AutoDroid-V2: Boosting SLM-based GUI Agents via Code Generation

Hao Wen[1], Shizuo Tian[1], Borislav Pavlov[1], Wenjie Du[1,*], Yixuan Li[1,*], Ge Chang[1,*], Shanhui Zhao[1,*], Jiacheng Liu[1,*], Yunxin Liu[1,2], Ya-Qin Zhang[1], Yuanchun Li[1,2,3,†]

[1] Institute for AI Industry Research (AIR), Tsinghua University
[2] Shanghai Artificial Intelligence Laboratory
[3] Beijing Academy of Artificial Intelligence (BAAI)

Abstract

Large language models (LLMs) have brought exciting new advances to mobile UI agents, a long-standing research field that aims to complete arbitrary natural language tasks through mobile UI interactions. However, existing UI agents usually demand powerful large language models that are difficult to be deployed locally on end-users' devices, raising huge concerns about user privacy and centralized serving cost. Inspired by the remarkable coding abilities of recent small language models (SLMs), we propose to convert the UI task automation problem to a code generation problem, which can be effectively solved by an on-device SLM and efficiently executed with an on-device code interpreter. Unlike normal coding tasks that can be extensively pre-trained with public datasets, generating UI automation code is challenging due to the diversity, complexity, and variability of target apps. Therefore, we adopt a document-centered approach that automatically builds fine-grained API documentation for each app and generates diverse task samples based on this documentation. By guiding the agent with the synthetic documents and task samples, it learns to generate precise and efficient scripts to complete unseen tasks. Based on detailed comparisons with state-of-the-art mobile UI agents, our approach effectively improves the mobile task automation with significantly higher success rates and lower latency/token consumption. Code is open-sourced at https://github.com/MobileLLM/AutoDroid-V2.

CCS Concepts

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

Keywords

Mobile UI Agent, Small Language Models, Mobile Task Automation, Script Generation

ACM Reference Format:

† Corresponding author: Yuanchun Li (liyuanchun@air.tsinghua.edu.cn).
* Work done during internships at AIR, Tsinghua University.

1 Introduction

Automating the control of mobile devices through natural language has long been a goal for mobile developers and researchers [1, 5, 22]. Agents powered by Large Language Models (LLM) and Vision Language Models (VLM, both LLM and VLM are refered as LLM for simplicity in this paper) have emerged as promising solutions for automating flexible and complex user tasks [20, 24, 34, 36, 41–43]. These agents function by interpreting user instructions, analyzing the graphical user interface (GUI) states, and predicting UI actions (e.g., touch, scroll, input text, etc.) to be performed in the target apps iteratively. Most of these LLM-based GUI agents operate in a step-by-step manner, *i.e.* querying the LLM at each GUI state for dynamic decision and reflection. They typically include the user's task and the observed GUI state (e.g., a screenshot or a text-based view hierarchy) in the LLM prompt, requesting the model to generate the required UI action. This iterative process continues until the task is completed. For this reason, we refer to these conventional designs as **"Step-wise GUI agents"** in this paper.

While step-wise GUI agents hold great potential, they mostly rely on powerful reasoning and reflection abilities, which are dominated by large cloud-based models (*e.g.* GPT-4o, Claude, etc.). Due to the high computational cost and/or closed-source nature of these models, they can hardly be deployed to mobile devices. This situation can lead to significant privacy and security risks of sharing each personal GUI page to the cloud [26, 47, 48], high user-side traffic consumption, and high server-side centralized serving costs, hindering large-scale deployment of GUI agents.

Among the possible countermeasures to this problem, **small language models (SLMs)** (There is no clear boundary between LLM and SLM. In this paper, we use SLM to represent the LLMs that can be deployed on the device.) and **efficient inference systems** have emerged as promising directions to enable on-device deployment of more powerful models [25, 32, 39]. The current capabilities of these SLMs are still far from their cloud-based counterparts, and the capability gap is anticipated to remain in the near future due to the scaling laws [19]. **Customizing a domain-specific SLM** can also be a potential solution. However, training a domain-specific model requires a large-scale, high-quality labeled dataset covering diverse usage scenarios, which are difficult to obtain in the UI task automation domain. Most existing mobile task automation datasets [6, 27, 30, 46]
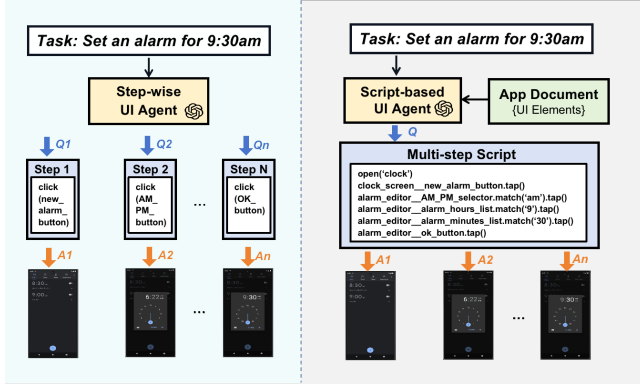
Figure 1: Comparison of conventional step-wise GUI agent and our script-based GUI agent. The step-wise GUI agent includes the user's task and the current GUI state in the LLM prompt ($Q1, \ldots Qn$), asking the model to generate a GUI action for each step ($A1, \ldots An$). The script-based agent inputs the task and app document (Q), prompting the LLM to generate a multi-step script for execution.

only have hundreds of human demonstrations. The largest dataset AitW [28] has 715k crowd-sourced task demonstrations, while the quality is low for most demonstrations (we compare the agents trained with such datasets in Section 4.2). Meanwhile, the LLM abilities used in existing GUI agents are diverse, including not only simple GUI grounding abilities that can be offered by a small tailored model [7, 12, 15], but also general reasoning and reflection abilities that can hardly be distilled to a much smaller model [4, 33, 43].

Here, we argue that the widely adopted step-wise mechanism of GUI agents may need to be changed in order to accommodate the requirements of practical on-device agents. First, step-wise decision-making requires high and frequent reasoning and reflection about the current state and progress, which are essentially challenging for SLMs. Second, even if the step-wise decisions can be made with reasonable accuracy, calling the model at each step is very compute-intensive, making it difficult to provide a smooth user experience with limited on-device computational resources.

Fortunately, we observe that **SLMs can be trained to have a unique but important strength in coding**, *i.e.* producing executable code based on user instructions [13, 29, 52]. This strength is largely attributed to extensive training on code datasets [9, 17], as it is relatively easier to collect large-scale, high-quality source code data compared to many other types of data. Remarkably, some on-device SLMs now achieve accuracy levels comparable to closed-source LLMs [52]. This raises an interesting question: *could the step-wise decision-making problem in GUI task automation be reframed as a code generation problem to better leverage the capability of on-device SLMs?*

To this end, we introduce AutoDroid-V2 to investigate **how to build a powerful GUI agent upon the coding capabilities of SLMs**. The key difference of AutoDroid-V2 is to complete GUI tasks by generating and executing multi-step scripts based on user instructions, rather than deciding one action at a time. We refer to this approach as the **"Script-based GUI Agent"**, which largely differs from traditional step-wise GUI agents, as illustrated in Figure 1.

Such a script-based agent can effectively address the aforementioned two problems of step-wise approaches: (1) Efficiency: Agents can generate a single script for a series of GUI actions to complete a task based on the user task, significantly reducing query frequency and consumption. (2) Capability: Script-based GUI agents rely primarily on the coding ability of SLMs, which have been proven effective by numerous existing studies on lightweight coding assistants.

A natural question that may be raised for script-based GUI task automation is why it is feasible to generate accurate scripts before knowing the actual situation at each runtime step. Conceptually, each app, although it may be very complicated, can be viewed as a finite state machine, where the number of states is predetermined in the app development stage. Thus, in principle, a human or AI with full information about the state machine is able to complete tasks without observing each state. Practically, realizing AutoDroid-V2 requires to further address two challenges. (1) **Understanding GUI elements**: It is necessary - but challenging - for LLMs to understand all GUI elements and their functions within an app to generate an accurate script. This difficulty arises from the complexity of mobile apps, which often involve numerous GUI states, each featuring over 100 GUI elements (e.g., buttons, icons, inputs) organized in a hierarchical structure, as well as complex transition relationships between GUI states. (2) **Dynamic Nature of Mobile apps**: Many GUI states and GUI elements can change in text, location, or size (*e.g.* contact names, note titles, dates). As a result, the generated script may become non-executable if a GUI element changes and cannot be located, due to the dynamic nature of GUI elements.

AutoDroid-V2 addresses these challenges by automatically generating a fine-grained yet concise app document, which guides agents in generating precise multi-step code to interact with UI elements and complete user tasks. The method consists of two stages: UI-Centric App Documentation and Document-Guided Code Generation. The UI-Centric App Documentation stage is responsible for analyzing the UI functionalities based on random exploring histories and recording the interactive elements. The Document-Guided Code Generation stage automatically generates code to interact with GUI elements, leveraging a domain-specific language (DSL) to efficiently handle runtime dynamicity.

We evaluate AutoDroid-V2 in two benchmarks with real execution environments and a total number of 226 tasks across 23 mobile apps, in comparison with strong baselines including AutoDroid [36], SeeClick [7], CogAgent [15], and Mind2Web [8]. The results demonstrate that AutoDroid-V2 can effectively enhance the accuracy and efficiency of on-device GUI agents, achieving 10.5%-51.7% higher task completion rate than baselines, reducing runtime input and output token consumption by 43.5x and 5.8x, reducing LLM inference latency by 5.7-13.4× compared to baselines.

