POLITECNICO DI TORINO

POLITECNICO DI TORINO

Master Degree course in Communications and Computer Networks Engineering

Master Degree Thesis

Adaptive GUI Test Evolution and Oracle Maintenance

Supervisors
Prof. Riccardo Coppola
Prof. Tommaso Fulcini

Candidate
Alessandro Poletti

Academic Year 2024-2025

Acknowledgements

To my family for their support and encouragement throughout my years of study.

Abstract

The mobile applications industry has grown significantly in the last few years, bringing companies to launch their own applications to the market as fast as possible, and support them with continuous updates. In this context, one of the primary concerns is the need for a thorough testing phase, which is often performed superficially or neglected altogether, being a costly and time-consuming activity. GUI testing in particular plays a crucial role, since it allows the simulation of direct user interaction.

As an application evolves, its GUI visual appearance, internal structure, and properties often change over time. This may mean that tests written for one version of the application could fail when executed on a subsequent version of the same application, not because of a malfunction of the application, but because the tests themselves are outdated. This issue is often described as fragility in software testing literature. As a consequence, developers should dedicate additional effort when a test fails after an update, understanding whether the test fails because of an actual defect of the application, or because it is outdated. In the latter case, the test needs to be manually repaired for the updated application.

Over the years, many solutions have been proposed to mitigate the amount of effort needed for test repair. Large Language Models (LLMs), with their ability to understand and generate natural language and other types of content to perform a wide range of tasks, represent a promising solution for addressing mobile GUI test repair challenges.

The goals of this study are: first, determine what are the most common causes for mobile GUI tests to break from one application version to another; second, understand how an LLM would help to reduce the effort needed for mobile GUI test repair; third, compare the performance of an LLM-based repair approach to a state-of-the-art repair tool, namely Healenium-Appium. To do so, we gathered 19 real-world Android applications with 61 broken GUI tests, and analyze the causes for them to break. Then, we developed an LLM-based approach for mobile GUI test repair, that leverages on multiple interaction with an LLM to generate repaired tests. Finally, the performance of our approach is compared to the baseline of Healenium-Appium.

In the experimental evaluation, our LLM-based approach was able to repair 45 of the 61 tests (73.8%) after only one interaction with the LLM, and 56 of the 61 tests (91.8%) after two or more interactions, outperforming the Healenium-Appium baseline by generating 50.8% and 68.8% more repaired tests respectively.

These results demonstrate that LLM-based repairs represent a very promising approach in addressing the fragility of mobile GUI tests. Our LLM-based approach is able to significantly reduce the manual effort required to maintain test suites as applications evolve, while achieving higher repair success rates than existing automated solutions. The integration of LLMs into mobile testing workflows can greatly help developers enhance test reliability and maintenance scalability.

Contents

Chapter 1

Introduction

The mobile application industry has grown significantly in the last few years, attracting the interest of several companies. Many companies have started offering support for mobile applications in most of their services. This brought the necessity for the companies to launch their applications on the market in the shortest possible time, and support them with continuous update. In this context, one of the main concern is for quality assurance, and in particular for a thorough software testing phase, which is often performed superficially or neglected altogether, being a costly and time-consuming activity.

Testing is a crucial phase of the software development process, allowing to verify that a software product functions as intended, and helping to prevent bugs and improve performance. Graphical User Interface (GUI) testing in particular plays an important role in mobile applications, as it simulates the direct user interaction.

Mobile applications typically receive a lot of updates over time. As an application evolves, its visual appearance, internal structure, and properties often change. This has an impact on the testing process. A test that was written for a certain version of an application may fail when executed on a subsequent version of the same application, not because it reflects an actual malfunction of the application, but because the test itself is outdated. This issue is often described as fragility in software testing literature. This means that developers should dedicate additional effort when the application changes, to keep the tests up to date. This is a cumbersome task: for each test whose execution fails on the updated version of the application, developers have to spend time to understand whether the failure is caused by a malfunction of the application, or the test is outdated and has to be repaired.

During the years, many solutions have been proposed to help reduce the effort for developers to repair the tests. The growth of Large Language Models (LLMs) in recent years, with their ability to understand and generate natural language and other types of content to perform a wide range of tasks, may represent a promising solution for this task. Some researchers have already started proposing tests repair approaches integrated with LLMs.

This research brings the following contributions:

- an analysis on what are the causes of mobile GUI tests failure and their incidence when applications are updated;

- a multi-step LLM-based approach for mobile GUI tests repair, developed to understand how an LLM behave in practice when used for tests repair;
- a comparison between the performance of our LLM-based repair approach and a state-of-the-art repair tool, namely Healenium-Appium.

We gathered 19 real-world Android applications, each one with a base version, a series of GUI tests written for the base version, and an updated version. For each application, we determined which tests fail when executed on the updated version and what are the causes of failure. Then, we applied our LLM-based repair approach on the broken tests. Finally, we used Healenium-Appium on the same tests to compare the results.

This research is organized as follows: in chapter 2, after a brief introduction to Android and mobile GUI testing, we explore the most promising studies regarding LLMs and testing. In chapter 3, we explain the methodology of the research, introducing the goals, the applications used, and the repair process. Finally, in chapter 4, we present the results regarding the causes of test failure and their incidence, our repair approach, and the comparison with Healenium-Appium.

Chapter 2

Background

2.1 GUI Testing of Mobile Applications

Software testing is a fundamental phase in the software development lifecycle. Software testing is the process of evaluating and verifying that a software product or application functions as intended, helping to prevent bugs and improve performance. Automated testing has become essential to meet the market demands for fast delivery of high-quality software. Testing can be performed on different levels of system abstraction, from the source code to the Graphical User Interface (GUI).

GUI-based applications allow users to interact with the underlying code through a pictorial human-machine interface. This is done through "events", such as mouse clicks, mouse drags, keyboard commands or shortcuts, etc. Today, GUI-based applications are everywhere in practice, and a well-designed GUI is often key to the success of a software application. For this reason, software developers are dedicating considerable effort to modeling, implementing and testing software systems at a GUI level of abstraction.

The growth of the mobile application industry has attracted the interest of several companies, which have begun offering support for mobile applications in most of their services. This brought the necessity for companies to launch their own applications to the market in the shortest possible time. In this scenario, one of the primary concerns is the need for a thorough testing phase, which is often performed superficially or neglected altogether, being a costly and time-consuming activity.

In this context, GUI testing plays an important role, as it allows the simulation of direct user interaction and, especially when automated, facilitates the rapid and reliable demonstration of the functional correctness of the application under test. A suite of automated tests also allows for the repeated execution of the same tests on demand, ensuring the non-regression of the application defects if the tests are successful.

The largest share of the mobile application market is controlled by Android. First released in 2008, Android is an operating system developed by a consortium of developers known as the Open Handset Alliance, led by Google, designed primarily for touchscreen-based devices, such as smartphones and tablets. Its architecture consists of several layers, including the Linux Kernel, the Hardware Abstraction Layer (HAL), native libraries, the Android runtime, the Java API framework and the user-facing apps.

The topmost layer is the user-facing apps, which are the apps that the user interacts with directly. Among them, Android comes with a set of core applications such as the home screen, phone dialer, email, and internet browser.

