LELANTE: LEveraging LLM for Automated ANdroid TEsting

Haz Sameen Shahgir
University of California, Riverside
Riverside, USA
hshah057@ucr.edu

Shamit Fatin
Bangladesh University of Engineering and Technology (BUET)
Dhaka, Bangladesh
1805055@cse.buet.ac.bd

Mehbubul Hasan Al-Quvi
Bangladesh University of Engineering and Technology (BUET)
Dhaka, Bangladesh
1805007@cse.buet.ac.bd

Sukarna Barua
Bangladesh University of Engineering and Technology (BUET)
Dhaka, Bangladesh
sukarnabarua@cse.buet.ac.bd

Anindya Iqbal
Bangladesh University of Engineering and Technology (BUET)
Dhaka, Bangladesh
anindya@cse.buet.ac.bd

Sadia Sharmin
Bangladesh University of Engineering and Technology (BUET)
Dhaka, Bangladesh
sadiasharmin@cse.buet.ac.bd

Md. Mostofa Akbar
Bangladesh University of Engineering and Technology (BUET)
Dhaka, Bangladesh
mostofa@cse.buet.ac.bd

Kallol Kumar Pal
Samsung R&D Institute Bangladesh (SRBD)
Dhaka, Bangladesh
kallol.kumar@samsung.com

A. Asif Al Rashid
Samsung R&D Institute Bangladesh (SRBD)
Dhaka, Bangladesh
ahmad.asif@samsung.com

Abstract

Given natural language test case description for an Android application, existing testing approaches require developers to manually write scripts using tools such as Appium and Espresso to execute the corresponding test case. This process is labor-intensive and demands significant effort to maintain as UI interfaces evolve throughout development. In this work, we introduce *LELANTE*, a novel framework that utilizes large language models (LLMs) to automate test case execution without requiring pre-written scripts. *LELANTE* interprets natural language test case descriptions, iteratively generate action plans, and perform the actions directly on the Android screen using its GUI. *LELANTE* employs a screen refinement process to enhance LLM interpretability, constructs a structured prompt for LLMs, and implements an action generation mechanism based on chain-of-thought reasoning of LLMs. To further reduce computational cost and enhance scalability, *LELANTE* utilizes model distillation using a foundational LLM. In experiments across 390 test cases spanning 10 popular Android applications, *LELANTE* achieved a **73%** test execution success rate. Our results demonstrate that LLMs can effectively bridge the gap between natural language test case description and automated execution, making mobile testing more scalable and adaptable.

CCS Concepts

• **Software and its engineering** → **Software testing and debugging**; • **Computing methodologies** → Planning for deterministic actions; Natural language processing.

Keywords

Automated Android Testing, Large Language Models (LLMs), GUI Refinement, Model Distillation, Prompt Engineering

ACM Reference Format:
Haz Sameen Shahgir, Shamit Fatin, Mehbubul Hasan Al-Quvi, Sukarna Barua, Anindya Iqbal, Sadia Sharmin, Md. Mostofa Akbar, Kallol Kumar Pal, and A. Asif Al Rashid. 2025. LELANTE: LEveraging LLM for Automated ANdroid TEsting. In *Evaluation and Assessment in Software Engineering (EASE '25), June 17–20, 2025, Istanbul, Turkiye.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3756681.3757032

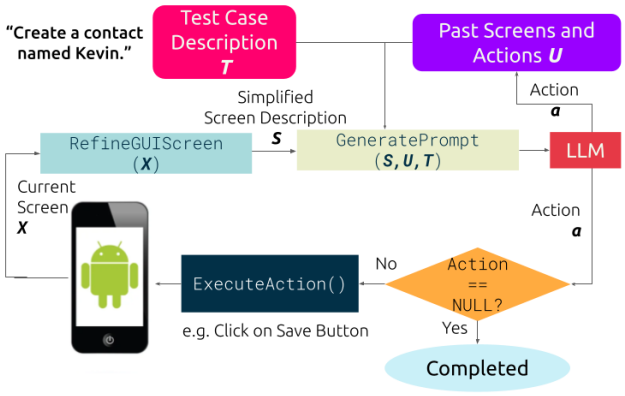


Figure 1: Overview of the *LELANTE* test case execution process. Given a natural language test case description, *LELANTE* incrementally refines the screen GUI representation and executes the test case step by step until completion.

