LogiDroid: Individual Functional Test Generation via Business Logic Extraction and Adaptation

YAKUN ZHANG, Harbin Institute of Technology, Shenzhen, China and Nanjing University, China
ZIHAN WANG, Harbin Institute of Technology, China
XINZHI PENG, Harbin Institute of Technology, Shenzhen, China
ZIHAO XIE, Harbin Institute of Technology, Shenzhen, China
XIAODONG WANG, Key Lab of HCST (PKU), MOE; SCS, Peking University, China
XUTAO LI, Harbin Institute of Technology, Shenzhen, China
DAN HAO, Key Lab of HCST (PKU), MOE; SCS, Peking University, China
LU ZHANG, Key Lab of HCST (PKU), MOE; SCS, Peking University, China
YUNMING YE*, Harbin Institute of Technology, Shenzhen, China

Functional testing is essential for verifying that the business logic of mobile applications aligns with user requirements, serving as the primary methodology for quality assurance in software development. Despite its importance, functional testing remains heavily dependent on manual effort due to two core challenges. First, acquiring and reusing business logic from unstructured requirements remains difficult, which hinders the understanding of specific functionalities. Second, a significant semantic gap exists when adapting business logic to the diverse GUI environments, which hinders the generation of test cases for specific mobile applications.

To address the preceding challenges, we propose *LogiDroid*, a two-stage approach that generates individual functional test cases by extracting business logic and adapting it to target applications. First, in the Knowledge Retrieval and Fusion stage, two LLM-based agents (i.e., a Semantic-Retrieval Agent and a Knowledge-Fusion Agent), are employed to construct a functional test dataset, retrieve relevant test cases, and extract structured business logic for the target functionality. Second, in the Context-Aware Test Generation stage, two LLM-based agents (i.e., a Perception–Interaction Agent and a Decision–Generation Agent), jointly analyze the extracted business logic and the real time GUI environment to incrementally generate context adaptive functional test cases. This design allows LogiDroid to accurately understand application semantics and use domain expertise to generate complete test cases with verification assertions. We assess the effectiveness of LogiDroid using two widely-used datasets that cover 28 real-world applications and 190 functional requirements. Experimental results show that

*Corresponding author.

Authors' addresses: Yakun Zhang, zhangyk@hit.edu.cn, Shenzhen Key Laboratory of Internet Information Collaboration, Harbin Institute of Technology, Shenzhen, Shenzhen, China, State Key Laboratory of Novel Software Technology and Nanjing University, Nanjing, China; Zihan Wang, 2023112983@stu.hit.edu.cn, Harbin Institute of Technology, Harbin, China; Xinzhi Peng, 2023311107@stu.hit.edu.cn, Shenzhen Key Laboratory of Internet Information Collaboration, Harbin Institute of Technology, Shenzhen, Shenzhen, China; Zihao Xie, 25s151153@stu.hit.edu.cn, Shenzhen Key Laboratory of Internet Information Collaboration, Harbin Institute of Technology, Shenzhen, Shenzhen, China; Xiaodong Wang, wang_xiaodong@stu.pku.edu.cn, Key Lab of HCST (PKU), MOE; SCS, Peking University, Beijing, China; Xutao Li, lixutao@hit.edu.cn, Shenzhen Key Laboratory of Internet Information Collaboration, Harbin Institute of Technology, Shenzhen, Shenzhen, China; Dan Hao, haodan@pku.edu.cn, Key Lab of HCST (PKU), MOE; SCS, Peking University, Beijing, China; Lu Zhang, zhanglucs@pku.edu.cn, Key Lab of HCST (PKU), MOE; SCS, Peking University, Beijing, China; Yunming Ye, yym@hit.edu.cn, Shenzhen Key Laboratory of Internet Information Collaboration, Harbin Institute of Technology, Shenzhen, Shenzhen, China.

ACM 1557-7392/2026/7-ART
https://doi.org/10.1145/3829085

LogiDroid successfully tested 40% of functional requirements on the FrUITeR dataset (an improvement of over 25% compared to the state-of-the-art approaches) and 65% on the Lin dataset (an improvement of over 55% compared to the state-of-the-art approaches). These results demonstrate the significant effectiveness of LogiDroid in functional test generation.

CCS Concepts: • **Software and its engineering** → *Software testing and debugging*.

Additional Key Words and Phrases: Functional testing, GUI test generation, Large language model

1 Introduction

In the past decade, mobile applications have evolved into an indispensable form of software. As of 2025, over four million applications are available in application stores [8], which makes quality assurance a critical priority for developers. Among various quality assurance methodologies, functional testing is essential as it directly verifies the business logic and user experience from an end-user perspective.

Functional testing for mobile applications primarily relies on generating *individual test cases* for different functionalities accessed through Graphical User Interfaces (GUIs). A GUI test case typically involves executing an ordered sequence of *events* on GUI *widgets* that belong to specific GUI *states*, accompanied by *assertions* to verify whether the outcomes align with developer expectations. However, existing automated testing approaches [17, 30, 39, 44, 51, 63, 64, 70, 74] focus mainly on defect detection, such as identifying crashes or resource leaks. These approaches often lack a deep semantic understanding of application functionalities, rendering them unable to generate test cases targeted at specific business logic. With the emergence of Large Language Models [13, 21, 61] (LLMs), several recent approaches [40, 69, 72, 76, 78] explore the target applications and leverage the reasoning capabilities of LLMs to generate GUI event sequences. However, these approaches rely on general-purpose knowledge, which is insufficient for understanding application-specific functionalities and adapting to the dynamic GUI behaviors required for complex functional verification. Moreover, they primarily generate event sequences and are unable to produce assertions, which limits their practical value for real-world functional testing. As a result, functional testing remains heavily dependent on manual effort. This reliance not only limits efficiency and scalability but also introduces quality risks due to subjective inconsistencies.

Despite the increasing demand for functional testing automation, achieving high-quality functional test generation still faces two core challenges.

Challenge 1: Acquisition and Reuse of Business Logic. The generation of high-quality functional test cases relies heavily on understanding the complex business logic of target functionalities [37, 56, 79, 80]. This logic encompasses the specific operational rules and decision-making processes that define a functional requirement. For example, the business logic of a "registration" functionality including entering the registration state, filling all the necessary inputs, and validating the registration result through specific state changes. However, such expertise typically resides in unstructured natural language descriptions or scattered historical test cases, making it difficult to formalize [15, 48]. Meanwhile, since recent approaches still mainly rely on general knowledge from the foundation models, they lack an explicit mechanism to extract, represent, and reuse business logic across applications, making them difficult to generate effective functional test cases.

Challenge 2: Semantic understanding and adaptation of GUI environments. Even when business logic is available, grounding it in the diverse and dynamic GUI environments of mobile applications remains a significant challenge. Each application implements its functionalities through unique GUI widgets, intricate interaction flows, and application-specific layout structures [79]. For example, a "registration" functionality may involve entirely different widgets, state transitions, and validation assertions across different shopping applications. As a result, the same functionality-level logic may correspond to different concrete GUI operations in different applications. This semantic gap makes it difficult to map abstract business logic to the specific GUI implementation of a target application [78], often leading to generated test cases that fail to execute the intended business scenario accurately. Such difficulty is further increased by existing approaches that mainly treat functional test generation as direct

exploration on the target application, without an explicit mechanism to bridge functionality-level knowledge and GUI-level execution.

We observe that although the functionalities of mobile applications vary, those with similar functionalities often exhibit related business logic and testing patterns. This observation indicates that knowledge reuse can be leveraged to generate individual test cases effectively. To this end, we propose **LogiDroid**, a two-stage approach that generates individual functional test cases through business logic extraction and adaptation to the target application. Detailed two stages are as follows.

Stage 1: Knowledge Retrieval and Fusion. This stage involves two LLM-based agents: a *Semantic-Retrieval Agent* and a *Knowledge-Fusion Agent*. The Semantic-Retrieval Agent is responsible for constructing a functional test dataset (containing 294 functional test cases from 71 applications) and automatically retrieving relevant test cases related to the target functionality within the dataset. Furthermore, the Knowledge-Fusion Agent semantically aligns and fuses the retrieved test cases, and extracts the business logic for the target functionality. **This stage establishes a transformation channel from vague requirements to structured business logic for subsequent test generation, effectively addressing the *Challenge 1*.**

Stage 2: Context-Aware Test Generation. This stage relies on the close collaboration between two LLM-based agents: a *Perception–Interaction Agent* and a *Decision–Generation Agent*. The Perception–Interaction Agent serves as the interaction interface between LogiDroid and the target application. It continuously captures multimodal data of target application (e.g., state images and widget texts), thereby providing rich context for decision-making. The Decision–Generation Agent acts as the core reasoning engine of LogiDroid. By jointly analyzing the business logic provided in Stage 1 and the real-time contextual information, it decomposes the testing task into multiple steps and incrementally generates context-adaptive test cases. **This stage maintains the depth of knowledge guidance during test generation while enhancing dynamic adaptability to environmental changes, effectively addressing the *Challenge 2*.**

We conduct a comprehensive evaluation to analyze the effectiveness of LogiDroid using 28 real-world mobile applications, 190 functional requirements, and corresponding test cases from two popular datasets (i.e., the FrUITeR dataset [6] and the Lin dataset [9]). We compare LogiDroid with the state-of-the-art (sota) approaches from both academia and industry, i.e., AutoDroid [72] and AppAgent [76]. On the FrUITeR dataset, the test cases generated by LogiDroid successfully validate 40% of the target functionalities, representing a 25% improvement over the baselines. On the Lin dataset, LogiDroid successfully tests 65% of the target functionalities, outperforming the baselines by 55%. *Note that, LogiDroid is able to generate complete test cases with assertions that the baselines fail to produce.* Overall, these results demonstrate that LogiDroid is effective in functional test generation for industrial applications.

The main contributions of this research are summarized as follows:

- **Methodological innovation.** Given the requirement description of a specific functionality, we propose LogiDroid, a novel approach that generates individual functional test cases (with assertions), which makes the large-scale functional testing possible in industry.
- **Technical design.** We design (1) a novel Knowledge Retrieval and Fusion technique to provide a reliable business logic for test generation; and (2) a novel Context-Aware Test Generation technique that combines multimodal information with domain knowledge to incrementally generate test cases adapted to the target application.
- **System implementation.** We develop a complete prototype tool and construct a functional test dataset containing 294 functional test cases from 71 applications across 13 categories. To promote research reproducibility, **the source code has been released as open source [2]**.
- **Experimental evaluation.** We conduct an empirical evaluation using real-world applications, demonstrating the effectiveness of LogiDroid.

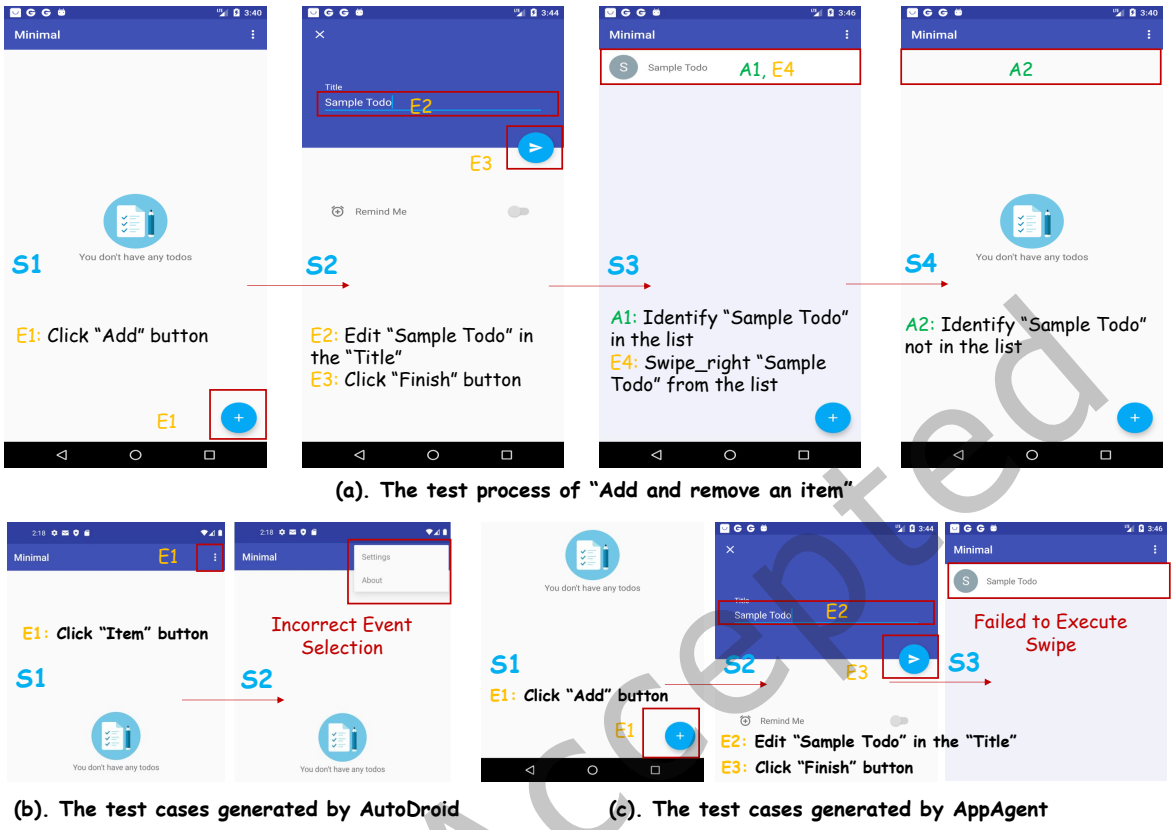

Fig. 1. The test process of "Add and remove an item" and the corresponding results of AutoDroid and AppAgent

2 Preliminaries

2.1 GUI Test Cases

A GUI test case (e.g., Figure 1(a)) typically consists of an ordered sequence of *events* performed on *widgets* across *states*, together with *assertions* that validate the observed outcomes against expected results. A GUI *state* (e.g., S1 in Figure 1(a)) denotes the observable interface of an application at a particular moment during execution, including the current screen and the visible GUI widgets on it. A GUI *widget* (e.g., "Add" button in Figure 1(a)) denotes an interactive element, such as a button, an input field, or a list item. An *event* (e.g., E1 in Figure 1(a)) denotes an action performed on a widget, such as clicking, editing, or swiping. An *assertion* (e.g., A1 in Figure 1(a)) denotes a checking step that determines whether an expected condition holds in the current or a subsequent GUI state, such as whether a target widget appears or disappears.