In order to separate the concerns of app's data, user interface and control flow, several architectural patterns have emerged, such as Model-View-Controller (MVC), Model-View-Presenter (MVP) and Model-View-ViewModel (MVVM). Google recommends the MVVM pattern, supporting it with several enhancements in its libraries. Initially, the decoupling of Android components relied on the declaration of UI layouts in XML files, which defined the static visual aspects of a component, and their inflation at runtime. These components could also be updated programmatically during their lifecycle, allowing for changes in visual aspects or internal state.

Recent recommendations from Google encourage developers to rely on the Jetpack Compose library for UI development. Jetpack Compose is a modern toolkit, consisting of a collection of Kotlin APIs, designed to simplify UI development. Based on a declarative programming model, it allows developers to define stateless components that are reusable, customizable, and easily testable.

Testing Android applications through their Graphical User Interface can be complex and time-consuming, due to their complexity and the need to replicate gesture-based interactions.

There are different criteria by which automated GUI testing of Android applications can be classified. One common classification in based on the widget location strategy. There are three main methods for identifying a widget during the execution of an application [3]:

1. Coordinate-based locators (first-generation locators): elements are identified by the coordinates relative to their position on the screen. They are considered outdated nowadays, due to the wide variety of screen sizes of Android-based devices.

2. Layout-based locators (second-generation locators): elements are identified using the value of their properties. These are the most used and widespread. They use a specific value of a layout property to retrieve the corresponding widget.

3. Image-based locators (third-generation locators): starting from a screenshot, an image recognition algorithm is used to detect the most similar screen element.

2.2 The fragility challenge

The purpose of an automated test suite is to ensure that all components of an app function correctly, with any test failure indicating a malfunction in the application. However, while this is generally true for low-level tests, GUI tests may fail due to changes in the locators of screen elements rather than actual issues with the application itself.

As an application evolves, its GUI visual appearance, internal structure, and properties often change over time. These changes may involve minor attribute modifications in specific widgets or a completely redesigned appearance on specific screens. As a result,

widget locators may change between releases, causing tests to fail, not because of a malfunction of the application, but because the widget's locator is no longer valid. Therefore, each test failure requires additional effort to understand whether the failure reflects an actual defect, and to fix the broken locators. This issue, commonly encountered in GUI testing, is referred to in the literature as fragility.

An example is illustrated in Figure 2.1, which shows the evolution of the home screen of a weather application. The 'Search' button that was in the top right corner of the screen in the old version of the application has become a Floating Action Button (FAB) in the newer version. If a test that was written for the older version of this application is executed on the updated version, it will fail, since it will be looking for the 'Search' button using the old locator resource-id="com.a5corp.weather:id/search", instead of the new locator resource-id="com.a5corp.weather:id/fab" used in the newer version of the application.

Figure 2.1. Evolution of the SimpleWeather main screens.

This means that test suites must keep up with GUI changes in the application. In this context, the growth in recent years of Large Language Models represents a promising solution for enhancing GUI test evolution and repair.

2.3 LLMs for GUI testing

Large Language Models (LLMs) are a type of artificial intelligence systems trained on a vast amount of data, making them capable of understanding and generating natural language and other types of content to perform a wide range of tasks. Their ability to understand intent, generate syntactically correct code, and reason about errors makes

them promising candidates for addressing GUI testing challenges. In recent years, many studies have been done regarding the use of LLMs in GUI test generation and GUI test migration, both for desktop and mobile applications. In this section, the most promising of these studies are explored.

2.3.1 GPTDroid

Liu et al. developed GPTDroid [5], an LLM-based system for automated GUI testing of Android applications. It follows a Q&A approach, asking the LLM to play the role of a human tester and enabling the interaction between the LLM and the application under test.

GPTDroid is realized with two nested loops.

In the outer loop it extracts the GUI context of the current page and encodes it into a prompt for the LLM, then decodes the LLM's answer into operation scripts to execute the app.

Specifically, the GUI context includes the information of the app, the GUI page currently tested, and all the widgets on the page. The first is extracted from the AndroidMaincast.xml file, while the other two are extracted from the view hierarchy file obtained via UIAutomator. The app information includes the name of the app and the name of all its activities. It facilitates the LLM to gain a general perspective about the functions of the app. The GUI page information includes the activity name of the page, all the widgets represented by the "text" field or "resource-id" field (the first non-empty one in order), and the widget position of the page. It facilitates the LLM to capture the current snapshot. Widget information provides the inherent meaning of all the widgets on the page. For each widget GPTDroid extracts "text", "hint-text", and "resource-id" field (the first non-empty one in order), "class" field, and "clickable" field. To avoid a widget having an empty textual field, the "text" of parent node widgets and sibling node widgets is also extracted.

The extracted information is organized according to linguistic patterns and prompt generation rules for inputting into the LLM.

6 linguistic patterns are used. Three patterns are used to describe the current GUI page and correspond respectively to the app information, the GUI page information and widget information. The other three patterns are designed for operation and feedback questions. For the operational questions, the required operation or input text is asked to the LLM. For the feedback question, the LLM is informed if the previous operation is not applicable, so that it can re-try.

The linguistic patterns are combined into three prompt generation rules. Test prompt is used to inform the LLM of the of the current status and query for the next operation. Feedback prompt is used to inform the LLM that an error occurred and re-try querying for the next operation. Start prompt is designed to start the testing of the app, and is only used once; it provides a global overview of the app.

After inputting the generated prompt, the LLM will generate the executive command in a natural language sentence. To overcome the challenge of mapping natural language to the app for execution, the LLM is provided with the output template, including available operations and operation primitives, which can be mapped directly to the instruction for

the app execution.

In the inner loop, a Testing Sequence Memorizer keeps the record of testing, including the set of tested functions, the testing path of activity, the set of tested activities with page visits number and the set of tested widgets of the current page with widget visits number. This helps retaining the knowledge during the testing process and understanding the functional aspects of the mobile app.

A Functionality-aware Memory Prompt has been constructed to ask the LLM what functionality is currently being tested and whether it is completed. Specifically, it consists of four patterns, related to the explored functionalities, the covered activities, the recently tested operations, and what function is currently tested. It refers to the testing sequence memorizer in each iteration to fetch the latest information.

Both the prompts for the outer loop and the inner loop are inputted into the LLM for querying the next operation.

GPTDroid uses VirtualBox and the python library pyvbox for running and controlling the Android x86 OS, Android UIAutomator for extracting the view hierarchy files, and Android Debug Bridge (ADB) for interacting with the app under testing. For the LLM, the GPT-3.5-turbo model is used.

GPTDroid has been evaluated on the apps of the Themis benchmark, which contains 20 open-source apps with 34 bugs in GitHub, and on 73 apps (with 109 bugs) selected amongst the 50 most popular apps of each category in Google Play. The baselines include 10 commonly-used and state-of-the-art automated testing techniques (Monkey, Droidbot, TimeMachine, WCTester, Stoat, Ape, Fastbot, ComboDroid, Humanoid and Q-testing) as well as 5 other baselines made of integrated versions of the above-mentioned testing tools (QTypist integrated with Droidbot and Ape, Toller integrated with Stoat, Vet integrated with WCTester and Ape).

