Beyond Static GUI Agent: Evolving LLM-based GUI Testing via Dynamic Memory

Mengzhuo Chen[1,2,3,*], Zhe Liu[1,2,3,*], Chunyang Chen[4], Junjie Wang[1,2,3,†],
Yangguang Xue[1,2,3], Boyu Wu[1,2], Yuekai Huang[1,2,3], Libin Wu[1,2,3], Qing Wang[1,2,3,†]
[1]Institute of Software Chinese Academy of Sciences, Beijing, China;
[2]University of Chinese Academy of Sciences, Beijing, China;
[3] State Key Laboratory of Complex System Modeling and Simulation Technology ISCAS, Beijing, China;
[4]Technical University of Munich, Munich, Germany;
[*]Both authors contributed equally to this research. †Corresponding author;
{chenmengzhuo23, liuzhe181}@mails.ucas.edu.cn, junjie@iscas.ac.cn, wq@iscas.ac.cn

***Abstract*—The development of Large Language Models (LLMs) enables LLM-based GUI testing to interact with graphical user interfaces by understanding GUI screenshots and generating actions, which are widely applied in industry and academia. However, current approaches test each app in isolation, lacking mechanisms for experience accumulation and reuse. This limitation often causes GUI testing approaches to miss deeper exploration and fail to trigger bug-prone functionalities. To address this, we propose `MemoDroid`, a three-layer memory mechanism that augments LLM-based GUI testing with the ability to evolve through repeated interaction. `MemoDroid` designs episodic memory to capture functional-level testing traces, reflective memory to summarize issue patterns and redundant behaviors, and strategic memory to synthesize cross-app exploration strategies. These memory layers are dynamically retrieved and injected into LLM prompts at runtime, enabling the agent to reuse successful behaviors, avoid ineffective actions, and prioritize bug-prone paths. We implement `MemoDroid` as a lightweight plugin, which can be integrated into existing LLM-based GUI testing approaches. We evaluate `MemoDroid` on real-world apps from 15 diverse app categories. Results show that `MemoDroid` enhances GUI testing performance across five baselines, with activity and code coverage increasing by 79% - 96% and 81% - 97%, and bug detection improving by 57% - 198%. Ablation studies confirm the contributions of each memory layer. Furthermore, `MemoDroid` detects 49 new bugs in 200 popular apps, with 35 confirmed fixes and 14 acknowledged by developers, showing its practical value in memory-driven GUI testing.**

***Index Terms*—Large Language Model, Android app, Memory Mechanism, Automated GUI Testing, GUI Agent**

I. Introduction

Graphical User Interface (GUI) testing is essential for ensuring the quality and reliability of modern mobile applications [1]–[5]. Traditional automated GUI testing approaches typically rely on pre-defined exploration scripts or hard-coded strategies, which limit their adaptability to complex and evolving user interfaces. With the rise of Large Language Models (LLMs), researchers and practitioners have begun to build intelligent GUI agents for mobile app testing [6]–[10] and automation [11]–[14]. These GUI agents leverage LLMs' multimodal understanding capabilities to interpret GUI screenshots, reason about app content, and autonomously generate action sequences.

Despite their flexibility and general-purpose reasoning, current LLM-based GUI testing approaches are typically designed for single-session testing, each testing task is executed from scratch, without any memory of prior executions or ability to accumulate targeted experience. While LLMs have world knowledge and reasoning capabilities, they lack specialized, app-specific learning during the testing process. As a result, these LLM-based GUI testing approaches often repeat ineffective interactions and cannot refine their strategies across multiple testing sessions, which limits their testing ability to improve over time. In contrast, human testers continuously build knowledge through hands-on practice, gradually learning to focus on bug-prone functionality, explore untouched areas, and avoid redundant actions.

Many real-world apps share common UI patterns and functional modules. Even within the same app, testers often need to perform testing across multiple versions, updates, or usage scenarios. In such cases, accumulated experience becomes crucial as testers rely on past knowledge to decide which interactions are likely to improve coverage, which functionalities and actions are historically more bug-prone, and which exploration strategies have proven effective in similar contexts. Unlike current LLM-based GUI testing approaches that treat each task as isolated, human testers learn from repeated exposure [15], gradually building abstract knowledge of what to explore, what to avoid, and how to adapt. While some recent work explores rule-based memory reuse or external knowledge augmentation [16]–[18], these approaches typically depend on heuristic rules or static knowledge bases, requiring manual alignment across app structures and limiting adaptability. There is still a lack of a dynamic, experience-driven memory mechanism that allows LLM-based GUI testing systems to grow over time, reflecting on past behaviors to summarize issue actions, and reusing learned strategies to guide future testing more effectively.

Recent advances across both academia and industry highlight the importance of memory in enabling agents to learn and adapt over time. In language agents [19]–[23], cognitive architectures such as Memochat [24], MemGPT [25], and recent frameworks like the Cognitive Architectures for Language

2643-1572/25/$31.00 ©2025 IEEE
DOI 10.1109/ASE63991.2025.00135

Agents (CoALA) [26] demonstrate how memory systems support better planning, reflection, and behavior reuse across long-horizon tasks. In domains like code generation [27]–[29] and robotics [30], [31], memory has also been shown to enhance agent decision-making by leveraging past traces and outcomes. However, memory design for automated GUI testing remains unexplored and poses domain-specific challenges. Unlike the language tasks, where prior prompts or context can be directly reused, automated GUI testing involves multimodal, dynamic interaction traces that must be abstracted, organized, and matched across heterogeneous app contexts. The memory must not only record low-level test traces but also generalize over issues, summarize functional actions, and offer adaptive guidance in future tasks.

Inspired by cognitive memory theories that emphasize learning through retention, abstraction, and reuse [32], [33], we propose `MemoDroid`, a three-layer memory mechanism for enhancing LLM-based GUI testing. Unlike prior approaches [6], [7], [9], [34] that treat each testing session in isolation, `MemoDroid` accumulates and reuses testing experience across tasks and applications. The *interaction-level episodic memory* segments and encodes functional testing traces, the *function-level reflective memory* summarizes repeated behaviors and identifies failure patterns, and the *app-level strategic memory* synthesizes exploration strategies across apps. During testing, `MemoDroid` dynamically retrieves relevant memory based on current GUI state, stagnation signals, or functional transitions, and injects it into the LLM prompt to guide decision-making. We implement `MemoDroid` as a lightweight plugin that can be integrated into existing LLM-based GUI testing systems without modifying their core architecture. It also allows memory accumulated from previous testing sessions to be reused in future tasks, which enables long-term improvement.

Designing `MemoDroid` involves several key challenges. First, a challenge lies in how to capture functional-level interaction experience in a structured and reusable form. `MemoDroid` addresses this by segmenting continuous testing traces into discrete functional units based on UI transitions and action semantics. Each unit is represented by annotated screenshots, structured action sequences, and an interaction graph, and is further encoded into a multimodal embedding to support downstream retrieval and analysis. Second, a key question is how to abstract and summarize testing behaviors for effective reuse. `MemoDroid` constructs reflective memory by merging semantically similar function units, analyzing execution patterns such as repeated actions or unresponsive operations, and generating high-level reflections through MLLM-based summarization. These summaries provide a concise and behavior-aware abstraction of past testing outcomes. Third, generalizing testing knowledge across apps requires identifying transferable strategies beyond app-specific traces. `MemoDroid` synthesizes strategic memory by aggregating reflections across apps and prompting the LLM to generate app-level exploration plans. At runtime, `MemoDroid` invokes memory based on GUI similarity and incorporates the retrieved memory into the prompt to inform testing decisions.

We evaluate `MemoDroid` on real-world Android apps from 15 diverse categories. Results show that `MemoDroid` enhances GUI testing performance across five baseline approaches, with activity and code coverage increasing by 79% - 96% and 81% - 97%, respectively, and bug detection rate increasing by 57% - 198%. An ablation study confirms that all three memory layers contribute to performance gains. Further experiments reveal that expanding the memory pool across 12 rounds leads to progressively better testing outcomes, demonstrating the benefit of experience accumulation. Finally, we conduct usefulness experiments on 200 popular apps on Google Play by integrating baselines with `MemoDroid`, and detect 49 previously unknown crash bugs, 35 of which have been fixed and 14 have been confirmed by developers.

The contributions of this paper are as follows:

- We propose a three-layer memory mechanism, `MemoDroid`[1], to enhance LLM-based GUI testing, consisting of episodic, reflective, and strategic memory layers that support trace retention, experience summarization, and strategy reuse.
- We automatically construct and release a structured memory pool covering 60 Android apps[1], which can be reused to improve future LLM-based GUI testing agents.
- Effectiveness and usefulness evaluation of the `MemoDroid` in real-world apps with practical bugs detected and confirmed.