Our work makes the following technical contributions:

(1) We introduce the first GUI agent driven by document-guided multi-step program generation, presenting a promising solution combining the remarkable coding capabilities of SLMs and efficient code-based task execution.

(2) We introduce a scalable automated app document generation method that can compress UI information from an app exploring history, as well as an interpreter that ensures reliable and efficient script execution.

(3) We conduct extensive experiments to demonstrate the efficiency and effectiveness of our approach, offering insights into the advantages and limitations of our script-based GUI agents and existing step-wise approaches.

2 Related Work and Motivation

2.1 GUI-based Mobile Task Automation

The goal of mobile task automation is to automatically complete user tasks by interacting with mobile devices. A GUI-based task automation agent takes a natural language task related to a mobile app and outputs a sequence of GUI actions executable on a smartphone. A **task** is a multi-step functionality request from the user intended for completion on a smartphone, often lacking explicit instructions. A **GUI State** reflects the current status or condition of the user interface in the app, usually structured as a GUI tree of controls represented by images and text. A **GUI element** is a control in the GUI state that human users can interact with, such as a button, a text box, an input field, and a slider. A **GUI action**, performable by the user or an agent on the device screen, is defined by (GUI element, action). Action represents how the target GUI element is manipulated, including "click", "long click", "input <text>", and "swipe <direction>".

2.2 Related Works

LLM-based GUI Agents. With the rise of large foundation models, there has been a trend leveraging LLMs [21, 36, 37, 44, 50] and Vision Language Models (VLMs) [40, 45, 49] to guide mobile task automation. These models enable more flexible and scalable task automation, typically executing tasks in a step-wise manner. Some research efforts focus on code generation for web page task automation [14, 31], relying primarily on the HTML to generate actions within the current screen. In contrast, AutoDroid-V2 adopts a full task-level script generation approach for mobile GUI agents. LLM-based GUI agents often rely on sensitive data—like screenshots and interaction logs—that may be sent to remote servers. This creates serious privacy risks, including data breaches and misuse [43]. High data traffic and server costs also hinder large-scale deployment. On-device LLM-powered agents are therefore essential for secure and affordable mobile task automation.

SLM-based GUI Agents. Researchers have explored various techniques to customize domain-specific SLMs for GUI-based mobile task automation. One area of focus involves enhancing the grounding capability of SLMs, which refers to locating GUI elements based on user instructions within a single GUI screen [7, 12, 15]. These methods gather extensive GUI-domain training data and use it to fine-tune a vision language model, thereby improving their ability to follow GUI domain-specific instructions. Other studies aim to acquire domain-specific knowledge about particular apps through exploration and use it to enhance SLMs' task-solving capabilities [2, 36]. However, the limited reasoning abilities of SLMs and large computational resource consumption hinder their widespread use, and we try to improve the efficiency and effectiveness of these methods by introducing document-guided script generation.

2.3 Script-based GUI Agents: Opportunities and Challenges

The limitations of current methods highlight the need for a more accurate and efficient approach to automating mobile UI tasks beyond traditional step-wise UI agents. Recent studies have shown that SLMs can perform well on challenging coding tasks [29, 52], and synthesizing executable code is more advantageous than using Text/JSON actions in agents [35]. This motivates the development of a script-based on-device GUI agent to address the identified limitations, offering the following benefits:

(1) Reduction of Query Overhead: Instead of querying the model for each individual step, script-based GUI agents can generate a multi-step action plan based on a single piece of code. This approach significantly reduces the overhead associated with multiple queries. (2) Enhanced Domain-specific Knowledge Utilization: By building an API document specifically for code generation, GUI agents can better leverage app-domain knowledge in a manner more familiar to language models. (3) Improved Action Tracking and Termination Control: Executing GUI actions based on code lines shifts the responsibility of action tracking from the model to a code status tracker. This approach is more controllable and avoids the issues of misalignment and hallucination, leading to more reliable task completion.

Although there is great potential for script-based on-device GUI agents, it still faces significant challenges.

Dynamic nature of mobile apps. For mobile apps, many UI states and elements are dynamic, frequently changing in text, location, or size (e.g., titles, dates, names). This variability increases the difficulty of both extracting app-domain knowledge and executing auto-generated scripts. To generate multi-step scripts that can be directly executed within app UIs, the system needs to provide all UI information within the entire app for summarizing knowledge. These dynamic UI elements need to be identified and abstracted to give an overview of the app. When executing the script, the system needs to locate the dynamic UI elements and perform actions according to the code, even when the elements change.

Extensive UI element space. A typical UI may contain over 100 elements with different XML descriptions, expanding the element space in the exploration history to tens of thousands. Describing all these elements for generating code is both redundant and potentially misleading, complicating the task of identifying the necessary UI elements for a given task. Additionally, UI elements often have relationships, such as inclusion (where some elements are nested within others, *e.g.* a song's title and artist within a single song item in Figure 3) or dependency (where one UI element is editable only after another has been clicked). It is still difficult for LLMs or SLMs to understand complex element relationships only based on a static UI document.

Limited Context Length and Computational Resources Due to limited computational power, reduced memory capacity, and energy constraints, on-device SLMs often have shorter context lengths (e.g., 4096 for Llama 3.1 8B [9]). As apps grow more complex, including detailed app usage patterns in the prompt can easily exceed the context length. Without this usage pattern, script generation loses valuable guidance and insight into app usage. Moreover, for a given user task, only a small portion of GUI elements may be relevant, yet

they can be buried within a long app usage context. In such cases, the on-device SLMs may struggle to identify and utilize the relevant information [18].

3 Our Approach: AutoDroid-V2

We introduce AutoDroid-V2 to address the aforementioned challenges of on-device GUI agents. The core idea behind AutoDroid-V2 is automatically constructing an informative app document and guiding the LLMs/SLMs to generate domain-specific device-control scripts.

As shown in Figure 2, AutoDroid-V2 operates in two stages. **Offline Stage:** AutoDroid-V2 first constructs an app document by analyzing the app exploration history (*details in §3.1*). The document serves as the fundamental guidelines for flexible script generation, which is desired to be concise (easy to process by LLMs) and precise (helpful for generating complicated executable code). We introduce various techniques (AI-guided GUI state compression, element XPath auto-generation, GUI dependency analysis, etc.) to achieve these goals. The built document is then used to synthesize large-scale user tasks for fine-tuning the local LLM (*details in §3.2*). Given the importance of training data in LLM customization, we adopt sandbox validation and tree-based search techniques to improve the data quality. **Online Stage:** Given a user's task request, AutoDroid-V2 calls the customized local LLM to generate a multi-step script and executes the script with our domain-specific interpreter (*details in §3.3*). Running the script reliably (*i.e.* able to handle runtime dynamics) and efficiently are two main objectives of the interpreter, which are addressed with our dependency-enhanced execution and prompt compression/caching techniques.

3.1 Automated App Document Generation

The goal of automated app document generation is to capture and analyze essential GUI elements in an app, guiding the backbone SLM to be aware of possible interaction use cases even without real-time observation. The input consists of random exploration traces, modeled as sequences of <GUI State, GUI Action> pairs. Following existing works [21, 36, 45], we suppose that the exploration traces can be generated by GUI testing tools [23] that explore apps randomly or through stochastic greedy depth-first strategies, or by a user trace monitor that records app usage in the background [11]. The exploration phase operates independently of user instructions, and the resulting traces are often redundant or aimless, containing many repeated actions and GUI elements. The output is a concise app document containing typical GUI states, important elements, and GUI transition relationships, as shown in Figure 3. Two primary challenges arise in this process: reducing the GUI element space and modeling transition relationships between GUI elements.

3.1.1 Functionality-aware GUI States Grouping. To ensure a structured and meaningful representation of the app's interface and interaction logic, AutoDroid-V2 first organizes the GUI states by iterating over the exploration history. Specifically, AutoDroid-V2 merges similar GUIs into an abstract GUI state, where similarity is determined by both **layout similarity** and **functionality similarity**.

Layout similarity refers to how the visual components (e.g., buttons, text fields, etc.) are arranged within the GUI. Let $T_1, T_2, ...T_n$ represent a set of GUI trees observed in the exploration traces. The

layout of each tree, T_i, is extracted by removing detailed content and repeated sibling elements. If a group $T_1, T_2, ...T_n$ shares the same layout l_i, they are considered layout-similar.

However, layout similarity alone is insufficient. Many different GUI states in an app may share the same layout due to common development patterns. For instance, a date selection screen and a background theme selection screen in a calendar app might both use a similar layout (e.g., a popup with a series of checkboxes). Despite having the same layout, these two screens serve different functions. To address this, we introduce functionality similarity, which considers the tasks or actions that users can perform within a particular GUI state. This ensures that even if the layouts are similar, the underlying functionality is accurately captured and differentiated. AutoDroid-V2 uses LLMs (*e.g.* GPT-4o) to assess and classify each GUI group based on its functionality after the layout-based grouping is performed. Specifically, for each step in the exploration history, AutoDroid-V2 constructs a prompt that includes the functionalities of previous GUI states (initially an empty list), and the current GUI state. It sends the prompt to the LLM to determine whether the current GUI state corresponds to any of the previously identified functionalities. If a match is found, the LLM maps the current GUI state to that existing functionality. If not, it generates a new name and description, which are then added to the functionality list.

Then, we can represent each abstract GUI state $S^{(i)}$ after grouping as $(functionality^{(i)}, layout^{(i)})$, where $functionality^{(i)}$ refers to the task or function associated with the GUI state (e.g., "date selection" or "theme selection") and $layout^{(i)}$ represents the shared visual structure.

