Towards Test Generation from Task Description for Mobile Testing with Multi-modal Reasoning

Hieu Huynh
Hai Phung
Hao Pham
minhhieu2214@gmail.com
20127018@student.hcmus.edu.vn
20127155@student.hcmus.edu.vn
University of Science, VNU-HCM
Ho Chi Minh city, Vietnam

Tien N. Nguyen
University of Texas at Dallas
Dallas, Texas, USA
tien.n.nguyen@utdallas.edu

Vu Nguyen*
University of Science, VNU-HCM
Katalon Inc.
Ho Chi Minh city, Vietnam
nvu@fit.hcmus.edu.vn

Abstract

In Android GUI testing, generating an action sequence for a task that can be replayed as a test script is common. Generating sequences of actions and respective test scripts from task goals described in natural language can eliminate the need for manually writing test scripts. However, existing approaches based on large language models (LLM) often struggle with identifying the final action, and either end prematurely or continue past the final screen.

In this paper, we introduce VISIDROID, a multi-modal, LLM-based, multi-agent framework that iteratively determines the next action and leverages visual images of screens to detect the task's completeness. The multi-modal approach enhances our model in two significant ways. First, this approach enables it to avoid prematurely terminating a task when textual content alone provides misleading indications of task completion. Additionally, visual input helps the tool avoid errors when changes in the GUI do not directly affect functionality toward task completion, such as adjustments to font sizes or colors. Second, the multi-modal approach also ensures the tool not progress beyond the final screen, which might lack explicit textual indicators of task completion but could *display a visual element* indicating task completion, which is common in GUI apps. Our evaluation shows that VISIDROID achieves an accuracy of 87.3%, outperforming the best baseline relatively by 23.5%. We also demonstrate that our multi-modal framework with images and texts enables the LLM to better determine when a task is completed.

1 Introduction

In mobile testing, testers often simulate user's tasks on the application when performing end-to-end testing. Coming from the requirements, each task includes a natural language description of its goal to be performed on the GUI of the application under test (AUT). One common approach to generating test cases and scripts for automated testing on the AUT is to record user's actions. The testing tools (e.g., Espresso [2], Appium [1], and [3]) are used to launch the AUT in a controlled test environment. Testers manually carry out the actions in the test environment to achieve the task, allowing for interactions with the AUT, e.g., tapping buttons, entering text, navigating between screens, etc. As the actions are performed, the test environment records all the steps. This recording includes details like which elements are interacted with (e.g., buttons, text fields) and what inputs are provided. The recorded

sequence of actions is then transformed into a test script by the test environment. The tester sometimes could directly write the test script to complete the task. This script can be modified or executed in future testing sessions in a controlled test environment.

Unfortunately, manually performing and recording actions or writing scripts for every task in mobile testing can be time-consuming and error-prone [10, 23]. It also hinders the prospect of automating the software development process from requirement engineering to testing. Automating this process speeds up test creation and scales it, while also reducing the likelihood of mistakes [22]. Early methods to automate this process introduced structured, domain-specific languages such as e.g., Gherkin and Cucumber [9] to assist testers in defining actions and behaviors.

With advances in machine learning (ML), large language models (LLMs)-based solutions have emerged to automate this process [15, 18, 26, 32]. These LLM-based approaches rely more on the textual content of the page to decide the next action and if the task is accomplished. While achieving successes, they have experienced a common shortcoming in *identifying the final action, and either stop too early or continue beyond the actual final screen*. The reason is that the DOM structure/texts do not always accurately reflect the screen content. For example, in a photo app, switching from the front camera to the back camera may not have a clear textual indication of completion, apart from the visual change on the screen.

This paper introduces VISIDROID, a *multi-modal*, automated mobile testing framework that enhances LLMs' comprehension of a GUI page by analyzing both its visual representation and textual content. *Our hypothesis is that this multi-modal approach can enable an LLM to make more precise decisions about subsequent actions as well as the decision on the completeness of the actions for the given task*. The multi-modal approach enhances an LLM in two significant ways. First, it enables the LLM to avoid prematurely terminating a task when textual content alone provides misleading signals of task completion. For instance, a confirmation screen may contain text similar to that of a task's final screen, potentially causing confusion. However, incorporating visual cues on the final screen allows the tool to recognize that the task is not yet completed. Additionally, visual input helps the LLM avoid errors when changes in the GUI do not directly affect functionality toward task completion, such as adjustments to font sizes or colors. Second, the multi-modal approach also ensures that the LLM stops at the correct action and does not progress beyond the final screen. For example, the final GUI might lack explicit textual indicators of task completion but

---

*Corresponding author.

could display visual elements that enable the LLM to recognize the task's conclusion accurately. *These visual changes can complement the textual contents to signify the completeness of the task in GUI apps.* For example, a checkbox is marked as WiFi is enabled for the Wifi-activation task, which is easily recognized on the screen.

Specifically, VISIDROID takes the goal of a task described in natural language and automatically interacts with the testing environment to perform the actions on the app and generate the test scripts without human intervention. Initially, an LLM is prompted to decide the first action based on the description. The test environment then executes this action. The image of the GUI page, along with its textual content, is processed by another LLM in a multi-modal setup. This LLM is responsible for deciding whether the task has been completed and for termination.

VISIDROID also combines *multi-modal framework* with *task memory* and *persistent memory* (i.e., accumulated experience) that work together to enhance the LLM's ability to learn from recent interactions and make informed decisions for next actions or completion. *Short-term (task) memory* allows the LLM to retain details on the most recent steps and their outcomes, ensuring essential context and continuity. The task memory scheme provides the LLM with information on past actions, GUI descriptions, and changes to the page after each action. Meanwhile, *persistent memory* leverages a broader set of patterns and knowledge from past interactions, enabling the LLM to recognize similarities with past sequences of actions in analogous situations encountered earlier to apply to the current one. For this, VISIDROID uses a self-reflection mechanism to reflect on the executed task, assessing how the task is achieved and the experience gained. Both (in)correct sequences are stored as persistent memories to assist VISIDROID in future tasks.

We evaluate VISIDROID using the dataset provided by Wen *et al.* [15], including 150 goal-based tasks on 12 Android applications. Out of 150 tasks, VISIDROID generated 131 exact-match sequences of actions, resulting in an *exact-match accuracy* of **87.3%**, outperforming the best baseline *AutoDroid* [15] (accuracy of 70.7%), which is a **23.5%** relative improvement. The generated exact-match action sequences are also used to create test scripts for automated regression testing, achieving a *successful execution rate* of **82.4%**.

In brief, this paper makes the following key contributions:

1. Multi-modal Reasoning of LLMs. VISIDROID supports automated generation of sequences of actions and test cases/scripts by integrating vision capability of LLMs with memory mechanisms.

2. Extensive Empirical Evaluation. Our evaluation demonstrates VISIDROID's effectiveness and its out-performance over the state-of-the-art approaches. Our data and code are available [4].

2 Motivation

Fig. 1 shows an example of an Android app, which is launched within a testing environment, e.g., Espresso or Appium. A tester is provided with a task of *"setting an alarm at 8:00 AM"*. (S)he is expected to manually write the test script or manually perform the actions on the Android GUI through one or multiple screens within the testing environment to achieve the task. (S)he achieves the task description by clicking the "Alarm" button, then entering the specified time and clicking on the "OK" button for confirmation (SCREEN 3)
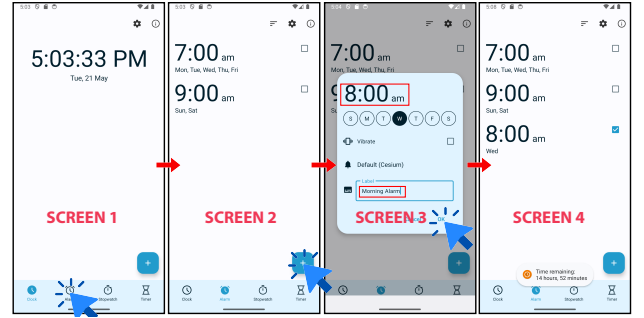

Figure 1: Task: *"Set Alarm at 8:00am"*. Existing methods relying on DOM/text contents incorrectly stop at SCREEN 3 because its DOM/texts contain *"8:00am"*, which matches the phrase in the task's description (the final screen is SCREEN 4)

to reach the final SCREEN 4. The test environment records the sequence of actions into a test script for later test generation and reuse.