Accordingly, a GUI test case is a sequence of state transitions driven by events and validated by assertions, as illustrated in Figure 1. This terminology serves as the basis of LogiDroid, which takes a functional requirement as input and generates a functional test case for the target application.

2.2 Illustrative Example

Figure 1 illustrates the test workflow of the "Add and remove an item" functionality in a to-do application [9]. The test case is designed to validate whether a user can successfully add a to-do item and subsequently remove it after finishing the task. We use this case to explain our design motivation. It also helps to illustrate the working mechanism of LogiDroid in later sections.

The test case comprises four GUI states (S1-S4), four events (E1-E4), and two assertions (A1-A2). The test process begins by triggering a click event on the Add button (E1). Subsequently, the text "Sample todo" is input into the Title box (E2), followed by a click operation on the Finish button (E3). Assertion A1 confirms the successful addition of an item by checking whether the text "Sample todo" appears in a new state (S3). Then, a swipe-right operation (E4) is performed on the added item. Finally, assertion A2 validates the removal of the item by detecting whether the item disappears from the state (S4).

We evaluate the performance of two representative functional testing approaches, AutoDroid [72] and AppAgent [76], on the aforementioned test functionality. As shown in Figure 1(b) and Figure 1(c), AutoDroid and AppAgent both fail to generate valid test cases for this functionality. This limitation primarily arises from the lack of effective mechanisms for domain knowledge acquisition and reuse. Consequently, they struggle to understand the functional semantics deeply and generate useful event sequences. Specifically, AutoDroid deviates from the intended workflow by selecting an incorrect event, while AppAgent fails to execute the required swipe interaction and therefore cannot complete the item removal functionality. Furthermore, they fail to generate the necessary verification assertions for the target application.

The significant disparity between test cases generated by existing approaches and target functionalities hinders their direct application in industrial scenarios. For this reason, functional testing for the mobile application still relies on extensive manual correction and maintenance. Therefore, it is imperative to develop innovative approaches capable of generating high-quality functional test cases.

3 LogiDroid

Given the target application and its functional requirements, LogiDroid automatically generates test cases to verify the corresponding functionalities. As illustrated in Figure 2, the overall workflow of LogiDroid consists of two primary stages. The first stage is the **Knowledge Retrieval and Fusion**. This stage aims to retrieve test cases from historical repositories that are semantically relevant to the current functional requirements. It then distills business logic for the corresponding functionalities, providing a reliable domain knowledge to guide subsequent test generation (see Section 3.1). The second stage is the **Context-Aware Test Generation**. In this stage, LogiDroid explores the target application in real-time to capture multimodal information, including GUI layouts and visual screenshots. By integrating the business logic acquired from the first stage with this multimodal context, the system employs a progressive decision-making mechanism to decompose functional requirements into executable test sequences. Note that, unlike existing approaches [44, 72, 76] that only generate events, LogiDroid generates comprehensive test cases with both events and assertions (see Section 3.2).

3.1 Knowledge Retrieval and Fusion

Given the specific functionality to be verified, the Knowledge Retrieval and Fusion stage aims to retrieve functionally similar instances from existing real-world test cases and distill expert-level domain knowledge from them. However, accurately matching the functional requirements within a massive repository of test cases and subsequently extracting reusable domain knowledge remains ***a critical challenge***. To address this challenge, LogiDroid utilizes a dual-agent configuration consisting of the Semantic-Retrieval Agent and the Knowledge-Fusion Agent, as depicted in Figure 2.

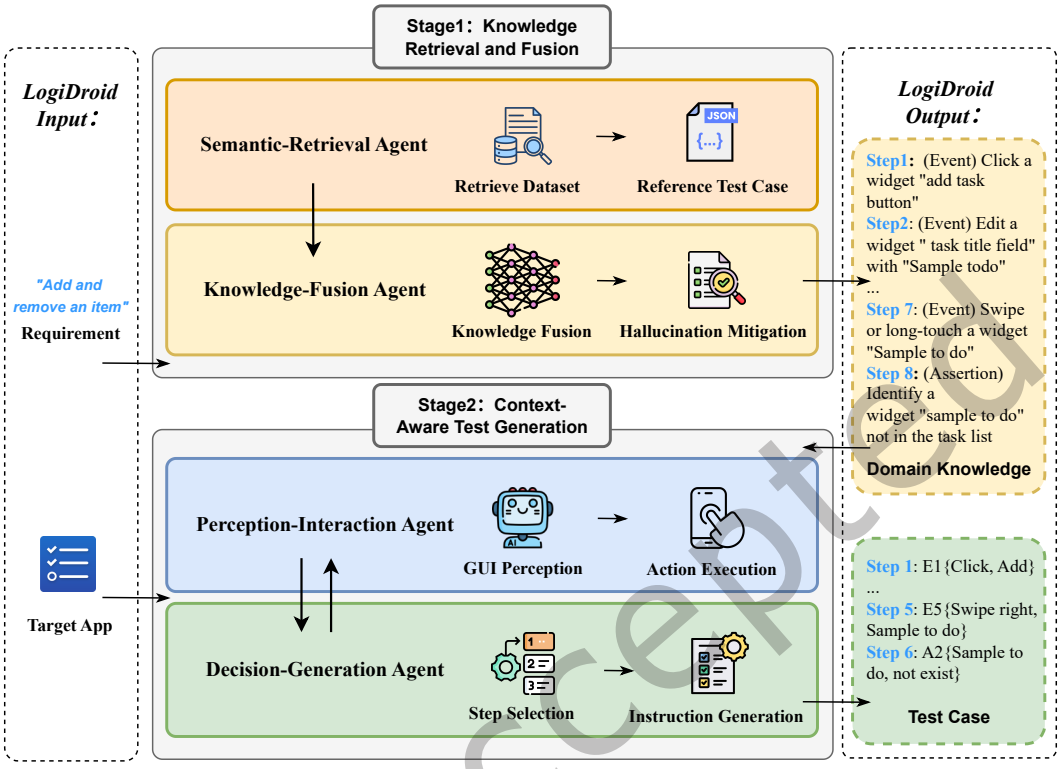

Fig. 2. Overview of LogiDroid

Specifically, we construct a functional test dataset including functional test cases for diverse functionalities. The **Semantic-Retrieval Agent** retrieves relevant test cases within the dataset (see Section 3.1.1). Subsequently, the **Knowledge-Fusion Agent** processes these retrieved cases to distill domain knowledge into structured documentation. This documentation captures the business logic and expert-level testing patterns required to detect the target functionality, serving as a foundation for guiding the subsequent generation process (see Section 3.1.2).

3.1.1 Semantic-Retrieval Agent. In industrial software environments, large repositories of functional test cases contain domain expertise and business logic essential for validating specific functionalities. Despite their significant reference value, these test cases are frequently scattered across heterogeneous sources, which makes their effective use difficult. Therefore, it is crucial to construct a high-quality test dataset and develop mechanisms to retrieve the relevant test cases.

To address the preceding challenge, LogiDroid automatically constructs the functional test dataset and designs the test case retrieval. This process is divided into two main components: *Dataset Construction* and *Test Case Retrieval*.

Dataset construction. This component details the systematic collection of functional test cases and the methodology for generating functional summaries.

Test case collection. To construct the functional test dataset, we systematically collect test cases through the following two channels.

(1) Existing datasets. We collect 95 valid test cases from two established datasets in the application testing domain: the Lin dataset [9] and the FrUITeR dataset [6]. These cases from 28 applications cover six common application scenarios, such as news and shopping.

(2) Open-source projects. To further enhance the diversity and coverage, we systematically collect additional test cases from F-Droid [4], a prominent open-source repository for mobile applications. Similar to related research [31, 63, 81], we focus on eight popular categories (e.g., navigation, sport, and device) including one common category (i.e., shopping) with existing datasets. We firstly select the top 30 applications by download volume in each category. We then exclude applications that do not contain test cases, remaining 43 F-Droid applications with 199 functional test cases.

Functional summary generation. To facilitate semantic retrieval, we generate a concise functional summary for each test case to encapsulate the core functionalities it verifies.

Given the advanced comprehension and summarization capabilities of LLMs, we implement an LLM-based automated generation approach as an alternative to traditional manual authorship or rule-based techniques. This approach offers superior scalability for future dataset expansions while maintaining consistency in quality and semantic accuracy. To guide the LLM in accurately comprehending and executing the summarization task, we design a structured prompt template. This design is motivated by the fact that LLMs demonstrate superior performance when processing structured information, as such formats align more closely with the organizational patterns found in their underlying training data [24, 50, 78]. As illustrated in Table 1, the template consists of the following four parts.

(1) Task Definition. This part explicitly defines the core objective for the LLM, which is to distill the business logic of a complex functional test case into a concise and one-sentence summary. By establishing this specific goal, the model focuses on high-level functional intent rather than low-level implementation details.

(2) Input Object. This part provides the test case to be summarized alongside its corresponding application category. To facilitate model comprehension, LogiDroid imports data in JSON format, decomposing each test case into a series of events and assertions in a semi-structured representation. Specifically, each **event** includes a widget identifier, an operation type, and an optional parameter (as shown in "Step 2" of Table 1). Each **assertion** includes a widget identifier and a verification condition (as shown in "Step 4" of Table 1). To achieve both precision and conciseness, each widget is characterized solely by three key attributes: `text`, `resource-id`, and `content-desc`.

(3) Demonstration Case. Leveraging the powerful in-context learning capabilities of LLMs [23], we provide two complete examples within the prompt template. These examples clearly demonstrate the mapping between raw test cases and their functional summaries, enabling the LLM to align its output with the required format and logic.

(4) Acceptance Criteria. To constrain the model's stochastic behavior and minimize the risk of hallucinations [35], this part enforces three constraints: (i) the output must be concise, adhering to a basic subject-verb-object structure; (ii) the summary must be in natural language rather than code; and (iii) the summary must focus on the primary operations of the test case while disregarding secondary implementation details. These criteria collectively ensure the precision and readability of the generated content.

Dataset Statistics. The final dataset includes 13 common categories, 99 functionalities, across 71 mobile applications, with 294 functional test cases and their corresponding summaries. The statistical details of the dataset are summarized in Table 2. To support efficient retrieval, LogiDroid implements a structured storage format where the application category serves as the primary index (Key_1), the embedding vector of the functional summary serves as the secondary index (Key_2), and the corresponding test case is stored as the value ($Value$).

Table 1. Prompt Template for Functional Summary Generation

AIM	EXAMPLE
Task Definition	You are a functional summary generator. Based on the test cases for the Android app, generate a natural and one-sentence description.
Input Object	Test case from a **[To-do]** app Step 1: (Event) Click a widget "add todo item button" Step 2: (Event) Edit a widget "user todo edit text" with "sample todo" Step 3: (Event) Click a widget "make todo floating action button" Step 4: (Assertion) Identify a widget "sample to do" in the state Step 5: (Event) Swipe right a widget "sample to do" Step 6: (Assertion) Identify a widget "sample to do" not in the state Functional summary:
Demonstration Case	**Example 1**: Test case from a Browser app Step 1: (Event) Click a widget "search" Step 2: (Event) Edit a widget "search" with "news" Step 3: (Event) Identify a widget "latest news" in the state Functional summary: Test the search functionality **Example 2**:...
Acceptance Criteria	Please generate the functional description for the **[To-do]** app. 1. Please keep it simple: only include at most the subject, verb, and object. 2. Please use natural English, not technical terms. 3. Please focus on the main actions, ignore the details.