II. Approach

This paper proposes `MemoDroid`, a three-layer memory mechanism designed to enhance LLM-based GUI testing by supporting the accumulation, abstraction, and reuse of experience across testing tasks. Unlike prior approaches that rely solely on short-term context within a single testing session, `MemoDroid` enables the LLM-based GUI testing (testing agent) to develop long-term, task-agnostic knowledge. This memory mechanism allows the agent to retain fine-grained execution traces, summarize testing outcomes, and derive reusable testing strategies, which is similar to how human testers gradually build domain-specific experience through repeated practice. As shown in Fig. 1, `MemoDroid` structures memory into three complementary layers: **Interaction-Level Episodic Memory**, **Function-Level Reflective Memory**, and **App-Level Strategic Memory**. The **Episodic Memory** records testing traces segmented by functional boundaries, including annotated screenshots, action logs, and GUI transitions. These traces are further encoded using visual-textual information and interaction graph embeddings for downstream reflection. The **Reflective Memory** consolidates repeated executions, identifies issue actions, and summarizes behavioral patterns through structured metadata and natural language generation. The **Strategic Memory** synthesizes multiple reflections not only from a single app but also across apps, producing generalized strategies that highlight priority

[1]We release the source code, dataset, and experimental results on our website https://github.com/TheMystery123/MemoDroid.

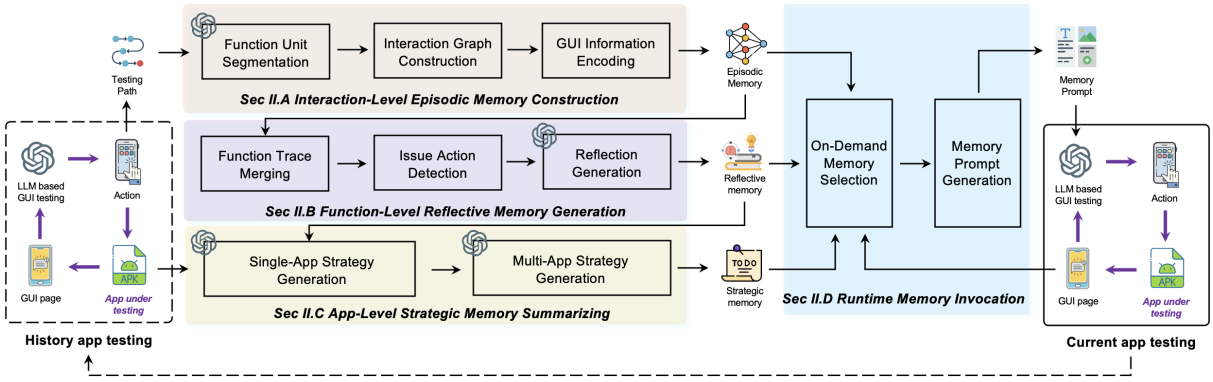


Fig. 1: Overview of MemoDroid

functionalities, recommended exploration sequences, and bug-prone behaviors.

MemoDroid incorporates a dynamic memory invocation mechanism throughout the testing process. It monitors exploration stagnation and repetitive behavior, then determines when to retrieve relevant memory based on GUI similarity. Depending on the testing scenario, MemoDroid retrieves and injects memory at the start of testing, mid-term testing, or the transition between functionalities. MemoDroid leverages testing experiences to guide follow-up testing more effectively, plan exploration paths more strategically, and achieve broader coverage while avoiding unnecessary repetition. We implement MemoDroid as a lightweight plugin that can be integrated into existing LLM-based GUI testing systems without modifying their core architecture.

A. *Interaction-level Episodic Memory Construction*

The interaction-level episodic memory captures the agent's testing traces at the granularity of app functionalities. Unlike prior approaches [9], [10], [35] that store full-session logs or unstructured action histories, MemoDroid segments the testing process into discrete function units, each representing a complete interaction sequence. This unit-based organization aligns with how app testing is typically conducted, where human testers also observe, verify, and document app behavior in terms of functionalities rather than isolated actions.

As shown in Fig. 1, the construction of episodic memory involves three components: (1) segmenting continuous interaction traces into function units based on GUI transitions and action semantics; (2) constructing an interaction graph for each unit, where nodes represent GUI pages and edges represent labeled actions; (3) encoding each unit using multimodal representations that integrate visual embeddings, textual interactions, and structural features.

*1) **Function Unit Segmentation**:* During the automated GUI testing process, the testing agent generates a continuous sequence of GUI states and actions. To organize these traces into episodic memory, we segment them into discrete function units, each representing a self-contained user task such as adding an expense or editing an entry. As shown in Fig. 1, the segmentation consists of three components:

Screenshot Annotation. We first annotate the screenshots captured at each interaction step by identifying the target component using its bounding box from the view hierarchy file. As shown in Fig. 2 (a), we draw red rectangles over the interactive components using OpenCV's `rectangle()` function. Each 10 consecutive steps is grouped as a sliding window, and the annotated screenshots are concatenated vertically using OpenCV's `vconcat()` to form a composite image. Within the image, each action is assigned an index from 1 to 10 for reference.

Textual Sequence Construction. For each screenshot in the function unit, we construct a textual sequence describing the interaction as shown in 2 (b). We extract its (1) page index, (2) activity name, (3) action type ('click', 'input', 'scroll', 'long_click'), and (4) the most informative component identifier, which selected in order of priority from the 'text' attribute, then 'label', and finally 'id'.

Functional Segmentation with MLLM. The visual-textual pair is input to a Multimodal Large Language Model (MLLM), which is prompted to analyze the functional structure of the sequence. Rather than outputting a binary decision over the whole window, the MLLM identifies the index ranges within the 10 steps that correspond to complete function units. As shown in 2, steps 1 to 6 are identified as a coherent function unit corresponding to the process of adding an expense. The generated description is: "This sequence completes an expense creation process, including amount entry, category selection, and confirmation." Once the units within a window are identified, the next window begins at the first unused step. For example, if 1 to 7 is a unit, the next window starts from step 7 and spans pages 7 to 16. This process continues iteratively until the entire trace is segmented. The output is a list of segmented function units. Each unit consists of: (1) a long image showing the interaction sequence; (2) a structured textual log of the involved steps; and (3) an MLLM-generated natural language summary describing the functionality.

*2) **Interaction Graph Construction**:* To structurally capture the flow of actions within each function unit, we construct a directed interaction graph $G = (V, E)$, where each node $v_i \in V$ corresponds to a GUI state (i.e., a screenshot and its associated

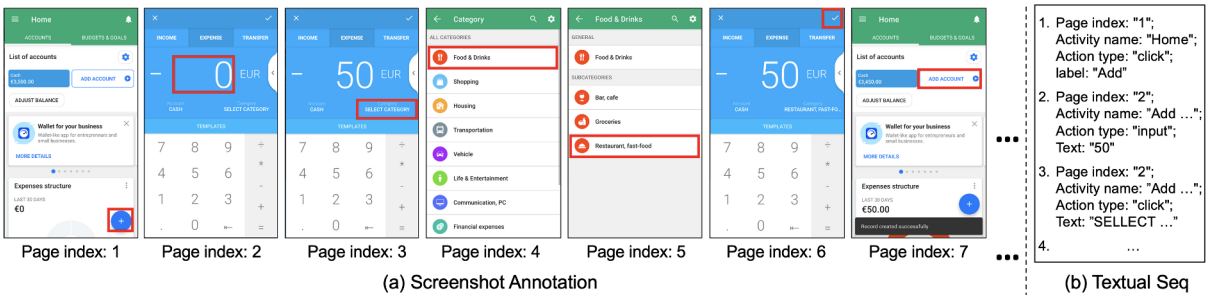


Fig. 2: Example of MemoDroid's screenshot annotation and textual sequence construction

view hierarchy), and each directed edge $e_{i \to j} \in E$ represents an interaction transitioning the app from v_i to v_j. Each edge is labeled with the action (e.g., click, scroll, input) and the corresponding component identifier (component's text, label, or ID). This labeling preserves both the transition semantics and component-level behavior. The interaction graph is later used during the embedding phase.

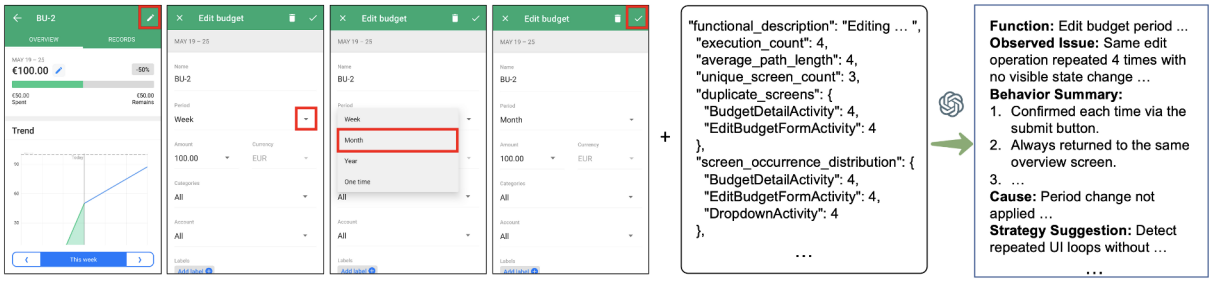
*3) **GUI Information Encoding**:* To enable memory retrieval and similarity comparison across function units, each segmented unit is encoded into a compact multimodal representation. We adopt a dual-encoder architecture based on the CLIP model [36]. For each function unit, the annotated screenshots are resized to a fixed resolution and individually passed through a pre-trained CLIP ViT-B/16 vision encoder, yielding one visual embedding per step. These embeddings are then aggregated via average pooling to obtain a single visual vector for the unit. In parallel, we encode the textual sequence of the function unit using the CLIP text encoder, yielding a global textual embedding that summarizes the interaction history in natural language. To incorporate behavioral structure, we apply a two-layer Graph Convolutional Network (GCN) over the interaction graph described in Section II-A2. Finally, we concatenate the visual, textual, and structural embeddings and project the result into a unified latent space using a fully connected layer with ReLU activation. The resulting vector is stored in episodic memory and used for future retrieval, clustering, and similarity alignment. In addition to encoding full function units, we also apply the same procedure to encode the first screenshot of each unit independently. This enables finer-grained retrieval in Section II-D, especially when only the first page of a function is visible at the beginning of new functions.