Several approaches were proposed to interact with the test environment and generate a sequence of actions on the Android app from the task description, e.g., GPTDroid [26], AutoDroid [37], DroidAgent [42], Guardian [32]. They struggle in determining the termination of a task. In Fig. 1, the correct final screen is SCREEN 4. However, the LLM-based models rely on the DOM content of the screen, leading to *prematurely and incorrectly stopping at SCREEN 3 in Fig. 1*, and do not select the "OK" button to confirm alarm creation. The reason is that SCREEN 3 also contains the text "8:00am" that *matches the same phrase in the task's textual description*, which might confuse the LLMs. If SCREEN 4 is provided, the LLMs might recognize that the task was finished as the toggle for the alarm was set at 8:00 am. See more cases where visual cues help in Section 5.2.4.

2.1 Key Ideas

*2.1.1 Key Idea 1 [***Multi-modal Framework for Mobile Test Script Generation***].* Building on the notion that "a picture is worth a thousand words," we harness LLMs' vision capabilities in a multi-modal framework to enhance their comprehension of a GUI page by analyzing both its image and content. *Our multi-modal approach aims to enable LLMs to make more accurate decisions on the completeness of the action sequence.* LLMs are expected to link the textual description of an image to the visual cues, ensuring that a task proceeds or stops correctly. In the same way, they can connect the task's goal to the visual layout of each screen in the Android app. As a result, we expect them to better identify the final screen when the task is completed as follows. First, **relying solely on textual signals can lead to errors, such as ending a task early when an earlier screen includes texts that are misleadingly similar to those on the final screen**. For example, a payment confirmation message might resemble a transaction completion screen. By integrating visual cues, like specific button layouts or icons on the final screen, the LLM avoids such mistakes. Second, **the visual changes are common to signify task completeness in GUI apps, while the DOM/text contents do not always reflect well the layouts**. The LLM can confidently identify the ending of a task using visual cues when textual indicators are absent. For instance, a final screen might lack completion text but include an image, such as a
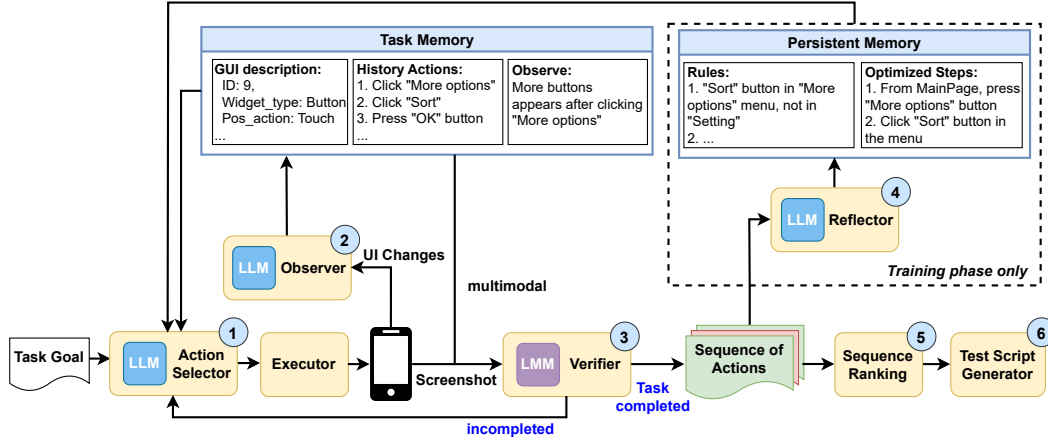
Figure 2: VɪsɪDʀoɪᴅ: Test Script Generation from Task Descriptions with Multi-Modal Reasoning

check mark signifying the completion of a configuration setting, a change in the current screen signifying the front and back cameras were switched, or the current color theme of the app was changed. This visual cue prevents the LLM from over-executing unnecessary actions beyond the task's end. Visual input also helps it ignore irrelevant changes, e.g., modifications to colors or formatting.

*2.1.2 Key Idea 2 [***Iterative LLM-based Agent Planning***]. In-spired by the ReAct planning [40], we develop an iterative LLM-based agent planning scheme for VɪsɪDʀoɪᴅ. In VɪsɪDʀoɪᴅ, the central LLM agent decides on the next action based on the observations of (1) the current GUI content observations, (2) the short-term memory of previous states and actions, (3) descriptions of content changes, and (4) the long-term memory of (in)correct action sequences. This chosen action is then executed on the current screen by a testing environment, and the resulting new content is used for the next iteration until the task is considered accomplished.

*2.1.3 Key Idea 3 [***Combination of Short- and Long-term Memories***]. A crucial aspect of navigating complex GUI tasks is main-taining awareness of previous steps and the evolving states of the GUI pages. For that, we integrate into VɪsɪDʀoɪᴅ a task memory mechanism that serves as a *short-term memory* of the current task execution, providing the LLMs with recent actions, GUI descriptions and any changes to the GUI page following each action. This task memory allows the LLMs to retain recent actions and their outcomes, enabling more informed decision-making in the next action. For example, in an Android app for retails, after a sequence of *"searching"* for a product, *"adding to cart"* the selected one, and *"going to cart"*, a likely next action is to *"proceed to checkout"*.

We also record the entire sequence of actions taken, along with their corresponding success or failure outcomes. We call it *persis-tent memory*, which draws from a wider set of past experiences, allowing the model to recognize patterns, similarities with prior failures/successes, and apply successful strategies. For example, a correct sequence *A* (*"Browse-Purchase-Pay"*) is recorded as fol-lows. Action 1: Tap on "Browse Categories.", Action 2: Select "Elec-tronics.", Action 3: Tap on a product (e.g., "Wireless Headphones"), Action 4: Tap "Add to Cart.", Action 5: Go to "Cart.", Action 6: Tap

"Proceed to Checkout.", Action 7: Enter payment details, Action 8: Tap "Confirm Purchase.", and Final Screen: "Order Confirmation" screen is displayed. Similarly, there is an incorrect sequence *B* (*"Search-Purchase-Pay"*) for a purchase task with payment: Action 1: Tap on "Search" and enter "Electronics.", Action 2: Tap on "Wireless Headphones.", Action 3: Tap "Buy Now.", Action 4: Tap "Confirm Purchase" *without entering payment details*, and Final Screen: "Error: Payment details missing" screen is shown.

Persistent memory enables the model to evaluate the correctness of actions by analyzing the outcomes of prior sequences. It can learn the patterns from successes: *The sequence includes steps like "Add to Cart" and "Proceed to Checkout" before confirming purchase. Entering payment details is necessary before final confirmation.* It can also learn the patterns from failures: *Skipping critical steps, such as entering payment details or adding items to the cart, leads to errors. Direct actions like "Buy Now" or "Confirm Purchase" without prerequisites often fail.* If the current task involves ensuring that the user reaches the "Confirmation" screen, persistent memory will guide the model to prioritize sequences similar to the correct sequence *A* while avoiding pitfalls shown in *B*. The model adjusts its behavior to ensure the correct sequence of actions is followed.

2.2 VɪsɪDʀoɪᴅ Overview

Figure 2 displays VɪsɪDʀoɪᴅ's workflow. It takes the description of a task as input, automatically interacts with the testing environment to perform the actions on the app, and generates the test scripts without human intervention. Its three key modules include:

1) **Iterative LLM-based Action Sequence Generation with a Multi-Modal Framework.** The iterative process revolves around the LLM agent Action Selector ①1 as follows. From the given app and task description, VɪsɪDʀoɪᴅ initiates the iterative process using the specified starting GUI page. At each iteration, the LLM agent en-gages in reasoning to determine the next action based on performed actions, UI changes, and experiences from persistent memory. This action, with the current GUI page, is fed into an executor, which executes the app at the selected page and presents the next page for the next iteration. Example actions include scrolling, touching, long-pressing, filling forms, or navigating back. The LLM can also

```
1  Action Selection Prompt = '''You are a testing agent... {system_prompt}
2  Your task is to {task}
3  You have completed the following actions:
4  {history_actions}
5  [Action 1]: Click button 'A'
6  [Observation 1]: Clicking 'A' navigate the app to page 'B'
7  ...
8  POSSIBLE NEXT ACTIONS: {candidate_actions}
9
10 Experience that can be relevent to this task: {persistent_memory}
11
12 Your job is to choose the most likely next steps to complete the task....
13 # The format of the JSON response must strictly follow these rules:
14 {
15     "chosen_action": ... (the index of the potential action that you choose)
16     "action_description": ... (a string describing the action you choose)
17     "reason": ... (a string describing the reason why you choose the action)
18 }...
```

Figure 3: Action Selection Prompt (Main Prompt)

pause if a loading screen is detected. This process iterates until Verifier ③ concludes that the task has been completed.

