AutoDroid: LLM-powered Task Automation in Android

Hao Wen[1], Yuanchun Li[1,2,†], Guohong Liu[1], Shanhui Zhao[1,*], Tao Yu[1,*], Toby Jia-Jun Li[3], Shiqi Jiang[4], Yunhao Liu[5], Yaqin Zhang[1], Yunxin Liu[1,2]

[1] Institute for AI Industry Research (AIR), Tsinghua University
[2] Shanghai Artificial Intelligence Laboratory
[3] Department of Computer Science and Engineering, University of Notre Dame
[4] Microsoft Research
[5] Global Innovation Exchange & Department of Automation, Tsinghua University

ABSTRACT

Mobile task automation is an attractive technique that aims to enable voice-based hands-free user interaction with smartphones. However, existing approaches suffer from poor scalability due to the limited language understanding ability and the non-trivial manual efforts required from developers or end-users. The recent advance of large language models (LLMs) in language understanding and reasoning inspires us to rethink the problem from a model-centric perspective, where task preparation, comprehension, and execution are handled by a unified language model. In this work, we introduce AutoDroid, a mobile task automation system capable of handling arbitrary tasks on any Android application without manual efforts. The key insight is to combine the commonsense knowledge of LLMs and domain-specific knowledge of apps through automated dynamic analysis. The main components include a functionality-aware UI representation method that bridges the UI with the LLM, exploration-based memory injection techniques that augment the app-specific domain knowledge of LLM, and a multi-granularity query optimization module that reduces the cost of model inference. We integrate AutoDroid with off-the-shelf LLMs including online GPT-4/GPT-3.5 and on-device Vicuna, and evaluate its performance on a new benchmark for memory-augmented Android task automation with 158 common tasks. The results demonstrated that AutoDroid is able to precisely generate actions with an accuracy of 90.9%, and complete tasks with a success rate of 71.3%, outperforming the GPT-4-powered baselines by 36.4% and 39.7%.

CCS CONCEPTS

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

KEYWORDS

Task Automation, Large Language Models, App Analysis

ACM Reference Format:
Hao Wen[1], Yuanchun Li[1,2,†], Guohong Liu[1], Shanhui Zhao[1,*], Tao Yu[1,*],, Toby Jia-Jun Li[3], Shiqi Jiang[4], Yunhao Liu[5], Yaqin Zhang[1], Yunxin Liu[1,2]. 2024. AutoDroid: LLM-powered Task Automation in Android. In *International Conference On Mobile Computing And Networking (ACM MobiCom '24), September 30–October 4, 2024, Washington D.C., DC, USA*. ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3636534.3649379

1 INTRODUCTION

Smartphone is one of the most sophisticated devices for individuals. With millions of mobile applications (apps for short) that have access to various embedded sensors and rich personal data, smartphones can be used for a lot of daily tasks such as ordering food, managing social networks, sensing and tracking health conditions, etc. Therefore, how to intelligently automate tasks on smartphones has become an attractive topic for mobile developers and researchers, due to its potential to significantly improve user experience and enable helpful virtual personal assistants.

The major approaches to mobile task automation can be classified as developer-based, demonstration-based, and learning-based techniques. Most existing commercial products (*e.g.* Siri, Google Assistant, Cortana, etc.) take a developer-based approach, which requires significant development efforts to

† Corresponding author: Yuanchun Li (liyuanchun@air.tsinghua.edu.cn).
* Shanhui Zhao and Tao Yu were student interns at Tsinghua University.

This work is licensed under a Creative Commons Attribution International 4.0 License.

ACM MobiCom '24, November 18–22, 2024, Washington D.C., DC, USA
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0489-5/24/09.
https://doi.org/10.1145/3636534.3649379

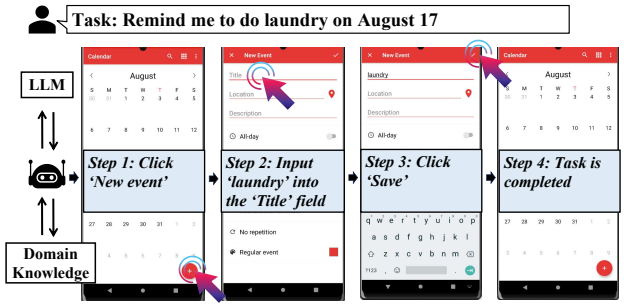


Figure 1: An illustration of LLM-powered mobile task automation. The agent interacts with the smartphone GUI to complete an arbitrary task, with the guidance of LLM and app domain knowledge.

support a new task. For example, to enable an automated task with Google Assistant, app developers need to identify the functionality which they want to trigger, configure and implement the corresponding intent, and register the intent with the assistant. When executing a task, the assistant uses natural language understanding (NLU) modules to map the user command to the intent, extract the intent parameters, and invoke the corresponding developer-defined function. Researchers have explored various methods to ease the development efforts. However, these methods still suffer from poor scalability, since they either require ad-hoc and/or large-scale human demonstrations of tasks (*e.g.* programming-by-demonstration approaches [2, 14, 15] and supervised learning approaches [3, 16, 35]) or require defining a clear reward for task completion (*e.g.* reinforcement learning approaches [10, 19, 39]). Due to the lack of scalability, there are few automated tasks supported today, even in the most popular apps.

Recently, the emergence of large language models (LLMs) like ChatGPT [28] and Claude [1] shows the promise in solving the scalability issue of task automation. Compared to traditional models, LLMs demonstrate unique abilities such as instruction following [36], step-by-step reasoning [44], and zero-shot generalization [11]. Such abilities are enabled by self-supervised learning on a huge corpus (more than 1.4 trillion tokens [38]) followed by tuning with human feedback. With these capabilities, researchers have managed to let LLMs invoke tools automatically, such as search engines [27], code interpreters [4], and third-party APIs [25, 30]. Similarly, using LLMs can potentially avoid the cumbersome manual efforts in mobile task automation. Meanwhile, connecting LLMs to smartphones can further unleash the power of LLMs in personal domains.

The goal of LLM-powered mobile task automation is to build an autonomous agent that can complete user-specified tasks by interacting with the smartphone. Although existing research [42] attempts to enable LLMs to understand mobile UIs, it simply relies on prompt engineering and does not utilize app-specific domain knowledge. AutoDroid combines the capabilities of LLM and the app-specific knowledge through dynamic app analysis, which enables it to handle arbitrary unseen tasks without manual efforts (illustrated in Figure 1). We identify three key problems to achieve this goal.

(1) **GUI Representation.** The input and output of task automators are graphical user interface (GUI) states and actions, unlike the natural language sentences LLMs can handle. To help LLMs better understand the GUI information and make precise interaction decisions, the GUI states and actions must be converted to text format while incorporating rich structured information.
(2) **Knowledge Integration.** Solving tasks with LLMs requires domain-specific knowledge about the applications. Unlike other tools studied in prior work (*e.g.* APIs) that LLMs can be easily configured to use, a smartphone app is usually a more complicated automata. LLMs need to navigate between different states to figure out how to complete the tasks.
(3) **Cost Optimization.** Querying LLMs is costly and compute-intensive, while completing a task with LLMs may involve many lengthy queries due to complexity of tasks and smartphone apps. Thus, it is desirable to optimize the efficiency of LLM queries to facilitate responsive task automation experience.

We introduce a mobile task automation framework, AutoDroid, to address the above problems. Overall, AutoDroid executes tasks by prompting the LLMs with an HTML-style text representation of GUI and querying for action guidance. To augment the LLMs with app knowledge, AutoDroid randomly explores the target apps and extracts UI transition graphs from them. By analyzing the UI states and transitions with LLMs, AutoDroid can convert the raw information to task completion knowledge, which is then integrated into the task automator by injecting foreseen functionalities into the prompts, matching relevant UI traces, or tuning the LLM parameters. The cost of querying LLMs is reduced by reducing and simplifying the queries based on app knowledge.

To systematically study the performance and challenges of LLM-powered task automation on Android, we build a benchmark with 158 manually labeled tasks from 13 open-source common mobile apps (Calendar, Messenger, Contacts, etc.). The source code and executable environments of the apps are provided for obtaining auxiliary information and reproducing task executions. The tasks include frequently asked how-to questions from the PixelHelp [16] dataset and common functionalities in the apps. For each task, we manually labeled the steps to complete the tasks, where each step is associated with both the GUI state and the GUI action. Our benchmark

evaluates the performance of LLM-powered task automation in terms of accuracy and cost.

We evaluate the effectiveness of our AutoDroid approach on the benchmark with different types of LLMs, including state-of-the-art online LLM services (GPT-3.5 and GPT-4) and open-source on-device LLMs (Vicuna). The results have demonstrated that AutoDroid can complete unseen tasks with a success rate of 71.3% with GPT-4, in which each action is selected with an accuracy of 90.9%. As compared with the baselines powered by off-the-shelf LLMs, the task completion rates are improved by 36.4% to 39.7%, and the average cost of querying LLMs is reduced by 51.7%.