1 Introduction

The rapid growth of the mobile application ecosystem has made Android the dominant platform for app development, with a market share consistently exceeding 70% [21]. This diversity allows for

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

EASE '25, Istanbul, Turkiye

© 2025 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-1385-9/25/06
https://doi.org/10.1145/3756681.3757032

innovation but also presents challenges for developers aiming to ensure consistent functionality across various devices, operating system versions, and screen sizes [4, 6, 12, 24]. As mobile applications evolve, rigorous testing becomes essential to maintaining software quality.

Application testing involves designing and executing **test cases**, which define scenarios to validate an application's functionality. Each test case typically includes a **test case description**, outlining the overall objective of the test. For example, a test case may verify whether a user can successfully log into an application, and its description would be "`Log in the application by entering user name and password`". In some cases, a test case may also specify a sequence of **test steps** or **actions**, which are atomic, human-performable interactions (e.g., `tapping a button`, `entering text`) that must be executed in order to complete the test case. For instance, the test steps for the login test case might include `launching the application`, `tapping the "Login" button`, `entering a valid email and password`, `pressing "Sign In"` and `verifying that the home screen appears`. Both test case descriptions and test steps are written in human-readable language and are not structured for direct machine interpretation.

Quality Assurance (QA) teams are responsible for creating test cases to ensure comprehensive coverage of an application's features and expected behavior. Human testers then execute these test cases by either following the specified test steps or, if no steps are provided, relying on their own understanding of the test objective. However, manually defining test steps is a labor-intensive process that requires continuous updates as applications evolve. This challenge is particularly pronounced in large development teams, where testing team may be unfamiliar with the application's functionality. In this work, we assume that test steps are not explicitly provided by the QA team. Therefore, the system must autonomously determine the appropriate sequence of actions to execute each test case.

Automated testing frameworks such as Appium [26] and Espresso [3] allow testers to write scripts for test cases that simulate **test steps** on an application. Although these tools ensure consistency and repeatability, they require significant manual effort and technical knowledge to create, maintain, and rewrite test scripts, especially as user interfaces evolve throughout development [2, 8, 13, 17, 23]. Record-and-replay mechanisms [22], which capture user actions for later execution, simplify the scripting process to some extent but struggle with dynamic UI modifications [9]. These limitations necessitate an automation tool that can directly interpret pre-existing human-readable test cases description in order to execute the test case without requiring manual intervention.

Recent advances in artificial intelligence, particularly in large language models (LLMs) such as GPT-4 [1] and LLaMA [27], have enabled new possibilities for automating human-readable tasks[10, 14, 16, 18, 31]. For example, AutoDroid [29] employs LLMs to automate Android tasks, demonstrating feasibility in interpreting and executing simple user commands. However, AutoDroid is not designed for structured test case execution, as it lacks mechanisms for dynamically adapting to UI changes and recovering from execution errors and plan for knowledge distillation. Furthermore it relies on foundational LLMs without incorporating model distillation techniques, making its execution computationally expensive and less practical for large-scale testing. These limitations make it unsuitable for comprehensive and scalable test automation.

In this work, we introduce ***LELANTE***, a novel framework that automates test case execution using LLMs, given only a natural language test case description. *LELANTE* eliminates the need for manually written test scripts by automatically generating a test case execution plan, consisting of required action sequence for executing the test case. *LELANTE* guides an LLM to generate actions iteratively using a structured prompt based on a refined screen representation, past action history, and test case description. The generated actions are executed on the application until *LELANTE* determines that the test case execution is complete. Figure 1 illustrates the core execution process of *LELANTE*. By automating this entire process, *LELANTE* significantly reduces the human effort required for software testing, allowing teams to conduct more extensive testing with minimal overhead. *LELANTE* eliminates the need for writing detailed and time-consuming test case steps, as it executes the test case directly from test case descriptions.