Vision Capability and Task Completion Decision: Once the LLM Action Selector ① chooses an action, the Executor executes it, and a screenshot of the updated GUI page is captured. The LMM (Large Multi-modal Model) Verifier ③ analyzes both the GUI description (in JSON format) and the actual screenshot, using a *multimodal approach to ensure more precise task completion decisions.* The Verifier ③ determines if the task has been successfully completed by examining the screenshot. If the task remains unfinished, the Verifier ③ asks Action Selector ① to perform the next action. This feedback mechanism is vital, as it can identify errors from the action history and trigger a back action to correct the sequence. The states of application before and after performing an action are compared to produce a list of UI changes. This change list is then provided to the LLM Observer ② , which observes major changes in the GUI page and stores this observation into task memory for the Action Selector ① to use in the next iteration.

2) **Task Memory:** As the process advances, the sequence of actions is logged. The tool maintains this history as a list of GUI pages, actions taken, and observations of screen changes. This serves as short-term memory, allowing the LLM to track recent actions and the evolving state of the GUI. After determining the next action, the action history is updated.

3) **Persistent Memory:** Prior research [35] has shown that reflection-based experience can enhance the LLM's performance via reinforcement learning with examples. In this, the LLM's reasoning is improved by incorporating few-shot examples of successful/-failed action sequences. These examples, generated by the Reflector ④ based on prior sequences, include rules such as: *"Confirming changes by clicking 'Save' button."* or *"Clicking to 'More...' to find the caption settings".* Task memory retains the immediate context and recent action outcomes, while persistent memory provides a repository of learned strategies and experiences. Such long-term knowledge in persistent memory is recorded as texts to the LLMs.

4) **Test Case Generation:** To facilitate testers to review and replay within the target app, VISIDROID concatenates each individual action from the produced action sequence into a test script.

3 Iterative Action Sequence Generation

Algorithm 1 Action Sequence Generation Algorithm
Input: *goal* (str), *pers_mem* (dict), *is_training* (bool) , *executor*

```
1:  pm ← pers_mem()
2:  procedure VISIDROID(goal, isTraining, executor)
3:      tm ← TaskMemory()              ▷ init an empty task memory
4:      tm.setGoal(goal)
5:      curUiState ← executor.curUiState()
6:      candActions ← extractActionables(curUiState)    ▷ buttons, text
    fields, checkboxes, etc.
7:      action ← actionSel.next(tm, pm, candActions)         ▷ ①
8:      prevUiState ← None
9:      while ! tm.finished and len(tm.actions) < MAX do
10:         curUiState, curScreen ← executor.execute(action)
11:         tm.addAction(action)
12:         uiChanges ← detUIChanges(prevUIState, curUIState)
13:         obs ← observer.observe(action, uiChanges)         ▷ ②
14:         tm.addObservation(obs)
15:         tm.finished ← verifier.verify(tm, pm, curScreen)     ▷ ③
16:         if ! tm.finished and len(tm.actions) < MAX then
17:             candActions ← extractActionables(curUiState)
18:             action ← actionSel.next(tm, pm, candActions)     ▷ ①
19:         prevUiState ← curUiState
20:     if isTraining then
21:         rules, optimizedSteps ← reflector.reflect(tm)       ▷ ④
22:         pm.updateRules(rules)
23:         pm.updateSteps(optimizedSteps)
24:     return tm.actions, tm.finished
```

3.1 VISIDROID Algorithm

Algorithm 1 outlines the main flow for generating an action sequence for a task. This function takes as input a goal-based task description (`goal`), persistent memory (`pm`), a flag indicating training or evaluation mode (`isTraining`), and the mobile interface the agent interacts with (`executor`). After processing, the function returns a list of actions and a boolean indicating if the task is complete.

Initialization: First, a task memory object is initialized to track the goal, executed actions, observations, and task completion status. The current state of the UI is captured using `executor.curUiState()`. A set of possible actions (e.g., pressing buttons, swiping) available in this UI state is then extracted.

Action Loop (Lines 7-19, Algorithm 1): The algorithm enters a loop involving three main components: Action Selector ① , Observer ② , and Verifier ③ . This loop continues until the Verifier ③ decides that the task's goal is completed or the maximum number of actions is reached.

1. Action Selector. The Action Selector ① takes responsible for selecting an appropriate action to perform on target element. The prompt is structured as Figure 3. At each iteration i, the Action Selector engages in reasoning $\mathcal{R}$ to determine the next action based on three external observations O.

a. The first observation O_1 is derived from short-term task memory, containing details about previously taken actions (Lines 3-5, Figure 3). For the initial action, the short-term memory is empty.

b. The second observation O_2, also derived from short-term task memory, includes a description of the GUI page after the latest action performed by the test environment. This observation is

```
1  Task Verify Prompt = '''You are a testing agent... {system_prompt}
2  Your task is to {task}
3  You have completed the following actions:
4  {history_actions}
5  [Action 1]: Click button 'A'
6  [Observation 1]: Clicking 'A' navigate the app to page 'B'
7  ...
8  Current UI screen: {screenshot}
9  Experience that can be relevant to this task: {persistent_memory}
10 Based on the task execution history, task end condition and the current
          state of the app, you need to verify the task execution if the task
          is done or not.
11 # The format of the JSON response must strictly follow these rules:
12 {
13     "screen_description": ... (a string describing current screen)
14     "task_done": ... (a boolean indicating if task is done)
15 }...
```

Figure 4: Multi-Modal Task-Verifying Prompt

produced by the Observer ② (Line 6, Figure 3). For the initial action, the starting page is used.

c. The third observation O_3 is a list of actionable UI elements extracted from the current UI state (Line 8, Figure 3). This list is structured in JSON format, with each actionable UI element paired with a set of associated actions (e.g., touch, long touch, swipe, input).

d. The fourth observation O_4 comes from persistent memory (initially empty) and consists of prior experience and optimized steps from previous executions of the same task (Line 10, Figure 3).

The Action Selector ① decides the next action to perform on the app under test a_{i+1} by applying the reasoning process $\mathcal{R}(O_1, O_2, O_3, O_4)$. The selected action is executed on the device via Executor and stored in the task memory for reference in next steps O_1.

2. Observer. The Observer ② is responsible for detecting major changes in the UI before and after an action's execution. Once an action is performed, the current UI state is compared with the previous state to identify any changes, additions, or removals (Line 13, Algorithm 1). Feeding all changes directly to the LLM for the Action Selector may be overwhelming. Thus, an intermediate step called *observing* is introduced. The Observer ② digests these UI changes and provides a concise summary of key changes, e.g., *"After pressing the 'Ok' button, a confirmation dialog appeared."* These observations are then stored in Task Memory for subsequent inference steps (O_2).

3. Multi-modal Verifier. After performing an action, VISIDROID captures the current state and screenshot, then invokes the Verifier ③ to check if the task has been achieved. Once the task is completed, the loop ends, and the result is returned. It uses a multi-modal verifier that leverages both text-based UI descriptions and screenshots, as visuals often convey more information than text in mobile UI testing. For instance, verifying visual styles of native Android elements (e.g., color, font, and opacity) is challenging with most mobile automated tools (e.g., Katalon, Appium). Tasks involving style verification thus benefit from a multi-modal approach.

Another challenge is defining the relative positioning of UI elements, critical for tasks like element repositioning, image scaling, and alignment. While absolute positions are represented by x and y coordinates, assessing spatial relationships among elements is complex. The multi-modal approach enables more precise verification of element positioning and layout.

The Verifier ③ receives five inputs: a **visual input** (current screenshot) and **four textual inputs** the goal, previous actions,

observations, and persistent memory. The structure of our prompt is shown in Figure 4. For multi-modal prompting, the screenshot is encoded as a base-64 string and included in the API request alongside the textual prompt directed to the LLM. We instruct the Large Multi-modal Model (LMM) Verifier ③ to return a task-completion status of 'yes' or 'no,' determining the ending.