3.1.2 Abstract GUI Elements. To reduce the GUI element space, we introduce abstract GUI elements in AutoDroid-V2, each of which may encapsulate several real GUI elements encountered during exploration. A **single abstract element** is typically one visible node in the GUI tree with which the user can interact. These elements can be classified into two types: **dynamic** and **static** elements. Static elements are those that remain constant regardless of changes in the app's status, such as the 'Search button' in Figure 3. Dynamic elements contain content that can change based on context, such as the 'song title' in Figure 3.

In addition to a single abstract element, AutoDroid-V2 introduces another abstract GUI element type called the **abstract element list**. This represents a collection of single elements that can be indexed or filtered, such as the 'song list' in Figure 3.

To address the dynamic nature of mobile apps and further reduce the GUI element space, AutoDroid-V2 merges a group of dynamic GUI elements (such as a group of specific songs in Figure 3) into one abstract GUI element (song_item), designating the parent node of these dynamic elements as the element list (song_list). We believe element list type can represent a large set of single elements in the app GUI because agents can interact with any of it by giving commands such as *<element_list>.match('name')* or *<element_list>[<idx>]* as mentioned in Section 3.2.1. This abstraction allows us to represent a large set of elements without needing to include every detailed element_list item in the document.

As shown in Table 1, an abstract element comprises four necessary components: *Name, Description, Identifier, Dependency* along with two optional components: *Action Effect, Options*. For the **element**
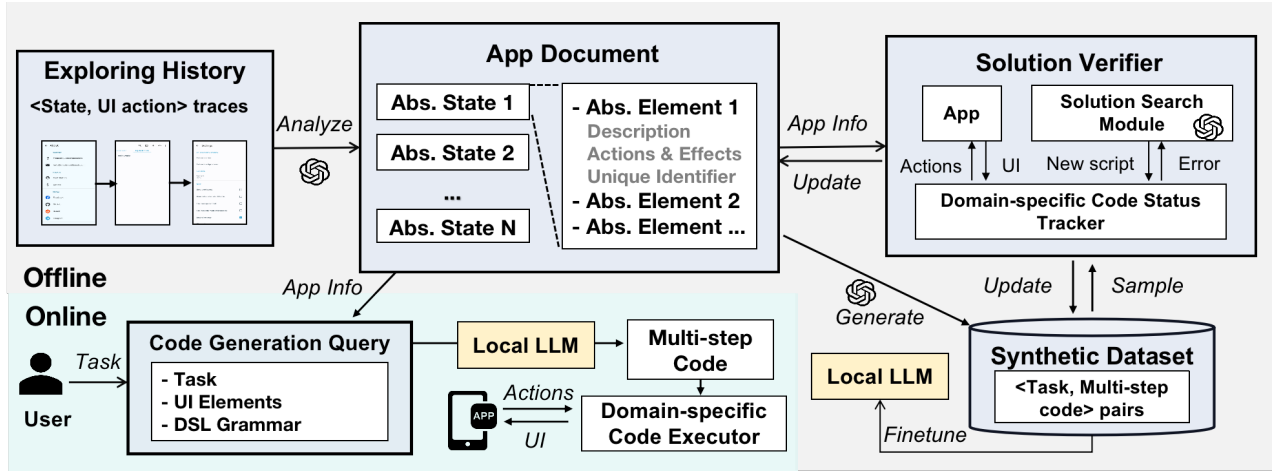
Figure 2: The architecture of AutoDroid-V2.
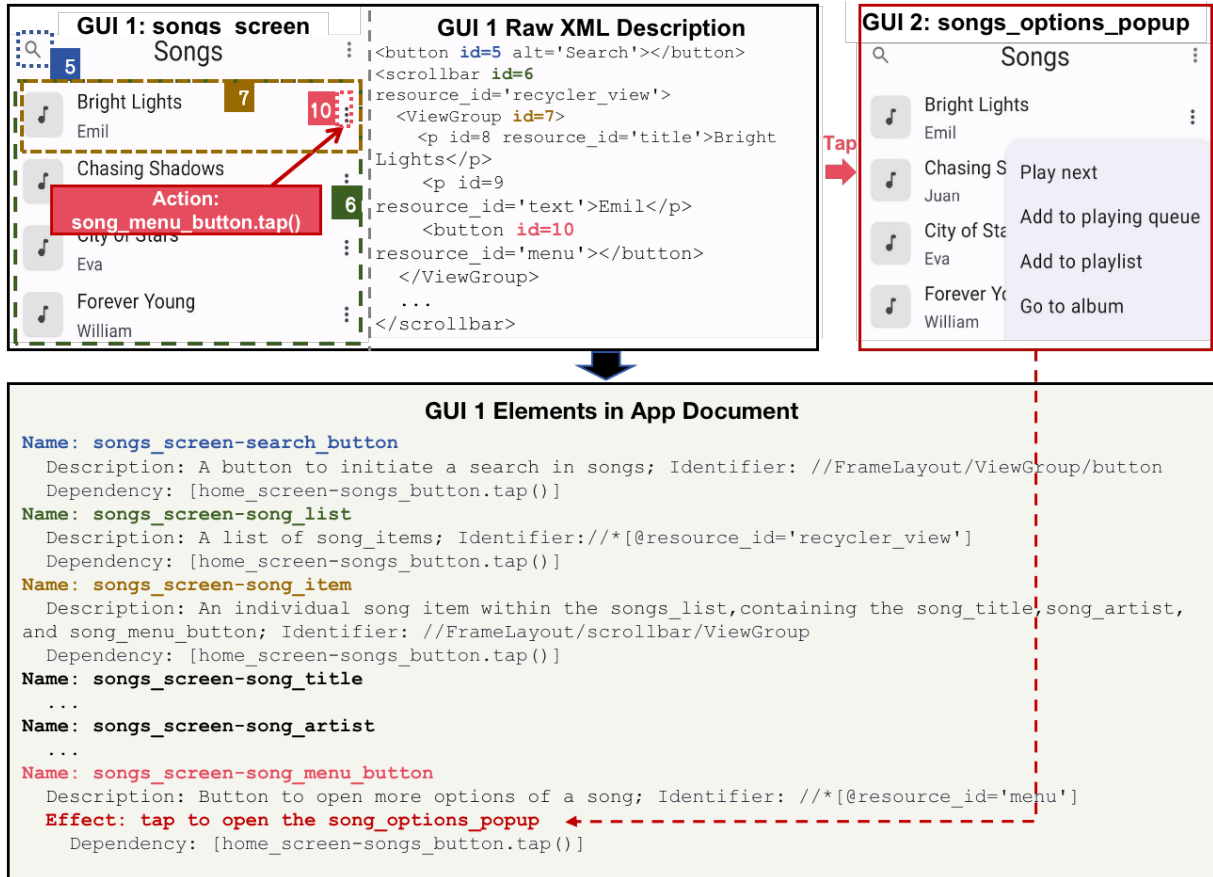
Figure 3: The document of an app containing essential abstract elements in one GUI.

name, the GUI state name is used as a prefix because there may be several elements with the same name that belong to different states and function differently. For example, tapping the 'more options' button in different GUIs can lead to different GUI states. **Identifier**

is used during runtime to locate and interact with the desired GUI element based on the rules defined in the script. AutoDroid-V2 uses XPath (XML Path Language) as the identifier for each GUI element. Due to the frequent changes in the number and properties of GUI

Table 1: Components of Abstract GUI Elements

Component	Description	Example
Name	A concise name for the element formatted as *state_name-element_name*.	songs_ui-song_list
Description	A brief description of the element's purpose and functionality.	A list of song_items
Identifier	Uniquely identifies and locates a specific GUI element within the app's GUI tree.	//*[@resource id='recycler view']
Options	All the possible keywords of the items in this element list. The items can be chosen by using '<element_list>.match(Options[i])' statement, which will be explained in Section 3.2.1.	<song_names>
Effect	describes the effect of performing actions on it.	tap to open the song_options_popup
Dependency	The navigation path for one GUI element from the other GUI screens.	Tap main_screen-open_song

elements, using a single XPath to locate the GUI element can be unreliable at runtime. Multiple XPaths form a prioritized queue for identifying GUI elements. The first (most precise) XPath specifies all element properties (*i.e.* t_e (text), alt_e (alternate text), r_e (resource id of e), $path_e$ (relative position of a GUI element from the root of the GUI tree)}, but may fail if some properties change at runtime. Subsequent XPaths omit changeable properties{$alt_e, r_e, path_e$}, {$r_e, path_e$}, etc. ($path_e$ is not used alone in the XPath because of the repeatness of UI elements). During execution, the system attempts to ground the GUI element using the XPath queue, starting with the most precise XPath and moving down the queue until a match is found.

To identify abstract GUI elements through an exploration history, we enumerate all GUI elements within every GUI state and use LLMs to determine the element: *Name, Description, and Options* for important elements. The *Effect* of each element is determined by the subsequent GUI state triggered by interaction. Additionally, LLMs are used to categorize elements as dynamic or static and encapsulate them for further processing. Specifically, for each GUI state from one abstract GUI state, AutoDroid-V2 constructs a prompt (shortened for clarity) to query LLMs (*e.g.* GPT-4o) to classify the GUI elements and their properties.

```
1  Imagine you are a UI annotator tasked with describing interactive
       elements of a mobile application in the form of a UI document
       . You are given:
2  <Current UI State>
3  <UI Action Performed on Current UI State> (including the
       functionality of the next GUI state)
4  <Previously Classified UI Elements in the Same Abstract GUI State>
5  <An example>
6  Now, you should output all new UI elements (i.e., those not
       present in the previously classified elements) that can be
       interacted with in the current UI state.
```

Listing 1: Structure of GUI element classification prompt.