By integrating category-based filtering with vector-based semantic matching, this multi-dimensional indexing mechanism enables the precise and efficient identification of specific functionalities.

Test case retrieval. Regarding the test requirements provided by the user, LogiDroid first encodes the requirements into an embedding vector and subsequently performs a similarity-based retrieval within the functional test dataset. By calculating the cosine similarity between the user's requirement vector and the functional summary vectors, LogiDroid selects the Top_{sim} results that exhibit the highest similarity scores and belong to the same application category. The test cases associated with these results contain valuable business logic, which serves as a critical reference for the subsequent test generation process and providing the necessary knowledge foundation for verifying specific functionalities.

3.1.2 Knowledge-Fusion Agent. To effectively reuse the domain expertise, the Knowledge-Fusion Agent focuses on extracting *core business logic* from a set of relevant test cases provided by the semantic-retrieval agent. For instance, in the context of registration functionalities, while specific implementations vary across different mobile applications (e.g., utilizing email-password combinations or username-phone number verification), the core business logic follows a standardized pattern. This pattern typically involves navigating to the registration state, inputting valid information, executing the registration action, and verifying the resultant state.

The primary objective of the Knowledge-Fusion Agent is to extract implementation-agnostic core business logic from multiple relevant test cases. By extracting away concrete implementation details, this agent provides

Table 2. The statistics of Functional Test dataset of LogiDroid

Category	Application	Test	Summary	Functionality
News	4	32	32	12
Shopping	7	29	29	12
Browser	12	51	51	10
To-do	5	10	10	2
Mail	2	4	4	2
Calculator	5	10	10	2
Note	6	28	28	14
Navigation	3	6	6	4
Draw	3	9	9	5
System	9	46	46	11
Device	7	29	29	9
Sport	3	24	24	11
Time	5	16	16	5
Total	**71**	**294**	**294**	**99**

reusable domain knowledge for subsequent test generation, effectively emulating the strategic guidance of a human testing expert.

Knowledge fusion. The Knowledge-Fusion Agent provides a structured fusion framework based on LLMs, with the template architecture illustrated in Table 3. This framework comprises four key parts designed to facilitate the extraction of reusable testing patterns.

First, in the "Task Definition", the model is explicitly directed to extract generalized business logic by fusing information from multiple retrieved test cases and the target requirement. Second, the "Input Object" supplies the set of relevant test cases identified by the Semantic-Retrieval Agent. Third, the "Demonstrate Case" provides a complete demonstration of the abstraction and fusion process, clearly defining the expected output format. Finally, the "Acceptance Criteria" enforce three rigorous constraints to enhance the quality of the generated output, including (i) the number of logic steps must remain within a reasonable range to maintain efficiency; (ii) the generated steps must strictly adhere to specified formatting conventions, where events follow the format "[Action] a widget [Widget] with [Value]" and assertions follow the format"Identify a widget [Widget] [Condition]"; (iii) the response must present the core business logic directly without any auxiliary explanations or code instructions.

Hallucination mitigation. To address the challenge of hallucinations in LLMs [35], LogiDroid implements an robust automated detection and feedback mechanism. Through the application of predefined rules, this mechanism rigorously assesses whether the synthesized output satisfies the required formatting conventions, remains within the boundaries for testing steps, and meets the criteria for logical relevance.

When an invalid output is detected, the system automatically generates corrective feedback and triggers a re-generation process, which continues iteratively until the output satisfies all predefined specifications. This closed-loop verification process effectively safeguards the reliability of the knowledge fusion stage and ensures the high quality of the final output.

3.2 Context-Aware Test Generation

While the core business logic extracted from the first stage provides domain expertise for functional verification, significant implementation differences across various mobile applications prevent this logic from being directly

Table 3. Prompt Template for Test Knowledge Fusion

AIM	EXAMPLE
Task Definition	You are a summarizer to fuse test knowledge for the functionality: **[Add and remove an item]** in a **[To-Do]** app.
Input Object	**Related Test Case 1**: Step 1: (Event) Click a widget "skip button" Step 2: (Event) Click a widget "new task button" ... Step 7: (Assertion) Identify a widget "sample to do" not in the state **Related Test Case 2**:...
Demonstrate Case	**Example**: Test knowledge for the functionality: [Test the search functionality] in a [Browser] app. Step 1: (Event) Click a widget "search" or "url" in the search bar Step 2: (Event) Edit a widget "search" or "url" in the search bar with "news" Step 3: (Event) Identify a widget "latest news" in the state
Acceptance Criteria	Please generate the test knowledge for the **[To-do]** app. 1. The generated test step do not too short or too long. 2. Please strictly use steps in the format of Event and Assertion (1) (Event) [Action] a widget [Widget] with [Value] (2) (Assertion) Identify a widget [Widget] [Condition] 3. Please do not include any code, XPATH, or scripting instructions

Algorithm 1 Agent Interaction Loop (Stage 2)

Input: App, Requirement R, Core Logic L
Output: Test Case T

while Task Not Complete **do**
  // Perception-Interaction Agent (PIA) & Decision-Generation Agent (DGA)
  $S_{curr} \leftarrow PerceiveState(App)$ ▷ **PIA**: Capture multimodal state
  $Step \leftarrow StepSelection(R, L, S_{curr})$ ▷ **DGA**: Align logic with state
  $Instr \leftarrow InstructionGeneration(Step, S_{curr})$ ▷ **DGA**: Generate executable action
  $Execute(App, Instr)$ ▷ **PIA**: Interact with device
  if $CompletionJudgment(Step, Instr, S_{curr})$ **then**
    ▷ **DGA**: Verify step completion
    $Index \leftarrow Index + 1$
  end if
end while
return T

applied to a target application. Bridging the gap between high-level business logic and application-specific events and assertions remains ***a challenge***. To address this challenge, we design two collaborative agents comprising the *Perception-Interaction Agent* and the *Decision-Generation Agent*, as illustrated in Figure 2.

Algorithm 1 shows the interaction of between the two agents in the second stage. First, the **Perception-Interaction Agent** (recalled Section 3.2.1) is responsible for dynamically capturing GUI states of the target application, providing real-time environmental context for the Decision-Generation Agent (Line 3). Second, the **Decision-Generation Agent** (recalled Section 3.2.2) synthesizes the multi-modal information of the target application with the core business logic from the first stage. This agent then generates an instruction sequence specifically adapted to the target application and its corresponding functionalities (Lines 4-5). Third, the **Perception-Interaction Agent** executing the instruction strategies derived from Decision-Generation Agent to the target application (Line 6). Through the alternating execution and closed-loop feedback of these two agents, this stage achieves an effective transformation from core business logic to executable test cases for specific functionalities.

3.2.1 Perception-Interaction Agent. There are two primary objectives of the Perception-Interaction Agent. First, this agent explores the target application by dynamically retrieving multi-modal information, which encompasses both visual data from screenshots and textual data from the GUI hierarchy. This multi-modal information is subsequently sent to the Decision-Generation Agent to facilitate the generation of instruction sequences. Second, upon receiving the instruction sequences produced by the Decision-Generation Agent, the Perception-Interaction Agent executes concrete operations on the target application. The Perception-Interaction Agent in a continuous loop operates continuously until a "Task Complete" signal is received, during which time it automatically records all interaction events and verification assertions to eventually synthesize them into a comprehensive functional test case.

The workflow of this agent consists of three critical components, which are GUI perception, action execution, and case synthesis.

GUI Perception. The Perception-Interaction Agent captures the visual and interactive widgets of the current state in real time by parsing the GUI hierarchy of the target application. This implementation involves two critical steps. First, this agent captures a screenshot of the current state to preserve complete visual information. Second, it transforms the state content into a structured natural language description. During this descriptive process, this agent extracts three core semantic attributes for each widget, specifically `text`, `content-desc`, and `resource-id`, while enumerating the operation types supported by each widget.

To maintain a logical structure for the preceding structured descriptions, LogiDroid organizes all widgets in the state according to a spatial order from top-left to bottom-right. This arrangement aligns with the natural browsing habits of users and forms a clear flow for state description. The "Input Object" portion in Figure 3 (a) provides a concrete example of the state description for the "S3" state in Figure 1.

Ultimately, the state screenshot and the natural language description of the GUI hierarchy serve as multi-modal inputs. These are transmitted together to the Decision-Generation Agent to provide comprehensive environmental context for verifying specific functionalities.

Action Execution. Based on the test instructions output by the Decision-Generation Agent (see Section 3.2.2), such as performing a click on a specific widget, this component is responsible for translating abstract test instructions into concrete actions. By invoking the underlying APIs of the mobile testing framework, this agent precisely executes predefined operations including clicks, text inputs, and swipes. Simultaneously, it monitors real-time state changes within the mobile application to facilitate assertion verification.

Case Synthesis. Upon the completion of the test sequence execution, this component performs a structured integration of the interaction events and corresponding assertions generated throughout the exploration process. By organizing these elements according to their execution order, this agent ultimately generates executable functional test cases including events and assertions. This process completes the transformation from dynamic interaction behaviors into standardized test cases for the target functionalities.

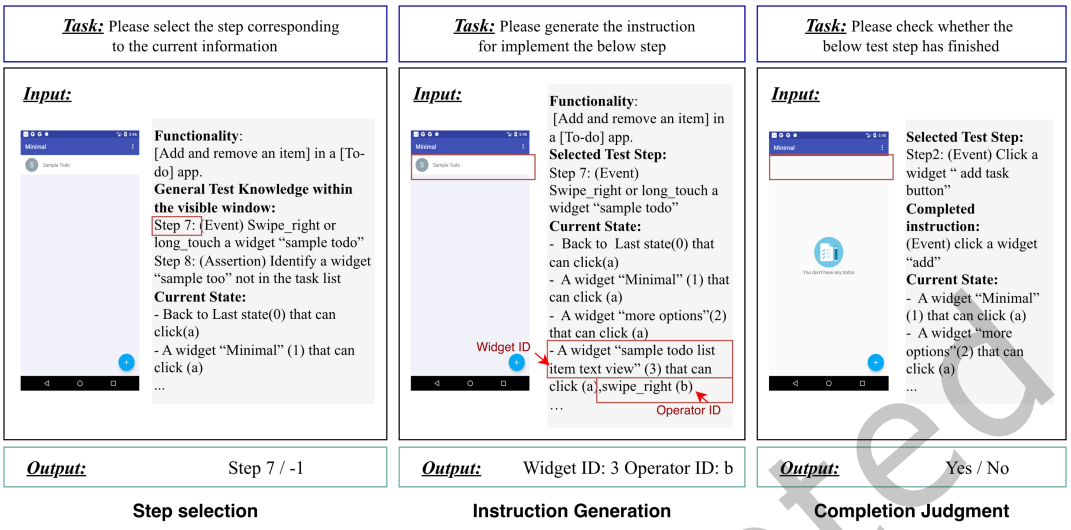

Fig. 3. Example of Decision-Generation Agent.

3.2.2 Decision-Generation Agent. The Decision-Generation Agent serves as the central coordination module of LogiDroid, bearing the critical responsibility of formulating testing strategies. This agent performs dynamic decision-making based on three inputs, which are the current state information, the requirement description of the target functionality, and the core business logic derived from the first stage. These decision strategies subsequently drive the Perception-Interaction Agent to execute specific actions.

In the decision-making process, this agent faces ***three challenges***. First, business logic often originates from a generalized synthesis of multiple applications. For instance, removing an item might require a "selection and click" in one app but a "swipe gesture" in another. Consequently, only a subset of these steps may apply to the target mobile application, necessitating precise step selection and adaptation. Second, after identifying the applicable steps, the agent must accurately map them to specific GUI widgets within the current state. This mapping is essential to generate executable instruction descriptions for subsequent events and assertions. Finally, due to the inherent limitations and hallucination issues of LLMs regarding task termination [35], models often struggle to perceive task completion independently, leading to redundant exploration. Therefore, an effective completion determination mechanism is required to avoid generating unnecessary operations.

To address the preceding challenges, the Decision-Generation Agent incorporates three collaborative components comprising step selection, instruction generation, and completion judgment, as illustrated in Figure 3. The detailed logic of this process is further elaborated in Algorithm 2.

Algorithm 2 illustrates the detailed execution flow of the **Decision-Generation Agent**. First, the **step selection** component initializes a sliding window mechanism to focus on a limited set of candidate steps and identifies the specific logic step that best aligns with the current GUI state (Line 6). Second, the **instruction generation** component synthesizes concrete operation instructions for the selected step (Line 17), employing a differentiated processing mechanism to handle event-based interactions or assertion-based verifications. Third, the **completion judgment** component evaluates whether the current step is completed (Line 19) to decide whether to proceed to the next step or retry, while incorporating a threshold mechanism to prevent infinite loops (Lines 20-27).