Our work makes the following technical contributions:

(1) To the best of our knowledge, this is the first work on enhancing mobile task automation by combining LLMs and app-specific knowledge. We build a benchmark for this problem.
(2) We introduce a novel UI representation method that connects smartphones with LLMs, a task synthesis method for augmenting LLMs with app knowledge, and various LLM query optimization techniques to reduce the cost of task automation.
(3) Through a comprehensive evaluation, we demonstrate the effectiveness of our approach and the potential to advance the field of mobile task automation.

2 BACKGROUND AND MOTIVATION

2.1 Mobile Task Automation

The goal of mobile task automation is to automatically complete different kinds of tasks given by users. Its input is an arbitrary task described with natural language and a mobile app to execute the task. The output is a sequence of UI actions that can be executed 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 **UI state**, visible to users on their mobile device, is an arrangement of controls depicted through images and text, typically organized as a GUI tree. A **UI action**, performable by the user or an agent on the device's screen, is defined by a tuple *(target element, action type, value)*. *Target element* refers to a control in the UI state, such as a button, text box, input field, or slider. *Action type* represents how the target element is manipulated. We consider three main types of smartphone interactions, including "click", "input", and "swipe". The *value* field is the text content of the "input" action, which is empty for other action types.

In contrast to existing methods that utilize LLMs to summarize or respond to queries about individual mobile UIs [41, 42], automating mobile tasks demands the capability to plan task solutions and an in-depth understanding of which UIs are essential for task completion. AutoDroid aims to achieve multi-step task automation by leveraging app-specific knowledge. Furthermore, unlike most existing approaches that require significant developer/user efforts [2] to enable automated tasks, we aim to achieve unsupervised task automation, *i.e.* support the automation of arbitrary tasks on black-box apps (whose internal mechanisms are unknown) without human effort. However, we assume that the apps are available for automated analyses, *e.g.* exploring the states, crawling the content, and analyzing the code. Such an assumption is reasonable because the app packages are all available for download and static/dynamic app analysis techniques have been extensively studied before [21–24].

2.2 Large Language Models

Large language models (LLMs for short) mainly refer to the Transformer-based [40] language models that contain billions of parameters and are trained on massive amounts of text data, such as ChatGPT [28], GPT-4 [29], PaLM [6], LLaMA [38], etc. These models exhibit capabilities that are not present in smaller models, including mathematical reasoning [7], program synthesis [4], and multi-step reasoning [44]. Specifically, LLM can perform the tasks better than the benchmark models trained on dedicated datasets. The input of an LLM is a prompt, which is an instruction to guide its generation of responses. The prompt is tokenized into tokens (words or subwords) before being fed into the LLM.

Researchers are actively exploring methods to enhance the problem-solving capabilities of LLMs by incorporating reasoning skills [44] and tool utilization [25, 30, 49]. These efforts aim to enable LLMs to use tools by teaching them to call APIs or to synthesize codes. However, task automation in smartphone apps is more complex since it is often related to the environment without documented interfaces.

2.3 LLM meets Mobile Task Automation

We believe that incorporating LLMs into mobile task automation brings unique advantages and strengths to both fields.

First, **LLMs have the potential to significantly advance the applications of mobile task automation.** The voice-controlled intelligent personal assistants (IPA) are typical applications of mobile task automation, aiming to provide intelligent, efficient, hands-free user experience on mobile devices. Such applications are not only useful in smartphones, but also in many other scenarios, including automotive in-vehicle infotainment (IVI) systems [31], wearable fitness trackers [34, 46], and VR/AR devices [12]. To support IPA services, developers usually have to manually configure the task workflows, which is a cumbersome process even for experienced developers. Researchers have also attempted to build agents that can directly manipulate GUI elements like human users [15, 16, 35, 42]. However, they usually require a lot of human demonstrations, step-by-step instructions, or clearly-designed

Figure 2: An example task of *"Remove all the events in the Calendar"*. The agent needs to tap on *"more options"* and *"settings"* on the first and second GUI, which do not have a direct semantic association with the ultimate goal. This association can be grasped more easily with app analysis.

task-specific reward functions for task completion [10, 19]. LLM-based agents can be better at GUI task automation with their strong language comprehension and reasoning abilities.

Second, **equipping LLMs with smartphones can significantly augment their abilities.** LLMs are trained with large-scale public data that contains rich commonsense and world knowledge, while they have limited knowledge about individual users and limited abilities to provide personalized services. Smartphones have been an important part of daily life by helping people connect with others, stay organized with calendars, navigate and get directions, control smart-home devices, and so on. If LLMs learn to use smartphone apps and access data siloed in them, they could become much better personal assistants with access to the rich sensors and personal data in mobile apps.

Yet, applying LLMs to mobile task automation involves several challenges, including GUI representation, knowledge integration, and cost optimization. First, LLMs are only capable of processing plain text data and cannot directly handle GUI or interact with it. Although the GUI state in Android can be represented as text using the UI Hierarchy Viewer or Accessibility Services, it is usually lengthy (about 40k tokens on average for each UI state) and difficult for LLMs to interpret. Second, LLMs lack knowledge and experience about certain applications, which may lead to incorrect execution of instructions. Figure 2 shows an example where a deep understanding of the app is needed to complete the task. It is difficult to determine solely based on semantics and prior knowledge that clicking on *'more options'* and then *'settings'* on the first two screens will lead to the screen containing the option to *'delete all events'*. Therefore, relying solely on prompt engineering for LLMs to produce common-sense solutions can result in mistakes. A better approach might be to let LLMs investigate and learn from mobile apps, gaining practical experience prior to undertaking tasks for users. Third, using LLMs for task completion may be costly. The price of querying ChatGPT API [28] is $1.5 / 1000K tokens. Even if we can deploy a private LLM service, the computational cost is still high. For example, inferring a single token with LLaMA-7B [38] takes 6.7 billion FLOPs, and the whole process of task completion may use over 2000 tokens.

3 OUR APPROACH: AUTODROID

We introduce AutoDroid, an LLM-powered end-to-end mobile task automation system to solve the aforementioned challenges. In the offline stage, AutoDroid obtains app-specific knowledge by exploring UI relations and synthesizing simulated tasks. In the online stage, AutoDroid continuously queries the memory-augmented LLMs to obtain guidance on the next action. The task is completed by following the LLM-suggested actions. AutoDroid adopts several techniques to improve the task completion rate and optimize the query cost. Figure 3 illustrates the workflow.

We explain the functioning of AutoDroid using the example of automating tasks in a calendar app: During the offline stage, AutoDroid explores the app by randomly clicking buttons on the screen and records the result in a UI Transition Graph (UTG) memory (*Step 1*). Next, it traverses all the UI elements in the UTG and summarizes the tasks they can accomplish (*Step 2*). During online operation, when the user issues a command such as "delete all the events in the calendar", the *Prompt Generator* generates a prompt based on the task, the UI state description, and relevant information stored in the *App Memory*. This information includes instructions on how to navigate to the GUI page that contains the "delete events" option. Subsequently, the *Privacy Filter* replaces any sensitive information in the prompt to safeguard privacy. The filtered prompt is then sent to the LLM. Once the LLM provides an answer, the *Task Executor* parses the action that can be executed on the smartphone and verifies its security before performing it. If the executor deems the action to be potentially risky, such as "delete all the events" in this particular task, it will seek confirmation from the user before proceeding. We will explain how AutoDroid does all of these in the rest of this section.

3.1 Task-oriented UI Prompting

UI prompting refers to the process of representing underlying UI information in text and injecting it into the prompt to query the LLM. The goal of UI prompting is to clearly present the UI textual and structural content to the LLM and restrict the output of the LLM to predict only valid UI interactions. Figure 4 showcases an example of AutoDroid converting a GUI interface into a prompt while completing the task.

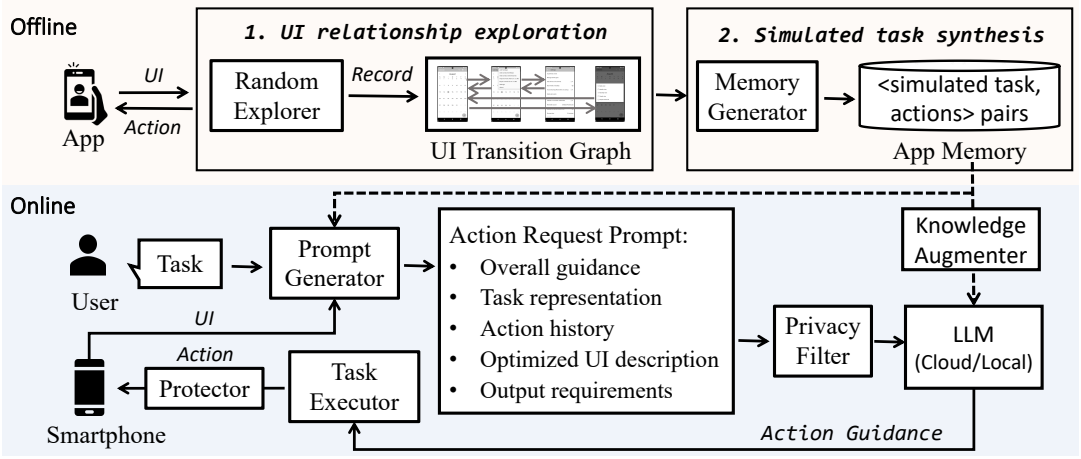


Figure 3: The workflow of AutoDroid.