3.2 Memories and Training

Memories. As seen in Figure 2, there are two types of memory: *Task Memory* and *Persistent Memory*. *Task Memory* is associated with a single execution; once the task ends, this memory is reset. *Task Memory* includes a list of performed actions (collected from Action Selector ①), a list of UI change observations after each action (collected from Observer ②), and the current UI state. In contrast, *Persistent Memory* consists of a list of rules and a list of optimized steps, generated by the Reflector ④ . *Persistent Memory* is built up during the *Training phase* and remains unchanged during the *Evaluation phase*. This memory is shared among executions.

Training Phase. VISIDROID employs a two-phased reflection strategy: the *Training Phase* and the *Evaluation Phase*. During the *Training Phase*, *Persistent Memory* is gradually built up after each execution and remains unchanged throughout the *Evaluation Phase*. Inspired by prior works [35], we employ a self-reflection mechanism in the *training phase*, without the need for human intervention. Once the agent completes a task, the Reflector module ④ reflects the generated sequence of actions against the task description to check if it successfully addressed the task goal or failed. The Reflector module builds reasoning on three pieces of information $\mathcal{R}_{\text{Reflect}}(O_1, O_2, O_3)$, where O_1 is the list of performed actions, O_2 is the list of UI change observations, and O_3 is the task description. $\mathcal{R}_{\text{Reflect}}$ can return in two formats, with two scenarios:

1. If the generated sequence fails to meet the task's goal, <"failed", [rules to avoid]>, VISIDROID instructs the LLM to return a set of rules based on the failed sequence to avoid in future runs. These rules are stored in *Persistent Memory*. For instance, in the first training iteration for the task *"Add the holidays of the United States to the calendar,"* the agent ended the task before clicking "Save," resulting in failure. Here, the Reflector extracted the (confirming) rule *"Ensure confirming the addition to avoid leaving the task incomplete"*.

2. If the generated sequence successfully addresses the task's goal, <"success", [rules to follow], [optimized steps]>. Although the task is addressed, it may have some excessive actions. VISIDROID then extracts the rules that can avoid unneeded actions and returns an optimal path for reference in the next runs. These rules and the optimal path are also stored in *Persistent Memory*. For example, for the task *"Sort contacts based on 'Date created' in descending order,"* the model initially navigated to the settings menu to find the sort option. After failing to find it, it returned to the "More options" menu, where it found the "Sort by" button. The initial steps were not optimal, so the Reflector provided the rule to go directly to the "More options" menu, and the optimized steps are ["More options"→"Sort by"→"Date"→"Descending"→"OK"].

3.3 Candidate Ranking and Test Generation

For each task, the agent generates multiple action sequences independently (three in our experiments) and selects the most appropriate one. The generated sequences are ranked using a heuristic solution based on *textual matching* with the task description.

VisiDroid splits a task description into a set of terms T, such as *"Alice"* and *"Favorites"* in *"Add contact Alice to Favorites"*. For each generated action sequence S_i, we gather the set U_i of interacted UI elements (e.g., `<Button content-desc="Add to Favorites"/>`). For each U_i, we define a scoring function:

$$\text{Score}(S_i) = \sum_{w \in T} \mathbb{I}(w \in U_i),$$

where $\mathbb{I}(w \in U_i)$ is an indicator function that equals 1 if the word w from T matches any word in U_i and 0 otherwise. The total score for each sequence S_i is the sum of matched words, representing its alignment with the description. The sequence S_i with the highest score is selected as the most relevant for the next step: $S_{\text{best}} = \arg\max_i \text{Score}(S_i)$. This scoring ensures that the sequence best matching the description terms is chosen to guide the LLM.

VisiDroid is proposed toward generating UI-level test cases and scripts for *automated regression testing* of mobile apps. For regression testing, *the oracle of a test* is derived from the previous execution of the app. VisiDroid is used in a scenario in which testers provide descriptions or goals of tasks that end-users would use the AUT and generate test cases and scripts. Generated test cases and scripts can be independently executed using tools like Appium or Katalon without using LLMs. Task descriptions or goals can come from requirements. With this scenario, test cases and scripts are similar to those generated using the popular record and replay approach [19], but it eliminates the need of testers performing the recording step manually on the AUT.

In VisiDroid, the generated sequence of actions is converted to test cases and scripts for regression testing. Formally, a generated test case T is represented as: $T = \left(\bigwedge_{i=1}^{n} a_i\right)$, where $a_1, a_2, \ldots, a_n$ are n actions (test steps) in the sequence. We developed an interpreter that converts action sequences into Appium test cases, enabling cross-device integration. During execution, each action in the generated sequence is interpreted as a test step, performing the designated UI action on the target element. If any step fails, the test case is terminated and marked as 'failed'.

4 Empirical Evaluation

To evaluate VisiDroid, we seek to answer the following questions:
RQ1. [Effectiveness on Generating Sequences of Actions]. How accurate does VisiDroid generate sequences of actions for a given task whose goal is described in natural language?
RQ2. [Effectiveness in Test Script Generation]. How well is VisiDroid in test script generation?
RQ3. [Ablation Study: Memory]. How much does our memory mechanism contribute to overall performance of VisiDroid?
RQ4. [Ablation Study: Multi-Modal]. How much does multi-modal reasoning contribute to overall performance of VisiDroid?
RQ5. [Ablation Study: Ranking]. How much does our ranking contribute to overall performance?
Dataset. We utilize the open-source DroidTask dataset, which was previously used in the-state-of-the-art AutoDroid [15]. This

dataset comprises 13 Android applications, each accompanied by an installation file and a set of task descriptions. These descriptions are concise and unguided, providing goal-based tasks without explicit instructions. Examples include tasks such as *"Add contact Bob to Favorites"* and *"Set app theme to light and save it"*. A total of 12 out of the 13 applications were successfully installed, while the provided Firefox's installation package could not be installed due to its deprecation with Android API 34. Consequently, our experimental analysis was conducted with VisiDroid and the baseline approaches—on those 12 applications, covering a total of 150 tasks.

5 Effectiveness in Action Generation (RQ1)

5.1 Empirical Methodology

Setup. We set up a controlled environment using the Android emulator from Android Studio. The emulator ran on a 64-bit Windows 11 with an R7-7840HS CPU (8 cores) and 32GB memory. For effectiveness and cost-efficiency, VisiDroid utilized GPT-4o (gpt-4o-2024-08-06) within its Action Selector (1), Observer (2), and Verifier (3). GPT-4 (gpt-4-turbo) was deployed in the Reflector (4) module to assess task failures and generate reflections.
Baselines. We compare VisiDroid with 3 LLM-powered task automation tools: AutoDroid [15], DroidAgent [18], and Guardian [32].
Metrics. For evaluation, we use the following metrics:

Exact-match. An action is considered as correct if it is required to complete the task in the context of the app. A sequence that contains all correct actions is evaluated as an exact-match sequence. The percentage of *Exact-match* for all the tasks is calculated as: $\%\textit{Exact-match} = \frac{\#Exact\text{-}match\ tasks}{\#Tasks}$.

Prefix-match. Prefix-match is calculated by determining the proportion of correct actions from the beginning of the sequence up to the first incorrect action. This metric reflects the percentage of an action sequence prefix that can be reliably reused. *Prefix-match* is reported as the average accuracy across all generated sequences.

Precision. If the prefix match of a sequence is zero (i.e., no initial correct actions), precision is also defined as zero, as the actions following an incorrect one are unlikely to be reusable within the app's context. Otherwise, the precision of a generated action sequence for a task is calculated as the ratio of correct actions to the total actions performed, representing how accurately a model identifies the next required action. $\%\textit{Precision} = \frac{\#Correct\ actions}{\#Generated\ actions}$. Average Precision is the average precision across all sequences.

Task Completion. This metric evaluates a model's ability to complete a task at the correct final step. A task is considered as completed if the goal of the task is achieved with the last correct action in the generated sequence. This metric is less stringent than *Exact-match*, allowing for additional actions in the middle of a sequence (e.g., additional actions to go back and forth from a screen). The percentage of task completion for all the sequences is defined as $\%\textit{Task Completion} = \frac{\#Completed\ tasks}{\#Tasks}$.

Task Coverage. This metric evaluates a model's ability to achieve a task's objective at covering the actions in a sequence. *Task Coverage* is less stringent than both *Task Completion* and *Exact-match*, allowing for additional actions either in the middle or at the end of a sequence. The percentage of task coverage for all sequences is defined as $\%\textit{Task coverage} = \frac{\#Covered\ tasks}{\#Tasks}$.