Through experiments across 390 test cases spanning 10 popular Android applications, *LELANTE* achieved a **73% test execution success rate**. Our findings demonstrate that LLM-based execution significantly improves the adaptability and scalability of mobile testing workflows, reducing the reliance on brittle, manually maintained scripts. This paper presents the following key contributions:

- A fully automated test case execution framework that eliminates the need for manually written scripts, allowing natural language **test case descriptions** to be executed directly and dynamically on the Android application.
- A structured prompt to guide the LLM and a GUI refinement process to extract relevant interactive elements from screen XML representations, reducing noise and improving the LLM's ability to accurately interpret the application interface and reason actions.
- Efficient distillation of open-source large language models for *LELANTE* execution, reducing computational overhead while maintaining accuracy.
- Empirical validation demonstrating a **73% test execution success rate** across **390 test cases**, showcasing the effectiveness of LLM-driven automation.

2 Problem Statement

Let T denote a natural language test case description, and S represent the state of the Android application system at any given time. The state S includes all observable and internal aspects of the system, such as the current screen layout, UI components, system settings, network conditions, and application runtime states. From S, we obtain a simplified and structured representation G, which captures only essential components of the state including UI elements and their properties, allowing for reasoning about available interactions.

At the start of testing, the Android device is in an initial state S_0. The execution of an action or a test step a_t causes a transition in the system, leading to a new state S_t, modeled by the state transition function Φ:

$$S_t = \Phi(S_{t-1}, a_t).$$

The function Φ captures how Android updates its state in response to user interactions. For instance, if the action a_t is "tap on a button," Φ updates S_{t-1} to reflect the resulting UI change, such as navigating to a new screen or displaying a pop-up dialog. Similarly, if a_t is "enter text in a field," Φ modifies S_{t-1} to include the new text input.

The test case description T implicitly defines an expected final state S_e, which specifies the conditions that must be satisfied for the test to be considered successful. For example, if T is `"Delete the contact named X"`, then S_0 represents the initial state where the contact exists in storage. The sequence of actions $\{a_1, a_2, \ldots, a_n\}$ may involve opening the contacts app, searching for the contact, selecting the delete option, and confirming the deletion. The expected final state S_e should then satisfy the condition that the contact is no longer retrievable in the storage system.

The goal of *LELANTE* is to generate a valid sequence of actions $\{a_1, a_2, \ldots, a_n\}$ such that the android application transitions from S_0 to S_e in compliance with the test case. At each step t, *LELANTE* acts as an agent that determines the next action:

$$a_t = f\big(T, \{a_1, a_2, \ldots, a_{t-1}\}, G\big),$$

ensuring that after n steps, the system reaches $S_n = S_e$. *LELANTE* employs iterative decision-making to automate Android testing to continuously adapt its actions based on the observed UI state.

A test case is considered successfully completed if, after executing a valid sequence of actions $\{a_1, a_2, ..., a_n\}$, the Android application's final state S_n matches the expected final state S_e implied by the test case description T. If the final state does not satisfy the conditions implied by the description, the test case is considered incomplete or failed.

3 Methodology

LELANTE begins with refining the application's graphical representation, eliminating unnecessary metadata while preserving only relevant interactive elements to create a structured and meaningful screen description (§ 3.1). This refined representation is then used to generate a structured prompt (§ 3.2), enabling the LLM to determine the next action based on the provided test case description and execution history. Once an action is selected, it is mapped to the corresponding UI element and executed on the device, with built-in mechanisms for ambiguity resolution and iterative GUI interaction (§ 3.3). To ensure cost-effective execution without reliance on third-party vendors, *LELANTE* employs model distillation using an open-source large language model, significantly reducing computational overhead while maintaining accuracy (§ 3.4).

3.1 Screen Representation Refinement

Android applications store screen layouts as XML structures, which, while comprehensive, are often verbose and inefficient for direct understanding and processing by LLMs. *LELANTE* applies a preprocessing step to refine the screen representation by extracting only the essential interactive and informative components from an XML screen description. The refinement process begins with XML parsing, removing elements that lack interaction potential, such as non-essential and non-interactive background images, labels, and decorative UI components. Interactive elements, including buttons, text fields, and checkable inputs, are retained. Each actionable element is assigned with an unique identifiers to enable referencing during execution. Some retained non-interactive elements include textual descriptions or icons. Since *LELANTE* relies on a purely text-based LLM for action inference, an image-to-text icon classification model is utilized to convert icon images into textual descriptions. Based on a Vision Transformer (ViT) model [5], the icon-classifier is integrated in *LELANTE* to recognize icons and include their semantic labels in the representation.