3.1.3 Forward and Backward Dependency. To provide necessary information for agents to plan solution steps, AutoDroid-V2 profiles the GUI element transition relationships by constructing an Element Transition Graph (ETG) and analyzing both the forward and backward dependency for each element. In the ETG, transitions are represented as triples (e_i, a_i, e_{i+1}). Here, e_i and e_{i+1} are nodes representing GUI elements, and a_i is the directed edge representing the action taken to transition from e_i to e_{i+1}. To build ETG, we define the root node e_r as the elements in the first GUI of the app after it is opened. For each action a_i observed during the exploration, we

locate the existing element e_i that a_i is performed on, and identify all the GUI elements ($e_{i+1}, e_{i+2}...$) that become available in the subsequent GUI state resulting from a_i. Edges are added to represent the transitions: $(e_i, a_i, e_{i+1}), (e_i, a_i, e_{i+2}), etc.$

After constructing ETG, we analyze the transition relationships between elements by defining two types of dependencies for each element e_i: the backward dependency and the forward dependency. **Backward dependency** records all the paths from other elements $e_1, e_2, ...e_{i-1}$ leading to e_i. **Forward dependency** represents the subsequent GUI state e_i leads to. At runtime, forward dependency is included in the prompt, providing the context by showing how one GUI element leads to the next sequence of interactions. If the script generated by the LLM fails to correctly trigger a transition between GUI elements, the backward dependency is then used to trace back through the possible previous interactions and identify an alternative path that can successfully reach the target GUI element.

3.2 Data Synthesis for SLM Customization

To improve the task automation capabilities of on-device SLMs, it is crucial to train them with large-scale data. We generate a large set of simulated user tasks of varying complexity based on the app document, along with their corresponding solutions. These synthetic task-solution pairs are used to fine-tune the SLMs, enabling them to create GUI-specific code with correct syntax and dependencies.

3.2.1 Large-scale Task & Solution Generation. AutoDroid-V2 generates tasks by sampling groups of elements from the app document, creating task-solution data of varying complexity levels. This approach enables on-device SLMs to learn to handle diverse tasks effectively. Task generation can be viewed as the inverse of task automation. In task automation, a user task $\mathcal{T}$ is mapped to a set of GUI elements and corresponding actions, forming a sub-graph G_s in the Element Transition Graph (ETG). Conversely, generating tasks is to map G_s to $\mathcal{T}$. AutoDroid-V2 iteratively samples a G_s from the ETG and queries LLMs to generate user tasks solvable with these elements. The complexity of a task is determined by the number and type of GUI elements involved and the number of actions required to complete the task. By varying the size of G_s, tasks of different complexities can be generated, denoted as $\{\mathcal{T}_1, \mathcal{T}_2, ..., \mathcal{T}_n\}$.

To regulate the task solution format and execute them on real devices, we develop a Python library to interpret the scripts. The design principle of this library is being concise and intuitive, allowing LLMs/SLMs to effectively learn through simple instructions and in-context examples. The library provides two categories of functions: GUI action APIs and information retrieval APIs. The GUI action APIs (including *tap, long_tap, set_text, scroll*) are designed to manipulate an element or a specific child element within an element list. These actions are invoked using methods like: *<element>.tap(), <element>.tap(child_element), <element>.set_text(<text>), or <element>.scroll(<direction>)* Information retrieving APIs (including *get_text, get_attributes, match, indexing*) facilitate the accurate identification of task-related GUI elements by gathering necessary information about them. Examples include *<element_list>.match(<text or attribute dict>)* or *<element_list>[<idx>]*.

Then, we design a detailed prompt to guide LLMs in generating executable code of user tasks for training. The prompt includes the user task, a simplified App Document (which details the name,

description, options, and effects of each element), a description of the domain-specific library, and the GUI elements in the current GUI (often the GUI elements in the original GUI screen of the app):

```
7   Imagine that you are an agent using a smartphone to complete a
        task. You need to write scripts to manipulate the GUI
        elements in the apps.
8
9   In the script, except for the common Python control flow (for, if-
        else, function def/calls, etc.), you can use the following
        APIs: <APIs in GUI domain-specific library>
10
11  You can use the following important GUI elements:
12  ...
13  <element i>: <description of element i>
14  Effect: <effect of element i>
15  Options: <options of element i>
16  ...
17
18  The available GUI elements in the current GUI:
19  <current available GUI elements>
20
21  Your task is: <the user task>
22
23  Your answer should follow this JSON format:
24
25  {
26      "plan": "<a high-level plan to complete the task>",
27      "elements": "<analyze the elements that could be used to
            complete the task>",
28      "script": "<the script to complete the task>"
29  }
```

Listing 2: Structure of the code generation prompt.

From the LLM's response, we can then parse an executable script. For example, for the task about the app shown in Figure 3, LLM can generate the following script:

```
31  '''
32  User task: Add the songs of Artist Emil to the Playlist 'Emil'.
33  '''
34  # Navigate to the Songs section
35  home_screen-songs_button.tap()
36  # Flag to control the scrolling loopdone = False
37  while not done:
38    for song_item in songs_screen-song_list:
39      song_artist = song_item.get_text(songs_screen-song_artist)
40      if song_artist == 'Emil':
41        song_item.tap(songs_screen-song_menu_button)
42        song_options_popup-add_to_playlist_button.tap()
43        add_playlist_screen-playlists.match('Emil').tap()
44
45    # Check if the end of the song_list has been reached
46    bottom = songs_screen-song_list.scroll("down")
47    if bottom:
48      done = True
```

Listing 3: LLM generated Python code for the user task.

3.2.2 Validation-based Script Revising. For each generated script, we validate it by executing it in a real environment before including it in the training dataset. Many of the generated scripts may not be executable due to LLM errors and app dynamics. Therefore, we need to handle encountered errors and revise the scripts accordingly during script validation. We introduce a domain-specific code status detector module that addresses errors by regenerating the script based on the error information. Specifically, for each task, we initially generate a code sample $C = C_0$, which is executed on a mobile device or emulator based on the domain-specific library mentioned in Section 3.2.1. If execution fails with error information $\mathcal{E}$, it indicates issues with C. The code status detector sends error information $\mathcal{E}$ as well as the GUI state where the error occurs to the LLM, which is responsible for regenerating the script based on this GUI state to continue execution.

Specifically, we categorize error $\mathcal{E}$ as sending an illegal action to a GUI element (such as inputting text into a button), matching or indexing a non-existent element in the screen (such as index out of range for an element list), or logic errors (the previous action does not result in the GUI state that contains the current element). This categorization makes it easier for LLMs to understand the error, rather than processing raw error information.

3.2.3 Tree-based Script Quality Improvement. Even if a script executes successfully, it may not align with the intended user tasks. Thus, AutoDroid-V2 uses an on-cloud LLM-based reward model [51] to evaluate whether the code C successfully achieves the task $\mathcal{T}$. This reward model takes as input the task $\mathcal{T}$, the executed code C, and a sequence of environment observation states $\mathcal{S} = (s_1, \ldots, s_n)$, outputting a judgment $\mathcal{J}$ and feedback $\mathcal{F}_{\text{out}}$. If the reward model deems the code C as complete, the task-code pair $(\mathcal{T}, C)$ is added to the dataset. Otherwise, the error information $\mathcal{E}$ and feedback $\mathcal{F}_{\text{out}}$ are recorded and used in a code search tree to regenerate a code sample.

The code search tree is a Depth-First Search (DFS) tree designed to generate code samples, where each node represents a possible attempt to achieve the task goal through code. The initial code C_0 resides at the root node of the tree. During the search process, new nodes are expanded in a depth-first manner. At each node, the LLM is queried to generate a new code C' based on the task $\mathcal{T}$, the current node's code C, the error information $\mathcal{E}$, and the feedback $\mathcal{F}_{\text{out}}$ from the reward model. The LLM generates new code sample C' that tries to address the error information $\mathcal{E}$ and the feedback $\mathcal{F}_{\text{out}}$, while achieving the task $\mathcal{T}$.

The new code sample C' is then validated again through the dynamic task execution validator. The search continues until a code sample C is accepted by the validator or the maximum number of search attempts is reached. The final task-solution pair $(\mathcal{T}, C)$ is then added to the final dataset, which will be used to fine-tune the SLM.

3.3 Script-based Runtime Task Execution

At runtime, AutoDroid-V2 queries the LLM to synthesize domain-specific script tailored to the user's task, which can be executed by the domain-specific code executor.

3.3.1 Runtime Dynamicity Handling. AutoDroid-V2 handles runtime dynamicity with two techniques. The first is the error handler, which is to re-generate the script after encountering runtime failures as described in Section 3.2.2. The second is the dependency-aware instruction execution.

At runtime, AutoDroid-V2 executes instructions from the script by grounding the target GUI element in the GUI screen to send proper GUI actions. AutoDroid-V2 introduces a dependency-enhanced instruction execution mechanism to handle incomplete scripts that may fail to account for all necessary navigation paths to a target GUI element. Specifically, let e_c represent the GUI element invoked in the script, and let T_c represent the current GUI tree, we match e_c by checking: 1) Whether T_c is the GUI state that e_c belongs to, in case the current GUI screen is another state that contains different GUI elements that share the identical XPaths with e_c. 2) The identifier of e_c can match one GUI element e_i in T_c. If not matched, AutoDroid-V2 leverages *backward dependency* of e_c to automatically navigate to it. The dependency represents potential navigation sequences that

can guide the GUI testing process to the desired element. Specifically, AutoDroid-V2 iteratively selects and executes actions from the dependency paths that are most likely to align with the current GUI context. If the element is still not found after completing all dependencies, AutoDroid-V2 raises an error and delegates the issue to the error handler module, signifying potential deficiencies in the app document or code logic.