B. Function-Level Reflective Memory Generation

To enable experience abstraction and facilitate high-level memory reuse, MemoDroid constructs a function-level reflective memory by analyzing episodic memory and summarizing its behavioral characteristics. MemoDroid first merges functionally similar testing traces to reduce redundancy. It then identifies issue-triggering actions during testing, which are key interactions likely to reveal issues or abnormal behaviors. MemoDroid finally generates a natural language summary for each function unit by prompting an MLLM with the annotated screenshots and structured metadata from the function units.

*1) **Function Trace Merging**:* MemoDroid merges functionally similar units based on three aspects: the activity name sequence, the sequence of action-component pairs, and the GUI page structure across steps. For each function unit, MemoDroid extracts: (1) the activity name at each step; (2) the action type with its corresponding UI component identifier (text, label, or ID); and (3) a structure hash computed from the view hierarchy of each GUI page. We compute a weighted similarity score by combining the Levenshtein similarity of activity sequences, the Jaccard similarity of action-component sets, and the perceptual hash match score between GUI pages. Scores that exceed a predefined threshold of 0.8 (Based on our pilot study and existing work [5], [37], [38]) are considered equivalent and merged. After merging, we construct a JSON object to summarize the behavior across merged traces. This includes: the number of times the function was executed, the average path length, the number of unique screens, duplicate screen counts, screen occurrence distribution, the most frequent start page, and the functional description inherited from episodic memory.

*2) **Issue Action Detection**:* MemoDroid identifies and records issue actions within each function unit. These issue actions are stored as part of the reflective memory and provide actionable insights for strategy generation and bug identification. We detect three types of issue actions. For each, we record the associated component and screen to support traceability and related analysis. First, *unresponsive actions*, if the GUI remains unchanged after interaction. We compare the pre- and post-action screenshots and view hierarchy trees; if no visible or structural difference is found, the operation is classified as unresponsive. Second, *redundant actions*, when the same action is repeatedly executed on the same component across multiple steps. These patterns often indicate exploration inefficiencies or unintended loops. For each such case, we identify the component, its screen, and repetition count. Third, *bug-triggering actions*, if they cause a bug such as a crash. The triggering component and screen are recorded for each instance. All detected issue actions are embedded into the structured metadata of the reflective memory and used to generate natural language reflections and inform strategy-level planning.

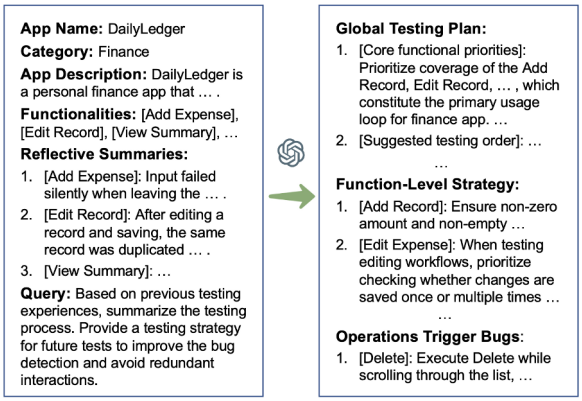


(a) Function unit (b) Testing Behavior Summary (c) Reflective memory

Fig. 3: Example of reflective memory generation

*3) **Natural Language Reflection Generation**:* MemoDroid generates a high-level natural language reflection for each function unit, which summarizes the testing behavior and issue actions. As shown in Fig. 3, the input consists of: (1) the image of the function unit; and (2) a *testing behavior summary*, which records testing behavior data aggregated during the merging (Section II-B1) and issue action detection (Section II-B2).

We construct a prompt that incorporates both the visual and structured information. This prompt is input for an MLLM, which is instructed to generate a concise reflection describing the observed issues, behavior summary, cause, and suggestions as shown in Fig. 3 (c). These reflections provide both human-readable insights and LLM-compatible summaries that can be recalled during future tests.



(a) Prompt (b) Strategic memory

Fig. 4: Example of strategic memory generation

C. App-Level Strategic Memory Summarizing

To support planning at the app level and facilitate experience transfer across applications, MemoDroid constructs a strategic memory layer that stores high-level testing strategies derived from reflective memory. These strategies summarize functional testing priorities, recurrent interaction risks, and bug-trigger actions commonly observed in specific app categories or usage scenarios. We construct a prompt that integrates the app metadata and the associated reflections. This prompt is provided to an MLLM, which is instructed to generate an app-level strategy summary.

*1) **App-Level Strategy Summary Generation**:* The input includes: (1) the app's name, category, and description; and (2) the set of natural language reflections. As shown in Fig. 4 (a), we integrate these inputs into a structured prompt, it ends with a query requesting a concise and reusable testing strategy. As shown in Fig. 4 (b), the output includes three parts: (1) a global testing plan indicating testing priorities and suggested execution order; (2) function-level strategies offering specific testing for key operations; and (3) actions that may trigger bugs. The generated strategy is stored in textual format as part of the app's strategic memory. It serves as reusable guidance when retesting the same apps or testing apps with similar structures or functionality.

*2) **Cross-App Strategy Abstraction**:* To further enhance the generalizability of testing knowledge, we abstract higher-level strategies by aggregating the app-specific summaries across multiple apps belonging to the same category, such as e-commerce, social media, or utilities. The input to this step consists of the natural language strategy summaries generated for each app in the previous stage. Each summary encapsulates the testing experiences and behavioral observations specific to one app. We group these summaries based on app categories. For each category, we concatenate the app-level strategy summaries and use an MLLM to synthesize a consolidated domain-level testing strategy, which is instructed to identify recurring patterns, frequent sources of bugs, and best practices that apply across the set of applications.

D. Runtime Memory Invocation

Runtime memory invocation refers to the process by which MemoDroid dynamically retrieves relevant memory entries during testing. This mechanism enables the system to leverage accumulated experience to guide decision-making in contextually appropriate moments. MemoDroid triggers memory invocation at the start of a new app, testing stagnation, or functional transitions. It analyzes the current GUI state, recent interaction history, and overall app metadata to retrieve related memory from the episodic, reflective, and strategic memory. Retrieved content is organized into a prompt, which is integrated into the LLM input to enhance its reasoning and action selection, without modifying the underlying testing framework.

*1) **On-Demand Memory Selection**:* To enable memory utilization during testing, MemoDroid adopts an on-demand

memory selection mechanism, where memory is retrieved only under specific runtime conditions. This selective strategy avoids unnecessary overhead while ensuring that guidance is provided when it is most impactful. MemoDroid defines three types of triggers: cold start, testing stagnation, and functional transitions.

Cold Start. At the beginning of testing a new app, MemoDroid retrieves app-level strategies from strategic memory. The retrieval is based on app textual information, including the app name, category, app description, and the list of activity names. These inputs are encoded into sentence embeddings and compared the app textual information of existing strategic memory entries using cosine similarity.

Mid-testing Stagnation. During testing, MemoDroid monitors the testing trace and detects stagnation based on: (1) more than 60% of recent actions revisit previously explored screens; (2) fewer than 3 unique action types appear in the past 10 steps; (3) no new function unit is completed within the last 20 steps. When stagnation is detected, MemoDroid queries the reflective memory using the encoding of the most recent function unit in Section II-A3. This encoding includes visual, textual, and graph structure information of the function unit.

Functional Transitions. After completing one function unit, MemoDroid anticipates a transition to a new functionality. It retrieves episodic memory entries related to the current GUI page, including the current page's screenshot and textual description, encoded using the same vision-language encoder as in Section II-A3. These embeddings are matched against the first-page embeddings of previously tested function units in memory. The retrieved episodic memory contains natural language summaries describing what functionality was previously explored from the similar UI page, including functionality description and the action sequences.

*2) **Memory Prompt Generation**:* After retrieving relevant memory, MemoDroid organizes them into structured prompts and injects them into the front of the prompt as the LLM's input of LLM-based GUI testing to guide testing decisions.

When strategic memories are retrieved during cold start, the app-level strategic memory are inserted into the prompt as high-level exploration guidance. Memory is prefixed with a system instruction indicating it originates from prior testing of similar apps and should be considered when planning initial actions. During testing stagnation, reflective memory are injected with the prompt to provide behavioral references, which describe the execution trace of a similar function unit. During functional transitions, MemoDroid retrieves function units from episodic memory whose starting pages match the current pages. The corresponding function unit testing tace are inserted into the prompt.