Algorithm 2 Decision-Generation Agent Execution Flow

Input: Requirement Description (R), Core Business Logic (L), Current State (S_{curr})
Output: Test Case (T)

1: Initialize sliding window W with size $Step_{num}$
2: $current\_step \leftarrow 0$; $is\_completed \leftarrow$ "Yes"
3: $executed\_cmds \leftarrow \emptyset$; $Candidate\_Steps \leftarrow$ GetStepsInWindow(W, L)
4: **while** Task Not Complete **do**
5: ▷ *Step Selection*
6: **if** $is\_completed =$ "Yes" **then**
7: $selected\_step \leftarrow$ StepSelection($R, S_{curr}, Candidate\_Steps$)
8: **if** $selected\_step = (-1)$ **then**
9: ▷ *No applicable step found*
10: Slide W forward
11: **continue**
12: **else**
13: $current\_step \leftarrow selected\_step$
14: $is\_completed \leftarrow$ "No"
15: **end if**
16: **end if**
17: ▷ *Instruction Generation*
18: GenerateInstruction($R, S_{curr}, current\_step$)
19: ▷ *Completion Judgment*
20: $is\_completed \leftarrow$ CompletionJudgment($R, current\_step, S_{curr}$)
21: **if** $is\_completed =$ "Yes" **then**
22: $current\_step \leftarrow current\_step + 1$
23: Reset attempts
24: **else**
25: Increment attempts
26: **if** attempts > threshold **then**
27: ▷ *Prevent infinite loops*
28: Skip step and Reset attempts
29: **end if**
30: **end if**
31: Update S_{curr} ▷ *Perceive new state via PIA*
32: **end while**

Step selection. In this component, as illustrated in Figure 3 (a), LogiDroid provides the LLM with three primary inputs: the requirement description of the target functionality, the core business logic generated by the Knowledge-Fusion Agent, and the GUI state information captured by the Perception-Interaction Agent.

To improve decision accuracy, LogiDroid employs a sliding window mechanism to manage the sequence of logic steps through three primary actions (line 7). First, the system uses a window of size $Step_{num}$ to restrict the focus of the LLM to a specific range of candidate logic steps. This design allows the model to precisely evaluate the applicability of each step relative to the current GUI state by analyzing a limited set of possibilities. Second, the model performs reasoning according to the order of steps within the window to select the logic step that best matches the current context of the mobile application. Third, if no applicable step exists within the window, the

model returns a specific identifier "(-1)", which triggers the sliding window to update and move to a new set of candidate steps (lines 8-9). This approach ensures that the agent effectively adapts business logic to the specific functionalities of the target application.

Instruction Generation. The instruction generation component, as illustrated in Figure 3 (b), processes three inputs: the target requirement description, the specific logic step identified by the step selection component, and the current state information provided by the Perception-Interaction Agent (see Section 3.2.1). Based on these inputs, the LLM generates concrete operation instructions, such as clicking the "Add" button in state S1 of Figure 1, and transmits them to the Perception-Interaction Agent for execution (line 17). To prevent redundant operations, LogiDroid maintains a record of completed test steps and their corresponding instructions. LogiDroid utilizes a differentiated processing mechanism for two distinct types of test steps.

Event-based steps. An event includes a widget, an action, and an optional input. For event-based steps, the LLM returns a specific combination of widget ID and operation ID, as shown in Figure 3 (b). This information is passed to the Perception-Interaction Agent and ultimately integrated into the final test case.

Assertion-based steps. An assertion includes a widget and a condition. GUI testing primarily involves two types of conditions [46, 81]. The first type of assertions verifies the existence of a widget on the current state, such as assertion A1 in Figure 1. For this type, LogiDroid inputs the current state and the logic step into the LLM to retrieve the corresponding widget ID. The second type of assertions verifies the disappearance of a widget that appeared in a previous state, such as assertion A2 in Figure 1. Since the target widget is absent from the current state, LogiDroid utilizes a state backtracking mechanism to identify the widget from historical states and retrieve its widget ID. Finally, once the widget is identified based on the preceding process, LogiDroid generates the appropriate assertion based on the widget ID and the corresponding conditions, and sends this assertion to the Perception-Interaction Agent for verification.

Through this processing mechanism, LogiDroid effectively manages diverse logic step requirements. This ensures both the accuracy and executability of events and assertions for complex functionalities within the mobile application.

Completion Judgment. To address the inherent limitations and hallucination issues of LLMs, LogiDroid implements an active query mechanism, as illustrated in Figure 3 (c). This mechanism decomposes the overall testing task into an ordered sequence of test steps and performs real-time state evaluation during the execution of each step.

Specifically, LogiDroid initiates the completion judgment process immediately after the LLM generates an operation instruction. During this phase, the model performs reasoning based on three features, which are the description of the current logic step, the sequence of already executed instructions, and the most recent state. The model must output a binary decision, where "Yes" indicates step completion and "No" indicates the step remains unfinished (line 19). If the step is judged as complete, LogiDroid proceeds to the selection phase for the next logic step (lines 20-21). If judged as incomplete, the agent continues to generate subsequent instructions targeting the current step (lines 24-26).

To prevent infinite loops during complex steps, LogiDroid enforces a maximum attempt limit. When the number of consecutive attempts for a single logic step reaches a predefined threshold, LogiDroid automatically skips that step and moves to the next one. This design ensures both thorough exploration of individual steps and the overall progress efficiency of the testing process. The active query mechanism offers two primary advantages. It maintains the integrity of the business logic through real-time state assessment and effectively handles complex scenarios requiring multiple operations, thereby ensuring high execution efficiency for the target mobile application and its functionalities.

Table 4. Statistics of Experimental Subjects

Dataset	Category	Apps	Cases	Events	Assertions	Reqs
FrUITeR	News	4	32	76	-	64
	Shopping	3	21	116	-	42
Lin	Browser	5	10	32	20	20
	To-Do	5	10	39	15	20
	Shopping	4	8	49	26	16
	Email	2	4	14	12	8
	Calculator	5	10	33	10	20
Total	-	28	95	359	83	190

4 Evaluation

To evaluate LogiDroid comprehensively, we conduct the research questions focusing on five key perspectives. First, we assess the effectiveness of LogiDroid in generating functional test cases for mobile applications in real-world industrial scenarios and compare LogiDroid with baseline approaches. Second, we investigate the specific contributions of different techniques within LogiDroid through an ablation study. Third, we evaluate the robustness of LogiDroid by examining whether its performance remains consistent across various underlying foundation models. Fourth, we analyze the efficiency of LogiDroid and the baselines. Fifth, we analyze the generalizability of LogiDroid on new applications. Details are as follows.

RQ1: How effective is LogiDroid compared with the baselines?
RQ2: How do LogiDroid's main techniques affect the GUI test generation?
RQ3: How robust is LogiDroid across different foundation models?
RQ4: How efficient is LogiDroid compared with the baselines?
RQ5: How does LogiDroid perform on new applications?

4.1 Experimental Setup

Experimental subjects. We utilize two widely-used functional testing benchmarks for evaluation, i.e., FrUITeR [6] dataset and Lin [9] dataset. These benchmarks cover various industrial-grade applications, including ABC News [1] and Firefox [5], providing a solid foundation for evaluating LogiDroid in real-world scenarios. Both datasets provide the target applications and developer-written functional test cases. Specifically, the test cases in the Lin dataset contain complete event sequences and assertions, while the FrUITeR dataset provides only event sequences.

We collect all installable applications and executable test cases from both datasets to form our experimental subjects. To enhance data diversity, we invite two volunteer engineers with 3–5 years of Android development experience to independently write requirement descriptions based on the actual functionality of the test cases. This dual-annotation mechanism aims to capture the diversity in how different individuals describe the same functionality. The final experimental dataset comprises 28 applications, 95 test cases (serving as *ground-truth test cases*), and 190 functional requirement descriptions (two independent descriptions per test case). Notably, compared with existing research [72, 76], our evaluation involves the largest number of mobile applications and categories. Table 4 presents the detailed statistics of the experimental subjects.

Baselines approaches. To comprehensively evaluate the performance of LogiDroid, we select two state-of-the-art approaches from both academia and industry for comparison, i.e., AutoDroid [72] and AppAgent [76].

AutoDroid [72] adopts a technical route combining static semantic understanding with dynamic exploration. This approach first comprehends the application semantics through static analysis and subsequently guides LLMs to generate test cases based on requirement descriptions.

AppAgent [76] is released by Tencent [11] Inc. (a fortune global 500 company). This approach learns to navigate and use new apps through two distinct modes: exploration and the observation of human demonstrations. By leveraging these learning mechanisms, it generates corresponding test cases for various functionalities.

Note that, existing functional test generation approaches generally suffer from a critical limitation involving the inability to generate effective assertions. As assertions represent core elements for verifying functional correctness, their absence severely restricts the practical value of these approaches in real-world industrial scenarios. In contrast, by incorporating retrieved domain knowledge to guide the LLM, LogiDroid is capable of generating functional test cases containing both complete event sequences and verification assertions for target applications.

Evaluation metrics. To evaluate LogiDroid and the baselines, we follow related studies [47, 78, 81] and utilize two evaluation metrics, i.e., perfect-rate and success-rate.

Perfect-rate: The proportion of generated test cases ($Test_t$) that are consistent with the ground-truth test cases ($Test_{gt}$). The evaluation process for perfect-rate is fully automated and requires no human intervention.

$$Perfect\text{-}rate = \frac{Test_t \cap Test_{gt}}{Test_t} \tag{1}$$

Success-rate: The proportion of generated test cases ($Test_t$) that successfully test the target functionality ($Test_{suc}$).

$$success\text{-}rate = Test_{suc} \,/\, Test_t \tag{2}$$

To quantify the generated test cases that successfully verify the target functionalities, we acknowledge that verification can be achieved through diverse sequences. For this reason, the developer-provided ground-truth represents only one feasible solution rather than an exhaustive standard. Since enumerating all potential valid cases is impossible, we systematically inspect those deviating from the ground-truth and consider them valid if they still fulfill the target requirement through semantically equivalent events and assertions. This manual verification involves the two volunteers previously responsible for requirement drafting. We provide them with an evaluation package containing the target application, the generated test cases, and the ground-truth test cases. To facilitate consistent judgment, we explicitly highlight the differences in events and assertions between each generated test case and its corresponding ground-truth test case. During evaluation, the volunteers execute both the ground-truth and generated test cases on the target application, and assess whether the generated test case still covers the essential functional steps and verification conditions required by the target functionality. For events and assertions, additional ones are considered acceptable if they do not hinder the testing objective, while replaced ones are regarded as valid if they produce the same functional effect as those in the ground-truth or correctly verify the expected functional outcome. Each volunteer first assesses validity independently, followed by group discussions to resolve any disagreements until reaching a final consensus. This multi-party mechanism maintains the reliability and objectivity of the evaluation results.

Implementation details. Regarding text vectorization, LogiDroid utilizes the bge-base-en-v1.5 [3] model to convert requirement descriptions and functional summaries into embedding vectors. This model is widely applied in semantic similarity calculation tasks [29, 29, 41]. Regarding parameter configuration, we optimize the three hyperparameters of LogiDroid through systematic preliminary experiments. For the sliding window size $Step_{num}$ and the number of retrieved similar test cases Top_{sim}, we conduct validation experiments on a candidate set {1, 2, 3} based on 20% of the test cases. The results indicated that LogiDroid achieves optimal performance when

Table 5. Overall effectiveness of LogiDroid and the baselines

Dataset	Approach	Perfect-rate	Success-rate
FrUITeR	LogiDroid	**20%**	**40%**
	AppAgent	9%	32%
	AutoDroid	15%	27%
Lin	LogiDroid	41%	57%
	LogiDroid*	**48%**	**65%**
	AppAgent	21%	42%
	AutoDroid	10%	27%

LogiDroid* denotes the variant evaluated without considering assertions.

$Step_{num} = 2$ and $Top_{sim} = 3$. For the LLM temperature, AppAgent provides an official setting, and we therefore follow its original configuration with a temperature of 0.0. In contrast, AutoDroid does not specify an official temperature setting. Thus, for LogiDroid and AutoDroid, we tune the temperature using the candidate set {0.0, 0.2, 0.4, 0.6, 0.8}. The results show that both approaches achieve the best overall performance when the temperature is set to 0.4. Therefore, we adopt this configuration for the final experiments.

4.2 RQ1: Effectiveness

We evaluate LogiDroid's effectiveness on the FrUITeR and Lin datasets, and compare it against two representative baselines: AutoDroid [72] and AppAgent [76]. The performance is evaluated based on two metrics, i.e., perfect-rate and success-rate. For fairness, all approaches utilize GPT-5 [61] as the underlying LLM.

Notably, we implement a rigorous evaluation protocol to avoid data leakage in the retrieval process of LogiDroid. When generating test cases for the specific functionalities of a target application, LogiDroid *removes all test cases associated with that application from the retrieval dataset*. Therefore, the system cannot access the ground-truth test case itself, nor any other test case from the same application. The retrieved test cases come from different applications with the target application, and thus involve different GUI implementations, interaction flows, and application contexts. In this sense, the retrieval source and evaluation target are separated at the application level, and the evaluation measures cross-application knowledge reuse rather than data leakage. By enforcing this data isolation, the evaluation provides an authentic reflection of LogiDroid's functional test generation capabilities.