Algorithm 1 Dependency-Aware Element Locating

Input: Target element E, current app instance App, dependency paths $\mathcal{D} = [D_1, D_2, \ldots, D_k]$, app document Doc
Output: Execution status S
1: **function** ELEMENTGROUNDING($E, App, \mathcal{D}$)
2:     **while** $E.identifier \notin current\_ui\_tree$ **and** *attempts < limit* **do**
3:         $Dependency \leftarrow$ Select from $\mathcal{D}$ based on which dependency includes an action matching the current GUI state
4:         **for** action in $Dependency$ **do**
5:             $current\_ui\_state \leftarrow$ GetCurrentUIState(App, Doc)
6:             **if** $action.element.ui\_state = current\_ui\_state$ **then**
7:                 $App.send\_action(Action)$
8:     **if** $E \notin Current$ **then**
9:         **return** Error   ▷ Section 3.2.2
10:     **else**
11:         **return** $CurrentUITree.FindElement(E.identifier)$

3.3.2 Enhancing Efficiency with Prompt Compression and Reuse.
Due to the limited computing resources of mobile devices and constrained context lengths of on-device SLMs, using the detailed app documents for task script generation (like how we synthesize data in Section 3.2.1) can be costly or even infeasible at runtime. Therefore, we propose to shorten the runtime query prompt. Specifically, a runtime code query prompt consists only of the basic instruction, the user task, and all the element names in the app document, which reduces the prompt length from 15.4k to 2.8k on average (most of which can be cached and reused). The feasibility of such simplifications is from fine-tuning, which trained the intrinsic task-agnostic knowledge into the model parameters and left only the task-dependent in the prompt for the customized SLMs' reference for different tasks. As such, this reduction does not affect the performance of on-device SLMs.

AutoDroid-V2 also employs prompt cache [10] techniques to further speed up on-device SLM inference, particularly in reducing the prefilling latency. The prompt cache stores the KV state of existing queries, enabling a new query to reuse the KV cache if it shares the same prefix with a previous query, which allows the new query to skip the computation of the shared part. The document portion of a prompt in AutoDroid-V2 is consistent across all queries and can be cached and reused for each user task. Additionally, the document statement accounts for most of the input prompt length (by 97.6%), so caching this part significantly reduces the prefilling latency.

4 Evaluation
We implement AutoDroid-V2 using Python and Java. SLMs are deployed on mobile devices using llama.cpp, a framework to enable LLM/SLM inference in C/C++. AutoDroid-V2 collects current GUI information via ADB (Android Debug Bridge). For each GUI action API, ADB commands are sent to the identified GUI element.

4.1 Experimental Setup
Dataset. We evaluate the effectiveness and efficiency of AutoDroid-V2 and baseline on two benchmarks: DroidTask [36] and AitW-subset [28]. DroidTask [36] is a mobile task automation dataset with 158 high-level tasks across 13 popular apps. AitW [46] is a large-scale dataset for Android device control. The original AitW dataset contains 417k tasks but lacks the exploration environment of the apps. So we choose the 'google apps' and 'general' subsets of the AitW dataset with 68 tasks using LlamaTouch [46] environment, which is executable and can be explored. The exploration trace is provided by the original datasets. Our document generation and task synthesis are independent of the test set.

Training Details. We perform supervised fine-tuning on our model using the automatically generated task-solution training set detailed in Section 3.2, training for one epoch with a batch size of 32, a learning rate of 1e-5, and a maximum sequence length of 8192. For the baseline AutoDroid [36], we use the same training parameters.

Hardware. We evaluate the performance of AutoDroid-V2 on two devices: (1) A OnePlus ACE 2 Pro with 8 3.2 GHz ARM-based cores (Snapdragon 8 Gen2 CPU) and Adreno™ 740 GPU. (2) a MacBook Pro with an Apple M2 Pro chip, featuring a 10-core CPU, a 16-core GPU, and 16 GB of unified memory, selected to represent devices with higher computational capacity. The local LLM Llama-3.1 [9] is deployed on the smartphone based on the llama.cpp framework [10]. The local LLMs are fine-tuned on an 8× A100 80GB server for 1 epoch, taking about 2.5 GPU hours.

Baselines. We mainly compare AutoDroid-V2 with 2 types of baselines. (1) VLM-based agents, including CogAgent [15], SeeClick [7]. CogAgent is an 18B visual language model specializing in GUI navigation tasks. SeeClick is a GUI agent built on Qwen-VL [3], with 9.6B parameters. (2) LLM-based agents, including AutoDroid [36], Mind2Web [8]. AutoDroid is a memory-augmented mobile task automation framework, which is similar to our framework but adopts a step-wise method for task automation. We evaluate AutoDroid using GPT-4o [16] and a fine-tuned Llama-3.1 [9] based on its automatically generated dataset. Mind2Web is a generalist agent for UI task automation, which we evaluate based on GPT-4o [16] and Llama-3.1 [9] respectively.

Metrics. Given a sequence of UIs $\{U_1, U_2, ..., U_n\}$ and corresponding actions $\mathcal{A} = \{A_1, A_2, ..., A_n\}$ performed by human annotators to complete a task T, if an agent generates a sequence of decisions $\hat{\mathcal{A}} = \{\hat{A}_1, \hat{A}_2, ..., \hat{A}_k\}$ on $\{\hat{U}_1, \hat{U}_2, ..., \hat{U}_k\}$, we use the following metrics to evaluate its performance:

- **Success Rate:** In DroidTask [36], a task is considered completed if the ground-truth action sequence $\mathcal{A}$ is a subsequence of the agent-generated sequence $\hat{\mathcal{A}}$, i.e., $\mathcal{A} \in \hat{\mathcal{A}}$. This metric reflects the agent's ability to complete the task. The LlamaTouch framework [46] evaluates task completion by sequentially comparing a task execution trace $\{\hat{U}_1, \hat{U}_2, ..., \hat{U}_k\}$ against selected essential states $\{U_{i_1}, U_{i_2}, ...\}$. A task is considered complete if the task execution trace includes all the essential states in order.
- **Reversed Redundancy Ratio:** As defined in AndroidArena [38], the reversed redundancy ratio is the ratio of the length of the ground-truth action sequence to the length of the agent-generated sequence, namely n/k. This metric measures the agent's efficiency when completing a task.

- **LLM Inference Latency:** We primarily measure the latency of LLM inference, which accounts for most of the system latency in on-device agents. Inference latency begins when the prompt is received by the LLM and ends when the final token of the output is generated.
- **Token Consumption:** We calculate token consumption based on the number of input and output tokens processed by the model.

4.2 Success Rate and Redundancy

4.2.1 DroidTask. We first evaluate the accuracy of AutoDroid-V2 and baselines in DroidTask, and the result is shown in Table 2. AutoDroid-V2 achieves an average task accuracy of 54.4%, significantly higher than the baselines, whose accuracy ranges from 10.5% to 43.9%. Among the baseline methods, AutoDroid [36] achieves the highest performance because it also adopts automatic task generation and fine-tuning techniques to enhance the performance of on-device LLMs. VLM on-device UI agents (CogAgent [15] and SeeClick [7]) exhibit lower accuracy because they rely solely on screenshot images, which are inadequate for text-heavy apps. Besides, the grounding capabilities of smaller on-device VLMs are limited, leading to frequent misplacement of UI elements.

AutoDroid-V2 outperforms baselines mainly because of the following reasons: Compared to methods that do not fine-tune LLMs, AutoDroid-V2 generates fine-tuned data that covers every GUI element in the app document, offering detailed insights into how to use the app. While AutoDroid [36] also fine-tunes LLMs by generating tasks and solutions, it relies on a step-wise method. In contrast, AutoDroid-V2 uses script-based fine-tuning data, which is more suitable for smaller LLMs. For the script-based method, the responsibility of determining whether a task is complete is shifted from the LLMs to the code executor. If a script runs without errors, it is considered complete, thus eliminating the need for the LLM to make a task completion decision. And AutoDroid is less effective at determining task completion [36]. AutoDroid tries to address this limitation by incorporating a small amount of manually annotated data from [6], this solution is neither scalable nor fully effective. Our experiments found that AutoDroid repeatedly executes actions for 25.5% The success rates for the Files, SMS, and Notes apps in AutoDroid-V2 are lower than expected, primarily because some high-quality fine-tuning data for these apps was mistakenly filtered out by the solution validation module, a problem discussed in Section 4.4.2. In practice, this issue could be addressed by exploring the app and pre-setting some user data within it.

For the reversed redundancy ratio, systems based on fine-tuned LLMs (AutoDroid [36], AutoDroid-V2) are better than systems based on original LLMs (Mind2Web [8]). LLMs can get a comprehensive overview of apps after fine-tuning and can efficiently navigate to the specific UI elements that are most likely to solve the task. Seeclick [7] and Cogagent [15] struggle to complete tasks in DroidTask, with their success rates being exceptionally low. Due to their inability to successfully complete most tasks, we do not include these two methods in the RRR statistics.