In addition to existing screen elements, *LELANTE* explicitly adds navigation buttons for scrolling and a 'Back' action, to the refined screen representation. Inclusion of the 'Back' button provides a built-in recovery action, enabling *LELANTE* to backtrack (guided by reasoning from LLM) and rectify erroneous action sequences autonomously. Figure 2 provides an example of this refinement process.

3.2 LLM-Driven Action Generation

Given natural language test case description, refined screen representation, and a record of previous actions, *LELANTE* constructs a structured prompt to guide the LLM in reasoning about the next action. The prompt is designed to enforce a structured chain-of-thought [28] reasoning process, ensuring that the model first thinks about how to reach the final state before generating appropriate action. The overall structure of the prompt is given in Appendix B.

A key challenge in this process was resolving ambiguity when the LLM selected an action. To address this, *LELANTE* assigns unique identifiers to each actionable UI element during the screen refinement stage, ensuring that the LLM can reference them precisely. Instead of generating free-text descriptions of UI elements, the LLM directly outputs the corresponding identifier, eliminating potential misinterpretations and reducing the need for additional disambiguation steps.

An important component of our structured reasoning mechanism is error recovery. During the execution, *LELANTE* analyzes the execution history and current screen to autonomously detect whether it has executed any incorrect or unintended action through iterative reasoning steps. Upon detection of an erroneous step, *LELANTE* chooses the explicitly provided 'Back' button in the refined screen representation to revert to an earlier screen state, from which it re-evaluates and selects an alternative action in subsequent steps to rectify its action plan. The reasoning process repeats the backtracking until a viable action path is found or until the test case is successfully completed.

To programmatically determine when a test case execution should end, *LELANTE* explicitly instructs the LLM, within the structured prompt described in Appendix B, to reason about whether the expected final state described in problem statement has been reached. Specifically, the LLM sets a boolean indicator when it detects completion, accompanied by a unique action identifier -1. *LELANTE* detects this special identifier, signaling termination of the ongoing test execution. Finally, a human tester manually validates if the actual application state matches the intended outcome from the test case description, confirming successful execution.

3.3 Action Execution Using GUI Interaction

After generating an action by LLM, *LELANTE* executes it on the application GUI using the identified UI element and applying the interaction (e.g, input a text or click a button). We use Appium to

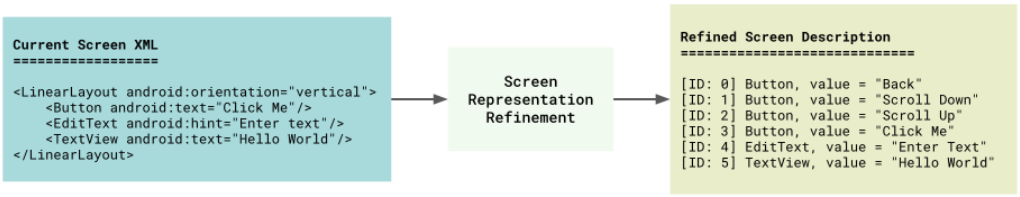


Figure 2: Screen representation refinement for LLM understanding. This process extracts essential and actionable elements (e.g., buttons, text fields) from the original screen representation (XML) and convert it to an optimized format suitable for LLM prompts.

execute LLM provided action into the Android GUI. Once the action is executed, the application state changes and the XML representation of the new screen is fetched by Appium. This screen along with previous actions and test steps are iteratively processed through the *LELANTE* pipeline until it decides that no further action is required.

3.4 Distillation for Cost-Effective Execution

While foundational LLMs (e.g., GPT-4o and Claude) demonstrate strong reasoning capabilities, their computational demands make them impractical for local or cheap test execution. To address this, *LELANTE* employs knowledge distillation [7], where a smaller model is fine-tuned using data generated by a larger model. This reduces computational overhead while preserving structured reasoning capabilities.

The distillation process began with human testers executing test cases using a foundational model (GPT-4o), during which we recorded its responses, including predicted actions and reasoning steps. We collected test cases for **30 Android applications** that were not included in the final evaluation set. To maintain high-quality training data, we filtered out erroneous actions, redundant backtracking, and inefficient execution paths, resulting in a fine-tuning dataset consisting of 4,000 test case steps.