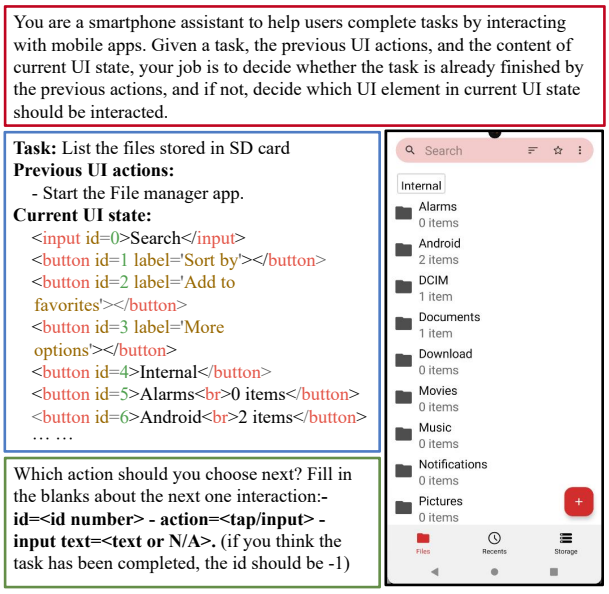


Figure 4: An illustration of the prompt used by AutoDroid. The content in red, blue, and green boxes are the overall guidance, the task representation, and the output requirements respectively. The 'current UI State' in the prompt refers to the UI displayed within the black box.

3.1.1 Converting GUI to Simplified HTML Representation. We develop a GUI parsing module to convert GUI to a simplified HTML representation that can be processed by LLMs. Researchers have found that LLMs are better at understanding HTML than natural-language-described UIs due to the large amount of HTML code in the training data of LLMs [42]. Therefore, we represent the GUI in HTML style, which can preserve the attribute information of UI elements. We use five types of HTML tags, namely <button>, <checkbox>, <scroller>, <input>, and <p>, which represent elements that can be clicked, checked, swiped, edited, and any other views

Table 1: The classes and properties of GUI elements

Class	Properties	Available action
<button>	ID, label, onclick (§3.2.2), text	click
<checkbox>	ID, checked, label, onclick (§3.2.2), text	check/uncheck
<scroller>	ID, scroll direction, label, text	scroll <direction>
<input>	ID, label, text, value	input <text>
<p>	ID, label, text	N/A

respectively. The properties included for each element are: ID (the order in which the element appears in the GUI tree), label (the content description that describes the function of the element), onclick (hints about the UI states that will be accessed upon clicking this button or checking/unchecking this checkbox, which will be introduced in §3.2.2), text (the text on the element), direction (scrolling direction, including up/down/left/right), checked (whether the checkbox is checked or not), value (the text that has been input to the text box). The classes and properties of GUI elements are shown in Table 1. We further simplify the DOM tree by pruning invisible elements and merging functionally equivalent elements, which will be introduced in §3.3. The texts of two merged UI elements are separated by "
", which represents a line break in HTML.

In our experiments, we observe that the agent generally does not proactively scroll on interfaces that can be scrolled vertically (shown in §6.2). However, having information about the scrolled interface is crucial for decision-making, especially when the target button is located on a scrolled portion of the interface that is not yet visible. Therefore, to provide the agent with comprehensive information, we need to include the components from the scrolled portion of the interface in the current UI state. To achieve this, for a given interface, AutoDroid first automatically scrolls through all scrollable components and records the information of the visible UI elements, and then provides this information to the LLM for decision-making. This approach offers two advantages. Firstly, it prevents LLMs from making blind selections when

they cannot see all the information on the interface. Secondly, it eliminates the need for LLMs to provide explicit instructions for scrolling, reducing the frequency of calling the LLM and lowering the associated computational overhead.

3.1.2 Restricting the Action Space with Selections. A key characteristic of UI task automation is that all agent actions need to be confined to the constraints of the underlying app *i.e.*, the agent can only perform actions of a supported action type on one of the existing UI elements. Thus, a challenge is to adapt LLMs, which are generative in nature, to such a discrete choice task. Hence, we impose the necessity for LLMs to produce results in a predetermined structure by completing the following requirement: "- id=<id number> - action=<tap/input> input text=<text or N/A> (in the event of task completion, id=-1)". LLMs must refrain from generating id or input in an arbitrary format.

3.2 Exploration-based Memory Injection

Exploration-based memory injection aims to provide app-related information to LLMs, enabling them to gain insights into apps, understand app utilization methods, and make effective decisions. However, there are challenges in utilizing automated app-related knowledge to assist LLMs in task automation, including: (i) The UI Transition Graph (UTG) obtained through random exploration cannot be directly processed by the LLM. (ii) Memory acquired solely through UI automation tools contains only UI and action data, without the essential information needed to directly enable task automation. This includes details about the specific UI elements and actions necessary to accomplish a particular task. (iii) An app may have numerous UI screens and UI elements (buttons, text boxes, etc.), exceeding the token length limit of LLMs if all of them are included in a prompt. To overcome these challenges, AutoDroid synthesizes simulated tasks based on the randomly explored UI graph. These simulated tasks serve as a guide for LLMs on how to accomplish a user task.

3.2.1 Simulated Task Generation. AutoDroid generates simulated tasks by analyzing the UI Transition Graph (UTG) as depicted in Figure 5. The UTG generated by the UI automator contains crucial information about the application, such as the connections between UIs and the presence of different UI elements on each screen. By summarizing the functionalities of all UI elements, we can gain a thorough understanding of the tasks that can be performed within the app and determine the corresponding UI elements required to execute them. As a result, AutoDroid parses all UI states and UI elements present in the UTG and extracts their functions by querying LLMs.

Specifically, UTG can be regarded as a directed graph, where the nodes and edges are all UI states and actions recorded by the random Explorer, denoted as $\mathbf{U}$ and $\mathbf{A}$ respectively. For each UI state $\mathbf{U}_i$, the memory generator queries LLM to summarize the functionalities of all the UI elements $\{e_i^j\}_{j=1}^{|U_i|}$, where $|U_i|$ denotes the number of elements in U_i. Note that AutoDroid only extracts the functionality of an element on the UI state that is closest to the initial UI if it appears on multiple UI states. After traversing all UI elements in the UTG, we obtain the *simulated task table* in the app memory containing n entries, where n represents the total number of UI elements on the UTG. Each entry in the table corresponds to a UI element e_i^j and is divided into three parts: <*Simulated task, UI states, UI elements*>. "Simulated task" represents the functionality of e_i^j that has been summarized by LLM, which can be perceived as a simulated task that can be completed by clicking this element. "UI elements" includes all the elements that were clicked, starting from the initial UI of the app and leading up to the attainment of U_i. "UI states" represents the sequence of UI states that were traversed from the initial UI state to U_i. This table provides the agent with information about the required operations to achieve each functionality, aiding the agent in planning how to complete a given task efficiently. Apart from the *simulated task table*, there is an additional table called the *UI function table* in the app memory. It provides a summary of the functionality associated with each UI state in the UTG. This information is obtained by querying the LLM to summarize the function of each UI state.

3.2.2 Augmenting Prompts with App Memory. The most straightforward approach to leveraging app-specific knowledge is to incorporate the app memory directly into the prompt, which can provide guidance to the LLM. However, this may exceed the maximum token limit of the LLM such as 4097 tokens for GPT-3.5 [28]. In many cases, only a few UI elements are necessary to complete a user's task. Hence, we selectively incorporate the most relevant UI information into the prompt.