Effectiveness results. Table 5 presents the evaluation results of LogiDroid compared with baseline approaches on the FrUITeR and Lin datasets. To ensure the reliability of the results, two volunteers (see Section 4.1) verify whether each generated test case successfully test the target functionality. We quantify inter-rate reliability using Fleiss' kappa coefficient [25]. The resulting value is 0.93, which satisfies the statistical standard of *"almost perfect agreement"*. This metric indicates that the evaluation results possess high consistency and credibility.

On the FrUITeR dataset, LogiDroid demonstrates a significant advantage. Specifically, 40% of the generated test cases successfully verified target functions, and 20% are perfectly identical to the ground-truth. Compared to baseline approaches, this result represents an improvement of over 25% in success-rate and over 33% in perfect-rate.

On the Lin dataset, LogiDroid exhibits consistent performance improvements across tasks. LogiDroid achieved a 57% success-rate, with 41% of test cases being perfectly identical to the ground-truth. Notably, the Lin dataset includes complete events and assertions, whereas baseline approaches only support event generation. Therefore,

Fig. 4. Illustrative Examples of Failure Cases

we additionally evaluated LogiDroid's performance without considering assertions (denoted as LogiDroid*). In this setting, LogiDroid* achieved a 65% success-rate and a 48% perfect-rate. This outperforms baseline methods by over 55% in success-rate and over 129% in perfect-rate.

Effectiveness analysis. We observe three findings from Table 5.

First, compared to AppAgent, LogiDroid achieves a significant improvement in the perfect-rate, with increases of 122% on the FrUITeR dataset and 129% on the Lin dataset. This advantage primarily stems from LogiDroid's knowledge retrieval and alignment mechanism. This mechanism extracts domain knowledge from historical test cases to generate high-quality business logic. In contrast, AppAgent lacks knowledge reuse capabilities. As a result, it generates test cases with numerous redundant operations, which compromise the overall quality of the test cases.

Second, compared to AutoDroid, LogiDroid achieves a significant improvement in the success-rate, with increases of 48% on the FrUITeR dataset and 141% on the Lin dataset. This performance boost is mainly attributed to LogiDroid's context-aware test generation mechanism. On one hand, it leverages multimodal information (i.e., textual and visual data) to deeply understand GUI semantics. On the other hand, it decomposes complex tasks

through a progressive decision-making process involving step selection, instruction generation, and completion judgment. In comparison, AutoDroid relies solely on textual information and employs a bulk generation strategy, making it difficult to ensure the completeness of test cases.

Third, unlike existing approaches, LogiDroid can generate assertions. Assertions are crucial for functional testing. The domain knowledge refined during our Knowledge Retrieval and Fusion stage contains high-quality, reusable business logic. This enables LogiDroid to generate critical assertions required for effective functional verification. Conversely, AppAgent and AutoDroid only generate basic operation sequences, which limits their practical application value.

Failure Analysis. To understand the weaknesses of LogiDroid, we manually analyze all failed test cases and identify three main reasons. We select three examples (see Figure 4) to illustrate these failure reasons and also discuss possible ways to address them. In this figure, the yellow events represent the correct events and the red events represent the wrong events.

First, the extracted business logic may fail to cover all necessary test steps for the target functionality, which leads to incomplete generated test cases. For example, state (a) in Figure 4 corresponds to the functionality of calculating the total amount for a bill of 56.6 with a 15% tip. The intended test procedure should first complete the required input setup and then trigger the calculation. However, the extracted business logic only includes the calculation step (i.e., E1) while missing the input-setup step. As a result, the generated test case cannot fully test the target functionality. A potential way to address this problem is to check whether the extracted business logic covers all key steps of the target functionality after knowledge fusion.

Second, the instruction generation component may produce incorrect events for the given test steps, resulting in a generated test case with incorrect interactions. For example, states (b) and (c) of Figure 4 correspond to the functionality of opening the contact page from application settings. The intended test procedure should open the profile entry, enter "Settings", and then open the contact entry. At the state (b), the instruction generation component produces the event of clicking the filter icon (i.e., E2), whereas the correct interaction should first be clicking the profile icon (i.e, E3) and then clicking the settings icon (i.e., E4). This happens because multiple candidate widgets in the current GUI state are semantically related, making the current generation process prone to confusion. Incorporating richer state information and widget attributes into the instruction generation component may help improve its effectiveness.

Third, the step selection component may choose an incorrect step in a hierarchical state, causing the generated test case to deviate from the intended functionality. We illustrate this problem using states (d) and (e) of Figure 4, which correspond to the functionality of opening the Terms page in an application. The intended test procedure should open "Settings", select the "About" entry, and then open "Terms". However, the step selection component does not select the "About entry" (i.e., E6) rather than randomly select E5. As a result, the generated test case fails to reach the target page and open "Terms" (i.e., E7). Incorporating hierarchical state information into the step selection process, and enabling it to better follow the intended procedure, may help reduce this problem.

> **Answer to RQ1:** LogiDroid is effective and significantly outperforms the state-of-the-art baselines in functional testing scenarios. Furthermore, the generated test cases by LogiDroid includes assertions that the state-of-the-art baselines fail to.

4.3 RQ2: Main Techniques

To evaluate the impact of LogiDroid's main techniques, we randomly select 50% of the applications from each category in the Lin dataset as our research subjects. Note that, we choose the Lin dataset as it offers broader category and complete event sequences with assertions, making it more representative for analyzing the contribution of individual approaches.

Table 6. Effectiveness of LogiDroid's Key Techniques

Approach	Perfect-rate	Success-rate
LogiDroid	30%	70%
LogiDroid (w/o Semantic-Retrieval Agent)	10%	40%
LogiDroid (w/o Knowledge-Fusion Agent)	5%	20%
LogiDroid (w/o Decision-Generation Agent)	10%	30%
AppAgent	20%	35%
AutoDroid	0%	10%

Experimental setup. To assess the contribution of core techniques, we design a series of ablation experiments in a controlled setting. These experiments analyze the influence of different agents on overall system performance.

For the Knowledge Retrieval and Fusion stage, we conduct two sets of ablation experiments. First, we construct the "LogiDroid (w/o Semantic-Retrieval Agent)" variant. After receiving requirement descriptions, this variant skips the cross-application retrieval process. It generates business logic using only the Knowledge-Fusion Agent based on the LLM's internal knowledge. Second, we develop the "LogiDroid (w/o Knowledge-Fusion Agent)" variant. This variant passes the requirement descriptions and retrieval results directly to the subsequent stages without any fusion processing.

For the Context-Aware Test Generation stage, the Perceptual-Interaction Agent represents a fundamental component. Removing it prevents the system from obtaining GUI information and generating test cases. Thus, we only conduct an ablation experiment by removing the Decision-Generation Agent, creating the "LogiDroid (w/o Decision-Generation Agent)" variant. In this setting, the system skips the specialized decision-making strategy. Instead, it makes decisions via the LLM directly.

Results analysis. Table 6 presents the results of the ablation study. The first row establishes the baseline performance of the full LogiDroid on a 50% sample of the Lin dataset. The subsequent three rows show the performance after removing each of the three agents. For reference, we also report the results of AppAgent and AutoDroid under the same experimental setting in this table.

The results demonstrate that the Semantic-Retrieval, Knowledge-Fusion, and Decision-Generation agents are all critical to the effectiveness of LogiDroid. Removing the Semantic-Retrieval Agent causes the success-rate to drop from 70% to 40%, which validates the vital role of case retrieval in domain knowledge extraction. Similarly, removing the Knowledge-Fusion Agent and the Decision-Generation Agent leads to success-rate declines to 20% and 30%, respectively. This proves that both agents are indispensable for distilling business logic and planning decisions.

Further analysis reveals three critical observations. First, the lack of a semantic-retrieval mechanism prevents LogiDroid from accessing external domain knowledge. This limitation forces the model to rely solely on the internal parameterized knowledge, which compromises test quality. Second, the absence of the knowledge-fusion mechanism has an even more profound impact because LogiDroid cannot extract generalized business logic from related cases. Without this fusion, the generated test cases lack the flexibility to adapt to the diverse implementations of various functionalities. Third, removing the decision-generating mechanism deprives the system of its ability to perform incremental task decomposition, which reduces the precision of the testing strategy. These observations validate the architectural design of LogiDroid where each agent works in synergy to maintain the effective generation of functional test cases.

Table 7. Robustness Evaluation Results of LogiDroid and Baseline Approaches

Approach	LLM	Perfect-rate	Success-rate
LogiDroid	GPT-5	30%	70%
	Gemini-2.5	20%	60%
LogiDroid*	GPT-5	50%	75%
	Gemini-2.5	25%	75%
AppAgent	GPT-5	20%	35%
	Gemini-2.5	15%	25%
AutoDroid	GPT-5	0%	10%
	Gemini-2.5	10%	10%

LogiDroid* denotes the variant of LogiDroid evaluated without considering assertions.

Answer to RQ2: LogiDroid's techniques are essential and contribute substantially to its superior performance in functional test generation.

4.4 RQ3: Robustness Analysis

To evaluate the performance consistency of LogiDroid across different foundation models, we replace the underlying LLM from *GPT-5* [61] with *Gemini-2.5* [22] while keeping all other experimental settings constant. This experiment uses the same data source as RQ2, which includes 50% of the applications in the Lin dataset. We compare LogiDroid against AppAgent [76] and AutoDroid [72]. This setup ensures consistent experimental conditions for cross-model comparison and validates the architecture of LogiDroid to different foundation models.

Results statistics. Table 7 illustrates the performance of various approaches on a 50% sample of the Lin dataset using Gemini-2.5 as the foundation model. The Lin dataset includes complete event sequences and assertions, which the baseline approaches AppAgent and AutoDroid cannot generate. To maintain a fair evaluation, we record the performance of LogiDroid without considering assertions, denoted as LogiDroid*. Under the full evaluation with assertions, LogiDroid achieved a 60% success-rate and a 20% perfect-rate. When assertions are excluded, LogiDroid* reached a 75% success-rate and a 25% perfect-rate. Compared to the baselines, LogiDroid improves the success-rate by over 67%, and the perfect-rate by over 200%.

Results analysis. The experimental results indicate that LogiDroid maintains excellent performance across different foundation models, demonstrating high model-agnosticism and robustness. Replacing the underlying LLM from GPT-5 with Gemini-2.5 does not compromise LogiDroid's performance advantage over baseline approaches. This result illustrates that the core architecture of LogiDroid adapts well to various foundation models. This trait is particularly crucial for practical applications. Users can adapt foundation models to their resource constraints and requirements without compromising performance stability.

Answer to RQ3: LogiDroid achieves stable performance across diverse foundation models, demonstrating the robust generalization of its architecture.

Table 8. Efficiency Statistics of LogiDroid and Baseline Approaches

Approach	Runtime [minutes]	Token
LogiDroid	6	31,796
AppAgent	8	34,936
AutoDroid	5	18,374

4.5 RQ4: Efficiency

To evaluate the efficiency of LogiDroid and baseline approaches, we performed experiments on the Lin dataset. We measure the average runtime and token usage for generating a single test case using LogiDroid, AppAgent, and AutoDroid.

Efficiency result. Table 8 presents the comparison of resource consumption during the test case generation process. The results show that LogiDroid takes an average of 6 minutes to generate a single test case, with an average token consumption of 31,796. Compared with AppAgent, LogiDroid reduces runtime by 25% and token consumption by 9%. Compared with AutoDroid, LogiDroid increases runtime by 20% and token consumption by 73%.

Efficiency analysis. Regarding resource efficiency, the token consumption of LogiDroid is lower than that of AppAgent but higher than that of AutoDroid. In terms of time efficiency, LogiDroid similarly ranks between the two baseline approaches.

Deep analysis reveals that the relatively high token consumption of LogiDroid primarily stems from the progressive decision-making mechanism of Decision-Generation Agent. This mechanism maintains the precision of test steps through multiple rounds of reasoning. Although there is room for optimization in runtime efficiency, the current efficiency level remains acceptable considering the significant improvements in testing accuracy (as shown in Section 4.2). Notably, through the effective knowledge reuse mechanism, LogiDroid sustains high-quality test generation while achieving superior token efficiency compared to AppAgent.

> **Answer to RQ4:** LogiDroid's efficiency is comparable to the baselines.

4.6 RQ5: Generalizability

To further evaluate the generalizability of LogiDroid, we conduct an additional study on a new dataset (i.e., the TEM dataset [10]). The LLM used in this evaluation is GPT-5.