For model adaptation, we employed **Low-Rank Adaptation (LoRA)** [11], a parameter-efficient fine-tuning approach, allowing us to optimize the model on local machines without excessive computational overhead. Because we employ a step by step execution model, we needed to train the model on the test steps rather than in full execution history. Our fine-tuned model retained the structured reasoning of the foundational model while significantly reducing inference costs, making *LELANTE* more practical for real-world deployment.

4 Evaluation

We evaluate *LELANTE* based on its accuracy and efficiency in executing natural language test cases, answering the following two research questions:

RQ1: How accurately and efficiently can *LELANTE* execute natural language test cases across various applications?

RQ2: How effectively can LELANTE recover from erroneous steps?

Our evaluation is performed using a dataset of *390 human-generated test cases* for *10 Android applications*. The minimum number of test steps required to complete each test case varied. Tests were conducted on physical Android devices (Samsung A14, A15, and A03 models) running Android 11, replicating realistic testing environments.

Technique	Model	Test Execution Success Rate	Error Recovery Rate	Execution Time per Step
AutoDroid	GPT-4o	71.3%	12%	20.5s
LELANTE	GPT-4o	**73%**	**78%**	11.8s
LELANTE	LLaMA-3.1 8B (Base)	60%	10%	**11.0s**
LELANTE	LLaMA-3.1 8B (Fine-tuned)	65%	55%	**11.0s**

Table 1: Test Execution Performance Metrics for Various Techniques and Models

Evaluation metrics. We use the following three metrics:

- **Test Execution Success Rate**: Percentage of test cases completely executed.
- **Error Recovery Rate**: Percentage of erroneous steps after which *LELANTE* autonomously detects mistakes, navigates backward, and chooses the correct alternative step.
- **Execution Time per Step**: Average time required to perform each test step.

Performance of LELANTE in test case execution. The evaluation results are summarized in Table 1. For RQ1, the **73% success rate** demonstrates *LELANTE*'s reliability and efficiency in executing natural language test cases description without predefined scripts. For RQ2, the **78% error recovery rate** highlights the framework's capability to dynamically handle execution errors. Due to limited prior work in structured test execution automation, we compare *LELANTE* against the closely related AutoDroid. The comparison emphasizes *LELANTE*'s strengths in error correction (66% higher), execution success rate (1.7% higher), and execution efficiency (8.7s lower time required per step) compared to AutoDroid. Furthermore, its use of distilled models enables cost-effective and scalable deployment.

Impact of Test Case Length on Execution Success. Minimum number of test steps needed to complete a test case can affects execution success rate. Figure 3 shows that most test cases have 3–10 steps (left) and that shorter test cases (fewer than 6 steps) succeed more often (right). Longer test cases see lower success due to increased interactions, error risks, and incorrect action selection. This highlights the need for refining test descriptions and improving error recovery.

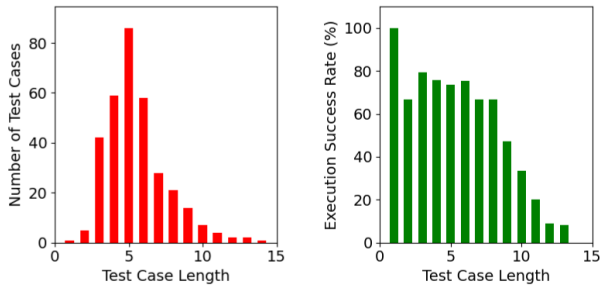


Figure 3: The left histogram shows test case length distribution, with most cases in the 3–10 step range. The right histogram shows that shorter test cases succeed more often, while longer ones have lower success due to increased interactions and error risks

Condition	Execution Success Rate
Original Test Cases	65%
Modified Test Cases	73%

Table 2: Impact of Test Case Modifications (Ambiguity Resolution of Test Case Descriptions) on Execution Success Rate

Impact of Ambiguity Resolution in Test Cases. Ambiguous test descriptions reduced execution reliability. For example, *"Change notification settings"* lacked details on which settings to modify, leading *LELANTE* to misinterpret steps. Refining descriptions (e.g., *"Disable email notifications from the Settings menu"*) improved success rates from 65% to 73% (Table 2). Guidelines for resolving ambiguity are in Appendix A.