AutoDroid determines the importance of a UI element in the app memory based on the similarity between its simulated task and the current user task. We use an embedding model (Instructor-XL [33]) that maps natural language sentences to a fixed-dimension embedding, where the embeddings of sentences with similar meanings are closer. The cosine similarity between the embeddings of the simulated task S and the current task T is denoted as $sim(E(S), E(T))$. Then, we can find k most similar simulated tasks in the app memory, denoted as $\{S_1, S_2, ..., S_k\}$. For each S_i, we can retrieve the corresponding *UI states* and the *UI elements* from the "simulated task table" in the app memory. In the online stage, if the current UI matches one of the *UI states* associated with S_i, we give hints about the UI elements that the random explorer interacted with in this UI state. This helps LLMs understand the outcome of interacting with the elements. Specifically,

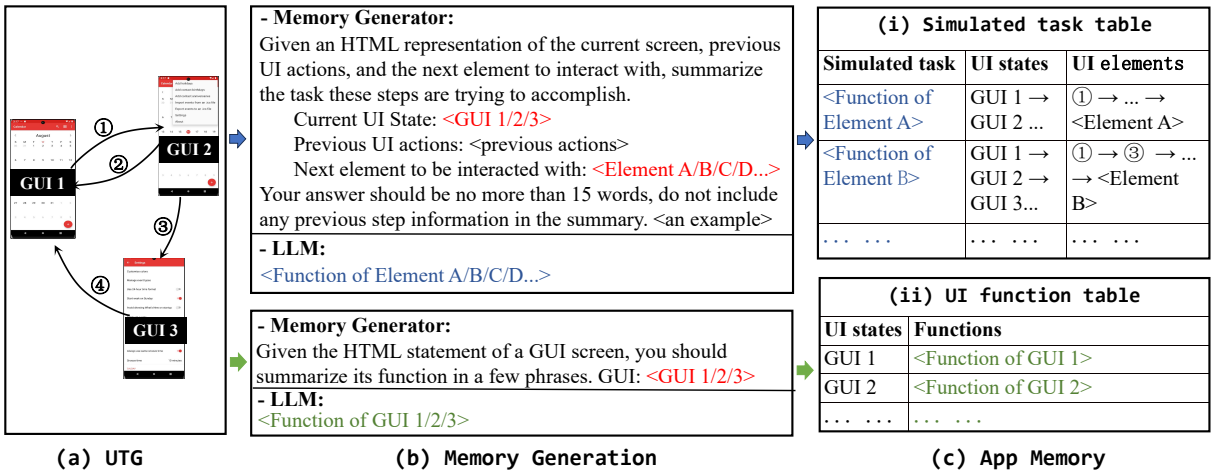


Figure 5: Workflow of offline simulated task synthesis. Given the UI Transition Graph (UTG), Memory Generator synthesizes a simulated task for each UI element with LLMs, then records the task-states-elements in the App Memory.

the prompt generator of AutoDroid will add a new property "onclick" to the HTML UI statement in the prompt (shown in Table1). In HTML, "onclick" is used to describe the event that will occur when users click a button, link, or image. In our prompt, the content of the "onclick" property refers to the functionality of the *UI states* that will be accessed after clicking this element, which is most relevant to completing the S_i. Algorithm 1 shows how to augment prompt with $\{S_1, S_2, ..., S_k\}$ and app memory M.

Algorithm 1 Prompt Augmentation

Input: Current user task T, k most similar simulated tasks in the app memory $\mathcal{S} = \{S_1, S_2 ... S_k\}$, App Memory M
Output: Final GUI state after completing the task T

function ONLINE-MAIN:
  $Guide \leftarrow GenerateGuide(M, \mathcal{S})$
  while T not completed **do**
    $UI \leftarrow$ Current GUI of SmartPhone
    if $UI \in Guide.UIs$ **then**
      $UI.elements.hint \leftarrow Guide.UI.elements.Function$
    end if
    $Prompt \leftarrow PromptGenerator(T, UI, History)$
    $Action \leftarrow LLM(Prompt)$
    $TaskExecutor.execute(Action)$
    $History.insert(Action)$
  end while
return *Current GUI*
end function

function GENERATEGUIDE($M, \mathcal{S}$):
  for each S_i of $\mathcal{S}$ **do**
    for each UI_i^j of $M.Simulated\_Task\_Table\{S_i\}$ **do**
      $Guide.UI_i^j.Element_i^j.hint \leftarrow M.Function\{UI_i^{j+1, j+2...}\}$
    end for
  end for
return *Guide*
end function

Take the task shown in Figure 2 as an example. Given the task *"Remove all the events in the calendar"*, AutoDroid can retrieve in the app memory and find the simulated task of the *"delete all events"* button in GUI 3 to be a relevant task. Additionally, AutoDroid can find that clicking *"more options"* and *"settings"* in GUI 1 and GUI 2 can lead to the target button. Therefore, if the current UI screen is GUI 1, the HTML description of *"more options"* in GUI 1 will change from "<button label='More options'></button>", to "<button label='More options' onclick='*navigate to GUIs that can: 1.add contact holidays and anniversaries, import and export events, manage settings, 2.Delete all events in the app, manage event reminders, etc.*'></button>".

3.2.3 Tuning Local LLM with App-specific Data. AutoDroid can also utilize smaller local LLMs (*e.g.* Vicuna-7B [5]) to make decisions, as a cost-effective alternative to larger on-cloud LLMs (*e.g.* GPT-3.5 [28]). However, the reasoning ability of these smaller LLMs is weaker than on-cloud LLMs, leading to a noticeable decrease in accuracy. It is observed that local LLMs still exhibit suboptimal performance even with the prompt augmentation methods introduced in §3.2.2. Researchers have found that fine-tuning using domain-specific data is an effective way to improve small LLM's abilities [5, 36]. Therefore, we can augment smaller LLMs by fine-tuning using app-specific data.

A key challenge in our scenario is how to generate high-quality (question, answer) pairs to fine-tune the LLM. A naive way is to directly synthesize these data pairs from the simulated task table of the app memory. For a simulated task S, the memory generator records a sequence of UI states $\{U_1, U_2, ..., U_k\}$ and the UI elements $\{e_1, e_2, ..., e_k\}$ to complete it. We can directly generate k data pairs $(q_i, a_i)_{i=1}^k$ based on this record. Specifically, q_i is a prompt generated based on the

task S, previous UI actions $\{A_1, A_2, ..., A_k\}$ (where the target elements are $\{e_1, e_2, ..., e_k\}$ and the action type is *click*), and the current UI state U_i. Then, the description of the action A_i can be the answer a_i. The rationale behind this approach is that: Based on the generation process of the app memory, we already know that when transitioning from interface U_i to complete task S, action A_i needs to be performed. Therefore, the correct answer of which action to choose given the state U_i should be A_i.

However, the answers generated in this way only include <*target element, action type, value*>, lacking detailed information or context. Thus, it is difficult for the local LLM to learn how to choose the correct action based on the prompt. If we include the reasons for choosing the target action in the answers, it will enhance the local LLM's understanding and enable it to learn how to reason based on the current task and UI [9]. Thus, we can ask larger LLMs (such as GPT-4 [29]) to answer the reason why A_i is chosen to complete task S, and prompt it to reason in a step-by-step manner like a Zero-shot Chain-of-Thought (0-shot CoT) [11]. The prompt sent to the larger LLM is mainly the same as Figure 4. Additionally, we provide the correct action to choose A_i, and prompt the LLM to reason about the correct action by changing the "output requirements" part to the following format:

Your answer should always use the following format: 1. Completing this task on a smartphone usually involves these steps: <?>. 2. Analyses of the relations between the task and the previous UI actions and current UI state: 3. Based on the previous actions, is the task already finished? <Y/N>. The next step should be <?/None>. 4. Can the task be proceeded with the current UI state? <Y/N>. Fill in the blanks about the next one interaction: - id=<id number> - action=<tap/input> - input text=<text or N/A>.

The answer to the above questions can be used as the answer in the (question, answer) pair for fine-tuning the local LLM. The thinking and reasoning data generated by these larger LLMs contains rich information and knowledge. Using it as answers to fine-tune smaller LLMs can enable it to mimic the emergent reasoning abilities of the large model. Besides leveraging the knowledge from larger LLMs, fine-tuning LLMs with app-specific data also has the bellow two advantages: (i) Learning from the UTG and incorporating the insights gained from it. (ii) Let smaller LLMs generate answers that adhere to the desired format instead of unrestricted formatting in the answers.

3.3 Multi-granularity Query Optimization

We observe that the primary source of overhead in AutoDroid arises from querying LLMs. Consequently, reducing the frequency of LM queries for each task will result in a reduction of AutoDroid's overhead. Additionally, as a more granular approach, pruning unnecessary tokens in the prompt, we can effectively decrease the computational cost of LLM.

3.3.1 Pruning Tokens by Merging Functionally Equivalent Elements. The HTML statement of UI described in §3.1 contains a lot of redundant information, which will increase the number of tokens and cause the LLM to overlook the most useful information. Therefore, We adopt two techniques to reduce the length of the text: First, we prune the elements without any visual or textual information (such as background or container items). Second, we merge functionally equivalent UI elements into one element and separate the originally different elements with a "
" delimiter, which means a line-break-like spacing in HTML. We merge UI elements based on two rules: (i) Based on UTG: If operating on these two UI elements leads to the same interface, we combine them into a single component. Specifically, if the starting and ending points of two edges representing actions in the UTG are the same, we merge the components they operate on. (ii) Based on UI tree analysis: We merge the non-interactive (plain text or image) UI leaf nodes sharing the same interactive ancestor (button, checkbox, text field, etc.) in the UI tree. For example, in the GUI screenshot shown in Figure 4, "Alarms" and "0 items" are two single plain-text nodes in the GUI tree that have a common clickable ancestor. Thus, we can merge them into an HTML statement: "<button id=5>Alarms
0 items</button>" instead of two single statements "<button id=5>Alarms</button>" and "<button id=6>0 items</button>".