The experimental evaluation shows that GPTDroid achieves an average activity coverage of 75% and an average code coverage of 66% across the 93 apps, while the best performing baselines achieve 57% and 55% respectively. An analysis on the average activity coverage at varying times also showed that GPTDroid achieve an higher coverage at any time, showing its efficiency on covering more activities in less time. GPTDroid also managed to detect 95 bugs (among the known ones) for the 93 apps, compared to the 66 of the best baseline. In terms of efficiency, at every time point, GPTDroid detects more bugs that the baselines.

GPTDroid was also evaluated against TimeMachine, Stoat with Toller, and Humanoid on its ability to detect new bugs. From 223 apps selected among the 50 most popular apps of each category in Google Play, GPTDroid was able to detect 53 new bugs, while the three baselines only detected 9.

2.3.2 AssistGUI

AssistGUI [4] by Gao et al. is a benchmark designed for Desktop GUI Automation. While previous similar studies where centered around web or mobile applications, AssistGUI is designed to be utilized in desktop productivity software. It spans five major categories of desktop tasks: office work, design, widget usage, system setting, and file manipulation, covering popular applications, such as Premiere Pro, After Effect, PowerPoint. This type

of apps poses a unique challenge compared to web and mobile apps because of their denser GUIs and demand for more complex operations and longer procedures.

Given an application, AssistGUI receives a user query in natural language that briefly describes a specific task, along with an instructional video as a supplement that more detailed illustrates how to complete it, and outputs a sequence of UI actions to fulfill the user's query. The task indicated by the query could include some user-customized requirements and not always align exactly with the operations shown in the video, so the model needs to modify the steps based on the instructional video.

AssistGUI introduces an Actor-Critic Embodied agent, based on LLMs, that is able to perceive the software environment, plan actions and execute them. It is made of four modules: the Planner, the GUI Parser, the Critic and the Actor.

The Planner outlines the key milestones and substasks of the task. Specifically, the LLM is first asked to extract the steps based on the subtitle of the video, then it is again asked to modify the steps according to the user query. The output is sent to the following modules.

The GUI Parser observes the GUI environment with the goal to convert an observed screenshot into a structured textual representation. Since desktop software tipically comprises a wide variety of UI elements, multiple tools are used to extract information. Specifically, metadata from the system are used for panel segmentation, an OCR model is employed to extract text from images, a pattern-matching method identifies icons, etc. The GUI information, including the meanings of UI elements and their spatial position coordinates, is represented panel by panel.

The Critic utilizes an LLM to assess the quality of the previous action by analyzing the screenshot taken before and after the execution. It outputs whether the previous action was executed correct and whether the current subtask is completed, with an explanation in case of a negative answer.

The Actor is in charge of generating actions. It is built upon an LLM. First it determines what is the current subtask based on the response of the Critic. Then, it generates an output action based on the current state of the observed sotware, the previous action taken, the current subtask and its corresponding milestone, while taking into account the Critic's feedback on the performance of the previous action.

The experimental results on AssistGUI are promising and outperforms existing methods in GUI automation for desktop applications, but also highlight the considerable challenges that remain in this field.

2.3.3 GERALLT

GERALLT [9] by Rosenbach, Heidrich, and Weinert is a system leveraging LLMs for esploratory GUI testing of real-life engineering software, with the goal of identifying unintuitive and incostistent parts of the interface. It is based on previous work by Liu et al. (GPTDroid) and Gao et al. (AssistGUI).

While prior research has primarily focused on mobile and web applications, GERALLT extends LLM-based testing techniques to real-world desktop environments. In particular, GERALLT was applied to the tool integration wizard of RCE, an open-source software platform for designing, implementing, and executing simulation toolchains, to automate

tests aimed at identifying unintuitive or overly complex GUI behavior. These types of tests were previously performed entirely manually due to the difficulty of implementing them using traditional testing approaches.

At the core of GERALLT are two components based on LLMs: one that explores GUI and execute actions, called "controller", and another that check the GUI after each action is performed, looking for unintuitive or inconsistent interactions, as well as functional errors, called "evaluator". A GUI Parser is also used to convert screenshots and metadata into a structured representation.

To mimic a human tester, GERALLT is asked to execute a loosely defined task, without the concrete steps to complete it, and to provoke unintuitive behavior while doing so. This is just like a human tester that can only rely on existing documentation and personal intuition.

GERALLT starts by taking a task as input an initializing an instance of RCE.

In each iteration, GERALLT construct a prompt for the controller, comprised of its role, the task, the documentation given by RCE, the structured representation of the GUI, the list of the possible actions, the actions previously taken and a concrete question about the next action to be executed in the GUI. After receiving the prompt, the controller outputs an action that GERALLT tries to execute on the RCE instance. The attempted action and whether it was successful is recorded in an action log.

After the execution, GERALLT construct a prompt for the evaluator, comprised of its role, the task, the executed action, the screenshots before and after the action was executed, the expected output format, and the request to determine unintuitive behavior of the GUI.

The experimental evaluation used the GPT-4o model over OpenAI's API. During nine isolated runs, each one on a fresh instance of GERALLT, in which 752 actions were performed on RCE, five unique issues with the tool integration wizard of RCE were found. The development team of RCE agreed on these issues, confirming that they constituted "blind spots" in the existing testing process.

2.3.4 Automated GUI testing for web applications by Zimmermann and Koziolek

Zimmermann and Koziolek [17] presented an approach to automated GUI testing in web applications that integrates the GPT-4 model with Selenium. The adoption of the GPT-4 model allows to bypass the hassle of fine-tuning models with pre-recorded user interaction data from human testers.

The GPT-4 model simulates user interactions. It receives the raw HTML code of the application under test, stripped of the unnecessary elements such as 'script' and 'style' code; this cleaner HTML code guides the AI model by providing an overview of current views and interaction possibilities. The Selenium WebDriver manages the interaction with the application by recording HTML snapshots of its state, identifying potential interaction targets, and tracking changes in the state of the Document Object Model (DOM) after each interaction. The updated state is reintroduced into the GPT-4 model, establishing a closed-loop feedback system.

A critical aspect of the experiment is the context size of the LLM, since it is particularly important in GUI testing to keep track of past interactions and generate meaningful new interactions. To mitigate the context size limitation of GPT-4, this approach includes the past interactions of the model and the current state of the application in every input prompt, to ensure that the LLM has full context of the whole testing history for each interaction.

The method has been employed to test a simple custom-built web calculator. The performances of this GPT-4 integrated approach has been compared to Monkey Tester and the GPT-3.5-Turbo model, consistently outperforming them, both in terms of higher coverage and faster coverage.

2.3.5 TEMdroid

TEMdroid [16] by Zhang et al. is a learning-based widget matching approach for GUI test migration. While existing migration approaches use manually defined matching functions, TEMdroid uses BERT to capture contextual information and learns a matching model to match widgets, making it capable of handling complex matching relations and adapt to diverse scenarios in real-world apps.

Given a source app, a source test case, and a target app as input, TEMdroid generates a migrated test case for the target app. TEMdroid is based on three components.