E. Implementation

In our implementation, we utilize the GPT-4o-mini[2] to handle both textual and visual information during the testing process. It obtains the view hierarchy file of the current GUI page through UIAutomator [39] to extract text information of the input widgets. MemoDroid is implemented as a lightweight plugin, which is compatible with existing LLM-based GUI testing systems and can be integrated without modifying their core architecture.

[2]https://openai.com/index/gpt-4o-mini-advancing-cost-efficient-intelligence/

III. Experiment Design

We design four research questions to evaluate the effectiveness, component contribution, and practical usefulness of MemoDroid:

- **RQ1 (Effectiveness Evaluation):** Can memory accumulated from previously tested apps improve GUI testing performance on new apps across categories?
- **RQ2 (Ablation Study):** How does each memory layer (episodic, reflective, strategic) contribute to the overall effectiveness of MemoDroid?
- **RQ3 (Memory Evolving Effect):** Does expanding the memory pool continuously improve GUI testing performance?
- **RQ4 (Usefulness Evaluation):** Can LLM-based GUI testing with MemoDroid discover unknown bugs in large-scale real-world apps?

A. Dataset

For RQ1 and RQ2, we evaluate whether testing memory from previously explored apps can enhance GUI testing performance on new apps. The experimental data consists of two parts: a *memory pool* and a *testing set*.

We do not adopt existing benchmarks such as Themis [40] due to their limited scale (20 apps) and outdated content (mostly before 2019), which are often incompatible with current Android systems. Following the collection principles of Themis, we curate our own dataset from F-Droid by selecting apps that: (1) were updated after January 2025, (2) are compatible with all baseline methods, (3) contain at least 12 activities, (4) support UIAutomator-based screenshot and view hierarchy extraction, and (5) include at least one manually verifiable functional bug.

We collect 90 apps that satisfy these criteria, covering 15 distinct categories with 6 apps per category. Among them, 60 apps (4 per category) are used to build the memory pool, while the remaining 30 apps (2 per category) form the testing set. The testing set apps are strictly excluded from memory construction to ensure fair evaluation.

For RQ3, we study whether an evolving memory pool leads to continuous improvements in testing performance. Specifically, we randomly partition the 60 apps used for memory construction into 12 groups, each containing 5 apps. These groups are incrementally added to the memory pool across 12 rounds. In round i, the memory pool consists of the first i apps. After each round, we test on the same 30 apps from RQ1 to measure how memory evolution affects generalization.

For RQ4, we assess the real-world usefulness of MemoDroid by applying it to 200 popular Android apps from Google Play, covering a diverse range of categories. These apps are selected based on popularity and recent update time.

We initialize the memory with the full set of 60 apps from RQ1 and evaluate whether memory-guided testing can uncover previously unknown bugs. Once potential issues are found, we report them to developers through GitHub or official email channels and track whether the bugs are confirmed or fixed.

B. Baselines

To evaluate the effectiveness of our proposed memory mechanism, we compare MemoDroid with five state-of-the-art LLM-based GUI testing approaches that do not incorporate memory support. Previous studies shows that these LLM-based methods are more effective than traditional GUI testing methods. However, we do not compare them with traditional methods again. **GPTDroid** [6] transforms GUI testing into a question-answering process with LLMs. It encodes GUI state into structured prompts, elicits next actions from the LLM. **DroidAgent** [8] introduces an intent-driven multi-agent framework for LLM-based testing. It generates semantic task goals and decomposes them into actionable steps. **AUITestAgent** [7] combines dynamic proxy organization and multi-dimensional GUI information extraction. **VisionDroid** [10] employs a multi-agent MLLM framework. It aligns GUI screenshots with textual information and coordinates three agents to generate the testing path. **Guardian** [9] enhances LLM-based GUI testing through external runtime enforcement. It optimizes the action space using domain constraints and supports dynamic re-planning by offloading inconsistent test paths, improving the stability and focus of LLM decisions. All baselines are evaluated using their released implementations or replicated pipelines following original configurations.

C. Experimental Setup

We implement MemoDroid as a lightweight plugin module integrated into existing LLM-based GUI testing frameworks. All experiments are conducted on a workstation equipped with an Intel Core i7-12700F CPU, 32GB DDR5 RAM, and a single NVIDIA GeForce RTX 3060 GPU (12GB), running Android emulator instances to ensure consistency across runs. For each experiment, we compare the performance of baseline testing methods with and without MemoDroid augmentation. To ensure fairness, both versions use the same LLM backbone and are initialized with identical configurations. Each tool is given 60 minutes to test each app, following the common practice in prior studies [40]–[42]. For apps requiring login, we register test accounts and pre-script login flows. Before each run, app data is cleared to avoid cross-run interference. We run each tool three times and obtain the average performance to mitigate potential bias.

To construct the memory pool used in RQ1, RQ2, and RQ4, we first run five LLM-based GUI testing baselines on 60 apps (Section III-A). Each app is tested independently, and the testing traces are recorded and organized into memory following the layered structure of MemoDroid. These traces are later encoded into episodic, reflective, and strategic memory components to enable experience reuse during testing of unseen apps. For each baseline, the same memory pool is used for fair comparison against its MemoDroid enhanced variant.

For RQ3, we begin with an empty memory and incrementally grow the pool by adding one app at a time from a preselected set of 60 apps. In each of the 12 rounds, the memory pool consists of the first i groups (i.e., $5 \times i$ apps). After each update, we evaluate testing performance on the same 30 apps used in RQ1.

D. Evaluation Metrics

We measure the performance from bug detection and testing coverage of our MemoDroid and baselines. For bug detection, we use the number of bugs detected by automated GUI testing tools. For testing coverage, we obtain the activity coverage and code coverage [4], [43]–[47], in which we treat the activities defined in the *AndroidManifest.xml* file of an Android app as the whole set of activities [5], [38], [48].

IV. Results and Analysis

A. Effectiveness of GUI Testing with MemoDroid (RQ1)

Table I shows the average testing performance of five baseline approaches, and they are integrated with MemoDroid across 30 testing apps. We observe that integrating with MemoDroid improves the performance of all baselines. Compared to their original versions, all memory-enhanced variants achieve higher bug detection, activity and code coverage.

TABLE I: Comparison between MemoDroid and baselines.

GUI Testing Method	Average Coverage		Average Bugs
	Activity	Code	
GPTDroid	0.29	0.27	0.50
GP + MemoDroid	**0.52 ↑ 79%**	**0.49 ↑ 81%**	**1.47 ↑ 194%**
DroidAgent	0.28	0.27	0.57
DA + MemoDroid	**0.55 ↑ 96%**	**0.51 ↑ 89%**	**1.70 ↑ 198%**
AUITestAgent	0.37	0.36	1.10
ATA + MemoDroid	**0.69 ↑ 86%**	**0.67 ↑ 86%**	**1.73 ↑ 57%**
VisionDroid	0.41	0.39	1.23
VD + MemoDroid	**0.77 ↑ 88%**	**0.73 ↑ 87%**	**2.10 ↑ 71%**
Guardian	0.40	0.38	1.30
GU + MemoDroid	**0.77 ↑ 93%**	**0.75 ↑ 97%**	**2.23 ↑ 72%**

Notes: "GP" is the GPTDroid, "DA" is the DroidAgent, "ATA" is the AUITestAgent, "VD" is the VisionDroid, "GU" is the Guardian.

Specifically, activity coverage increases by 79% - 96%, and code coverage improves by 81% - 97%, indicating that MemoDroid enables more effective activity and code exploration. In terms of bug detection, the number of bugs discovered increases by 57% - 198%. These results show that LLM-based GUI testing with MemoDroid not only expands coverage but also improves the ability to focus on bug-prone behaviors. The performance gains hold across both text-based and vision-based baselines, confirming the general applicability of MemoDroid.

