] Peter Shaw, Mandar Joshi, James Cohan, Jonathan Berant, Panupong Pasupat, Hexiang Hu, Urvashi Khandelwal, Kenton Lee, and Kristina Toutanova. 2023. From Pixels to UI Actions: Learning to Follow Instructions via Graphical User Interfaces. *arXiv preprint arXiv:2306.00245* (2023).

[55] Yongliang Shen, Kaitao Song, Xu Tan, Dongsheng Li, Weiming Lu, and Yueting Zhuang. 2023. Hugginggpt: Solving ai tasks with chatgpt and its friends in huggingface. *arXiv preprint arXiv:2303.17580* (2023).

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

[57] Yifan Song, Weimin Xiong, Dawei Zhu, Cheng Li, Ke Wang, Ye Tian, and Sujian Li. [n.d.]. Restgpt: Connecting large language models with real-world applications via restful apis. CoRR, abs/2306.06624, 2023. doi: 10.48550. *arXiv preprint arXiv.2306.06624* ([n. d.]).

[58] 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.

[59] 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).

[60] 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).

[61] 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.

[62] 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).

[63] 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).

[64] 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).

[65] 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.

[66] 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).

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

[68] 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.

[69] Jackie Yang, Monica S Lam, and James A Landay. 2020. Dothishere: multimodal interaction to improve cross-application tasks on mobile devices. In *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology*. 35–44.

[70] 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.

[71] Xizhou Zhu, Yuntao Chen, Hao Tian, Chenxin Tao, Weijie Su, Chenyu Yang, Gao Huang, Bin Li, Lewei Lu, Xiaogang Wang, et al. 2023. Ghost in the Minecraft: Generally Capable Agents for Open-World Enviroments via Large Language Models with Text-based Knowledge and Memory. *arXiv preprint arXiv:2305.17144* (2023).