4.2.2 AitW. AutoDroid-V2 outperforms the baselines by an average of 14.3%, as shown in Table 3. LLM-based GUI agents (CogAgent [15] and SeeClick [7]) perform better on AitW [28] than on DroidTask [36] because they were fine-tuned on the dataset and
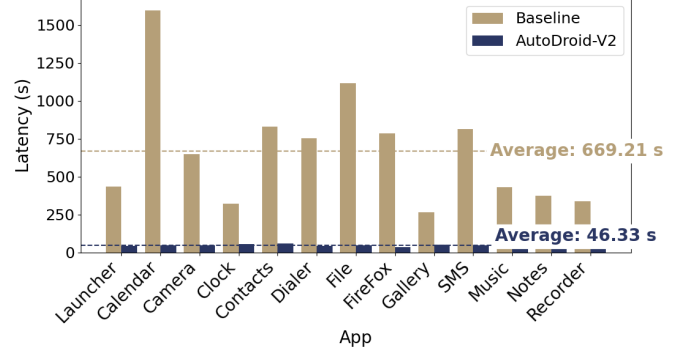
(a) LLM inference latency in Snapdragon 8 Gen 2.
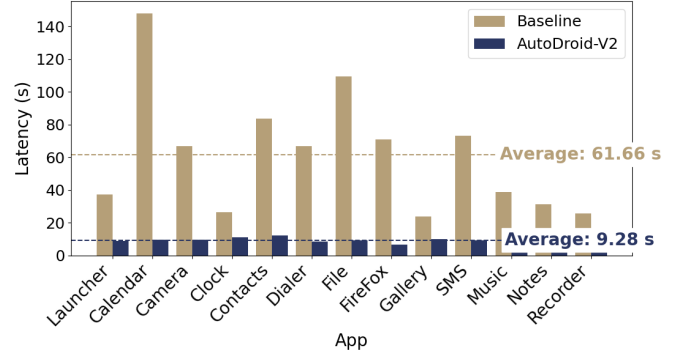
(b) LLM inference latency in Apple M2 Pro.

Figure 4: Average LLM inference latency of AutoDroid-V2 and baseline (AutoDroid) on DroidTask.

have a comprehensive understanding of the apps. The results demonstrate that AutoDroid-V2 consistently outperforms the baselines in complex apps like Chrome, Google Maps, and YouTube. This is because these apps often feature dynamic or deeply nested UI elements (*e.g.* hierarchical menus in Google Maps or dynamic content in YouTube driven by recommendation systems). Step-by-step agents may struggle to locate and interact with such elements consistently due to the limited context provided by a single screen. In contrast, AutoDroid-V2's script-based approach, with access to a full application overview, allows it to consider the potential effects of each action and anticipate future elements, leading to more effective solutions.

4.3 Latency and Cost

Figure 4 illustrates the on-device LLM inference latencies of AutoDroid-V2 and the step-wise method (AutoDroid [36]) across different apps on Snapdragon 8 Gen 2 and Apple M2 Pro. We quantize the fine-tuned Llama3.1 8B models of AutoDroid-V2 and AutoDroid to 8-bit precision and deploy them on mobile devices using the Llama.cpp framework [10]. The average LLM inference latency for each task of AutoDroid-V2 is 46.3s, compared to 669.2s for the baseline on Snapdragon 8 Gen 2, resulting in a 93.1% reduction in inference latency.

Table 2: Success rate on DroidTask. SR: Success Rate. RRR: Reversed Redundancy Ratio. 'ft' represents LLMs fine-tuned on GUI-specific dataset.

Metric	Method	Launcher	Calendar	Camera	Clock	Contacts	Dialer	File	FireFox	Gallery	SMS	Music	Notes	Recorder	Average
SR	SeeClick (Qwen-VL-9.6B-ft)	0%	0%	0%	9.1%	0%	0%	0%	0%	11.1%	13.3%	0%	0%	0%	2.7%
	CogAgent (CogVLM-17B-ft)	0%	0%	0%	9.1%	0%	0%	0%	0%	11.1%	13.3%	0%	0%	12.5%	3.3%
	Mind2Web (Llama-3-8B)	60.0%	41.2%	60.0%	50.0%	21.4%	33.3%	20.0%	12.5%	22.2%	40.0%	11.1%	35.7%	33.3%	34.4%
	AutoDroid (Llama-3-8B-ft)	80.0%	29.4%	50.0%	50.0%	21.4%	40.0%	**43.8%**	**87.5%**	44.4%	**33.3%**	44.4%	**35.7%**	**66.7%**	43.9%
	AutoDroid-V2 (Llama-3-8B-ft)	**80.0%**	**64.7%**	**66.7%**	**83.3%**	**50.0%**	**60.0%**	31.2%	75.0%	**66.7%**	26.7%	**55.6%**	28.6%	55.6%	**54.4%**
RRR	Mind2Web (Llama-3-8B)	21.1%	26.7%	42.6%	29.6%	39.4%	43.0%	38.9%	66.7%	75.0%	58.3%	23.1%	42.7%	50.7%	41.0%
	AutoDroid (Llama-3-8B-ft)	73.8%	82.4%	**92.3%**	70.8%	**88.9%**	**91.7%**	**95.2%**	67.6%	**100.0%**	95.0%	84.4%	95.0%	88.9%	86.3%
	AutoDroid-V2 (Llama-3-8B-ft)	**93.8%**	**94.7%**	91.7%	**74.7%**	88.1%	88.9%	95.0%	**76.4%**	96.7%	**100.0%**	**95.0%**	**100.0%**	**100.0%**	**90.5%**

Table 3: Success rate on AitW-subset. SR: Success Rate.

Method	Model	SR
Mind2Web	Llama-3.1-8B	27.1%
SeeClick	Qwen-VL-9.6B	30.8%
CogAgent	CogGLM-17B-ft	36.7%
AutoDroid	Llama-3.1-8B-ft	36.7%
AutoDroid-V2	**Llama-3.1-8B-ft**	**47.1%**

Table 4: Runtime token number of AutoDroid-V2 and Auto-Droid. 'Cached' denotes the cached prompt prefix that has been processed and can be reused. 'Remaining' denotes the remaining parts of the input prompt that come after the cached prefix.

	Method	Input Tokens			Output Tokens
		Cached	Remaining	Total	
Per Step	AutoDroid	66.0	452.1	518.1	127.4
	AutoDroid-V2	N/A	N/A	N/A	N/A
Per Task	AutoDroid	431.3	3021.2	3452.5	832.4
	AutoDroid-V2	2760.1	**67.9**	2828.0	**122.9**

This acceleration is mainly due to the script-based method, which reduces the number of LLM queries. For the step-wise method, the latency for each step remains stable, so the overall latency depends on the number of steps in a task. The Calendar app, for instance, has the highest average step count (13.4 steps per task), resulting in the highest latency. As the number of steps increases with more complex tasks, the latency for step-wise methods rises much faster than for the script-based method.

Table 4 shows the average runtime token consumption for AutoDroid-V2 and the baseline method. The cached tokens are the common prefix part that has already been processed, and the internal state of LLM has been stored in a file, which can be pre-loaded to memory and reused. The cached tokens refer to the common prefix that has already been processed, with the LLM's internal state stored in a file, allowing it to be preloaded into memory and reused. The remaining tokens are part of the input prompt following the cached prefix, which must be processed in real-time during model inference. For the step-wise method, the main part of a prompt is the GUI state, which cannot be cached because they are dynamic and frequent changes. Conversely, AutoDroid-V2 primarily uses the app document, which is static and can be cached. The latency for each response mainly depends on the remaining and output tokens. AutoDroid-V2 saves 97.8% of the remaining tokens and 85.2% of the output tokens, significantly reducing the computational burden.

The large portion of the common prefix in AutoDroid-V2 prompts allows for the use of the prefix key-value (KV) cache, which greatly reduces prefill time. LLM inference latency arises primarily from two phases: the prefill phase, where input text is prepared, and the decoding phase, where the model generates output based on the input and prior tokens. The prefill latency for AutoDroid-V2 is significantly reduced compared to the baseline (14.0s versus 85.3s per prompt) because most of the prompt has already been cached.

We examine token consumption and associated costs of AutoDroid-V2 while utilizing GPT-4o[16] across three stages: document generation, data synthesis, and solution validation. These stages represent

a one-time data collection process, during which GPT-4o is used to prepare the necessary data. Once the data are collected, the task execution process of AutoDroid-V2 does not require the intervention of GPT-4o. In the document generation stage, the average consumption of input tokens per application is 1.28M, and the average consumption of output tokens per application is 91.0K, resulting in a cost of $4.11 per application. The data synthesis stage consumes 2.98M input tokens and 38.2K output tokens per application at a cost of $7.83 per application. The solution validation stage demands the most substantial computational resources and incurs the highest costs, with an average consumption of 26.81M input tokens and 343.9K output tokens per application, leading to a cost of $70.48 per application.

4.4 Fine-grained Performance Analysis

4.4.1 Performance with Different LLMs. We use AutoDroid, the best-performing baseline on the Llama-3.8B model, as a reference and compare it with AutoDroid-V2 across different LLMs. The results presented in Figure 5 show the comparison of success rate and reversed redundancy ratio between AutoDroid-V2 and the baseline models for different LLMs: Llama3.2-3B, Qwen2.5-7B, and Llama3.1-8B, where AutoDroid-V2 achieves 44.6%, 50.0%, 54.4% success rate respectively. The reversed redundancy ratio of AutoDroid-V2 is stable, ranging from 90.5% to 93.0%. This is due to the generated scripts being designed to be straightforward and focused on task completion without including exploration or trial-and-error steps.

We also conduct experiments using powerful on-cloud LLMs (GPT-4o) as a reference for comparison with on-device LLMs, as shown in Table 5. The method employed by AutoDroid-V2 is described in Section 3.2. The success rate of AutoDroid-V2 is 9.3% higher than the baselines on average across two benchmarks. We summarize the failure cases observed with the AutoDroid-V2 below:
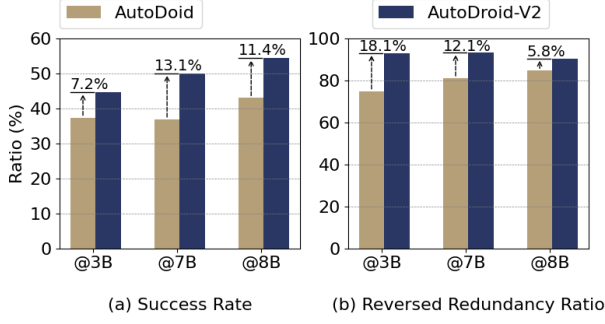
Figure 5: Success rate and reversed redundancy ratio of AutoDroid-V2 and baseline based on different LLMs. llama3.2-3b, qwen2.5-7b, llama3.1-8b

Table 5: Success rate on DroidTask based on on-cloud LLM (GPT-4o).