3.3.2 Reducing Query Times by Shortcuts and GUI Merging. GUI merging is to include several GUI states into one prompt if LLMs need them all to make decisions. The automatic scrolling introduced in §3.1.1 can accomplish this by skipping the intermediate steps like "scroll down". Without automatic scrolling, AutoDroid has to query LLMs at least twice to touch an element within the GUI after swiping, involving both scrolling and clicking. After merging the scrolled UIs into one prompt, we only need to call LLMs once and get the action *"Scroll down to Button A and touch it"*.

The shortcut is to execute simple actions directly with the help of the app memory. Although some steps are crucial and require a large model to make decisions, others are straightforward and do not require it. So if we can identify steps that are simple enough so that a local embedding model [33] can make decisions, we can reduce the number of queries. Specifically, let T, E, and $\{S_1, S_2, ...\}$ denote the user task, the embedding model, and the simulated tasks respectively. If we find $sim(E(S_k), E(T)) > \gamma$ where $S_k = \arg\max_{S_i \in \mathcal{S}} sim(E(S_i), E(T))$, then S_k is very similar to T, and accomplishing S_k is straightforward because we have a series of actions $\{A_k^1, A_k^2, ...\}$ in the app memory that navigate from the initial UI state to S_k. Thus we can perform S_k by the

task executor without calling LLM. γ is a hyper-parameter, the larger the value of γ, the stricter our criteria for selecting similar simulated tasks become. We observe that even if the shortcut navigates to UI states unrelated to the task, LLM is still able to identify issues and quickly navigate to the correct UI states.

4 IMPLEMENTATION

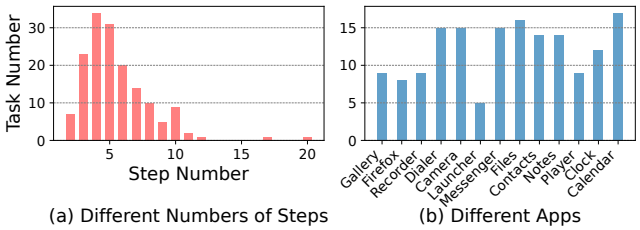
We implement AutoDroid using Python and Java. The local LLM Vicuna [5] is fine-tuned using PyTorch.

Identifying Risky Actions. Some actions may potentially alter local or server data, or cannot be undone once performed. These actions are considered risky and require user confirmation before being executed by the agent. For example, before calling a contact, AutoDroid needs to first prompt the user to verify the correctness of the action. If the user notices any errors in the number about to be dialed, they can manually make the necessary modifications. AutoDroid accomplishes this by prompting the LLM to identify risky actions, *i.e.* appending the sentence *"If this action potentially leads to a change of user data or server state that requires user confirmation, please answer requires_confirmation=Yes)"* to the prompt. In addition, AutoDroid also utilizes key phrases on the UI, such as "warning", to further identify potentially risky actions.

Eliding Private Information. We add a privacy filter that can mask the private information in the query. During online processing, it runs a Personal Identifiable Information (PII) scanner [26] that can detect sensitive information in the prompt, including name, phone number, email address, etc. This personal information is replaced with non-private words (*e.g.* "<name>"→"Alice") before sending the prompt to the cloud. After receiving the response from LLMs, AutoDroid maps the special words back to the original ones before parsing actions.

5 BENCHMARK

We introduce DroidTask, an Android Task Automation benchmark suite designed to evaluate the performance of end-to-end mobile task automation systems. DroidTask consists of 158 high-level tasks extracted from 13 popular apps. What sets our benchmark apart is that it not only provides tasks and corresponding GUI action traces but also offers the exploration memory and environment for the underlying apps. Agents can actively interact with the environment during the offline stage, gathering information about the apps and recording UTGs. All 13 apps used to collect the tasks are installed, granted necessary permissions, and can reproduce the GUI action traces in our environment. We will release the environment in the form of an Android Virtual Machine Snapshot, allowing researchers to restore the exact environment in which we collected our data. While previous benchmarks [3, 16, 35] also provide tasks and corresponding actions, they lack a reproducible environment. However, with the emergence of LLM-powered task automation methods [30, 47], which often require dynamic information about the environment for decision-making, our benchmark offers greater convenience for evaluating the performance of autonomous agents on mobile phones.



(a) Different Numbers of Steps (b) Different Apps

Figure 6: The distribution of tasks in DroidTask across different numbers of steps (a) and different apps (b).

We develop a system for collecting datasets that can interact with Android smartphones. The selected apps primarily consist of common mobile tools (such as contacts, dialer, camera, calendar, etc.) from F-Droid, a free and open-source app platform. For each app, we ask annotators to provide a list of 5-15 tasks described in natural language. To complete each task, annotators interact with the smartphone through a desktop computer in an iterative manner. During each iteration, the system displays the smartphone's user interface (UI) in its current state to the annotator, along with a list of available actions. Annotators can also directly observe the actual state of the smartphone. They can choose an action from the following options: 1. Touch <Button ID>, 2. Input <input text> to <EditBox ID>, 3. Swipe <Scroller ID> <direction> in the terminal. The distribution of tasks is shown in Figure 6.

6 EVALUATION

We conduct experiments to examine the accuracy and cost of AutoDroid in mobile task automation.

6.1 Experimental Setup

Dataset. We mainly evaluate AutoDroid on DroidTask (mentioned in §5). We also utilize MoTiF [3] dataset to train the baseline methods and fine-tune the LLMs. MoTiF [3] is a large-scale mobile app task dataset with more than 4.7k tasks (excluding tasks without valid demonstrations). It also provides the screenshot and the tree-based representation of the GUI screens that annotators interacted with when completing these tasks, but lacks the exploration environment of the apps.

Hardware. We evaluate the end-to-end performance of AutoDroid on a OnePlus ACE 2 Pro with 8 3.2 GHz ARM-based cores (Snapdragon 8 Gen2 CPU) and Adreno™ 740 GPU. The local LLM Vicuna-7B [5] is deployed on the smartphone

based on Machine Learning Compilation for LLM (MLC LLM) [37]. Additionally, it is deployed on an edge server equipped with 1 NVIDIA 3090 24G GPU to assess inference latency in an edge computing context. The Vicuna-7B model is fine-tuned on an 8× A100 80GB server for about 4 GPU hours.

Baselines. We choose META-GUI [35] and an existing LLM-based design for UI task automation [42] (referred to as LLM-framework) as our main baselines. META-GUI [35] is a training-based conversational agent on mobile GUI that can accomplish various tasks. We train it on the MoTiF [3] dataset. LLM-framework [42] is an LLM-based framework that enables diverse language-based interactions with mobile UIs. We also implement two relatively simple baselines, random performer (randomly selecting one UI element within each UI screen) and similarity-based (selecting the UI element that is semantically closest to the task using a SOTA embedding model [33]) performer.

Metrics. Given a sequence of UIs $\{U_1, U_2, ..., U_n\}$ in which human annotators performed actions $\mathcal{A} = \{A_1, A_2, ..., A_n\}$ to complete a task T, if one agent can make a sequence of decisions $\hat{\mathcal{A}} = \{\hat{A}_1, \hat{A}_2, ..., \hat{A}_n\}$ on $\{U_1, U_2, ..., U_n\}$, we use below two metrics to measure its performance:

(i) **Action Accuracy**: The ratio of the action $\hat{A}_i$ matching the ground-truth actions A_i, namely $P(\hat{A}_i = A_i)$. One action is right only if the target UI element and input text ("null" if there is no need to input) are both right. This metric reflects the ability of the agent to make correct decisions based on the available information.

(ii) **Completion Rate**: The probability of completing all the actions in one sequence correctly, namely $P(\hat{\mathcal{A}} = \mathcal{A})$. This metric reflects the probability of the agent being able to consistently and successfully complete a task.

6.2 Action Accuracy

We first evaluate the action accuracy of AutoDroid. The open-sourced LLM Vicuna-7B [5] is fine-tuned using the generated app-specific data, as mentioned in §3.2.3. For the closed-source LLM such as GPT-3.5 [28] and GPT-4 [29], which can not be fine-tuned directly, we augment them with automatically generated app memory, as mentioned in §3.2.2. The temperature of the LLMs is set to a lower value of 0.25 to encourage creativity while preventing it from being overly random. The action accuracy of AutoDroid and baselines is listed in Table 2. AutoDroid outperforms baselines on every action type, resulting in an overall accuracy improvement of 37.6%. Among all the actions, clicking is the simplest, only requiring the decision of the element ID. On the other hand, scrolling and inputting necessitate specifying the direction or value of the UI element, and determining completion entails considering all previous actions.It is also observed that with the LLM going larger, LLM-based methods outperform the model trained from scratch [35]. This is because the model has only been exposed to apps and tasks from specific datasets [3]. Thus, it will not perform well on new apps and tasks in the DroidTask. However, by accumulating sufficient prior knowledge and incorporating our memory integration, LLMs can engage in rational reasoning on how to solve problems on new apps. For scrollable UIs, AutoDroid will first browse and traverse all the components on the screen, eliminating the need for the "scroll" action. From the scroll accuracy of the baseline, it is observed that the probability of the agent actively selecting this action is very low. Thus, browsing and traversing first can improve the overall accuracy of the agent.