The Candidate-Widget Selection component identifies a sequence of candidate widgets from the target app for each source event or assertion in the source test case. TEMdroid extracts a sequence of source events and source assertions from a source test case, and launches the target application. TEMdroid tries to find a matched widget in the current GUI screen of the target application for each source event and assertion, according to the matching model. If TEMdroid does not find a matched widget in the current screen, a three-steps strategy is used: first, TEMdroid tries to find a target widget for the subsequent k source events and assertions, selecting the first widget that matches the source event or assertion. Second, TEMdroid tries to find a target widget in the screens directly or indirectly reached from the current GUI screen. Third, if two target widgets are found, the one with the higher similarity score is selected as the matched target widget; if only one target widget is found, this widget is deemed as the matched target widget; if no widget is found, TEMdroid did not found any matched target widget for this source event or assertion.

The Widget Matching component uses a learning-based matching model to identify matched target widgets from the candidate widgets.

Training the matching model present a challenge: the imbalance between a few positive samples, i.e., should-be-matched widget pairs, and many negative samples, i.e., should-not-be-matched widget pairs. To address this challenge, a two-stages training strategy is employed. In the first stage, a hard-negative sample miner is trained by using all the positive sample from the migration data and an equal number of randomly selected negative sample from the migration data. The miner is then applied to mine hard-negative samples, i.e., negative data that are easily misjudged as positive samples, in the migration data. In the second stage, a matching model is trained to identify the matching relations of the widgets, using all the positive samples in the migration data,

and all the mined hard-negative sample from the first stage, with a ratio of positive to negative of about 1:4.

The Event/Assertion Synthesis component synthesizes target events and assertions based on the matched widgets and the corresponding source events and assertions. The target test case is formed using the synthesized target events and assertions.

TEMdroid was evaluated on 34 real-world applications from the SemFinder dataset and the Craftdroid dataset. TEMdroid demonstrated to be effective in event matching, achieving a Top1 accuracy of 76% improving more than 17% over the baseline, and test case migration, with a F1-score of 89%, improving the baseline by 7%.

2.3.6 AutoDroid

AutoDroid [10] by Wen et al. is a task automation system for Android applications based on LLMs. The key to AutoDroid is to combine the commonsense knowledge of LLMs and app-specific knowledge through dynamic app analysis.

In the offline stage, AutoDroid obtains app-specific knowledge by randomly exploring the target app and extracting the UI Transition Graph (UTG). The UTG is then traversed to summarize the tasks that can be accomplished. During the online stage, the LLMs are continuously queried to obtain guidance on the next action.

AutoDroid employs a GUI parsing module to convert a GUI state into a simplified HTML representation that allows for a better understanding by the LLMs. The module uses five HTML tags, i.e. <button>, <checkbox>, <scroller>, <input>, and <p>, which represent respectively elements that can be clicked, checked, swiped, edited, and any other view. The properties are: ID (the order in which an element appears in the GUI tree), label (the content description that describe the function of an element), onclick (hints about the UI states that will be accessed by interacting with an element), direction (scrolling direction), checked, and value (the input text in a text box). To simplify the DOM tree, invisible elements are pruned and functionally equivalent elements are merged together, separated by
. AutoDroid also provides the LLMs with all the components from the scrolled portions of the interface. This prevents the LLMs from making blind decisions when they cannot see all the information on the interface, and eliminated the need for LLMs to provide explicit instructions for scrolling, reducing the frequency of calling the LLM and lowering the computational overhead.

AutoDroid uses exploration-based memory injection to provide information about the app to LLMs. This approach presents some challenges. The UTG cannot be directly processed by the LLM. Memory acquired through UI automation tools lacks essential information needed to directly enable task automation, like details about the specific UI elements and actions necessary to accomplish a particular task. A large number of UI screens and elements may cause to exceed the token length limit of the LLM. To overcome these challenges, AutoDroid synthesizes simulated task in the simulated task table, obtained by traversing the UTG and querying LLMs about the functions of the UI elements. The simulated task table contains an entry for each UI element in the UTG, with each entry made of three parts: simulated task, which represent the functionality, UI states, which represents the UI states traversed from the initial state, and UI elements, which represent all the elements clicked from the initial UI state. Another table, the

UI function table, obtained by querying the LLM, is used to provide a summary of the functionality associated with each UI state in the UTG.

Due to the token length limit of LLMs, only the most relevant UI information is incorporated into the prompt. The importance of an UI element is determined based on the similarity between its simulated task and the current user task.

AutoDroid can also use smaller local LLMs, like Vicuna-7B, as a cost-effective alternative to larger on-cloud LLMs, like GPT-3.5 or GPT-4. However, the reasoning capabilities of these smaller LLMs is noticeably weaker compared to the on-cloud alternatives. Fine-tuning using domain-specific data has been found to be an effective way to improve smaller LLMs abilities. This is done by generating question-answer pairs from the sequence of UI states and UI elements for a specific task in the simulated task table. From a simulated task S, with associated UI states U1, U2, ..., Uk and UI elements e1, e2, ..., ek, k data pairs (qi, ai) are generated, where qi is a prompt generated based on the task S, previous UI actions A1, A2, ..., Ak, and the current state Ui, while ai is the description of the action Ai. However, this answer only contains target element, action type, and value, lacking detailed information or context. For this reason, a larger in-cloud LLM is used to provide the reason on why the action Ai is chosen to complete the task S.

Some actions are considered risky, since they may alter local or server data, or cannot be undone once performed. These actions require user confirmation before being executed. Also, a privacy filter is added to mask private information in queries.

2.3.7 MACdroid

MACdroid [15] by Zhang et al. is an approach to test migration in which LLMs are used first to extract a general test logic from multiple existing test cases, and then to generate concrete test cases for a target app based on the extracted test logic. This new approach (abstraction-concretization paradigm) differs from existing migration paradigms which typically focus on widget-mapping from source apps to target apps.

MACdroid is based on two components. The Abstractor component aims to extract a general Test Logic (TL) for a target functionality based on multiple source test cases, while the Concretizer component aims to generate a target test case for the specified functionality of the target app.

The Abstractor includes three modules: TL Extraction, TL Summarization, and TL Validation.

The TL Extraction module extracts a sequence of structured test steps from each individual test case. This is done to facilitate comprehension by the LLM. To enhance the generalizability of the extracted logic, app-specific details are removed. There are two types of test steps: events and assertions. An event type test step is represented as a tuple with four elements: event type, widget, action to the widget, and optional input value. An assertion type test step is represented as a tuple with three elements: assertion type, widget, and a condition to check the widget. Since a widget typically has several attributes, only the text, content-desc and resource-id attributes are used to represent it accurately and concisely.

The TL Summarization module relies on the capabilities to understand and summarize information by LLMs to summarize a general test logic from the extracted individual test logics from the TL Extraction module. Specifically, a prompt with four parts (task description, input object, output example, and output requirements) was designed to interact with the LLM. The task description provides an overview guidance for the LLM to understand the aim of the module. The input object contains the individual test logics extracted by the TL Extraction module. The output example is included, since one-shot prompting has demonstrated superior performance compared to zero-shot setups. The output requirements guide the LLM in effectively summarize the general test logic. One requirement aims to enhance the comprehensiveness and generalizability of the general test logic, while the other focuses on including all the important information while remaining concise, making it clear while remaining within the word limit of LLMs inputs.