Experimental setup. The TEM dataset [10] consists of five newly collected popular applications from the Google Play Store [7] and ten corresponding functional test cases, covering five representative categories, i.e., browser, to-do, shopping, mail, and calculator, as summarized in Table 9. Since all applications in this dataset are newly introduced, the dataset provides an opportunity to further evaluate whether LogiDroid remains effective on previously unseen applications. Note that the TEM dataset only provides the application APKs and the target functionalities to be tested, but does not include ground-truth test cases. Therefore, we invite the same two volunteers (see Section 4.1) to construct the ground-truth test cases based on the given target functionalities. To assess the effectiveness of LogiDroid on this dataset, we follow the same evaluation process as described in Section 4.1 to evaluate LogiDroid, AppAgent, and AutoDroid.

Results. Table 10 presents the results of LogiDroid and the baselines on the TEM dataset. LogiDroid achieves a perfect-rate of 40% and a success-rate of 70%, outperforming AppAgent (20% and 50%, respectively) and AutoDroid (10% and 30%, respectively). These results show that the effectiveness of LogiDroid is not limited to the

Table 9. The TEM dataset used in RQ5

App	Version	Size	Download	Case	Event	Assertion
Web Browser	20.11.22	5M	5K+	2	6	4
Done	1.0	1.6M	50K+	2	5	3
FiveMiles	8.4.0	25.4M	10M+	2	10	2
Pro Mail	14.64.0	98.2M	1M+	2	7	3
Tip Calculator	2.6.2	4.3M	5K+	2	6	2

Table 10. Results on the TEM dataset

Approach	Perfect-rate	Success-rate
LogiDroid	40%	70%
AppAgent	20%	50%
AutoDroid	10%	30%

original evaluation benchmarks. Even on new applications, LogiDroid still achieves strong performance, further demonstrating its ability to reuse and adapt business logic across different applications.

Answer to RQ5: The results indicate that LogiDroid remains effective on previously unseen applications and achieves consistently better performance than the baselines on the new benchmark.

4.7 Threat to validity

We discuss potential threats to validity of our study across three primary dimensions.

A possible threat to external validity is the generalizability of our findings to other datasets. To mitigate this threat, we utilize a substantial number of applications and categories, exceeding the scale of most related work. Furthermore, we adopt all popular datasets in the field of functional testing as evaluation benchmarks. These datasets include various complex industrial mobile application examples that represent real-world scenarios. The diversity of these applications helps verify that LogiDroid effectively handles a wide range of functionalities.

A possible threat to internal validity involves potential errors in our implementation and experiments. To mitigate this threat, we manually analyze the test cases that fail to test the target functionalities. We also publish our implementation and experimental data to facilitate external validation. Regarding human evaluation, we invite two experienced developers as volunteers and provide them with a clear evaluation process. The resulting Fleiss' kappa coefficient of 0.93 confirms high consistency among evaluators, which maintains the objectivity of the assessment.

A possible threat to construct validity is evaluation metrics. To mitigate this threat, we follow related studies by utilizing two primary metrics to validate the effectiveness of the generated test cases. These metrics offer a reliable and objective basis for assessing the quality of the test cases generated by both the baseline approaches and LogiDroid.

Table 11. Method coverage of LogiDroid and AutoDroid

Category	App	Coverage number			Coverage capability	
		GT.	Logi.	Auto.	Logi.	Auto.
Browser	Privacy Browser	1566	1693	1559	99.7%	99.4%
	FOSS Browser	739	749	677	99.6%	91.6%
	Firefox	3068	3078	3290	100.0%	99.3%
To-Do	Minimal	2083	2075	1478	99.5%	63.5%
	Clear List	1770	1497	1520	84.6%	80.0%
	To-Do	1874	1787	1649	95.4%	87.2%
	Simply Do	91	80	80	87.9%	87.9%
	Shopping List	1756	2196	1546	91.2%	78.3%
Shopping	Geek	5757	5624	5153	97.3%	88.7%
	Yelp	15212	15392	15220	99.2%	98.9%
	Etsy	9109	8568	7776	91.0%	85.4%
	Wish	6962	6938	6602	99.4%	93.7%
Mail	K-9 Mail	3065	3052	1153	99.5%	37.6%
	Fast Email	2150	2150	1419	99.9%	65.7%
Calculator	Tip Calculator	123	118	118	95.9%	95.9%
	Simple Tip	1124	1107	1103	98.5%	97.9%
	Tip Plus	1529	1537	1515	100.0%	99.1%
	Free Tip	951	942	934	99.1%	98.2%
Overall Average		**3274**	**3255**	**2933**	**96.5%**	**86.0%**

5 Discussion

Test case collection. The key innovation of LogiDroid lies in extracting and fusing domain-specific knowledge into application functional testing. This process requires the systematic collection of functional test cases in real-world environments. The collection is feasible in practice, primarily based on the following two conditions.

First, a vast number of similar applications and their corresponding GUI test case resources exist in real-world environments [31, 55, 81]. Mainstream platforms like Google Play [7] and F-Droid [4] typically adopt standardized classification systems to group applications into specific categories such as Shopping and News. Meanwhile, open-source platforms like GitHub contain numerous mobile application projects with complete test cases, providing a rich source for the systematic collection of diverse functionalities.

Second, we develop a specialized automated collection tool that periodically crawls test cases from various channels and automatically generates functional summaries based on the method described in Section 3.1.1. The entire process operates without human intervention, which provides a reliable technical foundation for expansion and maintenance of large-scale test repositories. This automated workflow maintains the scalability of our constructed dataset while verifying that new entries align with existing knowledge structures.

Method coverage of LogiDroid. Method coverage is a commonly-used code coverage metric [32, 34, 71]. It shows whether the generated test cases can execute the methods related to the target functionality. In Section 4.2, we evaluate LogiDroid using success-rate and perfect-rate. To provide a more comprehensive understanding of the quality of the test cases generated by LogiDroid, we further evaluate their method coverage and compare it with that of the ground-truth test cases. For reference, we also evaluate the method coverage achieved by AutoDroid. Specifically, we design a new metric, *Coverage-capability*. The coverage-capability is calculated as the

ratio of common covered methods ($Cover_{common}$) between the generated test case and the ground-truth test case, to the total covered methods ($Cover_{gt}$) by the ground-truth test case:

$$Coverage\text{-}capability = \frac{Cover_{common}}{Cover_{gt}} \tag{3}$$

We use WALLMauer [16] to instrument the target apps and calculate the method coverage for each app. Note that, coverage-capability and the metrics in Section 4.2 are evaluated from different perspectives. In this way, the result trends of LogiDroid and AutoDroid presented here and in Section 4.2 may be related but different.

Table 11 shows the method coverage results for LogiDroid (denoted as Logi.) and AutoDroid (denoted as Auto.) for 18 applications that can be instrumented by WALLMauer [16] on the Lin dataset. For each approach, we report the number of covered methods and the coverage-capability. We also report the number of methods covered by the ground-truth test cases (denoted as GT.) for reference. On average, LogiDroid reaches 96.5% of the method coverage achieved by the ground-truth test cases, whereas AutoDroid reaches only 86.0%. In terms of the number of covered methods, LogiDroid achieves an average of 3255, which is closer to the ground-truth value of 3274 than AutoDroid's 2933. These results indicate that LogiDroid achieves the closest match to the ground-truth in terms of method coverage, demonstrating satisfied capability in generating high-quality test cases that effectively exercise the implementation logic of the target functionality.

Hallucination mitigation for semantic inconsistencies. While the current hallucination mitigation mechanism effectively maintains structural correctness and rule compliance, an important complementary aspect is the consideration of semantic inconsistencies. In practice, even when generated outputs satisfy predefined formatting constraints, they may still exhibit deviations from the intended functionality, such as omitting critical interaction steps, and generating assertions that do not faithfully reflect the expected outcomes. These inconsistencies arise from the challenge of maintaining stable alignment between abstract business logic and concrete application contexts, especially in dynamic and multimodal environments.

To further enhance reliability, three strategies may address semantic inconsistencies. First, a semantics-aware validation mechanism could be introduced by representing the extracted business logic as structured semantic constraints and then checking whether each generated test action remains consistent with these constraints. Second, an assertion-level checking mechanism could be incorporated by deriving expected functional outcomes from the extracted business logic and comparing them against the generated assertions, so as to assess whether generated assertions faithfully reflect the expected functional outcomes. Third, a cross-step consistency analysis mechanism could be developed to capture subtle semantic deviations through dependency analysis and temporal consistency constraints in long-horizon test generation. Exploring these strategies would complement the existing mitigation mechanism and further strengthen the correctness and robustness of generated test cases.

Scalability in large-scale knowledge repositories. Although the current evaluation is conducted on a dataset containing 294 functional test cases, this scale is sufficient for assessing the effectiveness of LogiDroid. Looking toward real-world deployment with substantially larger repositories, the scalability of LogiDroid remains feasible for two main reasons. First, functional test repositories in real-world settings are typically not flat collections, but are naturally organized by application category, functionality type, and business scenario, which provides an initial structure for narrowing the retrieval space. Second, LogiDroid only requires a small set of highly relevant candidates for subsequent knowledge fusion, rather than processing the entire repository during each generation process. Therefore, the framework does not rely on exhaustively traversing over the full repository for each target functionality, which helps keep the retrieval-and-generation pipeline scalable.

Effect of semantic summaries on retrieval accuracy. LogiDroid relies on condensed semantic summaries to support knowledge retrieval, which raises the concern that such summaries may omit important functional details and thereby reduce retrieval accuracy in large repositories. This risk is mitigated in LogiDroid for two

Table 12. LogiDroid Results based on the number of similar test cases

Similar Test	Test Num	Avg. Perf.	Avg. Suc.
0	3	0%	17%
1	4	50%	75%
2	3	33%	33%
3	16	22%	44%
≥ 4	69	30%	49%

main reasons. First, the semantic summaries are not produced through naive compression. Instead, they are generated through a structured summarization process over functional test cases, where the model is guided to focus on the core functional intent while taking as input semi-structured events, assertions, application category, and key widget attributes. This design helps preserve the functionality-level semantics needed for retrieval, rather than retaining only superficial textual patterns. Second, LogiDroid does not rely on a single summary for direct knowledge reuse. Instead, it combines category-based filtering with vector-based semantic matching to retrieve a small set of highly relevant candidates, and then performs knowledge fusion over multiple retrieved test cases to extract reusable business logic. As a result, even if an individual summary misses certain details, the subsequent multi-case retrieval and fusion process still provides a robust basis for accurate knowledge identification in large-scale repositories.

Influence of the number of similar test cases on LogiDroid. We conduct a statistical analysis to evaluate how the number of similar historical test cases retrieved for a target functionality affects the effectiveness of LogiDroid. Table 12 shows the average perfect-rate (denoted as Avg. Perf.) and success-rate (denoted as Avg. Suc.) of LogiDroid with different numbers of similar test cases in the combined Lin and FrUITeR datasets. The results show that, as the number of similar test cases increases, the effectiveness of LogiDroid generally improves. For example, the success-rate increases from 17% when no similar test case is available to 49% when at least four similar test cases are available. This result supports the intuition of LogiDroid that extracting business logic from multiple similar test cases is beneficial for functional test generation.

Note that, when the number of similar test cases is one, the success-rate is relatively high (i.e., 75%). However, this result is based on only four cases, which makes the statistics less stable. In addition, these cases are mainly from email applications, where the target applications and the retrieved source applications (i.e., the applications producing similar test cases) are highly alike. When the number of similar test cases is zero, LogiDroid still achieves a success-rate of 17%. This result indicates that LogiDroid still retains some capability to generate useful test cases from requirement descriptions and GUI reasoning even without similar historical cases.

6 Related Work

Functional test generation. Existing research proposes various functional test generation approaches for different operating systems. Regarding the Android system, Li et al. [43] proposed a test generation approach based on a matching model. This approach relies on manually written test logics and manually filtered application screenshots, selecting operation events appearing in the interface through model matching. DroidBot-GPT [73] first introduces LLMs to select operation events based on manually written test logics. AutoDroid [72], as an enhanced version of DroidBot-GPT, introduces an offline state relationship understanding mechanism to improve testing effectiveness. LLMDroid [69] leverages LLM guidance to strategically direct testing towards unexplored functionalities, thereby enhancing automated exploration coverage. AppAgent [76] and QTypist [49] learn the

functional operation logic of applications and generates corresponding test cases through two modes, which are autonomous exploration and observing human demonstrations.Similarly, ReuseDroid [40], LLMigrate [19] and Guardian [59] leverage LLMs to extract test intentions or enforce runtime constraints, generating functional test cases for target applications through a dynamic reasoning mechanism. Regarding the iOS system, AXNav [68] and ILvuUI [36] both use test logics and application screenshots as inputs, utilizing vision-based LLMs for event selection. Regarding the windows system, AssistGUI [27] is a GUI test generation framework specifically designed to adapt to that system.

There are two key differences between existing approaches with LogiDroid. First, for knowledge utilization, existing approaches mainly rely on the automatic exploration of the target application or the knowledge from the general LLMs, which affects the accuracy of testing. On the contrary, we propose a retrieval-augmented architecture to synthesize testing expertise from cross-application historical data in a principled manner. This strategy transforms the test generation process from undirected exploration into a knowledge-driven reasoning task, enabling LogiDroid to navigate complex functionalities of the mobile application with higher precision. Second, assertions are crucial for functional testing, yet existing approaches can only generate events and are unable to generate assertions, which limits their practical application value. However, LogiDroid supports the concurrent generation of both event sequences and semantic assertions.