5 Related Works

Automated mobile application testing has traditionally relied on script-based frameworks such as *Appium* and *Espresso* [3, 26], which allow developers to write scripts that simulate user interactions. While these tools provide consistency in test execution, they require extensive manual effort to maintain scripts as applications evolve, making them inefficient for large-scale automation. To mitigate script maintenance overhead, record-and-replay mechanisms, such as *RERAN* [15], capture user interactions for deterministic execution. However, these approaches remain fragile when handling dynamic UI changes requiring manual adaptation of the scripts and limiting their effectiveness in real-world scenarios. In contrast, LELANTE's LLM driven execution directly interprets human-language test cases without requiring any pre-written scripts.

To improve scalability, search-based techniques such as *Stoat* [25] and *Sapienz* [20] employ probabilistic and evolutionary algorithms for automated exploratory testing. Although effective in broadening test coverage, these methods often require substantial computational resources and manual validation of results. Recently, AI-driven methods such as *AutoDroid* [29] leverage LLMs to automate single-step UI interactions. However, AutoDroid primarily focuses on task execution without explicitly handling incorrect execution paths or error recovery. In contrast, *LELANTE* adopts an iterative, reasoning-driven approach specifically designed for testers, emphasizing robustness through dynamic error detection and recovery.

Multimodal frameworks integrating vision-language models [19, 30] have also been explored for UI understanding and verification. However, these models typically involve high computational costs due to heavy reliance on visual processing, limiting practical deployment at scale. *LELANTE*, on the other hand, relies solely on textual representation derived from a GUI refinement process (§ 3.1), which enhances the effectiveness of prompt engineering, improves model performance, and reduces deployment costs.

6 Discussion

LELANTE demonstrates a significant step in automating Android testing, achieving a 73% success rate and 78% error recovery. However, performance declines with longer test cases, highlighting challenges in handling ambiguity and error propagation. While GUI refinement improves execution, reliance on manual test case modifications suggests limitations in natural language understanding. Another limitation of *LELANTE* is its inability to autonomously verify if an executed test case successfully achieves its expected results. Consequently, human validation remains essential to confirm whether the expected final application state is correctly attained. Additionally, the current backtracking mechanism remains relatively slow during error recovery, necessitating improvements on reasoning mechanism and disambiguation strategy.

7 Conclusion and Future Work

In this work, we propose *LELANTE* , an LLM-driven framework for automating the execution of test cases for Android application. *LELANTE* utilizes a refined screen representation and a structured prompt to guide an LLM in generating execution plans for test cases. *LELANTE*'s reasoning-driven action selection and automated backtracking for error correction results in an enhanced execution efficiency and adaptability to dynamic UI changes. Our work paves the way for further advancements in AI-driven test automation, reducing developer effort, improving software quality and eliminating the need for manual script writing for test case automation. Future efforts will focus on automating test outcome (i.e., expected results) validation to reduce human dependency, optimizing backtracking performance, and investigating prompt-based or incremental learning techniques for leveraging historical execution data without extensive model fine-tuning.

Acknowledgments

This work was supported by Samsung R&D Institute Bangladesh (SRBD) and Bangladesh University of Engineering and Technology (BUET).

References

[1] 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).

[2] Riccardo Coppola, Luca Ardito, Maurizio Morisio, and Marco Torchiano. 2020. Mobile testing: new challenges and perceived difficulties from developers of the Italian industry. *IT Professional* 22, 5 (2020), 32–39.

[3] Android Developers. 2025. Espresso Testing Framework - Android Developers. https://developer.android.com/training/testing/espresso Accessed: 2025-03-03.

[4] Gianluca Dini, Fabio Martinelli, Ilaria Matteucci, Marinella Petrocchi, Andrea Saracino, and Daniele Sgandurra. 2018. Risk analysis of Android applications: A user-centric solution. *Future Generation Computer Systems* 80 (2018), 505–518.

[5] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby. 2021. An Image is

Worth 16x16 Words: Transformers for Image Recognition at Scale. In *International Conference on Learning Representations*. https://openreview.net/forum?id=YicbFdNTTy

[6] Hossein Falaki, Ratul Mahajan, Srikanth Kandula, Dimitrios Lymberopoulos, Ramesh Govindan, and Deborah Estrin. 2010. Diversity in smartphone usage. In *Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services* (San Francisco, California, USA) *(MobiSys '10)*. Association for Computing Machinery, New York, NY, USA, 179–194. doi:10.1145/1814433.1814453

[7] Jianping Gou, Baosheng Yu, Stephen J Maybank, and Dacheng Tao. 2021. Knowledge distillation: A survey. *International Journal of Computer Vision* 129, 6 (2021), 1789–1819.