The TL Validation module employs a rule-based method to ensure the quality of the summarized test logic, since the LLMs outputs may be irrelevant or fabricated with input data even when the output requirements are included. Three rules (irrelevant step, missing step, and ambiguous action) are designed to identify common issues related to the TL Summarization module, and for each issue a feedback for the TL Summarization module is generated to re-summarize the general test logic accordingly.

The general test logic summarized by the Abstractor cannot be used directly as a test case. The Concretizer aims to generate concrete test cases based on the events and assertions in the target app. This can be challenging due to the large number of candidates in the target app. A priority strategy is used, so that the LLM can select events and assertions from a smaller set, which are more relevant to test the target functionality. This privileged set of events and assertions is obtained by using one of the existing migration approaches.

The Concretizer includes three key modules: the Event/Assertion Matching module, the Event/Assertion Completion module, and the Test Validation module.

The Event/Assertion Matching module uses an LLM to match the events and assertions in the privileged set with the test steps in the general test logic. The prompt for the LLM is similar to the one for the TL Summarization module, including task description, input object, output example and output requirement. The main differences reside in the input object and output requirement. The input object includes the test step to match and privileged events and assertions that have not been matched by the preceding test steps. The output requirement emphasizes the need for type alignment (events with events, assertions with assertions) and that not every test step needs to match an event or assertion of the privileged events and assertions because these privileged events and assertions may not fully cover the target functionality.

The Event/Assertion Completion module selects events and assertions in the target app when the events and assertions from the privileged set cannot be matched to a given test step. The key parts are state description, event selection and assertion generation.

The state description allows the LLM to understand the semantics of GUI widgets and the actions that these widgets implement in the target app. The GUI state is converted into a natural language description to aid the LLM's understanding. Widgets are

represented by their semantics (text, content-desc, and resource-id) and the associated actions, and are organized according to their spacial locations.

In the event selection, given an event type test step from the general test logic and the current GUI state, MACdroid selects the appropriate events in the target app to complete this step with the LLM. MACdroid first generates a prompt for the LLM that includes an event step and the current state description. The LLM returns an event ID. MACdroid executes the corresponding event and performs a test validation on the updated GUI state. When the test validation passes, the event is incorporated into the GUI test case. The step is skipped if the LLM cannot select an appropriate event within a certain number of selected events.

An assertion is made of a widget and a condition. In GUI testing there are usually two types of condition, one that checks the presence of a widget in the current state, and one that checks the disappearance of a widget in the current state. For the first type of condition, the LLM selects the appropriate widget in the current state. For the second type of condition, the LLM tries to identify the widget from a previous state. The step is skipped if the LLM cannot select a widget after trying a certain number of widgets.

The Test Validation module uses four rules to identify possible inaccuracies of the LLM. For the Event/Assertion Matching module the common issues are incorrect type, i.e., matching an assertion with an event or viceversa, and irrelevant matching, i.e., including events and assertions that do not appear in the privileged set. For the Event/Assertion Completion module the two common issues are completion checking, i.e., the LLM may continue to select events and assertion for a test step without termination, and incorrect format, i.e., the output may not adhere with the required format.

MACdroid has been evaluated against TEMdroid and AutoDroid, on the FrUITeR and Lin datasets. When using GPT-3.5, MACdroid was able to generate tests that successfully test the 64% of the target functionalities on the FrUITeR dataset and 75% on the Lin dataset, improving the baselines by 191% and 42% respectively.

2.3.8 ScenGen

ScenGen [14] by Yu et al. is a scenario-based approach to automated GUI testing driven by multi-modal LLMs (MLLMs). Adopting a scenario-aware strategy allows to accurately identify user behavior patterns and understand the GUI widgets and layout from the perspective of human testers, ensuring comprehensive mobile GUI testing.

ScenGen views the iterative process in five stages, i.e. observation, decision-making, execution, result inspection and feedback, and recording. ScenGen consists of multiple LLM agents, with different agents responsible for different stages of the process. These agents include Observer, Decider, Executor, Supervisor, Recorder, and context memory.

The stateless nature of LLMs makes it necessary to manage the context supporting the test generation process. The memory of the LLM agents is divided in three parts: the Long-term memory includes information about devices, apps, and scenarios; the Working Memory contains the target scenario, the app and device under test, the executed test operations, and the interactive dialogues with the LLM. The Working Memory is constantly updated for individual testing and persists throughout the entire test generation task; the current app state, the app state before the most recent operation, and the results of GUI

widget recognition are stored in the Short-term Memory.

The Observer detects the GUI widgets and their attributes, providing precise data input for the next steps. From the GUI images, computer vision algorithms extract graphical widgets, while OCR algorithms identify textual widgets. The final widget recognition results are obtained by integrating and filtering the results of the graphical and textual recognition. ScenGen adopts a vision-based approach to identify widgets, instead of extracting information directly from the app layout files. This is done because widgets might be unaccessible from GUI layout files, while identifying widgets from app screenshots can achieve the effect of "what you see is what you get". After obtaining the final widget recognition results, the widgets are visualized by drawing bounding boxes on the original screenshots based on the coordinates of each widget. Each widget is also provided an ID. This visual representation, the original GUI image and the set of identified widgets are stored in the short-term memory.

The Decider generates detailed plans for the next step according to the current GUI state. The decision-making is an iterative process. It starts with a prompt that contains a brief introduction to the target scenario. Then, the current state is continuously evaluated to determine the appropriate next action. This is repeated until the target scenario is fully covered. The next action is determined based on the requirements of the target scenario, the current app state, the executed actions, the set of action types, and the collection of widgets on the current page. While LLMs are able to make logically correct decisions, they often struggle to precisely identify the specific location of widgets on a page. On the other hand, traditional visual algorithms can provide the precise pixel coordinates of widgets. ScenGen combines these two methods by dividing the action decision process in two parts: logical decision-making and widget localization.

In the logical decision-making process the LLM determines the next action based on the GUI images, the set of action types, the target scenario, and the executed actions. The output will include the name of the action type, a description of the action intent, a description of the target widget, eventual text to be entered for input actions, and eventual scroll direction for scroll actions. The widget location process aims to identify the precise pixel location of the target widget.

ScenGen employs a self-correction mechanism, in case the most recent decision operation does not pass the state transition verification. It involves two main steps: the cause analysis, in which the potential issues are identified, and the decision adjustment, in which decisions are modified based on the causes.

The Executor interprets and executes the planned actions. First, it parses the action decision results, and then it implements the specific actions through system interfaces, e.g. Android ADB.

The Recorder is responsible for saving execution execution instructions and contextual information of the testing process to the context memory. It records the execution results into the working memory and also monitors the app runtime logs to identify eventual bugs during the exploration.

The Supervisor check the executed operations and provides feedback based on the inspection results.

ScenGen has been evaluated on 92 apps, covering 10 different scenarios.

The effectiveness of the Decider has been evaluated in terms of accuracy, both before and after the self-correction mechanism. The initial decision accuracy ranges from 90.14% to 100%, while the final accuracy, i.e. after the intervention of the self-correction mechanism, ranges from 92.96% to 100%. The average accuracy is 96.08% and 97.06% respectively.