Bug detection for mobile applications. Based on different exploration strategies, bug detection approaches for mobile applications can be classified into four categories, which are random testing [12, 53, 65, 66], model-based approaches [17, 30, 39, 44, 51, 63, 64, 67, 70, 74], systematic testing approaches [14, 28, 54], and learning-based approaches [18, 20, 38, 45, 52, 58, 62, 75]. Representative research include Monkey [12] that adopts random exploration, AIMDROID [30] and Stoat [63] that combine static and dynamic analysis, SCENTEST [74] that utilizes event knowledge graphs to guide the exploration process, and SynthesiSE [28] that can dynamically infer Android model expressions.

The fundamental distinction between previous research and LogiDroid lies in different purposes. Existing approaches primarily focus on general GUI exploration and bug detection for the mobile application, yet they struggle to generate test cases that correspond to the specific requirements of individual functionalities. In contrast, LogiDroid shifts from undirected exploration to a knowledge-driven paradigm that generates precise functional test cases for diverse requirements.

LLM-based code generation. General Large Language Models like ChatGPT [13, 61] demonstrate significant potential in software engineering tasks, particularly in code generation, as evidenced by empirical study [33]. This success promotes the development of specialized models such as AlphaCode [42], CodeGen [57], and Incoder [26]. Researchers obtain these models by fine-tuning general LLMs with code corpora or through specialized training. Furthermore, a series of research studies have conducted in-depth exploration in the decoding stage. Zhang et al. [77] propose a planning-guided decoding algorithm based on Monte Carlo Tree Search, which improves code quality by exploring diverse program generation paths. Shi et al. [60] optimize output results by generating multiple program samples combined with test case re-ranking. These approaches fully exploit the potential of LLMs in program generation and debugging.

The key distinction between LogiDroid and existing LLMs for code generation lies in their object characters. Existing LLMs in code generation primarily focus on converting general natural language descriptions into general program code. They lack the capability to manage the complex interfaces and dynamic GUI states of a mobile application. In contrast, LogiDroid dynamically explores the target mobile application to capture real-time widget information and GUI state transitions. By learning domain-specific testing logic from these interactions, LogiDroid guides the generation process to emulate real-world user behavior for diverse functionalities.

7 Conclusion

In this paper, we propose LogiDroid, a two-stage approach that addresses the challenges of automated functional test case generation for mobile applications. By constructing a two-stage architecture, LogiDroid effectively resolves the core issues of domain knowledge acquisition and functional semantic understanding. Specifically, the approach utilizes Semantic-Retrieval and Knowledge-Fusion agents to achieve systematic accumulation and reuse of test knowledge, while it leverages Perception-Interaction and Decision-Generation agents to enable adaptive and progressive test generation in dynamic environments. Our evaluation of LogiDroid on 28 mobile applications and 190 functional requirements demonstrates that the approach significantly outperforms the state-of-the-art approaches, achieving substantial improvements in success-rate and perfect-rate. Notably, LogiDroid also exhibits advantages in generating test cases with complete verification assertions for diverse functionalities, providing a practical solution for large-scale industrial deployment.

In the future, we plan to focus on two aspects. First, we aim to optimize the decision-making mechanism under dynamic states, investigating the integration of reinforcement learning techniques to establish smarter exploration strategies. Second, we plan to construct a cross-platform test generation paradigm by establishing a unified abstraction approach, enabling the intelligent migration and adaptation of test cases across different mobile platforms.

Acknowledgments

We thank the anonymous TOSEM reviewers for their valuable feedback. This work was supported in part by the Shenzhen Science and Technology Program under Grant No. SYSPG20241211173609009, the National Natural Science Foundation of China under Grant No. 62272130, and the National Natural Science Foundation of China under Grant No. 62232003.

References

[1] 2025. *ABC News - Breaking News, Latest News and Videos.* https://abcnews.go.com/.
[2] 2025. *The artifact of LogiDroid.* https://github.com/ISSE-Lab/LogiDroid
[3] 2025. *Bge-base-en-v1.5.* https://huggingface.co/BAAI/bge-base-en-v1.5.
[4] 2025. *F-Droid: free and open source Android app repository.* https://f-droid.org/.
[5] 2025. *Firefox browser: fast, private and secure Web browser.* https://play.google.com/store/apps/details?id=org.mozilla.firefox.
[6] 2025. *FrUITeR dataset.* https://felicitia.github.io/FrUITeR.
[7] 2025. *Google Play store.* https://play.google.com/store/games
[8] 2025. *How many apps are currently available.* https://42matters.com/google-play-statistics-and-trends?.
[9] 2025. *Lin dataset.* https://github.com/seal-hub/CraftDroid.
[10] 2025. *TEM dataset: Popular Apps in Google Play Store.* https://github.com/YakZhang/TEMdroid/tree/main/Dataset/Usefulness_study.
[11] 2025. *Tencent Inc.* https://www.tencent.com/
[12] 2025. *UI/application exerciser Monkey.* https://developer.Android.com/studio/test/monkey.
[13] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774* (2023).
[14] Saswat Anand, Mayur Naik, Mary Jean Harrold, and Hongseok Yang. 2012. Automated concolic testing of smartphone apps. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering* (Cary, North Carolina) *(FSE '12)*. Association for Computing Machinery, New York, NY, USA, 1–11. https://doi.org/10.1145/2393596.2393666
[15] Chetan Arora, Tomas Herda, and Verena Homm. 2024. Generating Test Scenarios from NL Requirements Using Retrieval-Augmented LLMs: An Industrial Study. In *2024 IEEE 32nd International Requirements Engineering Conference (RE)*. 240–251. https://doi.org/10.1109/RE59067.2024.00031
[16] Michael Auer, Iván Arcuschin Moreno, and Gordon Fraser. 2024. WallMauer: Robust Code Coverage Instrumentation for Android Apps. In *Proceedings of the 5th ACM/IEEE International Conference on Automation of Software Test (AST 2024)* (Lisbon, Portugal) *(AST '24)*. Association for Computing Machinery, New York, NY, USA, 34–44. https://doi.org/10.1145/3644032.3644462
[17] Young-Min Baek and Doo-Hwan Bae. 2016. Automated model-based Android GUI testing using multi-level GUI comparison criteria. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering* (Singapore, Singapore) *(ASE '16)*.

Association for Computing Machinery, New York, NY, USA, 238–249. https://doi.org/10.1145/2970276.2970313

[18] Carlos Bernal-Cárdenas, Nathan Cooper, Kevin Moran, Oscar Chaparro, Andrian Marcus, and Denys Poshyvanyk. 2020. Translating video recordings of mobile app usages into replayable scenarios. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 309–321. https://doi.org/10.1145/3377811.3380328

[19] Benyamin Beyzaei, Saghar Talebipour, Ghazal Rafiei, Nenad Medvidović, and Sam Malek. 2025. Automated Test Transfer across Android Apps using Large Language Models. *Proceedings of International Symposium on Software Testing and Analysis* 2, ISSTA (June 2025), 2227–2250. https://doi.org/10.1145/3728975

[20] Nataniel P. Borges, Maria Gómez, and Andreas Zeller. 2018. Guiding app testing with mined interaction models. In *Proceedings of the 5th International Conference on Mobile Software Engineering and Systems* (Gothenburg, Sweden) *(MOBILESoft '18)*. Association for Computing Machinery, New York, NY, USA, 133–143. https://doi.org/10.1145/3197231.3197243

[21] Xiangping Chen, Xing Hu, Yuan Huang, He Jiang, Weixing Ji, Yanjie Jiang, Yanyan Jiang, Bo Liu, Hui Liu, Xiaochen Li, et al. 2025. Deep learning-based software engineering: progress, challenges, and opportunities. *Science China Information Sciences* 68 (2025), 11102. https://doi.org/10.1007/s11432-023-4127-5

[22] Gheorghe Comanici, Eric Bieber, Mike Schaekermann, Ice Pasupat, Noveen Sachdeva, Inderjit Dhillon, Marcel Blistein, Ori Ram, Dan Zhang, Evan Rosen, et al. 2025. Gemini 2.5: Pushing the frontier with advanced reasoning, multimodality, long context, and next generation agentic capabilities. *arXiv preprint arXiv:2507.06261* (2025).

[23] Qingxiu Dong, Lei Li, Damai Dai, Ce Zheng, Zhiyong Wu, Baobao Chang, Xu Sun, Jingjing Xu, and Zhifang Sui. 2022. A survey on in-context learning. *arXiv preprint arXiv:2301.00234* (2022).

[24] 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, 1–13. https://doi.org/10.1145/3597503.3608137

[25] Joseph L Fleiss. 1971. Measuring nominal scale agreement among many raters. *Psychological bulletin* 76, 5 (1971), 378.

[26] Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen-tau Yih, Luke Zettlemoyer, and Mike Lewis. 2022. Incoder: A generative model for code infilling and synthesis. *arXiv preprint arXiv:2204.05999* (2022).

[27] Difei Gao, Lei Ji, Zechen Bai, Mingyu Ouyang, Peiran Li, Dongxing Mao, Qinchen Wu, Weichen Zhang, Peiyi Wang, Xiangwu Guo, et al. 2023. ASSISTGUI: Task-Oriented Desktop Graphical User Interface Automation. *arXiv preprint arXiv:2312.13108* (2023).

[28] Xiang Gao, Shin Hwei Tan, Zhen Dong, and Abhik Roychoudhury. 2018. Android testing via synthetic symbolic execution. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering* (Montpellier, France) *(ASE '18)*. Association for Computing Machinery, New York, NY, USA, 419–429. https://doi.org/10.1145/3238147.3238225

[29] Yunfan Gao, Yun Xiong, Xinyu Gao, Kangxiang Jia, Jinliu Pan, Yuxi Bi, Yixin Dai, Jiawei Sun, Haofen Wang, and Haofen Wang. 2023. Retrieval-augmented generation for large language models: A survey. *arXiv preprint arXiv:2312.10997* 2, 1 (2023).

[30] Tianxiao Gu, Chun Cao, Tianchi Liu, Chengnian Sun, Jing Deng, Xiaoxing Ma, and Jian Lü. 2017. AimDroid: activity-insulated multi-level automated testing for Android applications. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 103–114. https://doi.org/10.1109/ICSME.2017.72

[31] Gang Hu, Linjie Zhu, and Junfeng Yang. 2018. AppFlow: using machine learning to synthesize robust, reusable UI tests. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Lake Buena Vista, FL, USA) *(ESEC/FSE 2018)*. Association for Computing Machinery, New York, NY, USA, 269–282. https://doi.org/10.1145/3236024.3236055

[32] Han Hu, Han Wang, Ruiqi Dong, Xiao Chen, and Chunyang Chen. 2024. Enhancing GUI Exploration Coverage of Android Apps with Deep Link-Integrated Monkey. *ACM Trans. Softw. Eng. Methodol.* 33, 6 (June 2024), 1–31. https://doi.org/10.1145/3664810

[33] Kai Huang, Xiangxin Meng, Jian Zhang, Yang Liu, Wenjie Wang, Shuhao Li, and Yuqing Zhang. 2024. An Empirical Study on Fine-Tuning Large Language Models of Code for Automated Program Repair. In *Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering* (Echternach, Luxembourg) *(ASE '23)*. IEEE Press, 1162–1174. https://doi.org/10.1109/ASE56229.2023.00181

[34] Muhammad Imran, Vittorio Cortellessa, Davide Di Ruscio, Riccardo Rubei, and Luca Traini. 2024. An Empirical Study on Code Coverage of Performance Testing. In *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering* (Salerno, Italy) *(EASE '24)*. Association for Computing Machinery, New York, NY, USA, 48–57. https://doi.org/10.1145/3661167.3661196

[35] Ziwei Ji, Tiezheng Yu, Yan Xu, Nayeon Lee, Etsuko Ishii, and Pascale Fung. 2023. Towards Mitigating LLM Hallucination via Self Reflection. In *Findings of the Association for Computational Linguistics: EMNLP 2023*. Association for Computational Linguistics, Singapore, 1827–1843. https://doi.org/10.18653/v1/2023.findings-emnlp.123

[36] Yue Jiang, Eldon Schoop, Amanda Swearngin, and Jeffrey Nichols. 2023. ILuvUI: Instruction-tuned LangUage-Vision modeling of UIs from Machine Conversations. *arXiv preprint arXiv:2310.04869* (2023).