The reason AutoDroid outperforms baselines is: (i) AutoDroid prunes and merges UI elements, which reduces the action space (from 36.4 to 13.2 choices per GUI state on average). (ii) The exploration-based memory can enhance the LLM with domain-specific knowledge about the apps, which will be detailed in §6.4. (iii) The output format of the fine-tuned model is aligned more closely with the format requirements specified in the output requirements. If the output is not standardized, the task executor would be unable to extract or recognize the element ID and action.

We further analyze why and how AutoDroid fails on some steps. We randomly sample 20 failure cases by AutoDroid (using GPT-4 [29] as the LLM), and categorize 3 typical failure causes, as explained below: 1. Multiple Correct Choices. In certain cases, there can be multiple valid ways to complete a task. Annotators may not be able to exhaustively list all the possible ways to complete a task, and if the agent attempts a different approach than what the annotators specified, it may be deemed incorrect. 2. Unable to accurately determine if the task has been completed. Sometimes AutoDroid mistakenly considers a task completed when it detects the presence of a specific UI element. 3. Lack of understanding of the GUI. AutoDroid occasionally overlooks important information or hints in the UI and makes decisions based on its prior experience. For example, in the task *"open the camera and record a short video, name it 'test.mp3' "*, the agent only needs to input 'test' into the "name" box. This is because the GUI indicates that the file extension '.mp3' is already displayed in the "file type" box. However, AutoDroid still selects 'test.mp3' as the input to the "name" box.

6.3 Task Completion Rate

The Task Completion Rate of AutoDroid and LLM-framework [42] is shown in Figure 7 (a). Note that we do not include the completion determination step for clear comparison. AutoDroid outperforms baseline by 40.5%, 26.4%, and 39.7% for Vicuna-7B, GPT-3.5, and GPT-4 respectively. We also show the completion rate of AutoDroid with and without

Table 2: Action accuracy of AutoDroid and baselines on DroidTask. Rand: Randomly selecting actions, Sim: Similarity-based action prediction, LLM-F: LLM-framework [42], Complete: Determining completion.

				Vicuna-7B		GPT-3.5		GPT-4	
Action	**Rand**	**Sim**	**MG**	LLM-F	AutoDroid	LLM-F	AutoDroid	LLM-F	AutoDroid
Click	2.3%	35.1%	25.3%	15.2%	74.5%	58.1%	72.1%	65.4%	91.2%
Input	0	0	0	0	40.0%	5.0%	62.5%	27.5%	82.5%
Scroll	2.5%	0	0	8.2%	N/A	0	N/A	0.6%	N/A
Complete	2.5%	0	N/A	4.4%	5.7%	0	41.8%	0	93.7%
Overall	2.3%	20.8%	22.4%	11.3%	**57.7%**	34.7%	**65.1%**	54.5%	**90.9%**

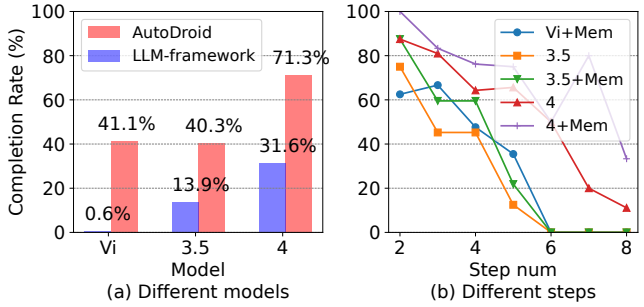


Figure 7: Task completion rate of AutoDroid with different LLMs and with varying numbers of steps. Vi: Vicuna-7B, 3.5: GPT-3.5, 4: GPT-4.

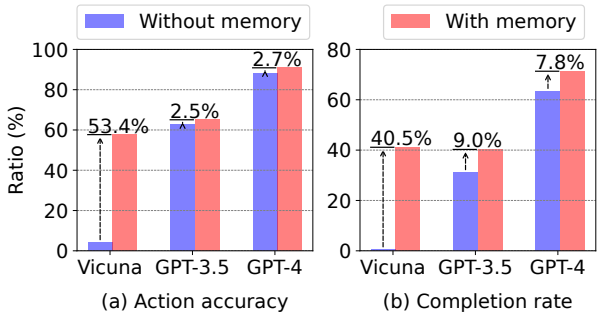


Figure 8: Action accuracy and task completion rate of AutoDroid, with and without memory augmentation.

memory augmentation in Figure 7 (b). As the number of steps increases, the overall completion rate decreases. This is because (i) the probability of each step being executed correctly decreases. (ii) Tasks that involve multiple steps often have multiple approaches to completion (e.g., creating a new contact by entering either the name or the phone number first). However, human annotators typically only annotate one approach, which can lead to the model's solution being mistakenly judged as incorrect. The actual completion rate in the real-system can be higher than the reported results, but we do not include real-system results since determining task completion can be ambiguous.

6.4 Ablation Study

6.4.1 Memory Injection. The action accuracy and task completion Rate of AutoDroid with and without memory is shown in Figure 8. We can observe that the improvement in overall completion rate is much higher than the improvement in single-step accuracy. This is because the introduction of memory allows LLMs to make crucial single-step decisions (such as the example in Figure 2). Although these critical steps account for a small proportion of all the action steps, they are essential for successfully completing certain tasks. Moreover, it can be observed that smaller models benefit more from the inclusion of memory in terms of task completion rate. This is because smaller models possess less prior knowledge, thus requiring more guidance from application-specific knowledge. However, even for larger models, the incorporation of memory remains meaningful. Limited model capacity cannot store the vast and ever-growing knowledge present in the world, making it difficult to stay updated on the evolving usage patterns of new applications. Therefore, automatic exploration and recording of their usage patterns play a crucial role in enabling LLMs to effectively use applications.

Table 3: Action accuracy and completion rate of AutoDroid based on Vicuna-7B with different fine-tuning techniques. Original: Vicuna-7B without fine-tuning. CoT: Fine-tuning with zero-shot chain-of-thought. Mo: Incorporating a small portion of MoTiF [3] dataset for fine-tuning.

Metric	Original	AutoDroid	No Mo	No CoT	No Mo&CoT
Action	11.3%	57.7%	51.9%	20.6%	51.9%
Completion	0.6%	41.1%	29.8%	0.6%	31.6%

6.4.2 Zero-shot Chain of Thought Fine-tuning. The action accuracy and task completion rate of AutoDroid based on Vicuna-7B [5] fine-tuned with and without Zero-shot Chain-of-Thought (0-shot CoT) [11] is shown in Table 3. Since the app memory automatically generated by AutoDroid only contains clicking and checking actions, the LLM fine-tuned merely on the app memory is poor on inputting and adjusting whether the task has been completed. Hence, we incorporate a small portion of manually annotated data for fine-tuning.

Specifically, we add only the input and completion judgment data from MoTiF dataset [3] into the app memory dataset. Note that the app and task in the MoTiF dataset [3] are unrelated to our dataset. Thus, adding this portion of data will not result in any test data leakage. It simply enables the model to learn to input and to determine the task's completion.

Vicuna-7B [5] fine-tuned with Zero-shot Chain-of-Thought data generated by app memory mixed with a small portion of MoTiF [3] (*AutoDroid*) can achieve 57.7% action accuracy and 41.1% completion rate on DroidTask, with an input accuracy of 40.0%. Without MoTiF [3] data (*"No Mo"*), the fine-tuned model can achieve 51.9% action accuracy, and the inputting accuracy is 0%. We observe that when there are no CoT and no MoTiF data (*"No Mo&CoT"*), the fine-tuned LLM can achieve a high accuracy rate with simple click actions, and it can generally handle tasks that involve only clicking. However, once the MoTiF dataset is introduced (*"No CoT"*) to teach the LLM additional types of actions (such as input and task completion judgments), the LLM is heavily misled by the completion of judgment tasks. As a result, it outputs a significant number of "task completed" instead of selecting actions correctly. Consequently, the action accuracy drops from 51.9% to 20.6%.

6.5 Cost Analysis

Runtime Cost. AutoDroid reduces runtime overhead by addressing two aspects: reducing the number of tokens per query and minimizing the frequency of query. In Figure 9 (a), we show the count for each prompt length. Our baseline [42] includes only visible leaf nodes in the UI tree, and contains 625.3 tokens within each prompt on average. AutoDroid merges functionally equivalent nodes in the UI tree and further simplifies the expression of properties, reducing the token count by nearly half (339.0 on average). There are two main benefits: (i) Reducing token length can significantly decrease the model's inference latency. (ii) For calling on-cloud LLM API, it can reduce costs. For example, for GPT-3.5 and GPT-4, the cost can be reduced from $0.938 and $18.76 to $0.509 and $10.17 every 1000 queries respectively on average.