Similar metrics have been use to evaluate the widget location effectiveness. The initial accuracy ranges from 54.55% to 97.14%, while the final accuracy ranges from 94.29% to 100%, showing great improvements thanks to the self-correction mechanism. The average accuracy is 82.69% and 97.76% respectively.

Regarding the test generation capability, 99 experiments showed ScenGen achieving a coverage rate of 86.87% and a success rate of 84.85%.

For the evaluation of the bug detection capability, Monkey, Stoat, PIRLTest and ScenTest were used as baselines. The first three approaches are representative of traditional automated testing approaches, while ScenTest is the first state-of-art scenario-based GUI testing approach. Withing the target testing scenarios, ScenGen was the one detecting more bugs, and all the bugs detected by the baselines were also detected by ScenGen.

2.3.9 Trident

Trident [6] by Liu et al. is a vision-driven, multi-agent collaborative automated GUI testing approach for detecting non-crash functional bugs, based on MLLMs. It is composed of three agents: Explorer, Monitor, and Detector.

The Explorer agent navigates through the app and visually inspects GUI pages, capturing view hierarchies and screenshots. Instead of relying on view hierarchy files to extract attribute information of app widgets, which can be inefficient as it may lose the structural semantics of the GUI when converting screenshots to text, Trident is able to use screenshot images as input directly, thanks to the capabilities of MLLMs. However, relying only on images can be limiting, so a method is used to align visual and textual information to enhance the MLLM's understanding of the GUI page semantics. For the text, the information extracted are app information, obtained from the AndroidManifest.xml file, and page and widget information, extracted from the view hierarchy file of the current GUI page using UIAutomator. The app information, such as the name of the app and the activities, provide the LLM with a general understanding of the app's functionalities. The page and widget information, such as "text", "hint-text", "resource-id", "class", "clickable", "long-clickable", "checkable", "scroll", and "bounds", facilitate the LLM in suggesting appropriate actions related to the widgets. For the image, a screenshot annotation method is used. Bounding boxes are drawn around each actionable widget, with different colors indicating different action types. Each widget is labeled with a number, ordered from top to bottom and from left to right, marking only the first widget in each row to reduce the overlapping annotations and letting the MLLM infer the others. A "legend of image" is also included in the prompt to facilitate the MLLM's understanding.

Each annotated widget in the screenshot is associated with the extracted GUI text information. Due to input length limits of the MLLM, detailed alignment information is omitted in prompt related to text and image information. It is instead used in a chain-of-thought mechanism, to verify whether the MLLM suggests interactive widgets according

to the correct logical process. The widget ID from the screenshot and the corresponding text are requested when querying general actions or text input. If the sources do not align, a potential error in the MLLM's decision making is inferred, and the MLLM is prompted to retry.

The Explorer prompt is composed of the GUI information and screenshots, the testing history provided by the Monitor agent, and the feedback from the Detector agent. The MLLM is queried about the interactive widget and the action to take.

The Monitor agent is the central control unit of Trident. It supervises the testing process, records the exploration history, and triggers the Detector at appropriate times. It continuously records the exploration sequence of the app, including view hierarchies and screenshots of the explored GUI pages, generating two views of testing history: an higher-level text view of the explored functionalities, and a detailed image view of testing records. These views are provided to the Explorer during the testing process, as a way to enhance the understanding of the testing process, while also addressing the token limit of LLM's inputs. During the monitoring process, the Monitor determines whether the current functionality is still under exploration. If so, it instruct the Explorer to continue, otherwise it pauses the Explorer and triggers the Detector to analyze potential bugs in the previously tested functionality.

The Monitor prompt is composed of the testing recording of the current function and the whole testing history. The MLLM is queried about the GUI page description and the currently tested function with its status.

The Detector agent identifies potential functional bugs by analyzing the logical transitions and interactions within the GUI pages. A functionality-driven chain-of-thought mechanism is employed to uncover potential logical inconsistencies for bug detection, analyzing the exploring sequence to uncover potential bug-triggering operations, or unusual or abnormal operations that might influence the exploration performance. The results are fed back to the Explorer to enhance the likelihood to uncover bugs and address any anomalies detected during the process.

Since it can be difficult for MLLMs to perform as well on domain-specific tasks such as this one, the MLLM is provided with examples to demonstrate what the instruction is. A basic dataset of examples of non-crash functional bugs has been built from issue reports of open source apps. Each example comprises a bug description, a bug screenshot, and a natural language description of the bug reproduction path. An excessive amount of examples might mislead the LLM and cause a decline in performances. To address this issue, a retrieval-based example method is used to choose the most suitable examples for the current GUI page. The examples are chosen based on the similarities between the bug's page description and the activity name of the tested page. Word2Vec is used to encode context information and calculate the cosine similarity. An empirical evaluation suggests that the performance increase with the number of examples, peaking with 4 examples and declining after that.

The Detector prompt is composed of bug examples, the functionality-driven chain-of-thought analyzing the functionality being tested, and the real exploration sequence in the form of screenshots. The MLLM is queried about bug detection and possible bug or exception paths.

The testing coverage and bug detection performance of Trident has been evaluated on three datasets: one is from the Odin and RegDroid datasets, one is from 83 apps selected among the 50 most popular apps of each category from Google Play, and one is crafted through bug injection on the second dataset, to avoid data leakage issues caused by the use of MLLMs. These datasets accounts for a total of 590 non-crash bugs. The baselines are the popular automated GUI testing tools Monkey, APE, Fastbot, Humanoid and GPTDroid, paired with the mobile app display issue detection techniques OwlEyes, NightHawk, DVermin, and DiffDroid. Six baselines of logical bug detection for mobile apps are also used: iFixdataloss, SetDroid, Genie, Odin, Android Lint, and LiveDroid. AppAgent and GPT-4 are also used as MLLM-based baselines. Trident is able to achieve an average activity coverage between 57% and 65%, and an average code coverage between 55% and 62%. This is between 21% to 27% and between 28% and 32% better than the best baseline (GPTDroid).

Trident has also been evaluated on its capability of detecting new bugs. This evaluation uses 187 popular apps as a dataset. Among these apps, Trident was able to detect 43 bugs in 42 apps never discovered before, all of them fixed/confirmed by the developers.

2.3.10 DroidAgent

DroidAgent [13] by Yoon et al. is an autonomous Android GUI testing agent, for semantic, intent-driven GUI testing automation, based on LLMs. DroidAgent generates natural language descriptions of specific tasks that can be achieved using the given App Under Testing (AUT), and then tries to interact with the GUI of the AUT with the intent of achieving those tasks. If the interaction is successful, DroidAgent will produce a test case script for the specific task. The end result is both a natural language description of the task and an executable test script.

DroidAgent comprises four main LLM-based agents: Planner, Actor, Observer, and Reflector. The Actor and Observer try to perform tasks, which are planned by the Planner and later reflected upon by the Reflector. It also includes three different memory modules (short-term, long-term, and spatial memory), with two modules, Task Retriever and Widget Retriever, in charge of extracting information from memory and format it for the agents.