Dataset	Metric	Mind2Web	AutoDroid	AutoDroid-V2
DroidTask	SR	65.2%	62.7%	67.1%
	RRR	92.5%	92.3%	90.1%
AitW-subset	SR	41.1%	30.9%	51.4%

Match Errors: The <element>.match(<text or attribute dict>) syntax finds target elements in the list that match the provided text or attribute dictionary. However, the provided texts may not always align with the actual element text. For example, in the task *"change the theme of the app to black"*, the LLM might generate *theme_options.match('black').tap()*, but the available options could be *"white", "dark", or "auto light/dark"*. In this case, the correct action would be to tap "dark". A solution is to use a lightweight embedding model to find the most similar text to the input.

Misunderstanding Syntax: We have observed instances where certain code statements are generated incorrectly. For example, *favorite_button.match('selected') == 'false'* should be written as *favorite_button.match('selected') == False*. Enhancements to the code generation capabilities of the foundation model could help address such issues.

Neglecting Confirmation Elements: Some tasks require confirmation at the final step, such as tapping elements like "save" or "confirm." These confirmation actions are sometimes overlooked in the generated code.

4.4.2 Impact of Script Validation. We also compare the success rate of AutoDroid-V2 with and without the train data validator module mentioned in Section 3.2.2 and Section 3.2.3. The LLM is Llama3.1 8B, and the average success rate of methods with and without a solution validator are 54.4%, and 51.9% respectively, improved by 2.5%. We observe that 4 applications such as Contacts, Files, Notes, and SMS experience a minor decrease in success number (1-3 tasks) after the validation step. These apps are highly user-specific, containing personal data such as contact names, file names, and notes. Validation becomes problematic for these apps because some tasks cannot be completed unless certain user-specific data is present. For instance, the task "Delete a contact named Alice"

cannot be completed if Alice is not in the user's contact list. Thus, some valid tasks and solutions are filtered out by the validator, even though the solutions themselves are correct. Consequently, these tasks are excluded from the training data, which diminishes the fine-tuned LLM's ability to complete such tasks. For other apps, tasks do not require specific user data (*e.g.* Camera, Calendar, Recorder). In these cases, the validator can filter out incorrect solutions, improving or maintaining the fine-tuned model's accuracy. One potential solution to this issue is to pre-set user data using a random explorer before validating the app. The task generator can then be provided with this user data, enabling it to generate tasks that can be validated successfully.

5 Discussion

One potential concern is how AutoDroid-V2 handles situations where structured text representations of GUIs are unavailable. Typically, AutoDroid-V2 identifies and interacts with the target GUI elements at runtime by matching their identifiers, which are text-based XPaths. However, some special mobile apps, such as Unity-based apps and Web-based apps, may not provide structured text representations. A possible solution is to recover structured GUI representations based on the visual features. Vision-Language Models (VLMs) have shown their great potential in this field, and it is foreseeable that structured GUI representations can be obtained by VLMs accurately and efficiently.

Another possible limitation of AutoDroid-V2 is the difficulty of handling highly dynamic user interfaces, such as those in web search engines and games. In these apps, it is difficult to plan the actions in advance, since the future states are usually unpredictable. In concept, AutoDroid-V2 can deal with such dynamic user interfaces, but it may need to regenerate the plan for each step, leading to degraded efficiency. Our future work will focus on how to seamlessly integrate the step-wise reasoning approach with our script-based approach. Future agents are expected to flexibly switch between the two modes for different types of user interfaces, handling dynamic content effectively while maintaining efficiency.

6 Conclusion

We present a document-guided, script-based, end-to-end system named AutoDroid-V2 to support mobile task automation using on-device SLMs. Experiments show that the script-based method significantly improves the efficiency and performance of GUI agents. We believe this approach has the potential to enable the full deployment of GUI agents on devices, achieving accuracy comparable to that of cloud-based GUI agents.

Acknowledgement

References

[1] Tanzirul Azim, Oriana Riva, and Suman Nath. 2016. ULink: Enabling User-Defined Deep Linking to App Content. In *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys '16)*. Association for Computing Machinery, New York, NY, USA, 305–318. https://doi.org/10.1145/2906388.2906416

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

[3] Jinze Bai, Shuai Bai, Shusheng Yang, Shijie Wang, Sinan Tan, Peng Wang, Junyang Lin, Chang Zhou, and Jingren Zhou. 2023. Qwen-VL: A Versatile Vision-Language Model for Understanding, Localization, Text Reading, and Beyond. *arXiv preprint arXiv:2308.12966* (2023).

[4] William E Bishop, Alice Li, Christopher Rawles, and Oriana Riva. 2024. Latent State Estimation Helps UI Agents to Reason. *arXiv preprint arXiv:2405.11120* (2024).

[5] Richard A. Bolt. 1980. "Put-that-there": Voice and gesture at the graphics interface. In *Proceedings of the 7th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '80)*. Association for Computing Machinery, New York, NY, USA, 262–270. https://doi.org/10.1145/800250.807503

[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 (ECCV)*.

[7] Kanzhi Cheng, Qiushi Sun, Yougang Chu, Fangzhi Xu, Yantao Li, Jianbing Zhang, and Zhiyong Wu. 2024. SeeClick: Harnessing GUI Grounding for Advanced Visual GUI Agents. arXiv:cs.HC/2401.10935

[8] Xiang Deng, Yu Gu, Boyuan Zheng, Shijie Chen, Samuel Stevens, Boshi Wang, Huan Sun, and Yu Su. 2024. MIND2WEB: towards a generalist agent for the web. In *Proceedings of the 37th International Conference on Neural Information Processing Systems (NIPS '23)*. Curran Associates Inc., Red Hook, NY, USA, Article 1220, 24 pages.

[9] Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. 2024. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783* (2024).

[10] Georgi Gerganov et al. 2023. llama.cpp. https://github.com/ggerganov/llama.cpp

[11] Google. 2023. Create your own accessibility service. https://developer.android.com/guide/topics/ui/accessibility/service. Accessed: 2023-11-11.

[12] Boyu Gou, Ruohan Wang, Boyuan Zheng, Yanan Xie, Cheng Chang, Yiheng Shu, Huan Sun, and Yu Su. 2024. Navigating the digital world as humans do: Universal visual grounding for gui agents. *arXiv preprint arXiv:2410.05243* (2024).

[13] Suriya Gunasekar, Yi Zhang, Jyoti Aneja, Caio César Teodoro Mendes, Allie Del Giorno, Sivakanth Gopi, Mojan Javaheripi, Piero Kauffmann, Gustavo de Rosa, Olli Saarikivi, et al. 2023. Textbooks are all you need. *arXiv preprint arXiv:2306.11644* (2023).

[14] Izzeddin Gur, Hiroki Furuta, Austin V Huang, Mustafa Safdari, Yutaka Matsuo, Douglas Eck, and Aleksandra Faust. 2024. A Real-World WebAgent with Planning, Long Context Understanding, and Program Synthesis. In *The Twelfth International Conference on Learning Representations*. https://openreview.net/forum?id=9JQtrumvg8

[15] Wenyi Hong, Weihan Wang, Qingsong Lv, Jiazheng Xu, Wenmeng Yu, Junhui Ji, Yan Wang, Zihan Wang, Yuxiao Dong, Ming Ding, and Jie Tang. 2024. CogAgent: A Visual Language Model for GUI Agents. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. 14281–14290.

[16] Aaron Hurst, Adam Lerer, Adam P Goucher, Adam Perelman, Aditya Ramesh, Aidan Clark, AJ Ostrow, Akila Welihinda, Alan Hayes, Alec Radford, et al. 2024. Gpt-4o system card. *arXiv preprint arXiv:2410.21276* (2024).

[17] Albert Q Jiang, Alexandre Sablayrolles, Antoine Roux, Arthur Mensch, Blanche Savary, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Emma Bou Hanna, Florian Bressand, et al. 2024. Mixtral of experts. *arXiv preprint arXiv:2401.04088* (2024).

[18] Huiqiang Jiang, Qianhui Wu, Xufang Luo, Dongsheng Li, Chin-Yew Lin, Yuqing Yang, and Lili Qiu. 2024. LongLLMLingua: Accelerating and Enhancing LLMs in Long Context Scenarios via Prompt Compression. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, Lun-Wei Ku, Andre Martins, and Vivek Srikumar (Eds.). Association for Computational Linguistics, Bangkok, Thailand, 1658–1677. https://doi.org/10.18653/v1/2024.acl-long.91

[19] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. 2020. Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361* (2020).

[20] Sunjae Lee, Junyoung Choi, Jungjae Lee, Hojun Choi, Steven Y Ko, Sangeun Oh, and Insik Shin. 2023. Explore, select, derive, and recall: Augmenting llm with human-like memory for mobile task automation. *arXiv preprint arXiv:2312.03003* 3, 7 (2023), 8.