In Table 4, we randomly select five baseline prompts and find the corresponding prompts optimized by AutoDroid. We measure their real latency on the Vicuna-7B [5] deployed on the smartphone as well as on the edge server. Our optimized prompt reduces inference latency by 21.3% on average. Note that the inference latency of LLMs on the smartphone and the edge server primarily depends on the number of output tokens. Therefore, when deploying the LLM on a mobile device, we do not require the LLM to output the Chain-of-Thought but rather output in the original manner shown in Figure 4. In the case of P_5, due to the excessive length of the baseline, it was truncated after outputting only one word, resulting in minimal inference latency.

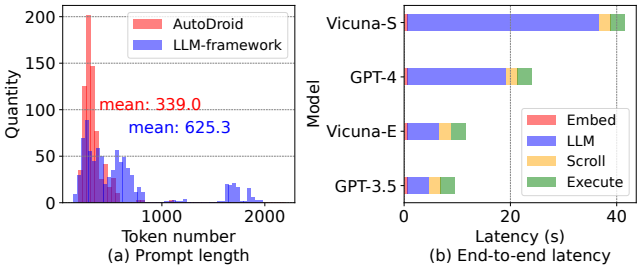


Figure 9: Overhead of AutoDroid and LLM-framework [42]. Left: The number of prompts with different token counts. Right: The component of per-step latency of AutoDroid based on 3 LLMs respectively. Vicuna-E: Vicuna-7B deployed on the edge server. Vicuna-S: Vicuna-7B deployed on the smartphone.

Table 4: Per-step inference cost of AutoDroid with Vicuna-7B deployed on the OnePlus ACE 2 Pro smartphone. LLM-F: LLM-framework [42]. $P_{1\sim5}$: Five random prompts from LLM-framework.

Prompt length / Inference latency	P_1	P_2	P_3	P_4	P_5
LLM-F input length (token)	252	401	460	559	719
AutoDroid input length (token)	299	280	233	177	233
On device LLM-F latency (s)	40.6	50.2	63.9	64.9	36.0
On device AutoDroid latency (s)	39.7	30.8	33.9	39.7	22.7
On cloud LLM-F latency (s)	4.4	5.5	6.4	16.1	6.5
On cloud AutoDroid latency (s)	4.2	8.8	4.9	5.3	5.6

Figure 9 (b) shows the component of per-step latency of AutoDroid. The Vicuna-7B model is deployed on the smartphone and on the edge server. On-cloud GPT-3.5 and GPT-4 models are accessed by making API calls. The embedding model [33] is deployed on an edge 1080 Ti GPU with 11 GB memory. Note that the latency in calling GPT-3.5 and GPT-4 is significantly influenced by network conditions, server load, and so on. Therefore, we make 10 measurements to calculate the average latency, but there still remains a considerable degree of instability. Calling LLMs (*"LLM"*) accounts for the majority of the latency, with 42.1%, 51.9%, 77.6%, and 87.1% of the latency based on GPT-3.5, Vicuna-7B (on-server), GPT-4, and Vicuna-7B (on-device) respectively. Therefore, reducing LLM calls can largely reduce the end-to-end overhead. Besides, Embedding the task and searching the most similar UI element (*"Embed"*) account for only 1.7% of the overhead, and only needs to be executed once for every task. Therefore, the overheads of finding the shortcuts and memory injection are acceptable.

We also conducted experiments on saving the number of calls based on merging GUI and shortcuts. On average, AutoDroid reduces LLM calls by 1.2 per task resulting in an overall decrease of 13.7% in the total number of calls using the GUI merging technique. Our shortcuts correctly guide

Table 5: Action accuracy and completion rate of AutoDroid based on GPT-4 with privacy information replacement and security confirmation on 5 apps in DroidTask. Priv: Privacy information replacement, Sec: Security confirmation.

Metric	Original	+Priv	+Sec	+Priv&Sec
Acc	92.9%	89.9%	89.9%	89.9%
Completion	75.4%	69.9%	68.5%	69.9%

LLMs in 75% of cases. Considering only the correct shortcuts, we save 38.02% of the number of steps, with an average savings of 1.73 steps per task.

Offline Preparation Cost. For every app, it takes about 0.5-1 hour to generate the UI Transition Graph (UTG), which is then analyzed to synthesize simulated tasks based on LLMs, taking about 5-10 minutes. Finally, the simulated tasks are mapped into high-dimensional vectors by an embedding model [33] for runtime lookup, which typically takes about 10 seconds on a desktop computer. The offline preparation is a one-time process and does not need to be performed again at runtime.

6.6 Influence of Security/Privacy Filtering

We ask the annotators of DroidTask to also determine whether each action could potentially change the state of the user or the app. If so, we consider the action to be risky and prompt the user to confirm whether to proceed with the action. We evaluate AutoDroid's accuracy in detecting risky actions in five apps that may contain risky actions (contacts, dialer, SMS messenger, clock, and calendar). We consider risky actions as positive examples and AutoDroid achieved a precision of 75.0% and a recall of 80.5%. We further show the influence of adding privacy information replacing and security confirmation into the prompt in Table 5. When privacy replacement and security confirmation are added, a decrease in accuracy and completion rate can be observed, which is acceptable.

7 RELATED WORK

UI Understanding and Automation. There has been growing interest in using machine learning techniques to comprehend and summarize user interfaces, enabling use cases such as accessibility and task-oriented bots. Key areas of research include: 1) Semantic analysis of GUIs to summarize functions [13, 18], interpret UI elements' purposes [17, 48], and address user questions related to the GUI [42, 43]. It is crucial for various interaction tasks such as UI automation and accessibility. 2) Mapping user instructions to UI elements [15, 16, 35]. These methods aim to to select the most relevant GUI elements for given tasks. 3) Mobile UI task automation [45, 49]. These methods build agents to complete tasks for users by performing actions on the GUI. AutoDroid, on the other hand, leverages the UI transition memory to complete complex, multi-step tasks on smartphones. The memory can help agents to understand richly informative UIs and the usage of apps, and augment the LLMs in reasoning and planning. After the first release of AutoDroid, there were various LLM-based UI agents proposed, which had been comprehensively summarized in a recent survey [20].

Augmented LLM. Although LLMs excel in tasks like question answering and text generation, they are still constrained by the information they can store in their fixed set of weights and context length. Therefore, researchers are augmenting LLMs with different tools, such as web browser [8, 27], APIs [25, 30], and other DNN models [32]. Unlike existing approaches that often depend on public APIs, our method does not require custom APIs, which are uncommon in mobile applications.

8 DISCUSSION

Randomness of LLMs. We can set the 'temperature' hyperparameter to 0 for consistent responses. But setting temperature too small will inhibit innovative answers, thereby potentially reducing the performance of our system. In our experiments, we set the temperature to 0.25. And we observe a 2.1% accuracy reduction when we set the 'temperature' of GPT-3.5 to 0. Conversely, increasing the temperature to 0.7 boosted action accuracy by 3.8%.

Increased latency limits the practical use of AutoDroid. Our work could be extended by a collaborative approach between LLMs and smaller models. We could call LLMs only once for each task to create a guideline based on the filtered domain-specific knowledge about the app. Subsequently, smaller models could be employed to associate these guidelines with UI elements [16, 35]. Introducing an instruction cache could further reduce latency by storing and reusing common commands, minimizing the need for repeated LLM invocations.

9 CONCLUSION

We present an LLM-powered mobile task automation system that can support arbitrary tasks without manual efforts. Experiment results have shown that our method can achieve effective task automation, outperforming existing training-based and LLM-based baselines. We believe that the synergy between the commonsense knowledge of LLMs and domain-specific knowledge in mobile apps can potentially bring truly intelligent and helpful personal assistants into reality.

ACKNOWLEDGEMENT

This work is supported by the National Key R&D Program of China (No.2022YFF0604501), NSFC (No.62272261), and Tsinghua University (AIR)–AsiaInfo Technologies (China) Inc. Joint Research Center.

REFERENCES

[1] Anthropic. 2023. Claude. https://www.anthropic.com/product.

[2] 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

[3] Andrea Burns, Deniz Arsan, Sanjna Agrawal, et al. 2022. A Dataset for Interactive Vision Language Navigation with Unknown Command Feasibility. In *European Conference on Computer Vision (ECCV)*.

[4] Mark Chen, Jerry Tworek, Heewoo Jun, et al. 2021. Evaluating Large Language Models Trained on Code. (2021). arXiv:cs.LG/2107.03374

[5] Wei-Lin Chiang, Zhuohan Li, et al. 2023. Vicuna: An Open-Source Chatbot Impressing GPT-4 with 90%* ChatGPT Quality. https://lmsys.org/blog/2023-03-30-vicuna/

[6] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, et al. 2022. PaLM: Scaling Language Modeling with Pathways. arXiv:cs.CL/2204.02311

[7] Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, et al. 2021. Training Verifiers to Solve Math Word Problems. *arXiv preprint arXiv:2110.14168* (2021).

[8] Xiang Deng, Yu Gu, Boyuan Zheng, et al. 2023. Mind2Web: Towards a Generalist Agent for the Web. arXiv:cs.CL/2306.06070

[9] Cheng-Yu Hsieh, Chun-Liang Li, Chih-Kuan Yeh, et al. 2023. Distilling step-by-step! outperforming larger language models with less training data and smaller model sizes. *arXiv preprint arXiv:2305.02301* (2023).