The task generation by the Planner is achieved by combining information from three different sources into a prompt for the LLM agent: high-level task history, which is extracted from the long-term memory and includes the textual summaries of the last N tasks and the M most relevant task knowledge; total and visited activities, which is extracted from the spatial memory; initial knowledge, which defines a virtual persona with an ultimate goal to be achieved within the app.

The Actor is in charge of choosing the next action. It receives a set of action types (such as "touch" or "set_text") and a list of widgets in the current screen, instead of a combination of the two. This helps reducing the number of tokens in the LLM prompt. It also receives information about the prior actions, the most recent response of the application, and the critique of the recent progress. The Actor selects an action type and the widget to apply the action to. The Actor also receives the action types "wait", "back", and "end_task", which are not directly derived from the widget of the page. Since LLMs

can quite effectively detect loading screen, the Actor can decide when to wait. "back" is for navigating back. "end_task" let the Actor conclude the task before the fixed max action limit.

Within DroidAgent, screens are represented with a JSON textual representation. The Observer summarize the outcome of an action, based on the difference between screens before and after the action was taken. The Actor is informed about the outcome of the action.

A "self-critique" component is incorporated into the Actor. To avoid the Actor going down on a wrong path, the self-critique element periodically generates feedback, based on the execution history up to that point and the current GUI state description. The prompt asks for a review on the task execution history and a suggested workaround plan if the Actor appears to be struggling. While the main conversation for the next action uses the GPT-3.5 model, the self-critique component adopts the more advanced model GPT-4.

The Reflector is activated when a task execution round finishes, to create a concise description of the result of trying to perform the task. The input for the Reflector component is the entire task execution history, including self-critique and all observations from the working memory, the current GUI state, and the ultimate goal. The Reflector is instructed to "derive memorable reflections to help planning next tasks and to be more effective to achieve the ultimate goal". This process helps avoiding that the overall system forgets useful knowledge acquired during task execution.

The long-term memory contains the history of performed tasks. The short-term working memory includes the current task's execution history, and it is cleared after each planning step. The spatial memory contains specific widget knowledge, to let the agent remember observation after interacting with a widget for future interactions.

DroidAgent has been evaluated on 15 apps. 10 apps are chosen from the 23 apps of the Themis dataset, while the other 5 are selected from FDroid. Monkey, DroidBot, Humanoid, and GPTDroid are chosen as the baseline. As evaluation metrics, screen coverage, with a specific focus on activity coverage, and "feature coverage" are employed. Regarding the LLMs, the Actor and Observer use the GPT-3.5 model with extended 16K context length, the summarization process of the widget retriever uses the standard GPT-3.5 model with 4K context length, while the Planner, Reflector, and self-critique module employ the GPT-4 model.

DroidAgent achieves, on average, an activity coverage of 60.7%, compared to the 51.4% of the best baseline, Humanoid. DroidAgent finds more difficult to navigate certain types of AUTs, where widgets do not have textual properties, or a single view contains different interactable subregions. DroidAgent also covers more features compared to Humanoid (on average: 13.9 vs 7.3), even when Humanoid has higher activity coverage, demonstrating that DroidAgent tends to engage in more meaningful interactions. At last, DroidAgent managed to find five crashes, while Humanoid only found four.

2.3.11 AlphaRepair

Xia et al. proposed AlphaRepair [11], the first cloze-style (or infilling-style) Automatic Program Repair (APR) approach that leverages on CodeBERT, a large pre-trained code

model, for APR without any fine-tuning/restraining on historical bug fixes.

APR techniques usually follow the Generate and Validate (G&V) approach, which consists in generating candidate patches to replace potential buggy locations. The patches that successfully pass the test are called plausible patches and are further inspected by the developers. Recently, researchers have started utilizing learning-based APR techniques, that leverage on Machine Learning or Deep Learning to generate patches. These techniques proved to be effective APR approaches, but they face some issues. The first issue is the quality and the quantity of training data: the training or fine-tuning of model is done using historical bug fixes, i.e., pairs of buggy and patched code. This data is usually obtained from the bug-fixing commits of open-source projects, which can also include unrelated edits. For this reason models are usually trained only on commits with few lines of changes, limiting the amount of training data. The second issue is the context representation, which provides crucial information. Current tools uses the context together with the buggy code, making it challenging for the model to distinguish the patch location within the context.

AlphaRepair distinguish itself from other learning-based APR techniques by directly learning how the correct code should look like, instead of turning buggy code into fixed code. The problem is viewed as a cloze task, in which the buggy code is replaced with a mask and the model replace the mask with the correct code. This approach does not need pairs of buggy and fixed code, as it directly makes use of the existing pre-trained model to generate code repairs. AlphaRepair does not require additional retraining or fine-tuning, since Masked Language Model training is done as part of the pre-training task in CodeBERT. This pre-training consist of giving a sequence of tokens as input, with a random set of tokens replaced with masks; the goal is to output the original tokens that have been masked out. The AlphaRepair approach is articulated in five steps:

1. Input processing: starting from a buggy project, the buggy line and the context are separated. The context before and after the buggy line is tokenized, with the mask in the middle. The buggy line is also provided as a comment.

2. Mask generation: multiple mask lines are generated using different templates. Each mask line replaces a buggy line and is tokenized together with the surrounding context as input for CodeBERT.

3. Patch generation: CodeBERT is queried iteratively to generate candidate patches that replace the mask line.

4. Patch re-ranking: CodeBERT is used again to rank the generated patches.

5. Patch validation: the candidate patches are validated against the test suite. The list of plausible patches is then given to the developers to examine.

AlphaRepair has been evaluated on Defects4J, outperforming state-of-the-art techniques. On a newer version of Defects4J (2.0) AlphaRepair showed even better performance, with 3.3x more fixes than the best baseline.

2.4 State of the art approach for repair

In recent years, many traditional approaches to mobile GUI test repair have been proposed, that do not leverage on LLMs.

METER [8] (Pan et al., 2020) is an approach to automated mobile GUI test repair that leverages on computer vision techniques to infer changes in the GUIs between two different versions of an application. METER only relies on screenshots to repair test scripts, thus making it applicable to apps targeting both open and closed source mobile platforms. The METER approach works as follows: given the base version and the updated version of an app, and a group of test scripts for the base version of the app, METER first runs each input test script on the base version of the app in order to record the intended behaviors of the tests; METER then checks if the intended behavior of a test is preserved when executed on the updated version of the app, via screen matching; in the case of a negative answer, the test is considered broken, and a sequence of test actions to replace it is constructed.

GUIDER [12] (Xu et al., 2021) is an approach to automated repair of GUI test scripts for Android applications. It uses both structural and visual information of widgets to better understand the evolution of widgets from the base version of the app to the updated version. Given the base version of an Android app, its test scripts, and the updated version of the same app, GUIDER first records the intended behavior of each test script by running it on the base version of the app; then, for each test action, GUIDER checks if it preserves its behavior when executed on the updated version of the app; in the case of a negative answer, GUIDER construct a replacement for the next one or two test actions.