[8] Giovanni Grano, Adelina Ciurumelea, Sebastiano Panichella, Fabio Palomba, and Harald C Gall. 2018. Exploring the integration of user feedback in automated testing of android applications. In *2018 IEEE 25Th international conference on software analysis, evolution and reengineering (SANER)*. IEEE, 72–83.

[9] Jiaqi Guo, Shuyue Li, Jian-Guang Lou, Zijiang Yang, and Ting Liu. 2019. Sara: self-replay augmented record and replay for android in industrial cases. In *Proceedings of the 28th acm sigsoft international symposium on software testing and analysis*. 90–100.

[10] Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. 2024. Large language models for software engineering: A systematic literature review. *ACM Transactions on Software Engineering and Methodology* 33, 8 (2024), 1–79.

[11] Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, Weizhu Chen, et al. 2022. Lora: Low-rank adaptation of large language models. *ICLR* 1, 2 (2022), 3.

[12] Manishaben Jaiswal. 2018. Android the mobile operating system and architecture. *Manishaben Jaiswal," ANDROID THE MOBILE OPERATING SYSTEM AND ARCHITECTURE", International Journal of Creative Research Thoughts (IJCRT), ISSN* (2018), 2320–2882.

[13] Pingfan Kong, Li Li, Jun Gao, Kui Liu, Tegawendé F Bissyandé, and Jacques Klein. 2018. Automated testing of android apps: A systematic literature review. *IEEE Transactions on Reliability* 68, 1 (2018), 45–66.

[14] Metin Konuk, Cem Baglum, and Ugur Yayan. 2024. Evaluation of Large Language Models for Unit Test Generation. In *2024 Innovations in Intelligent Systems and Applications Conference (ASYU)*. IEEE, 1–6.

[15] Wing Lam, Zhengkai Wu, Dengfeng Li, Wenyu Wang, Haibing Zheng, Hui Luo, Peng Yan, Yuetang Deng, and Tao Xie. 2017. Record and replay for android: Are we there yet in industrial cases?. In *Proceedings of the 2017 11th joint meeting on foundations of software engineering*. 854–859.

[16] Youwei Li, Yangyang Li, and Yangzhao Yang. 2024. Test-Agent: A Multimodal App Automation Testing Framework Based on the Large Language Model. In *2024 IEEE 4th International Conference on Digital Twins and Parallel Intelligence (DTPI)*. IEEE, 609–614.

[17] Jun-Wei Lin, Navid Salehnamadi, and Sam Malek. 2020. Test automation in open-source android apps: A large-scale empirical study. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. 1078–1089.

[18] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2024. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *Advances in Neural Information Processing Systems* 36 (2024).

[19] Zhe Liu, Cheng Li, Chunyang Chen, Junjie Wang, Boyu Wu, Yawen Wang, Jun Hu, and Qing Wang. 2024. Vision-driven automated mobile gui testing via multimodal large language model. *arXiv preprint arXiv:2407.03037* (2024).

[20] 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*. 94–105.

[21] Bahman Rashidi and Carol J Fung. 2015. A Survey of Android Security Threats and Defenses. *J. Wirel. Mob. Networks Ubiquitous Comput. Dependable Appl.* 6, 3 (2015), 3–35.

[22] Onur Sahin, Assel Aliyeva, Hariharan Mathavan, Ayse Coskun, and Manuel Egele. 2019. Randr: Record and replay for android applications via targeted runtime instrumentation. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 128–138.

[23] Kabir S Said, Liming Nie, Adekunle A Ajibode, and Xueyi Zhou. 2020. GUI testing for mobile applications: objectives, approaches and challenges. In *Proceedings of the 12th Asia-Pacific Symposium on Internetware*. 51–60.

[24] Anirban Sarkar, Ayush Goyal, David Hicks, Debadrita Sarkar, and Saikat Hazra. 2019. Android application development: a brief overview of android platforms and evolution of security systems. In *2019 Third International conference on I-SMAC (IoT in Social, Mobile, Analytics and Cloud)(I-SMAC)*. IEEE, 73–79.