We further manually review the testing records and analyze the reasons for the performance improvement of the 5 GUI testing approaches integrated with MemoDroid. First, the

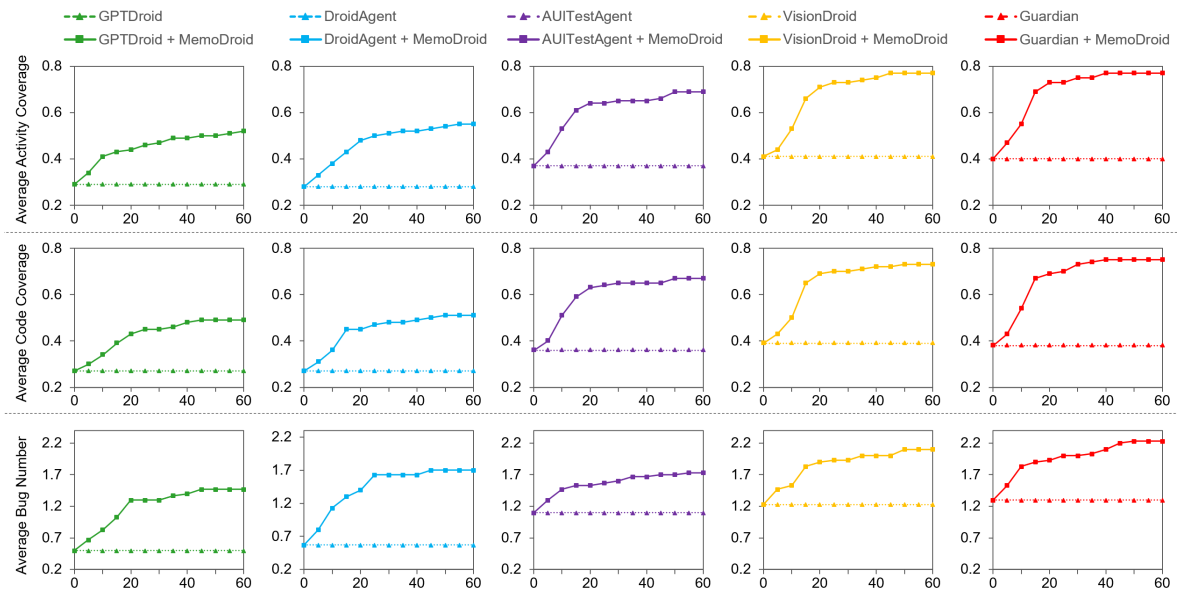


Fig. 5: Impact of memory pool evolving

episodic memory provides low-level trace reuse, which allows the GUI testing approaches to avoid redundant exploration and replicate effective actions. Second, the *reflective memory* identifies issue actions. For instance, It helps GUI testing approaches understand that submitting forms with empty fields or switching certain settings often leads to unexpected crashes. Third, the *strategic memory* provides high-level exploration strategies to assist GUI testing approaches in making decisions in unfamiliar applications.

B. *Ablation Study (RQ2)*

To evaluate the contribution of each memory layer in `MemoDroid`, we conduct an ablation study by selectively removing one layer at a time and comparing the performance with the full system. Specifically, we consider three variants: *w/o Episodic Memory*, *w/o Reflective Memory*, and *w/o Strategic Memory*, which delete memory from different layers.

As shown in Table II, removing any memory layer leads to a notable drop in performance, confirming that all three components are essential. Among them, reflective memory contributes the most. Removing it results in a 38% - 25% reduction in activity coverage, 37% - 25% in code coverage, and 39% - 19% fewer bugs detected. This highlights the importance of behavioral summarization and failure-driven guidance. Episodic memory also plays a key role by supporting low-level reuse. Its removal causes a 26% - 23% drop in activity coverage and a 29% - 22% drop in code coverage. Strategic memory contributes high-level testing guidance and yields a 22% - 15% reduction in activity coverage and a 22% - 16% drop in code coverage when removed.

Overall, these results demonstrate that `MemoDroid` achieves its effectiveness by combining all three types of memory: reflective memory offers the strongest improvement, while episodic and strategic memory provide complementary support for trace reuse and test planning.

TABLE II: Comparison between `MemoDroid` and variants.

GUI Testing Method	Average Coverage		Average Bugs
	Activity	Code	
GP + `MemoDroid`	**0.52**	**0.49**	**1.47**
w/o Episodic Memory	0.39 ↓ 25%	0.38 ↓ 22%	1.13 ↓ 23%
w/o Reflective Memory	0.36 ↓ 31%	0.34 ↓ 31%	1.00 ↓ 32%
w/o Strategic Memory	0.44 ↓ 15%	0.41 ↓ 16%	1.23 ↓ 16%
DA + `MemoDroid`	**0.55**	**0.51**	**1.70**
w/o Episodic Memory	0.41 ↓ 25%	0.39 ↓ 24%	1.30 ↓ 24%
w/o Reflective Memory	0.34 ↓ 38%	0.32 ↓ 37%	1.03 ↓ 39%
w/o Strategic Memory	0.43 ↓ 22%	0.41 ↓ 20%	1.37 ↓ 19%
ATA + `MemoDroid`	**0.69**	**0.67**	**1.73**
w/o Episodic Memory	0.52 ↓ 25%	0.51 ↓ 24%	1.40 ↓ 19%
w/o Reflective Memory	0.49 ↓ 29%	0.47 ↓ 30%	1.27 ↓ 27%
w/o Strategic Memory	0.55 ↓ 20%	0.53 ↓ 21%	1.50 ↓ 13%
VD + `MemoDroid`	**0.77**	**0.73**	**2.10**
w/o Episodic Memory	0.57 ↓ 26%	0.52 ↓ 29%	1.80 ↓ 14%
w/o Reflective Memory	0.53 ↓ 31%	0.49 ↓ 33%	1.70 ↓ 19%
w/o Strategic Memory	0.61 ↓ 21%	0.57 ↓ 22%	1.87 ↓ 11%
GU + `MemoDroid`	**0.77**	**0.75**	**2.23**
w/o Episodic Memory	0.59 ↓ 23%	0.57 ↓ 24%	1.77 ↓ 21%
w/o Reflective Memory	0.58 ↓ 25%	0.56 ↓ 25%	1.60 ↓ 28%
w/o Strategic Memory	0.62 ↓ 19%	0.59 ↓ 21%	1.83 ↓ 18%

Notes: "GP" is the GPTDroid, "DA" is the DroidAgent, "ATA" is the AUITestAgent, "VD" is the VisionDroid, "GU" is the Guardian.

C. *Impact of Memory Pool Evolving (RQ3)*

To evaluate whether the evolution of memory pools will gradually improve the performance of testing, we conduct a memory evolution experiment. Fig. 5 shows the results across five baselines. We observe an upward trend across

all three metrics as the memory pool evolves. This demonstrates that MemoDroid benefits from accumulated experience even when the memory is constructed from a small number of diverse apps. After about 5 apps, the baselines with MemoDroid begin to outperform them without MemoDroid. As more apps are added, the performance gap widens, confirming that MemoDroid can generalize across categories.

For instance, *VisionDroid + MemoDroid* improves its activity coverage from 0.41 (baseline) to 0.71 after 20 memory apps, while bug detection increases from 1.23 to 1.90. *Guardian + MemoDroid* shows a similar trend, with code coverage rising from 0.38 to 0.69. The improvement is smooth, indicating that MemoDroid can continually integrate new testing memory and experience to support long-term testing performance growth. These results validate the scalability of MemoDroid's memory mechanism. It not only enables individual reuse of prior testing history but also accumulates testing strategies and behavior patterns in a way that generalizes across unseen apps.

TABLE III: Information of the fixed bugs.

ID	App name	Category	Down	Version	GP	DA	ATA	VD	GU
1	Traveloka	Travel	50M+	5.20.0					
2	Yelp	Food	50M+	25.11					
3	BeautyCam	Photo	50M+	12.5.35					
4	Wattpad	Book	50M+	11.2.0	*	*	*	*	*
5	Strava	Health	50M+	4.2.5					
6	Klook	Travel	10M+	7.20.2					
7	HelloTalk	Edu	10M+	6.0.6					
8	Xbrowser	Commun	10M+	5.2.0					
9	MMWbO	Health	10M+	25.2.0	*	*			*
10	SportsTrack	Health	10M+	5.0.1					
11	Deliveroo	Food	10M+	3.215					
12	KiKUU	Shop	10M+	30.1.6		*			
13	Wink	Video	10M+	2.7.5	*			*	*
14	StepsApp	Health	10M+	6.0.12					
15	eSound	Music	10M+	6.11					
16	HabitNow	Product	5M+	2.2.3d					
17	Sectograph	Product	5M+	2.3.1					
18	Comera	Commun	5M+	5.0.23					*
19	NetSpeed	Tool	5M+	1.10.0					*
20	Speaky	Edu	5M+	3.3.5					
21	FSBrowser	Social	5M+	8.4.38	*		*		
22	Sketchbook	Art	5M+	6.10.0					
23	FlipaClip	Art	5M+	4.2.5					
24	ClickUp	Product	1M+	5.4.6				*	
25	Tawasal	Commun	1M+	5.4.0		*		*	*
26	VERO	Social	1M+	2.3.3					
27	Hilokal	Edu	1M+	12.15.3					
28	9Weather	Weather	1M+	1.134				*	*
29	BEA	Finance	1M+	10.0.1					
30	Joytify	Music	1M+	1.2.9					
31	Weezer	Music	1M+	2.4.4					
32	Vyke	Commun	1M+	1.20.3					
33	DailyExpen	Finance	1M+	249					
34	Supershift	Product	1M+	25.15					
35	Timeleft	Travel	500K+	3.3.0					

Notes: "GP" is the GPTDroid, "DA" is the DroidAgent, "ATA" is the AUITestAgent, "VD" is the VisionDroid, "GU" is the Guardian.