Table 1: Effectiveness on Generating Sequence of Actions (RQ1). Total number of tasks: 150.

Metric	Guardian	DroidAgent	AutoDroid	VISIDROID
#Exact-match tasks	0	21	106	**131**
#Completed tasks	2	36	113	**132**
#Covered tasks	54	105	126	**132**
Exact-match (%)	0	14	70.7%	**87.3%**
Task Completion (%)	1.3	24	75.3%	**88.0%**
Task Coverage (%)	36	70	84%	**88.0%**
Prefix-match (%)	10.6	28.8	80.6%	**90.5%**
Precision (%)	21.2	39.8	87.2%	**91.4%**

The relationship between the metrics on the tasks can be represented as (Exact-match tasks ⊆ Completed tasks ⊆ Covered tasks), indicating that *Exact-match* tasks are a subset of *Completed tasks*, which in turn are a subset of *Covered tasks*.

5.2 Empirical Results

5.2.1 **Result Analysis.** As seen in Table 1, the *Exact-match* metric indicates that 87.3% of VISIDROID's generated action sequences follow the correct path (without extra actions), indicating that *almost 90% of produced sequences of actions can be directly used to produce test scripts*. The *Prefix-match* metric showcases VISIDROID's ability to accurately generate prefixes of action sequences, with an 90.5% prefix-match rate. On average, 9/10 actions in a sequence are correctly reusable. Although AutoDroid has access to the app's initial context via UI Traverse Graph, VISIDROID's superior *Prefix-match* underscores the effectiveness of its *Long-Term Memory*, which enables the model to learn the optimal path from previous executions.

A sequence of actions is considered to cover a task (Covered tasks) if it reaches the task's objective, even if it includes some extra post-completion actions. Among all methods, VISIDROID shows the smallest gap between *#Exact-Match tasks* and *#Covered tasks*, with 99% of covered tasks ending correctly (131/132). That is, in 131/132 tasks, VISIDROID's produced sequences end at the correct final screens *without extra post-completion actions. This shows the benefit of* **multi-modal method** *with* **vision** *in* **finishing the task correctly** *without extra actions (see Ablation study).*
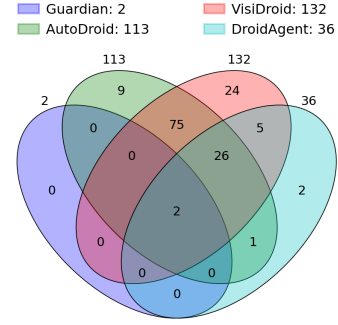
Precision reveals that 91.4% of actions generated are reusable. That is, on average, 9 actions are correct and only one action is extra in the middle of sequences of actions.

In comparison, VISIDROID outperforms all the baselines in all five metrics in Table 1. VISIDROID improves relatively over the best baseline AutoDroid by **23.5%** in *Exact-match*. It results are also significantly better than those of Guardian and DroidAgent.

5.2.2 **Task Completion Comparison.** AutoDroid [15] has failed to complete 13 tasks, 8 of which were finished by VISIDROID. Its limitations in task completion are acknowledged in their paper [15].

DroidAgent [18] shows the largest gap between *#Covered tasks* and *#Completed tasks*, indicating that while it can reach the task objective, it **struggles with determining when to end** in 13 tasks. The LLM takes on both roles of determining the next action and terminating the action loop, often overlooking when the goal has already been achieved and unnecessarily continuing the loop.

Guardian [32] shows the least number of exact-match tasks, completed tasks, and covered tasks, with only 0, 2, and 54 respectively.

Figure 5: Overlapping Analysis of Completed Tasks

This is because Guardian uses a rule with a fixed number of actions (15 actions). Given that our dataset does not contain any tasks that take exactly 15 steps, the *Exact-match* metric is understandably zero. During our email communication with Guardian's authors, they acknowledged this limitation, which will be currently addressed in their future work. The *#Completed tasks* metric is slightly more relaxed than *Exact-match*. That metric allows agents to have excessive steps in the middle but requires them to end once the task objective is reached. Thus, there are two tasks where Guardian ends the tasks correctly (with extra steps in the middle).

5.2.3 **Overlapping Analysis.** Figure 5 provides a Venn diagram for all results. VISIDROID completed 132/150 tasks, **uniquely resolving 24 tasks** in which none of the baseline approaches correctly finished. The best baseline (AutoDroid) completed correctly 9 unique tasks. VISIDROID shares 103 completed tasks with Auto-Droid, and 33 with DroidAgent. This represents the relative improvements in *Task completion* over Guardian, DroidAgent, and AutoDroid by 66.7X, 2.7X, and 16.9% respectively.

5.2.4 **Further Analysis and Examples on Cases Uniquely Detected by VISIDROID.** We further investigated **24** tasks that VISIDROID uniquely handled to highlight the effectiveness of the *Multi-modal Verifier*, particularly for tasks where visual images are crucial for determining task completion. We report the following:

The <u>first</u> category consists of tasks that require changes in visual images within an app. For example, tasks like *"Take a selfie video"* involve capturing a new video, which cannot be detected solely through textual changes. Similar examples include *"Switch to front camera"* and *"Show a photo."* The <u>second</u> category includes tasks that result in appearance changes, such as *"Switch to light theme"* or *"Change to left alignment (for texts)."* The <u>third</u> category involves tasks related to configuration settings that use toggles or radio buttons, e.g., *"Turn Wi-Fi off"*, *"Add a contact to favorites."* The <u>fourth</u> category covers tasks that involve small icon changes, such as *"Play a video/audio."* or *"Turn on word count"*. These four categories, (12 tasks) share a common challenge: they involve visual changes that are difficult to detect through text alone. Since these visual elements update on-screen without respective textual changes, baseline models often fail by making incorrect actions, such as switching the camera twice, toggling buttons multiple times, or continuously pressing the playback button – ultimately leading to task failure. In contrast, VISIDROID effectively recognized screen
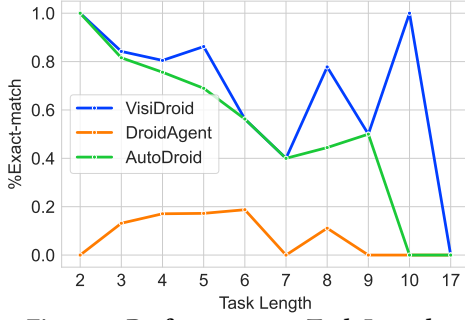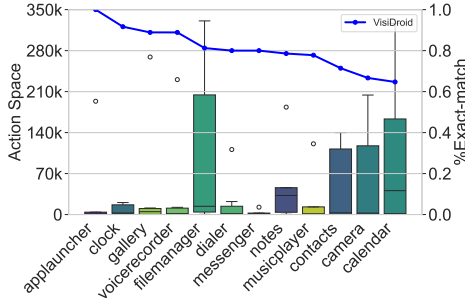
Figure 6: Performance on Task Lengths

Figure 7: Performance on Action Spaces

Figure 8: Token Count Analysis for VISIDROID

changes through visual analysis, ensuring accurate task completion. The last category involves the tasks requiring the use of multiple apps, e.g., *"Print image"* and *"Make call"*). DroidAgent failed these tasks because it focuses on a single app.

5.2.5 **Limitations.** The 9 tasks that VISIDROID did not complete correctly but AutoDroid did were mainly due to unclear task descriptions. Some descriptions confuse a setting option with a command. For example, the task *"Send long messages as MMS"* refers to a setting that allows sending long messages as MMS instead of using SMS. However, our agent misinterprets this as a command to send some long messages as MMS, thus generating an action chain to send a long message. AutoDroid, leveraging the UI graph, navigates to the correct screen. This type of tasks accounts for 6 out of 9 cases.

5.2.6 **Performance on Tasks with Different Lengths.** We stratified the results on the Exact-Match metric on all the best-performing approaches across tasks of varying lengths (Figure 6). We exclude Guardian in the graph due to its low Exact-Match. For tasks of length 2 (i.e., two actions), both VISIDROID and AutoDroid achieve 100% Exact-match. As the task length increases to 3-17, VISIDROID either outperforms or matches AutoDroid, demonstrating consistent performance even in more complex tasks exceeding 10 steps. For the cases of infeasible tasks (which do not exist in our dataset), VISIDROID will stop at the pre-defined maximum number of actions.