[37] Manabu Kamimura, Akihiko Matsuo, and Yoshiharu Maeda. 2015. Measuring Business Logic Complexity in Software Systems. In *2015 Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 370–376. https://doi.org/10.1109/APSEC.2015.26

[38] Yavuz Koroglu, Alper Sen, Ozlem Muslu, Yunus Mete, Ceyda Ulker, Tolga Tanriverdi, and Yunus Donmez. 2018. QBE: QLearning-based exploration of Android applications. In *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*. 105–115. https://doi.org/10.1109/ICST.2018.00020

[39] Duling Lai and Julia Rubin. 2020. Goal-driven exploration for Android applications *(ASE '19)*. IEEE Press, 115–127. https://doi.org/10.1109/ASE.2019.00021

[40] Xiaolei Li, Jialun Cao, Yepang Liu, Shing-Chi Cheung, and Hailong Wang. 2025. Reusedroid: A vlm-empowered android ui test migrator boosted by active feedback. *arXiv preprint arXiv:2504.02357* (2025).

[41] Xiaoxi Li, Jiajie Jin, Yujia Zhou, Yuyao Zhang, Peitian Zhang, Yutao Zhu, and Zhicheng Dou. 2025. From Matching to Generation: A Survey on Generative Information Retrieval. *ACM Trans. Inf. Syst.* 43, 3 (May 2025), 1–62. https://doi.org/10.1145/3722552

[42] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. 2022. Competition-level code generation with alphacode. *Science* 378, 6624 (2022), 1092–1097.

[43] Yang Li, Jiacong He, Xin Zhou, Yuan Zhang, and Jason Baldridge. 2020. Mapping natural language instructions to mobile UI action sequences. *arXiv preprint arXiv:2005.03776* (2020).

[44] Yuanchun Li, Ziyue Yang, Yao Guo, and Xiangqun Chen. 2017. DroidBot: a lightweight UI-guided test input generator for Android. In *Proceedings of the 39th International Conference on Software Engineering Companion* (Buenos Aires, Argentina) *(ICSE-C '17)*. IEEE Press, 23–26. https://doi.org/10.1109/ICSE-C.2017.8

[45] Yuanchun Li, Ziyue Yang, Yao Guo, and Xiangqun Chen. 2020. Humanoid: a deep learning-based approach to automated black-box Android app testing. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering* (San Diego, California) *(ASE '19)*. IEEE Press, 1070–1073. https://doi.org/10.1109/ASE.2019.00104

[46] Jun-Wei Lin, Reyhaneh Jabbarvand, and Sam Malek. 2020. Test transfer across mobile apps through semantic mapping. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering* (San Diego, California) *(ASE '19)*. IEEE Press, 42–53. https://doi.org/10.1109/ASE.2019.00015

[47] Jun-Wei Lin and Sam Malek. 2022. Gui test transfer from web to android. In *Proceedings of the 15th IEEE International Conference on Software Testing, Verification and Validation (ICST 2022)*. 1–11. https://doi.org/10.1109/ICST53961.2022.00011

[48] Hanyue Liu, Marina Bueno García, and Nikolaos Korkakakis. 2024. Exploring multi-label data augmentation for llm fine-tuning and inference in requirements engineering: A study with domain expert evaluation. In *2024 International Conference on Machine Learning and Applications (ICMLA)*. IEEE, 432–439. https://doi.org/10.1109/ICMLA61862.2024.00064

[49] Zhe Liu, Chunyang Chen, Junjie Wang, Xing Che, Yuekai Huang, Jun Hu, and Qing Wang. 2023. Fill in the Blank: Context-Aware Automated Text Input Generation for Mobile GUI Testing. In *Proceedings of the 45th International Conference on Software Engineering* (Melbourne, Victoria, Australia) *(ICSE '23)*. IEEE Press, 1355–1367. https://doi.org/10.1109/ICSE48619.2023.00119

[50] 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 preprint arXiv:2305.09434* (2023).

[51] Zhe Liu, Chunyang Chen, Junjie Wang, Yuekai Huang, Jun Hu, and Qing Wang. 2022. 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* (New Orleans, LA, USA) *(CHI '22)*. Association for Computing Machinery, New York, NY, USA, 1–14. https://doi.org/10.1145/3491102.3501903

[52] Zhe Liu, Chunyang Chen, Junjie Wang, Yuekai Huang, Jun Hu, and Qing Wang. 2023. Nighthawk: Fully Automated Localizing UI Display Issues via Visual Understanding . *IEEE Transactions on Software Engineering* 49, 01 (Jan. 2023), 403–418. https://doi.org/10.1109/TSE.2022.3150876

[53] Aravind Machiry, Rohan Tahiliani, and Mayur Naik. 2013. Dynodroid: an input generation system for Android apps. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering* (Saint Petersburg, Russia) *(ESEC/FSE 2013)*. Association for Computing Machinery, New York, NY, USA, 224–234. https://doi.org/10.1145/2491411.2491450

[54] Ke Mao, Mark Harman, and Yue Jia. 2016. Sapienz: multi-objective automated testing for Android applications. In *Proceedings of the 25th International Symposium on Software Testing and Analysis* (Saarbrücken, Germany) *(ISSTA 2016)*. Association for Computing Machinery, New York, NY, USA, 94–105. https://doi.org/10.1145/2931037.2931054

[55] Leonardo Mariani, Mauro Pezzè, Valerio Terragni, and Daniele Zuddas. 2021. An evolutionary approach to adapt tests across mobile apps. In *2021 IEEE/ACM International Conference on Automation of Software Test (AST)*. 70–79. https://doi.org/10.1109/AST52587.2021.00016

[56] Bilgin Metin, Martin Wynn, Aylin Tunalı, and Yağmur Kepir. 2025. Business Logic Vulnerabilities in the Digital Era: A Detection Framework Using Artificial Intelligence. *Information* 16, 7 (2025), 585.

[57] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022. Codegen: An open large language model for code with multi-turn program synthesis. *arXiv preprint arXiv:2203.13474* (2022).

[58] Ju Qian, Zhengyu Shang, Shuoyan Yan, Yan Wang, and Lin Chen. 2020. RoScript: a visual script driven truly non-intrusive robotic testing system for touch screen applications. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 297–308. https://doi.org/10.1145/3377811.3380431

[59] 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* (Vienna,

Austria) *(ISSTA 2024)*. Association for Computing Machinery, New York, NY, USA, 958–970. https://doi.org/10.1145/3650212.3680334

[60] Freda Shi, Daniel Fried, Marjan Ghazvininejad, Luke Zettlemoyer, and Sida I Wang. 2022. Natural language to code translation with execution. *arXiv preprint arXiv:2204.11454* (2022).

[61] Aaditya Singh, Adam Fry, Adam Perelman, Adam Tart, Adi Ganesh, Ahmed El-Kishky, Aidan McLaughlin, Aiden Low, AJ Ostrow, Akhila Ananthram, et al. 2025. Openai gpt-5 system card. *arXiv preprint arXiv:2601.03267* (2025).

[62] Helge Spieker, Arnaud Gotlieb, Dusica Marijan, and Morten Mossige. 2017. Reinforcement learning for automatic test case prioritization and selection in continuous integration. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Santa Barbara, CA, USA) *(ISSTA 2017)*. Association for Computing Machinery, New York, NY, USA, 12–22. https://doi.org/10.1145/3092703.3092709

[63] Ting Su, Guozhu Meng, Yuting Chen, Ke Wu, Weiming Yang, Yao Yao, Geguang Pu, Yang Liu, and Zhendong Su. 2017. Guided, stochastic model-based GUI testing of Android apps. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering* (Paderborn, Germany) *(ESEC/FSE 2017)*. Association for Computing Machinery, New York, NY, USA, 245–256. https://doi.org/10.1145/3106237.3106298

[64] Ting Su, Yichen Yan, Jue Wang, Jingling Sun, Yiheng Xiong, Geguang Pu, Ke Wang, and Zhendong Su. 2021. Fully automated functional fuzzing of Android apps for detecting non-crashing logic bugs. *Proc. ACM Program. Lang.* 5, OOPSLA (Oct. 2021), 1–31. https://doi.org/10.1145/3485533

[65] Jingling Sun, Ting Su, Jiayi Jiang, Jue Wang, Geguang Pu, and Zhendong Su. 2023. Property-Based Fuzzing for Finding Data Manipulation Errors in Android Apps. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (San Francisco, CA, USA) *(ESEC/FSE 2023)*. Association for Computing Machinery, New York, NY, USA, 1088–1100. https://doi.org/10.1145/3611643.3616286

[66] Jingling Sun, Ting Su, Kai Liu, Chao Peng, Zhao Zhang, Geguang Pu, Tao Xie, and Zhendong Su. 2023. Characterizing and Finding System Setting-Related Defects in Android Apps. *IEEE Trans. Softw. Eng.* 49, 4 (April 2023), 2941–2963. https://doi.org/10.1109/TSE.2023.3236449

[67] Jingling Sun, Ting Su, Jun Sun, Jianwen Li, Mengfei Wang, and Geguang Pu. 2024. Property-Based Testing for Validating User Privacy-Related Functionalities in Social Media Apps. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering* (Porto de Galinhas, Brazil) *(FSE 2024)*. Association for Computing Machinery, New York, NY, USA, 440–451. https://doi.org/10.1145/3663529.3663863

[68] Maryam Taeb, Amanda Swearngin, Eldon School, Ruijia Cheng, Yue Jiang, and Jeffrey Nichols. 2023. Axnav: Replaying accessibility tests from natural language. *arXiv preprint arXiv:2310.02424* (2023).

[69] Chenxu Wang, Tianming Liu, Yanjie Zhao, Minghui Yang, and Haoyu Wang. 2025. LLMDroid: Enhancing Automated Mobile App GUI Testing Coverage with Large Language Model Guidance. 2, FSE (June 2025), 1001–1022. https://doi.org/10.1145/3715763

[70] Jue Wang, Yanyan Jiang, Ting Su, Shaohua Li, Chang Xu, Jian Lu, and Zhendong Su. 2022. Detecting non-crashing functional bugs in Android apps via deep-state differential analysis. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Singapore, Singapore) *(ESEC/FSE 2022)*. Association for Computing Machinery, New York, NY, USA, 434–446. https://doi.org/10.1145/3540250.3549170

[71] Wenyu Wang, Dengfeng Li, Wei Yang, Yurui Cao, Zhenwen Zhang, Yuetang Deng, and Tao Xie. 2018. An empirical study of Android test generation tools in industrial cases. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering* (Montpellier, France) *(ASE '18)*. Association for Computing Machinery, New York, NY, USA, 738–748. https://doi.org/10.1145/3238147.3240465

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

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

[74] Shengcheng Yu, Chunrong Fang, Mingzhe Du, Zimin Ding, Zhenyu Chen, and Zhendong Su. 2024. Practical, Automated Scenario-Based Mobile App Testing. *IEEE Transactions on Software Engineering* 50, 7 (July 2024), 1949–1966. https://doi.org/10.1109/TSE.2024.3414672

[75] Shengcheng Yu, Chunrong Fang, Mingzhe Du, Yuchen Ling, Zhenyu Chen, and Zhendong Su. 2024. Practical Non-Intrusive GUI Exploration Testing with Visual-based Robotic Arms. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering* (Lisbon, Portugal) *(ICSE '24)*. Association for Computing Machinery, New York, NY, USA, 1–13. https://doi.org/10.1145/3597503.3639161

[76] Chi Zhang, Zhao Yang, Jiaxuan Liu, Yanda Li, Yucheng Han, Xin Chen, Zebiao Huang, Bin Fu, and Gang Yu. 2025. AppAgent: Multimodal Agents as Smartphone Users. In *Proceedings of the 2025 CHI Conference on Human Factors in Computing Systems (CHI '25)*. Association for Computing Machinery, New York, NY, USA, 1–20. https://doi.org/10.1145/3706598.3713600

[77] Shun Zhang, Zhenfang Chen, Yikang Shen, Mingyu Ding, Joshua B Tenenbaum, and Chuang Gan. 2023. Planning with large language models for code generation. *arXiv preprint arXiv:2303.05510* (2023).

[78] Yakun Zhang, Chen Liu, Xiaofei Xie, Yun Lin, Jin Song Dong, Dan Hao, and Lu Zhang. 2026. GUI test migration via abstraction and concretization. *ACM Transactions on Software Engineering and Methodology* 35, 2 (2026), 1–29.

[79] Yakun Zhang, Wenjie Zhang, Dezhi Ran, Qihao Zhu, Chengfeng Dou, Dan Hao, Tao Xie, and Lu Zhang. 2024. Learning-based widget matching for migrating gui test cases. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*. 1–13.

[80] Yakun Zhang, Qihao Zhu, Jiwei Yan, Chen Liu, Wenjie Zhang, Yifan Zhao, Dan Hao, and Lu Zhang. 2024. Synthesis-Based Enhancement for GUI Test Case Migration. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 869–881.

[81] Yixue Zhao, Justin Chen, Adriana Sejfia, Marcelo Schmitt Laser, Jie Zhang, Federica Sarro, Mark Harman, and Nenad Medvidovic. 2020. FrUITeR: a framework for evaluating UI test reuse. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Virtual Event, USA) *(ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 1190–1201. https://doi.org/10.1145/3368089.3409708