D. Usefulness Evaluation (RQ4)

We follow the settings in RQ1 in Section IV-A and run both the original baseline approaches and the baseline approaches integrated with our MemoDroid to test 200 apps. For the 200 apps, MemoDroid detects 93 crash bugs in 66 apps, of which 49 bugs in 47 apps are never discovered before. Furthermore, as shown in Table III, only 7 of these new bugs are detected by the best baseline without MemoDroid, and the bugs detected by these baselines are all subsets of the baseline with MemoDroid. We submitted these 49 bugs to developers, and all of them have been fixed/confirmed so far (35 fixed and 14 confirmed, none of them rejected). This further demonstrates the effectiveness of our proposed MemoDroid in assisting automated GUI testing approaches in detecting bugs. Due to space limitations, Table III presents a part of fixed/confirmed bugs, and the full lists can be found on our website[1]. We also collect the feedback from developers regarding the bugs we submitted."This bug affects a core function and somehow slipped through our internal testing.", "We weren't aware of this bug." These feedbacks also demonstrate the importance of MemoDroid in assisting automated GUI testing approaches.

V. Discussion

A. Effectiveness Across App Versions

We evaluate whether MemoDroid can improve GUI testing across different versions of the same app. In practice, mobile apps evolve incrementally through versioned releases, introducing layout refinements, new features, or behavioral changes. To simulate real-world evolution, we select five historical versions (v2.0 to v5.0) of a real-world finance app, Wallet [49] (Download number: 10M+), covering both minor revisions (e.g., v2.0 to v2.1) and major upgrades (e.g., v2.4 to v4.0/v5.0). At the start of testing, the memory is empty and progressively accumulates experience from earlier versions.

TABLE IV: Performance of different versions of an app.

Testing Method	Activity Coverage (different versions)					All
	v 2.0	v 2.1	v 2.4	v 4.0	v 5.0	Bugs
GPTDroid	0.23	0.25	0.25	0.26	0.23	1
GP+MemoDroid	**0.23**	**0.33**	**0.39**	**0.41**	**0.43**	**2**
DroidAgent	0.27	0.26	0.28	0.26	0.29	1
DA+MemoDroid	**0.27**	**0.35**	**0.41**	**0.41**	**0.46**	**2**
AUITestAgent	0.29	0.28	0.30	0.29	0.31	1
ATA+MemoDroid	**0.29**	**0.37**	**0.38**	**0.44**	**0.49**	**3**
VisionDroid	0.33	0.32	0.35	0.29	0.31	2
VD+MemoDroid	**0.33**	**0.45**	**0.57**	**0.59**	**0.59**	**4**
Guardian	0.30	0.34	0.29	0.33	0.34	3
GU+MemoDroid	**0.30**	**0.47**	**0.53**	**0.61**	**0.64**	**5**

Notes: "GP" is the GPTDroid, "DA" is the DroidAgent, "ATA" is the AUITestAgent, "VD" is the VisionDroid, "GU" is the Guardian.

As shown in Table IV, we compare the performance of baselines with and without MemoDroid. Without memory, baseline coverage remains relatively low and fluctuates across versions (e.g., VisionDroid ranges from 0.29 to 0.35). With memory support, all baselines with MemoDroid show improvements as the versions progress. For instance, Guardian with MemoDroid improves activity coverage from 0.30 (v2.0) to 0.64 (v5.0), while bug detection increases from 3 to 5. These results suggest that MemoDroid effectively supports version-aware testing by reusing past execution knowledge. Interestingly, performance gains are evident even when the app undergoes major UI changes, such as between v2.4 and v4.0, indicating that MemoDroid's reflective and strategic memory

is able to abstract and transfer testing knowledge across moderate structural shifts. This demonstrates the potential of memory-guided testing to enhance regression analysis and ensure robustness in evolving mobile applications.

B. Memory Transfer Across Platforms

We further evaluate whether the testing memory constructed from Android apps can benefit GUI testing on a different platform (Web). We select two representative web GUI testing tools, SeeAct [50] and GBST [51], and test them on ten real-world Web Apps. We reuse the memory pool constructed from the 60 Android apps used in RQ1, without incorporating any platform-specific knowledge from the web domain.

We compare each Web testing tool's original version with the integrated `MemoDroid` version, using a 60-minute testing time budget per app. The original SeeAct and GBST detect 3 and 1 bugs, respectively. While they integrated with `MemoDroid` detect 7 and 4 bugs. This demonstrates that the memory captured from Android tasks can effectively support web GUI testing by providing generalized testing strategies. Despite platform differences in UI structure and interaction modality, `MemoDroid` retrieves reflective and strategic memory that captures high-level exploration goals. This helps guide the GUI testing tools toward semantically meaningful actions and reduces ineffective actions.

C. Insights From Memory Content

Beyond quantitative improvements, `MemoDroid` produces reflective and strategic memory entries that offer actionable insights into app behavior. These summaries can guide both automated agents and human testers in identifying bugs and optimizing testing strategies. For example, in one reflective memory, `MemoDroid` observed that in a Spend Tracker [52] app, accessing the "Add Transaction" page without selecting a transaction type (e.g., income or expense) and immediately tapping "Save" led to a crash. Based on this insight, the GUI testing tool systematically explored variations of this flow in other similar apps, successfully triggering similar bugs. Overall, these insights demonstrate that `MemoDroid`'s memory is not merely a static log but a dynamic experience. In future work, we plan to explore how these memory summaries can assist human testers in prioritizing test scenarios, debugging edge cases, and designing testing strategies.

D. Threats of Validity

The first threat relates to the potential overlap between the apps in the LLM's training data and those used in experiments. Since LLMs are trained on large-scale datasets, including open-source repositories, some of the apps may have been seen during training, potentially influencing the results. To mitigate this threat, we specifically selected apps that were updated after March 2025 for RQ4. This ensures that the apps are new and unseen by the LLM, providing a more robust evaluation of `MemoDroid`.

The second threat is hallucination in LLMs. We provide output examples in the prompt to mitigate its occurrence and ensure the robustness of our `MemoDroid`. Despite these mitigation strategies, hallucinations may still occur. We also design pre-defined recovery rules that automatically correct non-executable actions.

VI. Related Work

A. Automated GUI Testing

To ensure the quality of mobile apps, many researchers study the automatic generation of large-scale test scripts to test apps [53]. Since Android apps are event-based [54]–[57], the most common automated testing approaches are model-based automated GUI testing approaches [3], [6], [38], [41], [46], [58]–[69], which design corresponding models through the analysis of the apps. Due to the lack of consideration for the semantic information of the app's GUI pages, the coverage of model-based approaches is still low. Researchers further proposed human-like testing strategies and designed LLM-based automated GUI testing approaches. GPTDroid [35] used GPT-3.5 to generate the testing script. AppAgent [70] enabled the agent to operate apps through a simplified action space. DroidAgent [8] generated high-level, realistic tasks for GUI testing based on app-specific functionalities. AUITestAgent [7] is used for step-oriented testing. AXNav [71] generated annotated videos to visually and interactively review accessibility test results. UXAgent [72] achieved LLM-driven automated usability testing by simulating thousands of user interactions. TestAgent [73] generated and executed test cases through multimodal perception. However, they focus on improving testing performance within single-session testing tasks, without mechanisms for modeling or reusing prior testing experience. In this paper, we design a memory mechanism that enables LLM-based GUI agents to accumulate and reuse testing knowledge.

B. Memory Mechanisms in LLM-based Agents

Memory emerges as a critical component in enhancing the reasoning and planning capabilities of LLM-based agents, especially in tasks that require multi-step interactions and long-term context [24], [74], [75]. In the domain of dialogue agents, recent work introduces persistent memory modules to track long-term user preferences and conversational history [19]–[23], often leveraging memory or embedding-based retrieval to support context retention. Similarly, embodied agents in robotics and game environments [76]–[79] employ memory to capture past trajectories and outcomes, enabling agents to refine future decisions based on prior experiences. Software automation agent demonstrated the use of memory in planning complex action sequences by chaining prompts or retrieving execution traces [80], [81]. These approaches typically rely on prompt chaining or retrieval-augmented generation (RAG) to feed historical context into LLMs. However, they treat memory as a history buffer or document corpus, limiting their ability to reason about higher-level task abstraction or reuse strategies across contexts. Unlike these works, our approach targets memory construction in the specific setting of LLM-based GUI testing, which requires a structured representation of interaction traces and support for cross-task generalization.

VII. Conclusion

This paper proposes MemoDroid, a three-layer memory mechanism that augments LLM-based GUI testing with the ability to evolve through repeated interaction. These memory layers are dynamically retrieved and injected into LLM prompts at runtime, enabling the agent to reuse successful behaviors, avoid ineffective actions, and prioritize bug-prone paths. We implement MemoDroid as a lightweight plugin, which can be integrated into existing LLM-based GUI testing approaches. Evaluation shows that integrating memory improves test coverage and bug detection across diverse apps and platforms. We contribute a reusable memory pool constructed from 60 Android apps, offering a foundation for future research on memory-augmented testing.

In the future, we will expand this memory repository to cover a broader spectrum of apps, with the long-term goal of building a comprehensive and evolving memory base, which could empower agents to generalize better.

Acknowledgment

This work was supported by the National Natural Science Foundation of China Grant No. 62232016, 62402483, 62072442, 62402484, Major Program of ISCAS Grant No. ISCAS-ZD-202401 and ISCAS-ZD-202302, Innovation Team 2024 ISCAS (No. 2024-66), Basic Research Program of ISCAS Grant No. ISCAS-JCZD-202304.