5.2.7 **Performance on Tasks with Different Action Spaces.** At each screen, there may exist multiple feasible actions. The action space consists of all possible actions at all the screens for a task. We aim to analyze *how accurate VISIDROID performs when the action space gets larger*. In Figure 7, we report the action space for each task and group them by application. The action space for each task is calculated as $A = (a_0 \times a_1 \times \ldots \times a_e)$, where a_0 is the number of
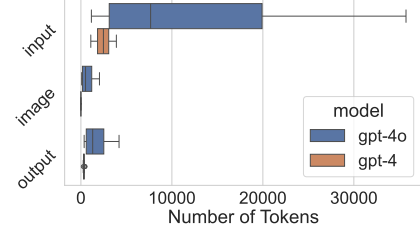
actions in the first state, and a_e is the number of actions in the last state. An increase in the action space A can be attributed to two reasons: a complicated UI at each state leading to a large a_i, or a complicated task, leading to a large e. Four apps—*Contacts, Camera, Calendar*, and *File Manager*—exhibit large action spaces, with their upper quartiles exceeding **100,000** possible actions. The tasks on the other **9** apps have an average action space of **20,000**.

As seen, despite complexity, VISIDROID maintains strong performance on the *File Manager* app, achieving over **80%** exact-match accuracy with the largest action space. For *Contacts, Camera*, and *Calendar*, we maintain an accuracy on exact-match higher than **65%** although the action spaces are **5x** larger than other apps. **This level of accuracy in challenging scenarios highlights VISIDROID's robustness when handling cases with large spaces of actions**.

5.2.8 **Token Count Analysis.** Overall, the *text input tokens* are much larger than the *output tokens* and *image input tokens* as required by VISIDROID. Thus, we conducted a cost analysis by measuring tokens needed for OpenAI API for all tasks. The *image input tokens* are the smallest, requiring only 85 tokens for each verification in *low-resolution mode* of OpenAI API, and the median image tokens among all tasks are around 500 tokens. *Observer, Verifier*, and *Reflector* are the most costly components. *Observer* and *Verifier* are called each time VISIDROID interacts with the app under test, to summarize what has changed and whether the task is completed. *Observer* takes the UI change list as input; therefore, whenever the app navigates to another screen, all elements of the old screen are replaced by the new one, resulting in a long change list and, consequently, a large input token size. As for *Reflector*, although the input and output tokens (not using image tokens) are relatively small compared to those of GPT-4o, it uses GPT-4, which results in a higher cost per token. On average, a task requires 13,419 input tokens (text + image), 1,670 output tokens for GPT-4o, and 2,463 input tokens, 337 output tokens for GPT-4. This shows that **the cost for image inputs is acceptable** with the gain of higher performance.

6 Effectiveness in Test Script Generation (RQ2)

In this experiment, we aim to evaluate VISIDROID's usefulness in regression testing via generating test scripts from a generated sequence of actions. From 150 sequences of actions derived from *RQ1*, we generated 150 corresponding test scripts. All of them were executed on the corresponding AUTs to verify whether they could run successfully. We used as the ground truth the final screens from the regression runs of the AUT. Although the test cases and scripts are interpreted one-to-one from the action sequences, they may fail during the execution due to various reasons such as flakiness [33]. A successfully executed test script requires (1) the successful

Table 2: Success Rates of Test Scripts generated by VISIDROID

Test scripts for	Successful execution (%)	
All tasks	108/150	= 72.0%
Completed tasks	108/132	= 81.8%
Exact-match tasks	100/131	= 82.4%

```
1  #Task: add event 'VisitParents' on July 30, remind 1 hour before, save it
2  d = appium.driver()
3  d.find_element(content_desc, "New Event").click()
4  d.find_element(text, "Title").fill("VisitParents")
5  d.find_element(id, ".date_picker").click()
6  d.find_element(text, "Prev month").click()
7  d.find_element(text, "Prev month").click()
8  d.find_element(text, "30").click()
9  d.find_element(text, "OK").click()
10 d.find_element(text, "10 minutes before").click()
11 d.find_element(text, "1 hour before").click()
12 d.find_element(content_desc, "Save").click()
```

Figure 9: Test script for adding event to Calendar app

```
1  #Task: Export contacts to a .vcf file 'classmate.vcf'
2  d = appium.driver()
3  d.find_element(text, "More options").click()
4  d.find_element(text, "Export contacts to file").click()
5  d.find_element(text, "contacts_2024_09_12_19_17_15").fill("classmate.vcf")
6  d.find_element(text, "OK").click()
7  d.find_element(text, "Save").click()
```

Figure 10: Test script for exporting contacts in Contacts

execution of all test steps and (2) achieving the testing goal by the end of the script. Therefore, only a sequence of actions from a completed task or an exact match can meet the second requirement.

In total, without human intervention, we have 108 successfully executed test scripts out of 150 tasks, marking a success rate of 72%. These 108 successful test scripts come from 132 completed tasks, resulting in **a success rate of 81.8%** for completed tasks. If we only generate test scripts for exact-match sequences, the success rate remains nearly 82%. For action sequences of completed tasks, there are 24 non-executable test scripts (132-108), while for exact-match sequences, there are 23 non-executable test scripts (118-131).

Several reasons contribute to test inexecutability as we explain in the following cases. The dynamic factor (e.g., time) is one reason of test script inexecutability, affecting the element names and root location of the date picker. In Figure 9, we show the test script generated for the task *"add the event of 'VisitParents' on July 30, remind me 1 hour before, save it"*. This task requires the agent to interact with a date picker to navigate to July 30 from the current day. However, the date when generating the sequence of actions is different from that of executing the test script, thus, following the generated sequence led to an incorrect date. Although this is an exact-match sequence and all 10 steps can be executed, the task goals were not reached at the end, resulting in a failed script.

The second case is demonstrated in Figure 10, where the task is to export all contacts under the *Contacts* app to a file. While executing the script, the test case failed at Line 5, where it finds a text box with the text *"contacts_2024_09_12_19_17_15"*, which is dynamic due to the current timestamp of the system. Therefore, the test driver cannot find such a text box, leading to a test failure.

Apart from the time factor, *UI delay or unexpected popups* also contribute to test inexecutability. Especially for apps that require loading a large amount of UI resources, such as *File Manager, Gallery,* and *Camera,* UI delays often occur. Those issues can be resolved by applying auto-healing, smart-wait features from test environments.

7 Ablation Study (RQ3–RQ5)

We aim to evaluate the key design components in VISIDROID: 1) *Persistent Memory*, 2) *Multi-modal Verifier*, and 3) *Sequence Ranking*. To achieve this, we build three additional variants of VISIDROID: VISIDROID without *Persistent Memory* (No-mem), VISIDROID without *Multi-modal Verifier* (No-vis), and VISIDROID without *Sequence Ranking* (No-rank). The results for these variants are in Table 3. We used the same metrics as in RQ1, with an additional one: *#Premature tasks*, which measures the number of tasks that did not reach the task objectives but were terminated prematurely by the model.

7.1 Ablation on Persistent Memory (RQ3)

Table 3: Contributions of VISIDROID's components

Metric	VisiDroid	No-mem	No-vis	No-rank
#Exact-match tasks	131	79 (-52)	108 (-23)	321
#Completed tasks	132	89 (-43)	111 (-21)	341
#Covered tasks	132	92 (-40)	113 (-19)	-
#Premature Tasks	5	29 (+24)	14 (+9)	-
#Total Tasks	150	150	150	450 (x3)
Exact-match(%)	87.3	52.7 (-34.6)	72.0 (-15.3)	71.3 (-16.0)
Task Completion(%)	88.0	59.3 (-28.7)	74.0 (-14.0)	75.8 (-12.2)
Prefix-match(%)	90.5	67.3 (-23.2)	82.0 (-8.5)	-
Precision(%)	91.4	73.0 (-18.4)	84.2 (-7.2)	-

VISIDROID without *Persistent Memory* (VISIDROID-mem) is implemented by discarding No-Mem and *Reflector* module ④ from Figure 2 and the *experience* section (Line 10) in the prompt of Figure 3. That is, No-Mem does not require training; every task is run independently three times without training. The rest of the design is retained.