[25] 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*. 245–256.

[26] Appium Team. 2025. Appium Documentation. https://appium.io/docs/en/latest/ Accessed: 2025-03-03.

[27] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. 2023. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971* (2023).

[28] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems* 35 (2022), 24824–24837.

[29] 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*. 543–557.

[30] Qinzhuo Wu, Weikai Xu, Wei Liu, Tao Tan, Liujian Liujianfeng, Ang Li, Jian Luan, Bin Wang, and Shuo Shang. 2024. MobileVLM: A Vision-Language Model for Better Intra- and Inter-UI Understanding. In *Findings of the Association for Computational Linguistics: EMNLP 2024*, Yaser Al-Onaizan, Mohit Bansal, and Yun-Nung Chen (Eds.). Association for Computational Linguistics, Miami, Florida, USA, 10231–10251. doi:10.18653/v1/2024.findings-emnlp.599

[31] Daniel Zimmermann and Anne Koziolek. 2023. GUI-Based Software Testing: An Automated Approach Using GPT-4 and Selenium WebDriver. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering Workshops (ASEW)*. IEEE, 171–174.

A Guidelines for Formatting Test Cases

Creating well-structured **test case descriptions** is crucial for ensuring the optimal performance of *LELANTE*'s LLM-based processing pipeline. Poorly written test cases can introduce **ambiguity**, making it difficult for the model to interpret and execute the correct actions. For example, instead of saying *"Add category inside the Edit Category window"*, a clearer version would be *"Add a new category named* `Elysium` *from inside the Edit Category window"*.

Additionally, test cases should be structured to provide only **essential details** while maintaining **precision**. If an action can be performed in multiple ways, only **one method** should be specified, such as *"Set call vibration as* `Basic call` *in Call alerts and ringtones"* instead of just *"Set call vibration as* `Basic call`*"*. **Unnecessary details** should be avoided when there is only **one way** to complete a task. When necessary, **explicit directions** should be provided to help locate options within the UI, such as clarifying that *"Gallery settings are located in the drawer"*. Finally, test case descriptions should always include the **final confirmation step** to prevent premature completion—an instruction like *"Update Samsung account profile picture by taking a picture"* should be revised to *"Update Samsung account profile picture by taking a picture. Click* `Done` *to save"*.

B Prompt structure

To ensure accurate test execution, *LELANTE* employs a structured prompt shown below that guides the LLM in understanding the test case, evaluating prior actions, and determining the next step.

```
You are given the textual description of a screen, past actions,
      and an overall goal. Some actions have already been taken in
      the past and you need to determine the immediate next action
      needed to achieve the goal. Do the past actions indicate
      that the goal has already been achieved? Output a JSON with
      keys:
"goal_action_plan": "string, a goal is made up of a series of
      actions that need to be taken to achieve it. Write a
      detailed description of the actions required. Some actions
      may already have been completed. The next action may not be
      directly on the screen, but an action on the screen could
      lead to it. Think strategically, step by step, considering
      the goal and past actions.",
"past_actions_summary": "string, a short description of what
      actions were taken in the past",
```

```
"no_further_action_needed": "string, whether the past actions
      fully achieve the goal or not. Start with 'Past Actions
      indicate ...' or 'Past Actions do not indicate ...'. Don't
      forget to save if instructed.",
      "no_further_action_needed_bool": "boolean, true or false",
"immediate_next_action": "string, a detailed explanation of the
      next action to perform, based on the goal and past actions.
      Think step by step.",
"current_screen_actions": [ [action: str, ID: int] ],
"selected_current_screen_action": [reasoning: str, action: str,
      ID: int],
"repeating_past_action": "string, describing if a past action is
      being repeated, excluding scrolling or going back.",
"repeating_past_action_bool": "boolean, true or false. False for
      scrolling or going back.",
"id": "number/integer, the ID of the element to interact with.
      Use -1 if no action is needed.",
"text_input_value": "string, only EditText and
      AutoCompleteTextView can accept textinputvalue. For other
      elements, use '<NOVALUE>'"
```

Current Screen: **<Refined Current Screen Description>**
Overall Goal: **<Test Case Description>**
Past Actions: **<Past Actions Selected by LELANTE>**