[10] Peter C Humphreys, David Raposo, Tobias Pohlen, et al. 2022. A data-driven approach for learning to control computers. In *International Conference on Machine Learning*. PMLR, 9466–9482.

[11] Takeshi Kojima, Shixiang (Shane) Gu, Machel Reid, et al. 2022. Large Language Models are Zero-Shot Reasoners. In *Advances in Neural Information Processing Systems (NeurIPS 2022)*, Vol. 35. 22199–22213.

[12] Sunjae Lee, Hoyoung Kim, Sijung Kim, et al. 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 (MobiCom '22)*. Association for Computing Machinery, New York, NY, USA, 690–702. https://doi.org/10.1145/3495243.3560522

[13] Gang Li and Yang Li. 2023. Spotlight: Mobile UI Understanding using Vision-Language Models with a Focus. In *The Eleventh International Conference on Learning Representations*. https://openreview.net/forum?id=9yE2xEj0BH7

[14] 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.

[15] 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 (MobiSys '18)*. Association for Computing Machinery, New York, NY, USA, 96–109. https://doi.org/10.1145/3210240.3210339

[16] Yang Li, Jiacong He, Xin Zhou, Yuan Zhang, and Jason Baldridge. 2020. Mapping Natural Language Instructions to Mobile UI Action Sequences. In *Annual Conference of the Association for Computational Linguistics (ACL 2020)*. https://www.aclweb.org/anthology/2020.acl-main.729.pdf

[17] Yang Li, Gang Li, Luheng He, Jingjie Zheng, Hong Li, and Zhiwei Guan. 2020. Widget Captioning: Generating Natural Language Description for Mobile User Interface Elements. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Association for Computational Linguistics, Online, 5495–5510. https://doi.org/10.18653/v1/2020.emnlp-main.443

[18] Yang Li, Gang Li, Xin Zhou, Mostafa Dehghani, and Alexey A. Gritsenko. 2022. VUT: Versatile UI Transformer for Multimodal Multi-Task User Interface Modeling. In *International Conference on Learning Representations (ICLR)*. https://openreview.net/forum?id=rF5UoZFrsF4

[19] 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 (SIGIR '21)*. Association for Computing Machinery, New York, NY, USA, 1420–1430. https://doi.org/10.1145/3404835.3462905

[20] Yuanchun Li, Hao Wen, Weijun Wang, et al. 2024. Personal LLM Agents: Insights and Survey about the Capability, Efficiency and Security. *arXiv preprint arXiv:2401.05459* (2024).

[21] 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

[22] Yuanchun Li, Ziyue Yang, Yao Guo, and Xiangqun Chen. 2019. Humanoid: A deep learning-based approach to automated black-box android app testing. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1070–1073.

[23] Chieh-Jan Mike Liang, Nicholas D. Lane, Niels Brouwers, et al. 2014. Caiipa: Automated Large-Scale Mobile App Testing through Contextual Fuzzing. In *Proceedings of the 20th Annual International Conference on Mobile Computing and Networking (MobiCom '14)*. Association for Computing Machinery, New York, NY, USA, 519–530. https://doi.org/10.1145/2639108.2639131

[24] Hao Lin, Jiaxing Qiu, Hongyi Wang, Zhenhua Li, Liangyi Gong, Di Gao, Yunhao Liu, et al. 2023. Virtual Device Farms for Mobile App Testing at Scale: A Pursuit for Fidelity, Efficiency, and Accessibility. In *Proceedings of the 29th Annual International Conference on Mobile Computing and Networking (MobiCom '23)*. Association for Computing Machinery.

[25] Pan Lu, Baolin Peng, Hao Cheng, Michel Galley, Kai-Wei Chang, Ying Nian Wu, Song-Chun Zhu, and Jianfeng Gao. 2023. Chameleon: Plug-and-Play Compositional Reasoning with Large Language Models. arXiv:cs.CL/2304.09842

[26] Microsoft. 2023. PII Codex. [Online]. Available at: https://github.com/EdyVision/pii-codex.

[27] Reiichiro Nakano, Jacob Hilton, Suchir Balaji, et al. 2022. WebGPT: Browser-assisted question-answering with human feedback. (2022). arXiv:cs.CL/2112.09332

[28] OpenAI. 2022. ChatGPT. [Online]. Available at: https://openai.com/blog/chatgpt/.

[29] OpenAI. 2023. GPT-4 Technical Report. arXiv:cs.CL/2303.08774

[30] Shishir G. Patil, Tianjun Zhang, Xin Wang, and Joseph E. Gonzalez. 2023. Gorilla: Large Language Model Connected with Massive APIs. arXiv:cs.CL/2305.15334

[31] Preeti Rani and Rohit Sharma. 2023. Intelligent Transportation System Performance Analysis of Indoor and Outdoor Internet of Vehicle (IOV) Applications Towards 5G. *Tsinghua Science and Technology* (2023). https://doi.org/10.26599/TST.2023.9010119

[32] Yongliang Shen, Kaitao Song, Xu Tan, Dongsheng Li, Weiming Lu, and Yueting Zhuang. 2023. HuggingGPT: Solving AI Tasks with ChatGPT and its Friends in Hugging Face. arXiv:cs.CL/2303.17580

[33] Hongjin Su, Weijia Shi, Jungo Kasai, et al. 2023. One Embedder, Any Task: Instruction-Finetuned Text Embeddings. In *Findings of the Association for Computational Linguistics: ACL 2023*. Association for Computational Linguistics, Toronto, Canada, 1102–1121. https://aclanthology.org/2023.findings-acl.71

[34] Boyuan Sun, Qiang Ma, Shanfeng Zhang, Kebin Liu, and Yunhao Liu. 2017. iSelf: Towards Cold-Start Emotion Labeling Using Transfer Learning with Smartphones. *ACM Trans. Sen. Netw.* 13, 4, Article 30 (sep 2017), 22 pages. https://doi.org/10.1145/3121049

[35] 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).

[36] Rohan Taori, Ishaan Gulrajani, Tianyi Zhang, et al. 2023. Stanford Alpaca: An Instruction-following LLaMA model. https://github.com/tatsu-lab/stanford_alpaca. *GitHub repository* (2023).

[37] MLC team. 2023. MLC-LLM. https://github.com/mlc-ai/mlc-llm

[38] Hugo Touvron, Thibaut Lavril, Gautier Izacard, et al. 2023. LLaMA: Open and Efficient Foundation Language Models. (2023). arXiv:cs.CL/2302.13971

[39] Daniel Toyama, Philippe Hamel, Anita Gergely, et al. 2021. AndroidEnv: A Reinforcement Learning Platform for Android. abs/2105.13231 (2021). arXiv:cs.LG/2105.13231 http://arxiv.org/abs/2105.13231

[40] Ashish Vaswani, Noam Shazeer, Niki Parmar, et al. 2017. Attention is All You Need. In *Proceedings of the 31st International Conference on Neural Information Processing Systems (NIPS'17)*. Curran Associates Inc., Red Hook, NY, USA, 6000–6010.

[41] Sagar Gubbi Venkatesh, Partha Talukdar, and Srini Narayanan. 2022. UGIF: UI Grounded Instruction Following. *arXiv preprint arXiv:2211.07615* (2022).

[42] 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 (CHI '23)*. Association for Computing Machinery, New York, NY, USA, Article 432, 17 pages. https://doi.org/10.1145/3544548.3580895

[43] Bryan Wang, Gang Li, Xin Zhou, Zhourong Chen, Tovi Grossman, and Yang Li. 2021. Screen2Words: Automatic Mobile UI Summarization with Multimodal Learning. In *The 34th Annual ACM Symposium on User Interface Software and Technology (UIST '21)*. Association for Computing Machinery, New York, NY, USA, 498–510. https://doi.org/10.1145/3472749.3474765

[44] Jason Wei, Xuezhi Wang, Dale Schuurmans, et al. 2022. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. In *Advances in Neural Information Processing Systems*, S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh (Eds.), Vol. 35. Curran Associates, Inc., 24824–24837. https://proceedings.neurips.cc/paper_files/paper/2022/file/9d5609613524ecf4f15af0f7b31abca4-Paper-Conference.pdf

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

[46] Jian Xu, Qingqing Cao, Aditya Prakash, Aruna Balasubramanian, and Donald E. Porter. 2017. UIWear: Easily Adapting User Interfaces for Wearable Devices. In *Proceedings of the 23rd Annual International Conference on Mobile Computing and Networking (MobiCom '17)*. Association for Computing Machinery, New York, NY, USA, 369–382. https://doi.org/10.1145/3117811.3117819

[47] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2023. ReAct: Synergizing Reasoning and Acting in Language Models. In *International Conference on Learning Representations (ICLR)*.

[48] Xiaoyi Zhang, Lilian de Greef, Amanda Swearngin, 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 (CHI '21)*. Association for Computing Machinery, New York, NY, USA, Article 275, 15 pages. https://doi.org/10.1145/3411764.3445186

[49] 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