As seen, *Persistent Memory* contributes significantly to accuracy. *Without it, the percentage of exact-match tasks drops by 34.6% (52 fewer tasks), and the task completion rate drops by 28.7% (43 fewer tasks).* Our investigation shows that in 24 out of these 52 cases, tasks ended prematurely without achieving the objective due to the absence of persistent memory. Failures typically occur because the model skips confirmation steps, e.g., clicking *"Save"* or *"Ok"* after modifying settings, which are essential for task completion. During training, the model gains experience on such tasks, which enhances performance in the evaluation phase. Experiences such as *"Always confirm changes by tapping the 'Save' button to apply new settings"* are generated by the LLM and stored in *Persistent Memory* for future predictions on similar tasks.

Apart from experience rules, *Persistent Memory* also provides a set of optimized steps, which drive the evaluation phase to precisely address the task goal. Without these optimized steps, the agent struggles to decide the correct actions, as seen in *Prefix-match* and *Precision*, where *Prefix-match* experiences a downgrade of 23.3%.

7.2 Ablation of Multi-Modal Verifier (RQ4)

For the variant in this case, we omit the image input from Figure 4, leaving the verifying prompt with only *four textual inputs*: the goal, previous actions, observations, and persistent memory. This transforms the *Multi-modal Verifier* into a purely text-based verifier. As seen in Table 3, removing the image input reduces both task completeness and proper task ending. This change increases the gap between *#Covered tasks* and *#Exact-match tasks* (from 1 to 5), indicating that *more tasks fail to end correctly without vision*.

Most of those cases occur in where the task causes a UI change. For example, tasks like *"take a short selfie video"* during RQ1 analysis could only be resolved by VisiDroid. In this case, No-vis failed by repeatedly pressing the shutter button. Similarly, tasks like *"change the alignment to center"*, which result in direct UI changes, encounter the same problem. In contrast, visual images are useful in detecting those changes. The absence of vision capacity in No-vis also affects task completeness, leading to premature task endings in 14 cases. See Section 5.2.4 for more case studies where visual analysis helps.

Importantly, *even without vision,* No-vis *achieves slightly better* Exact-match *than the best baseline, AutoDroid. This shows the higher effectiveness of memory-powered iterative action sequence generation. Our multi-modal analysis also improves further over the baselines.*

7.3 Ablation of Sequence Ranking (RQ5)

We run the model 3 times independently for each task, which results in 150x3 tasks, then the we use the Sequence ranking to rank those 3 solutions and choose the best. The results for VisiDroid-rank are in Table 3. Ablating Sequence Ranking results in more solutions are selected and less efficient. Sequence ranking contributes to 16% of Exact-match and 12.2% of Task Completion. Without it, more incorrect sequences require manual validation, tripling labor cost.

8 Threats to Validity

Our dataset may not represent typical Android apps. Our analysis shows that the dataset has a diverse set of Android apps varying in task length and action space. The dataset used in our experiments consists of 12 apps that occasionally exhibited instability when running on the simulator during the training, evaluation, and test execution phases. In our experiment, if a crash occurred, the execution was retried. During the test execution phase when test scripts were executed using Appium, the test execution encountered flakiness issues such as the instability of Appium, UI loading time, and changes in the states of the AUT [33]. Hallucination of the LLM is another threat to validity, affecting the consistency and accuracy of our study. To reduce hallucination, we set the temperature at zero, reran the test cases several times, and manually examined all test cases to verify whether they are valid. Several test cases failed during execution due to flakiness, but they turned out to be valid. Different wording in a prompt might lead to different results, but the LLM was fairly robust in our experiments. We used GPT-4 for a fair comparison with the baselines using the same model. To address potential generalizability concerns, we designed prompts that are compatible with various LLMs and made them publicly available.

9 Related Work

Android GUI Testing. Our work is closely related to the following GUI testing approaches. *GPTDroid* [26] treats mobile GUI testing as a Q & A task, using GPT-3 to predict the next action based on static GUI context and dynamic testing context. *AutoDroid* [15] employs a two-phase exploration strategy: offline random exploration builds a UI Transition Graph, which guides *autonomous action planning* during the online phase. *DroidAgent* [18] is an autonomous test agent that generates and executes tasks independently using a planner and an actor. The planner sets a task, and the actor executes actions until the task goal is reached or an action limit is met. For evaluating DroidAgent on natural language tasks, we adapted its input to focus on one task goal at a time, halting the planner after task completion. *Guardian* [32] is an external runtime framework that integrates domain-specific knowledge, managing interactions between the LLM and the AUT. It lacks visual representations.

In comparison, all the above approaches struggle with identifying the final action, and either end prematurely or continue past the final screen. In VisiDroid, for Android GUI apps, several visual cues and changes can indicate task completion, e.g., changes in the visual GUI elements. First, GPTDroid has an LLM chat with the mobile apps by passing the GUI page information to LLM, and builds a specialized neural network to decode the LLM's output into actionable steps. We excluded GPTDroid from comparisons because it does not fit with the usage scenario of VisiDroid, which works with a testing environment. Second, VisiDroid distinguishes itself from AutoDroid [15] and DroidAgent [18], which rely on autonomous planning strategies with LLM-based planners. Instead, VisiDroid employs an iterative approach, Third, Guardian [32] also leverages LLM's reasoning capabilities but focuses on refining the action space for better re-planning. Guardian has set a fixed number of actions (15), thus struggling with task completion decision.

Test Generation and LLMs. For unit testing, several studies propose white-box solutions for test script generation [7, 20, 29, 31]. ML-powered automation solutions are also employed to enhance various aspects of API testing, e.g., generating test cases [8, 30, 36] and realistic test inputs [5]. In the GUI test generation, Google's Monkey tool employs a strategy based on random app exploration [13]. Humanoid [21] represents a learning-based strategy. Model-based strategies are also utilized [6]. Advancements in LLMs have facilitated various tasks, including code generation [11, 14, 24, 34], automated program repair [17, 38, 39, 41, 43], and testing [16, 18, 25, 32]. GUI testing has utilized LLMs for test case generation [18, 20, 27, 28], test inputs [25], and bug reproduction [12].

10 Conclusion

Novelty. We present VisiDroid, a multi-modal mobile testing framework utilizing LLMs with vision capabilities to generate test scripts from GUI tasks described in natural language. We found that visual changes and cues are commonly used to indicate task completeness in GUI applications, whereas DOM or text content does not always accurately reflect the layout. Moreover, relying solely on textual signals can lead to errors, such as prematurely concluding a task when an earlier screen contains text that misleadingly resembles that of the final screen. Evaluation on 150 goal-based tasks across 12 Android apps shows its superior performance over the baselines.

From sequences of actions to perform tasks, VISIDROID generates test scripts, effectively converting task descriptions into automated regression tests. As a practical implication, it enables testers to efficiently generate automated test scripts from end-user task descriptions, potentially replacing the current practice of recording the tester's actions in the GUI and writing test cases manually. It also facilitates the process of generating test cases and scripts from requirement specifications with minimal human involvement.

Impacts on Future Research. This work provides evidence for the advantages of **multi-model analysis** by combining textual and visual components for test generation, suggesting several future research directions in software testing. One future avenue is **automated GUI bug detection**, where models analyze inconsistencies between expected textual outputs and actual visual renderings to identify UI issues such as misaligned elements, overlapping text, or inaccessible components. Using multi-modal analysis for **automated generation of test oracles** is important towards more autonomous testing where tester's involvements in the testing activities are reduced. Deciding whether a functionality performs well is taken at least partially by models. Another direction is **adaptive GUI test case generation**, where multi-modal LLMs dynamically adjust test cases based on visual changes, ensuring robust testing even when UI layouts evolve. **User experience evaluation** leverages LLMs to assess GUI usability by detecting visual clutter, poor contrast, or inefficient navigation flows. Finally, **visual reasoning for automated accessibility testing** could enhance inclusivity by detecting *color contrast violations*, *improper screen reader support*, or missing alt texts, *ensuring compliance with accessibility standards*.

Data Availability. Our data and code are available at [4].