References

[1] A. Developers, "Ui/application exerciser monkey," 2012.

[2] H. N. Yasin, S. H. A. Hamid, and R. J. Raja Yusof, "Droidbotx: Test case generation tool for android applications using q-learning," *Symmetry*, 2021.

[3] Z. Dong, M. Böhme, L. Cojocaru, and A. Roychoudhury, "Time-travel testing of android apps," in *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE, 2020, pp. 481–492.

[4] Y. Li, Z. Yang, Y. Guo, and X. Chen, "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, 2019, pp. 1070–1073.

[5] M. Pan, A. Huang, G. Wang, T. Zhang, and X. Li, "Reinforcement learning based curiosity-driven testing of android applications," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2020, pp. 153–164.

[6] Z. Liu, C. Chen, J. Wang, M. Chen, B. Wu, X. Che, D. Wang, and Q. Wang, "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*, 2024, pp. 1–13.

[7] Y. Hu, X. Wang, Y. Wang, Y. Zhang, S. Guo, C. Chen, X. Wang, and Y. Zhou, "Auitestagent: Automatic requirements oriented gui function testing," *arXiv preprint arXiv:2407.09018*, 2024.

[8] J. Yoon, R. Feldt, and S. Yoo, "Autonomous large language model agents enabling intent-driven mobile gui testing," *arXiv preprint arXiv:2311.08649*, 2023.

[9] D. Ran, H. Wang, Z. Song, M. Wu, Y. Cao, Y. Zhang, W. Yang, and T. Xie, "Guardian: A runtime framework for llm-based ui exploration," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2024, pp. 958–970.

[10] Z. Liu, C. Li, C. Chen, J. Wang, B. Wu, Y. Wang, J. Hu, and Q. Wang, "Vision-driven automated mobile gui testing via multimodal large language model," *arXiv preprint arXiv:2407.03037*, 2024.

[11] C. Zhang, Z. Yang, J. Liu, Y. Han, X. Chen, Z. Huang, B. Fu, and G. Yu, "Appagent: Multimodal agents as smartphone users," *arXiv preprint arXiv:2312.13771*, 2023.

[12] J. Wang, H. Xu, J. Ye, M. Yan, W. Shen, J. Zhang, F. Huang, and J. Sang, "Mobile-agent: Autonomous multi-modal mobile device agent with visual perception," *arXiv preprint arXiv:2401.16158*, 2024.

[13] H. Wen, Y. Li, G. Liu, S. Zhao, T. Yu, T. J.-J. Li, S. Jiang, Y. Liu, Y. Zhang, and Y. Liu, "Autodroid: Llm-powered task automation in android," in *Proceedings of the 30th Annual International Conference on Mobile Computing and Networking*, 2024, pp. 543–557.

[14] H. Wen, H. Wang, J. Liu, and Y. Li, "Droidbot-gpt: Gpt-powered ui automation for android," *arXiv preprint arXiv:2304.07061*, 2023.

[15] J. Itkonen, M. V. Mäntylä, and C. Lassenius, "Test better by exploring: Harnessing human skills and knowledge," *IEEE Software*, vol. 33, no. 4, pp. 90–96, 2015.

[16] Y. Su, D. Liao, Z. Xing, Q. Huang, M. Xie, Q. Lu, and X. Xu, "Enhancing exploratory testing by large language model and knowledge graph," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–12.

[17] Y. Su, Z. Han, Z. Xing, X. Xia, X. Xu, L. Zhu, and Q. Lu, "Constructing a system knowledge graph of user tasks and failures from bug reports to support soap opera testing," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, 2022, pp. 1–13.

[18] J. Wang, Y. Li, Z. Chen, L. Chen, X. Zhang, and Y. Zhou, "Knowledge graph driven inference testing for question answering software," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–13.

[19] Y. Shao, L. Li, J. Dai, and X. Qiu, "Character-llm: A trainable agent for role-playing," *arXiv preprint arXiv:2310.10158*, 2023.

[20] C. Li, Z. Leng, C. Yan, J. Shen, H. Wang, W. Mi, Y. Fei, X. Feng, S. Yan, H. Wang *et al.*, "Chatharuhi: Reviving anime character in reality via large language model," *arXiv preprint arXiv:2308.09597*, 2023.

[21] Z. M. Wang, Z. Peng, H. Que, J. Liu, W. Zhou, Y. Wu, H. Guo, R. Gan, Z. Ni, J. Yang *et al.*, "Rolellm: Benchmarking, eliciting, and enhancing role-playing abilities of large language models," *arXiv preprint arXiv:2310.00746*, 2023.

[22] J. Zhou, Z. Chen, D. Wan, B. Wen, Y. Song, J. Yu, Y. Huang, L. Peng, J. Yang, X. Xiao *et al.*, "Characterglm: Customizing chinese conversational ai characters with large language models," *arXiv preprint arXiv:2311.16832*, 2023.

[23] Z. Kaiya, M. Naim, J. Kondic, M. Cortes, J. Ge, S. Luo, G. R. Yang, and A. Ahn, "Lyfe agents: Generative agents for low-cost real-time social interactions," *arXiv preprint arXiv:2310.02172*, 2023.

[24] J. Lu, S. An, M. Lin, G. Pergola, Y. He, D. Yin, X. Sun, and Y. Wu, "Memochat: Tuning llms to use memos for consistent long-range open-domain conversation," *arXiv preprint arXiv:2308.08239*, 2023.

[25] C. Packer, V. Fang, S. Patil, K. Lin, S. Wooders, and J. Gonzalez, "Memgpt: Towards llms as operating systems." 2023.

[26] T. Sumers, S. Yao, K. Narasimhan, and T. Griffiths, "Cognitive architectures for language agents," *Transactions on Machine Learning Research*, 2023.

[27] Y. Tsai, M. Liu, and H. Ren, "Rtlfixer: Automatically fixing rtl syntax errors with large language model," in *Proceedings of the 61st ACM/IEEE Design Automation Conference*, 2024, pp. 1–6.

[28] D. Chen, H. Wang, Y. Huo, Y. Li, and H. Zhang, "Gamegpt: Multi-agent collaborative framework for game development," *arXiv preprint arXiv:2310.08067*, 2023.

[29] Y. Li, Y. Zhang, and L. Sun, "Metaagents: Simulating interactions of human behaviors for llm-based task-oriented coordination via collaborative generative agents," *arXiv preprint arXiv:2310.06500*, 2023.

[30] Q. Xie, S. Y. Min, P. Ji, Y. Yang, T. Zhang, K. Xu, A. Bajaj, R. Salakhutdinov, M. Johnson-Roberson, and Y. Bisk, "Embodied-rag: General non-parametric embodied memory for retrieval and generation," *arXiv preprint arXiv:2409.18313*, 2024.

[31] J. Mei, Y. Ma, X. Yang, L. Wen, X. Cai, X. Li, D. Fu, B. Zhang, P. Cai, M. Dou *et al.*, "Continuously learning, adapting, and improving: A dual-process approach to autonomous driving," *arXiv preprint arXiv:2405.15324*, 2024.

[32] F. I. Craik and J. M. Jennings, "Human memory." 1992.

[33] A. D. Baddeley, *Human memory: Theory and practice*. psychology press, 1997.

[34] Y. Li, C. Zhang, W. Yang, B. Fu, P. Cheng, X. Chen, L. Chen, and Y. Wei, "Appagent v2: Advanced agent for flexible mobile interactions," *arXiv preprint arXiv:2408.11824*, 2024.

[35] Z. Liu, C. Chen, J. Wang, M. Chen, B. Wu, X. Che, D. Wang, and Q. Wang, "Chatting with gpt-3 for zero-shot human-like mobile automated gui testing," *arXiv preprint arXiv:2305.09434*, 2023.

[36] A. Radford, J. W. Kim, C. Hallacy, A. Ramesh, G. Goh, S. Agarwal, G. Sastry, A. Askell, P. Mishkin, J. Clark, G. Krueger, and I. Sutskever, "Learning transferable visual models from natural language supervision," in *ICML*, 2021.

[37] Z. Liu, C. Chen, J. Wang, Y. Huang, J. Hu, and Q. Wang, "Guided bug crush: Assist manual gui testing of android apps via hint moves," in *Proceedings of the 2022 CHI conference on human factors in computing systems*, 2022, pp. 1–14.

[38] T. Su, G. Meng, Y. Chen, K. Wu, W. Yang, Y. Yao, G. Pu, Y. Liu, and Z. Su, "Guided, stochastic model-based gui testing of android apps," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, 2017, pp. 245–256.

[39] UIAutomator, "Python wrapper of android uiautomator test tool." https://github.com/xiaocong/uiautomator, 2021.

[40] T. Su, J. Wang, and Z. Su, "Benchmarking automated gui testing for android against real-world bugs," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, pp. 119–130.

[41] T. Gu, C. Sun, X. Ma, C. Cao, C. Xu, Y. Yao, Q. Zhang, J. Lu, and Z. Su, "Practical gui testing of android applications via model abstraction and refinement," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 269–280.