[21] Sunjae Lee, Junyoung Choi, Jungjae Lee, Munim Hasan Wasi, Hojun Choi, Steve Ko, Sangeun Oh, and Insik Shin. 2024. MobileGPT: Augmenting LLM with Human-like App Memory for Mobile Task Automation. In *Proceedings of the 30th Annual International Conference on Mobile Computing and Networking (ACM MobiCom '24)*. Association for Computing Machinery, New York, NY, USA, 1119–1133. https://doi.org/10.1145/3636534.3690682

[22] Yuanchun Li, Hao Wen, Weijun Wang, Xiangyu Li, Yizhen Yuan, Guohong Liu, Jiacheng Liu, Wenxing Xu, Xiang Wang, Yi Sun, et al. 2024. Personal llm agents: Insights and survey about the capability, efficiency and security. *arXiv preprint arXiv:2401.05459* (2024).

[23] Yuanchun Li, Ziyue Yang, Yao Guo, and Xiangqun Chen. 2017. DroidBot: A Lightweight UI-Guided Test Input Generator for Android. In *Proceedings of the 39th International Conference on Software Engineering Companion (ICSE-C '17)*. IEEE Press, 23–26. https://doi.org/10.1109/ICSE-C.2017.8

[24] Xiao Liu, Bo Qin, Dongzhu Liang, Guang Dong, Hanyu Lai, Hanchen Zhang, Hanlin Zhao, Iat Long Iong, Jiadai Sun, Jiaqi Wang, et al. 2024. AutoGLM: Autonomous Foundation Agents for GUIs. *arXiv preprint arXiv:2411.00820* (2024).

[25] Zhenyan Lu, Xiang Li, Dongqi Cai, Rongjie Yi, Fangming Liu, Xiwen Zhang, Nicholas D Lane, and Mengwei Xu. 2024. Small language models: Survey, measurements, and insights. *arXiv preprint arXiv:2409.15790* (2024).

[26] Xinbei Ma, Yiting Wang, Yao Yao, Tongxin Yuan, Aston Zhang, Zhuosheng Zhang, and Hai Zhao. 2024. Caution for the Environment: Multimodal Agents are Susceptible to Environmental Distractions. *arXiv preprint arXiv:2408.02544* (2024).

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

[28] 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:cs.LG/2307.10088

[29] Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, et al. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950* (2023).

[30] 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).

[31] Heyi Tao, Sethuraman T V, Michal Shlapentokh-Rothman, Tanmay Gupta, Heng Ji, and Derek Hoiem. 2024. WebWISE: Unlocking Web Interface Control for LLMs via Sequential Exploration. In *Findings of the Association for Computational Linguistics: NAACL 2024*, Kevin Duh, Helena Gomez, and Steven Bethard (Eds.). Association for Computational Linguistics, Mexico City, Mexico, 3693–3711. https://doi.org/10.18653/v1/2024.findings-naacl.234

[32] Chien Van Nguyen, Xuan Shen, Ryan Aponte, Yu Xia, Samyadeep Basu, Zhengmian Hu, Jian Chen, Mihir Parmar, Sasidhar Kunapuli, Joe Barrow, et al. 2024. A Survey of Small Language Models. *arXiv preprint arXiv:2410.20011* (2024).

[33] Junyang Wang, Haiyang Xu, Haitao Jia, Xi Zhang, Ming Yan, Weizhou Shen, Ji Zhang, Fei Huang, and Jitao Sang. 2024. Mobile-Agent-v2: Mobile Device Operation Assistant with Effective Navigation via Multi-Agent Collaboration. *arXiv preprint arXiv:2406.01014* (2024).

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

[35] Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Zhang, Yunzhu Li, Hao Peng, and Heng Ji. 2024. Executable Code Actions Elicit Better LLM Agents. In *ICML*. arXiv:2402.01030

[36] Hao Wen, Yuanchun Li, Guohong Liu, Shanhui Zhao, Tao Yu, Toby Jia-Jun Li, Shiqi Jiang, Yunhao Liu, Yaqin Zhang, and Yunxin Liu. 2024. AutoDroid: LLM-powered Task Automation in Android. In *Proceedings of the 30th Annual International Conference on Mobile Computing and Networking (ACM MobiCom '24)*. Association for Computing Machinery, New York, NY, USA, 543–557. https://doi.org/10.1145/3636534.3649379

[37] Hao Wen, Hongming Wang, Jiaxuan Liu, and Yuanchun Li. 2023. Droidbot-gpt: Gpt-powered ui automation for android. *arXiv preprint arXiv:2304.07061* (2023).

[38] Mingzhe Xing, Rongkai Zhang, Hui Xue, Qi Chen, Fan Yang, and Zhen Xiao. 2024. Understanding the weakness of large language model agents within a complex android environment. In *Proceedings of the 30th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*. 6061–6072.

[39] Jiajun Xu, Zhiyuan Li, Wei Chen, Qun Wang, Xin Gao, Qi Cai, and Ziyuan Ling. 2024. On-device language models: A comprehensive review. *arXiv preprint arXiv:2409.00088* (2024).

[40] An Yan, Zhengyuan Yang, Wanrong Zhu, Kevin Lin, Linjie Li, Jianfeng Wang, Jianwei Yang, Yiwu Zhong, Julian McAuley, Jianfeng Gao, et al. 2023. GPT-4V in Wonderland: Large Multimodal Models for Zero-Shot Smartphone GUI Navigation. *arXiv preprint arXiv:2311.07562* (2023).

[41] Keen You, Haotian Zhang, Eldon Schoop, Floris Weers, Amanda Swearngin, Jeffrey Nichols, Yinfei Yang, and Zhe Gan. 2024. Ferret-UI: Grounded Mobile UI Understanding with Multimodal LLMs. *arXiv preprint arXiv:2404.05719* (2024).

[42] Jinliang Yuan, Chen Yang, Dongqi Cai, Shihe Wang, Xin Yuan, Zeling Zhang, Xiang Li, Dingge Zhang, Hanzi Mei, Xianqing Jia, Shangguang Wang, and Mengwei Xu. 2024. Mobile Foundation Model as Firmware. In *Proceedings of the 30th Annual International Conference on Mobile Computing and Networking (ACM MobiCom '24)*. Association for Computing Machinery, New York, NY, USA, 279–295. https://doi.org/10.1145/3636534.3649361

[43] Chaoyun Zhang, Shilin He, Jiaxu Qian, Bowen Li, Liqun Li, Si Qin, Yu Kang, Minghua Ma, Qingwei Lin, Saravan Rajmohan, Dongmei Zhang, and Qi Zhang. 2024. Large Language Model-Brained GUI Agents: A Survey. arXiv:cs.AI/2411.18279 https://arxiv.org/abs/2411.18279

[44] Chaoyun Zhang, Shilin He, Jiaxu Qian, Bowen Li, Liqun Li, Si Qin, Yu Kang, Minghua Ma, Guyue Liu, Qingwei Lin, et al. 2024. Large language model-brained gui agents: A survey. *arXiv preprint arXiv:2411.18279* (2024).

[45] Chi Zhang, Zhao Yang, Jiaxuan Liu, Yucheng Han, Xin Chen, Zebiao Huang, Bin Fu, and Gang Yu. 2023. AppAgent: Multimodal Agents as Smartphone Users. arXiv:cs.CV/2312.13771

[46] Li Zhang, Shihe Wang, Xianqing Jia, Zhihan Zheng, Yunhe Yan, Longxi Gao, Yuanchun Li, and Mengwei Xu. 2024. LlamaTouch: A Faithful and Scalable Testbed for Mobile UI Task Automation. In *Proceedings of the 37th Annual ACM Symposium on User Interface Software and Technology (UIST '24)*. Association for Computing Machinery, New York, NY, USA, Article 132, 13 pages. https://doi.org/10.1145/3654777.3676382

[47] Xinyu Zhang, Huiyu Xu, Zhongjie Ba, Zhibo Wang, Yuan Hong, Jian Liu, Zhan Qin, and Kui Ren. 2024. PrivacyAsst: Safeguarding User Privacy in Tool-Using

[48] Yanzhe Zhang, Tao Yu, and Diyi Yang. 2024. Attacking Vision-Language Computer Agents via Pop-ups. *arXiv preprint arXiv:2411.02391* (2024).

[49] Zhuosheng Zhang and Aston Zhang. 2024. You Only Look at Screens: Multimodal Chain-of-Action Agents. In *Findings of the Association for Computational Linguistics: ACL 2024*, Lun-Wei Ku, Andre Martins, and Vivek Srikumar (Eds.). Association for Computational Linguistics, Bangkok, Thailand, 3132–3149. https://doi.org/10.18653/v1/2024.findings-acl.186

[50] Zhizheng Zhang, Xiaoyi Zhang, Wenxuan Xie, and Yan Lu. 2023. Responsible Task Automation: Empowering Large Language Models as Responsible Task Automators. arXiv:cs.AI/2306.01242

[51] Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric Xing, Hao Zhang, Joseph E Gonzalez, and Ion Stoica. 2023. Judging LLM-as-a-Judge with MT-Bench and Chatbot Arena. In *Advances in Neural Information Processing Systems*, A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine (Eds.), Vol. 36. Curran Associates, Inc., 46595–46623. https://proceedings.neurips.cc/paper_files/paper/2023/file/91f18a1287b398d378ef22505bf41832-Paper-Datasets_and_Benchmarks.pdf

[52] Qihao Zhu, Daya Guo, Zhihong Shao, Dejian Yang, Peiyi Wang, Runxin Xu, Y Wu, Yukun Li, Huazuo Gao, Shirong Ma, et al. 2024. DeepSeek-Coder-V2: Breaking the Barrier of Closed-Source Models in Code Intelligence. *arXiv preprint arXiv:2406.11931* (2024).

Large Language Model Agents. *IEEE Transactions on Dependable and Secure Computing* 21, 6 (2024), 5242–5258. https://doi.org/10.1109/TDSC.2024.3372777