References

[1] 2024. *Appium.* https://appium.io
[2] 2024. *Espresso.* https://developer.android.com/training/testing/espresso
[3] 2024. Katalon Studio Automation Testing Tool. https://katalon.com/
[4] 2025. *VisiDroid.* https://github.com/visidroid/visidroid
[5] Juan C Alonso, Alberto Martin-Lopez, Sergio Segura, Jose Maria Garcia, and Antonio Ruiz-Cortes. 2022. ARTE: Automated Generation of Realistic Test Inputs for Web APIs. *IEEE Transactions on Software Engineering* 49, 1 (2022), 348–363.
[6] Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, Bryan Dzung Ta, and Atif M Memon. 2014. MobiGUITAR: Automated model-based testing of mobile apps. *IEEE Software* 32, 5 (2014), 53–59.
[7] Andrea Arcuri, Gordon Fraser, and Juan Pablo Galeotti. 2014. Automated unit test generation for classes with environment dependencies. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering.* 79–90.
[8] Vaggelis Atlidakis, Patrice Godefroid, and Marina Polishchuk. 2019. Restler: Stateful rest api fuzzing. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE).* IEEE, 748–758.
[9] Cucumber [n. d.]. Cucumber. https://cucumber.io/docs/gherkin/.
[10] Gustavo da Silva and Ronnie de Souza Santos. 2023. Comparing Mobile Testing Tools Using Documentary Analysis. In *2023 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM).* 1–6. doi:10.1109/ESEM56168.2023.10304798
[11] Yihong Dong, Xue Jiang, Zhi Jin, and Ge Li. 2024. Self-collaboration code generation via chatgpt. *ACM Transactions on Software Engineering and Methodology* 33, 7 (2024), 1–38.
[12] Sidong Feng and Chunyang Chen. 2024. Prompting Is All You Need: Automated Android Bug Replay with Large Language Models. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering* (Lisbon, Portugal) *(ICSE '24).* Association for Computing Machinery, New York, NY, USA, Article 67, 13 pages. doi:10.1145/3597503.3608137
[13] Google. 2022. https://developer.android.com/studio/test/other-testing-tools/monkey.
[14] Qiuhan Gu. 2023. Llm-based code generation method for golang compiler testing. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering.* 2201–2203.
[15] Hao, Yuanchun Wen, Guohong Li, Shanhui Liu, Tao Zhao, Toby Yu, Shiqi Jia-Jun Li, Yunhao Jiang, Yaqin Liu, Yunxin Zhang, and Liu. 2023. Empowering LLM to use Smartphone for Intelligent Task Automation. *arXiv:2308.15272* (2023).
[16] Hieu Huynh, Quoc-Tri Le, Tien N. Nguyen, and Vu Nguyen. 2024. Using LLM for Mining and Testing Constraints in API Testing. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering* (Sacramento, CA, USA) *(ASE '24).* Association for Computing Machinery, New York, NY, USA, 2486–2487. doi:10.1145/3691620.3695341
[17] Matthew Jin, Syed Shahriar, Michele Tufano, Xin Shi, Shuai Lu, Neel Sundaresan, and Alexey Svyatkovskiy. 2023. Inferfix: End-to-end program repair with llms. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering.* 1646–1656.
[18] Juyeon, Robert Yoon, Shin Feldt, and Yoo. 2023. Autonomous Large Language Model Agents Enabling Intent-Driven Mobile GUI Testing. *arXiv:2311.08649v1* (2023).
[19] Wing Lam, Zhengkai Wu, Dengfeng Li, Wenyu Wang, Haibing Zheng, Hui Luo, Peng Yan, Yuetang Deng, and Tao Xie. 2017. Record and replay for android: Are we there yet in industrial cases?. In *Proceedings of the 2017 11th joint meeting on foundations of software engineering.* 854–859.
[20] Caroline Lemieux, Jeevana Priya Inala, Shuvendu K Lahiri, and Siddhartha Sen. 2023. Codamosa: Escaping coverage plateaus in test generation with pre-trained large language models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE).* IEEE, 919–931.
[21] 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.
[22] Mario Linares-Vásquez, Kevin Moran, and Denys Poshyvanyk. 2017. Continuous, evolutionary and large-scale: A new perspective for automated mobile app testing. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME).* IEEE, 399–410.
[23] Mario Linares-Vásquez, Cárlos Bernal-Cardenas, Kevin Moran, and Denys Poshyvanyk. 2017. How do Developers Test Android Applications?. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME).* 613–622. doi:10.1109/ICSME.2017.47
[24] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2024. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *Advances in Neural Information Processing Systems* 36 (2024).

[25] Zhe Liu, Chunyang Chen, Junjie Wang, Xing Che, Yuekai Huang, Jun Hu, and Qing Wang. 2022. Fill in the blank: Context-aware automated text input generation for mobile gui testing. *arXiv preprint arXiv:2212.04732* (2022).

[26] Zhe Liu, Chunyang Chen, Junjie Wang, Mengzhuo Chen, Boyu Wu, Xing Che, Dandan Wang, and Qing Wang. 2023. Chatting with GPT-3 for Zero-Shot Human-Like Mobile Automated GUI Testing. arXiv:2305.09434 [cs.SE] https://arxiv.org/abs/2305.09434

[27] Zichuan Liu, Chao Chen, Jianing Wang, Mingming Chen, Bin Wu, Xiaopeng Che, Dan Wang, and Qiang Wang. 2023. Chatting with GPT-3 for Zero-Shot Human-Like Mobile Automated GUI Testing. *arXiv preprint arXiv:2305.09434* (2023).

[28] Zhe Liu, Chunyang Chen, Junjie Wang, Mengzhuo Chen, Boyu Wu, Xing Che, Dandan Wang, and Qing Wang. 2024. Make llm a testing expert: Bringing human-like interaction to mobile gui testing via functionality-aware decisions. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–13.

[29] Stephan Lukasczyk and Gordon Fraser. 2022. Pynguin: Automated unit test generation for python. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*. 168–172.

[30] Alberto Martin-Lopez, Sergio Segura, and Antonio Ruiz-Cortés. 2021. RESTest: Automated Black-Box Testing of RESTful Web APIs. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '21)*. Association for Computing Machinery.

[31] Shabnam Mirshokraie, Ali Mesbah, and Karthik Pattabiraman. 2015. Jseft: Automated javascript unit test generation. In *2015 IEEE 8th international conference on software testing, verification and validation (ICST)*. IEEE, 1–10.

[32] Dezhi Ran, Hao Wang, Zihe Song, Mengzhou Wu, Yuan Cao, Ying Zhang, Wei Yang, and Tao Xie. 2024. Guardian: A Runtime Framework for LLM-Based UI Exploration. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2024)*. Association for Computing Machinery, New York, NY, USA, 958–970. doi:10.1145/3650212.3680334

[33] Alan Romano, Zihe Song, Sampath Grandhi, Wei Yang, and Weihang Wang. 2021. An empirical analysis of UI-based flaky tests. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 1585–1597.

[34] Gabriel Ryan, Siddhartha Jain, Mingyue Shang, Shiqi Wang, Xiaofei Ma, Murali Krishna Ramanathan, and Baishakhi Ray. 2024. Code-aware prompting: A study of coverage-guided test generation in regression setting using llm. *Proceedings of the ACM on Software Engineering* 1, FSE (2024), 951–971.

[35] Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2024. Reflexion: Language agents with verbal reinforcement learning. *Advances in Neural Information Processing Systems* 36 (2024).

[36] Emanuele Viglianisi, Michael Dallago, and Mariano Ceccato. 2020. Resttest-gen: automated black-box testing of restful apis. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*. IEEE, 142–152.

[37] Hao Wen, Yuanchun Li, Guohong Liu, Shanhui Zhao, Tao Yu, Toby Jia-Jun Li, Shiqi Jiang, Yunhao Liu, Yaqin Zhang, and Yunxin Liu. 2023. AutoDroid: LLM-powered Task Automation in Android. *Proceedings of the 30th Annual International Conference on Mobile Computing and Networking* (2023). https://api.semanticscholar.org/CorpusID:261277501

[38] Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. 2023. Automated program repair in the era of large pre-trained language models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 1482–1494.

[39] Chunqiu Steven Xia and Lingming Zhang. 2024. Automated program repair via conversation: Fixing 162 out of 337 bugs for $0.42 each using chatgpt. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 819–831.

[40] 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)*.

[41] Xin Yin, Chao Ni, Shaohua Wang, Zhenhao Li, Limin Zeng, and Xiaohu Yang. 2024. Thinkrepair: Self-directed automated program repair. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 1274–1286.

[42] Juyeon Yoon, Robert Feldt, and Shin Yoo. 2023. Autonomous Large Language Model Agents Enabling Intent-Driven Mobile GUI Testing. arXiv:2311.08649 [cs.SE] https://arxiv.org/abs/2311.08649

[43] Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. 2024. Autocoderover: Autonomous program improvement. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 1592–1604.