[42] Y. Li, Z. Yang, Y. Guo, and X. Chen, "Droidbot: a lightweight ui-guided test input generator for android," in *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*. IEEE, 2017, pp. 23–26.

[43] Y. He, L. Zhang, Z. Yang, Y. Cao, K. Lian, S. Li, W. Yang, Z. Zhang, M. Yang, Y. Zhang *et al.*, "Textexerciser: feedback-driven text input exercising for android applications," in *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020, pp. 1071–1087.

[44] P. Liu, X. Zhang, M. Pistoia, Y. Zheng, M. Marques, and L. Zeng, "Automatic text input generation for mobile testing," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 2017, pp. 643–653.

[45] Y. L. Arnatovich, L. Wang, N. M. Ngo, and C. Soh, "Mobolic: An automated approach to exercising mobile application guis using symbiosis of online testing technique and customated input generation," *Software: Practice and Experience*, vol. 48, no. 5, pp. 1107–1142, 2018.

[46] J. Wang, Y. Jiang, C. Xu, C. Cao, X. Ma, and J. Lu, "Combodroid: generating high-quality test inputs for android apps via use case combinations," in *ICSE*, 2020, pp. 469–480.

[47] W. Wang, W. Yang, T. Xu, and T. Xie, "Vet: identifying and avoiding ui exploration tarpits," in *FSE*, 2021, pp. 83–94.

[48] "Android," https://developer.android.google/topic/, 2022.

[49] "Wallet," https://play.google.com/store/apps/details?id=com.droid4you.application.wallet, 2025.

[50] B. Zheng, B. Gou, J. Kil, H. Sun, and Y. Su, "Gpt-4v(ision) is a generalist web agent, if grounded," in *Forty-first International Conference on Machine Learning*, 2024. [Online]. Available: https://openreview.net/forum?id=piecKJ2DlB

[51] D. Zimmermann and A. Koziolek, "Gui-based software testing: An automated approach using gpt-4 and selenium webdriver," in *2023 38th IEEE/ACM International Conference on Automated Software Engineering Workshops (ASEW)*. IEEE, 2023, pp. 171–174.

[52] "Spend tracker," https://play.google.com/store/apps/details?id=com.thebudgetingapp.thebudgetingapp&hl=en, 2025.

[53] Q. Xie and A. M. Memon, "Designing and comparing automated test oracles for gui-based software applications," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 16, no. 1, pp. 4–es, 2007.

[54] S. Anand, M. Naik, M. J. Harrold, and H. Yang, "Automated concolic testing of smartphone apps," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, 2012, pp. 1–11.

[55] T. Wu, X. Deng, J. Yan, and J. Zhang, "Analyses for specific defects in android applications: A survey," *Frontiers of Computer Science*, pp. 1–18, 2019.

[56] R. Jabbarvand, J.-W. Lin, and S. Malek, "Search-based energy testing of android," in *ICSE*. IEEE, 2019, pp. 1119–1130.

[57] R. Matinnejad, S. Nejati, and L. C. Briand, "Automated testing of hybrid simulink/stateflow controllers: industrial case studies," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, 2017, pp. 938–943.

[58] N. Mirzaei, J. Garcia, H. Bagheri, A. Sadeghi, and S. Malek, "Reducing combinatorics in gui testing of android applications," in *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. IEEE, 2016, pp. 559–570.

[59] S. Yang, H. Wu, H. Zhang, Y. Wang, C. Swaminathan, D. Yan, and A. Rountev, "Static window transition graphs for android," *Automated Software Engineering*, vol. 25, no. 4, pp. 833–873, 2018.

[60] W. Yang, M. R. Prasad, and T. Xie, "A grey-box approach for automated gui-model generation of mobile applications," in *International Conference on Fundamental Approaches to Software Engineering*. Springer, 2013, pp. 250–265.

[61] X. Zeng, D. Li, W. Zheng, F. Xia, Y. Deng, W. Lam, W. Yang, and T. Xie, "Automated test input generation for android: Are we really there yet in an industrial case?" in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2016, pp. 987–992.

[62] K. Mao, M. Harman, and Y. Jia, "Sapienz: Multi-objective automated testing for android applications," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, 2016, pp. 94–105.

[63] M. Chen, Z. Liu, C. Chen, J. Wang, B. Wu, J. Hu, and Q. Wang, "Standing on the shoulders of giants: Bug-aware automated gui testing via retrieval augmentation," *Proceedings of the ACM on Software Engineering*, vol. 2, no. FSE, pp. 825–846, 2025.

[64] Z. Liu, C. Chen, J. Wang, M. Chen, B. Wu, Y. Huang, J. Hu, and Q. Wang, "Unblind text inputs: predicting hint-text of text input in mobile apps via llm," in *Proceedings of the 2024 CHI Conference on Human Factors in Computing Systems*, 2024, pp. 1–20.

[65] Z. Liu, C. Chen, J. Wang, M. Chen, B. Wu, Z. Tian, Y. Huang, J. Hu, and Q. Wang, "Testing the limits: Unusual text inputs generation for mobile app crash detection with large language model," in *Proceedings of the IEEE/ACM 46th International conference on software engineering*, 2024, pp. 1–12.

[66] Z. Liu, C. Chen, J. Wang, X. Che, Y. Huang, J. Hu, and Q. Wang, "Fill in the blank: Context-aware automated text input generation for mobile gui testing," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 1355–1367.

[67] Z. Liu, C. Chen, J. Wang, Y. Su, Y. Huang, J. Hu, and Q. Wang, "Ex pede herculem: Augmenting activity transition graph for apps via graph convolution network," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 1983–1995.

[68] Y. Huang, J. Wang, Z. Liu, S. Wang, C. Chen, M. Li, and Q. Wang, "Context-aware bug reproduction for mobile apps," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 2336–2348.

[69] Z. Liu, C. Chen, J. Wang, Y. Su, and Q. Wang, "Navidroid: a tool for guiding manual android testing via hint moves," in *Proceedings of the ACM/IEEE 44th international conference on software engineering: companion proceedings*, 2022, pp. 154–158.

[70] Z. Yang, J. Liu, Y. Han, X. Chen, Z. Huang, B. Fu, and G. Yu, "Appagent: Multimodal agents as smartphone users," *arXiv preprint arXiv:2312.13771*, 2023.

[71] M. Taeb, A. Swearngin, E. Schoop, R. Cheng, Y. Jiang, and J. Nichols, "Axnav: Replaying accessibility tests from natural language," in *Proceedings of the 2024 CHI Conference on Human Factors in Computing Systems*, 2024, pp. 1–16.

[72] Y. Lu, B. Yao, H. Gu, J. Huang, J. Wang, Y. Li, J. Gesi, Q. He, T. J.-J. Li, and D. Wang, "Uxagent: An llm agent-based usability testing framework for web design," *arXiv preprint arXiv:2502.12561*, 2025.

[73] Y. Li, Y. Li, and Y. Yang, "Test-agent: A multimodal app automation testing framework based on the large language model," in *2024 IEEE 4th International Conference on Digital Twins and Parallel Intelligence (DTPI)*. IEEE, 2024, pp. 609–614.

[74] W. Wang, L. Dong, H. Cheng, X. Liu, X. Yan, J. Gao, and F. Wei, "Augmenting language models with long-term memory," *Advances in Neural Information Processing Systems*, vol. 36, pp. 74 530–74 543, 2023.

[75] J. Tack, J. Kim, E. Mitchell, J. Shin, Y. W. Teh, and J. R. Schwarz, "Online adaptation of language models with a memory of amortized contexts," *arXiv preprint arXiv:2403.04317*, 2024.

[76] Z. Wang, S. Cai, A. Liu, Y. Jin, J. Hou, B. Zhang, H. Lin, Z. He, Z. Zheng, Y. Yang *et al.*, "Jarvis-1: Open-world multi-task agents with memory-augmented multimodal language models," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2024.

[77] M. Yan, R. Li, H. Zhang, H. Wang, Z. Yang, and J. Yan, "Larp: Language-agent role play for open-world games," *arXiv preprint arXiv:2312.17653*, 2023.

[78] G. Wang, Y. Xie, Y. Jiang, A. Mandlekar, C. Xiao, Y. Zhu, L. Fan, and A. Anandkumar, "Voyager: An open-ended embodied agent with large language models," *arXiv preprint arXiv:2305.16291*, 2023.

[79] X. Zhu, Y. Chen, H. Tian, C. Tao, W. Su, C. Yang, G. Huang, B. Li, L. Lu, X. Wang *et al.*, "Ghost in the minecraft: Generally capable agents for open-world environments via large language models with text-based knowledge and memory," *arXiv preprint arXiv:2305.17144*, 2023.

[80] S. Lee, J. Choi, J. Lee, M. H. Wasi, H. Choi, S. Y. Ko, S. Oh, and I. Shin, "Explore, select, derive, and recall: Augmenting llm with human-like memory for mobile task automation," *arXiv preprint arXiv:2312.03003*, 2023.

[81] L. Zheng, R. Wang, X. Wang, and B. An, "Synapse: Trajectory-as-exemplar prompting with memory for computer control," *arXiv preprint arXiv:2306.07863*, 2023.