COSER [1] (Cao et al., 2024) is an approach to mobile GUI test repair that utilizes both external semantic information from the GUI elements and internal semantic information from the source code. The information extracted from the source code helps better reflecting the intended behavior of the GUI elements. COSER receives as input the base version of an application, a test script for it, and the updated version of the app; it outputs the repaired test script for the updated version of the app. COSER comprises four main components: the Script Executor parses the input test script into an internal representation, and executes the script to provide information to the subsequent stages; the Semantic Extractor extracts both the internal and external semantic information of each element of the GUI; the Transition Analyzer extracts a static transition graph from the source code, refining it using dynamic information; the Scripts Repairer incorporates all the information extracted to repair the input test script.

2.4.1 Healenium

Healenium is an open-source library for automated testing. It leverages machine learning to improve the reliability and resilience of tests. It is able to automatically detect and heal test failures caused by changes in the UI. Advanced algorithms are used to analyze test results, determine the root cause of failures, and make the necessary adjustment for the test to continue functioning correctly. Among many other supported testing frameworks, Healenium can be integrated in Appium, to help testing mobile applications as well.

Given the base version of an application and its tests, Healenium works as follow: first, tests are run successfully on the base version of the application, allowing Healenium to save the successful locators as a baseline for the next test execution. When the UI changes and a test script fails because an element is not found, Healenium catches NoSuchElement exception, and triggers its algorithms to generate a list of healed locators; a score is

associated to each locator, and the locator with the highest score is taken to replace the old broken one. At the end of the test run, Healenium generates the report with all detailed information about healed locator, a screenshot, and a feedback button on healing success.

Chapter 3

Methodology

The goal of this research is to analyze the incidence and the causes for which GUI tests for Android applications become outdated and break as the applications evolve. Then we want to understand the effectiveness of an LLM-based approach in repairing the outdated tests, compared to more traditional approaches.

3.1 Research Questions

The Research Questions (RQs) that drive this research are the following:

RQ1: What is the incidence and what are the causes of Android GUI tests failure?

RQ2: What is the performance of an LLM-based approach in Android GUI tests repair?

RQ3: How does our LLM-based approach for Android GUI tests repair compare to the Healenium-Appium baseline?

For this research we gathered a set of real world Android applications. For each application we have a base version, its GUI tests, and an updated version.

To answer RQ1, we firstly executed the tests of each application on the base version, to understand their behavior, and verify that they could run without issues in our environment. Then, each test was executed on the updated versions of the applications. For each test we determined whether it passes on the updated version or not, and for all the failures we carefully annotated where the test fails and what are the causes. Since a test can have multiple causes of failure, but only the first one is evident from the execution, a further analysis was done, with the aid of Appium Inspector, to determine all the possible causes of failure for the tests.

For RQ2, we designed an LLM-based approach for mobile GUI tests repair. This approach is applied to all the tests whose execution failed on the updated version of their application, with the aim to generate a repaired version of each test, i.e. a test that passes when executed on the updated version of its application, while also maintaining its intended behavior in testing widgets and functionalities. The repair process in described in details in section 3.4.

Finally, in RQ3 we want to compare the performance of our LLM-based approach for mobile GUI tests repair to the repair approach of Healenium-Appium. Similarly to RQ2, the Healenium-Appium repair approach was applied to each test whose execution failed on the updated version of its application. The results are annotated and compared to the ones of the LLM-based approach.

3.2 Project Selection

The set of applications used in this research comes from the applications used in ExtRep [7] (Long et al., 2025; https://github.com/anonymousproject25/ExtRep). These applications are all available in their GitHub repository; each application comes with the APK files for both the base version and the updated version, together with the GUI tests written in Appium for the base versions; in total the ExtRep dataset includes 40 applications with 112 tests. For various reasons, some applications from this dataset had to be excluded: some applications could not be installed on the emulator in use; one application had visualization problems on Appium Inspector, which hindered an accurate analysis of the application; on one application, Appium tests were too slow to be executed in a reasonable time; some applications did not have any tests that needed to be repaired. Since we noticed that not all the applications were tested thoroughly by the tests that were already written, we added some new tests. In the end, 18 applications were used from the ExtRep dataset, with 87 tests.

In addition, we also used an open source application, Now in Android, for which we wrote 26 Appium tests that follows the logic of the GUI tests on the application's repository on GitHub.

The details of the applications used in this research are listed in Table 3.1.

3.3 Operational Settings

This research utilized GitHub Copilot, powered by the model Claude Sonnet 3.7 Thinking (Anthropic, 2025), as the LLM of choice. The web version of the tool was used (https://github.com/copilot), instead of the desktop one integrated in Visual Studio Code, for its ability to support images.

Visual Studio Code was used as the IDE for all the ExtRep tests. Android Studio Meerkat (version 2024.3.1) was used to run Now in Android, from the source code, and its tests.

The applications were deployed on two Android emulators provided by Android Studio: the Google Nexus 5X with Android 6.0 is used for all the applications from the ExtRep dataset, while the device Pixel_3a_API_34_extension_level_7_x86_64 with Android 14.0 is used for Now in Android.

Appium Inspector was used to analyze the applications under test. Appium Inspector is a GUI assistant tool for Appium that provides a visual inspection of the application under test. It shows the screenshot of the current page of the application, along with its page source, and it includes various features to interact with the app. Appium Inspector was also used to extract the view hierarchy files and take screenshots of the applications.

Application	Base Version	Updated Version	# of tests
AnkiDroid	v2.6	v2.13.0	7
AntennaPod	v2.1.3	v2.5.0	5
AppLaunch	v2.0.0	v3.1.1	5
BookDash	v2.8.0	v2.9.2	3
Counter	v13	v18	6
Currency	v1.0	v1.35	5
DIDA	v5.7.0.0	v5.8.8.1	8
DNS66	v0.4.1	v0.6.8	6
DumbphoneAssistant	v0.4	v0.5	2
Fwknop	v1.0.0	v1.2.3	2
Mgit	v1.5.1	v1.6.1	2
Now in Android	v0.0.5	main branch	26
OpenBikeSharing	v1.0	v1.10.0	5
RedReader	v1.14.0	v1.23.1	6
Sgit	v1.3.0	v1.3.2	2
SimpleCalculator	v2.0.0	v3.1.2	3
SimpleWeather	v4.1.0	v5.3.2	3
SoundRecorder	v1.2.5	v1.3.0	4
SunTimesWidget	v0.11.0	v0.15.13	13

Table 3.1. List of the applications used in the research.

The Android Debug Bridge (ADB) was used to install the applications on the emulated devices from the APK files.

3.4 Repair Process

The existing tests for the applications from ExtRep were written with Appium 1. Since Appium 1 is deprecated and no longer supported, we dedicated some time to rewrite these tests with Appium 2, which is the current and actively supported version. The new tests added by us are written directly with Appium 2.

Now in Android did not have existing Appium tests, but instead it had a series of GUI tests already written using the Jetpack Compose UI Test framework. Starting from these tests, we used the same logic to write 26 Appium tests for Now in Android.

The tests for the applications from ExtRep are all written in Python, while the tests for Now in Android are written in Java.

After setting up all the applications and their tests, the repair process starts, according to the following steps:

1) Each test is first executed on the base version of its application. This was done to understand the behavior of the tests, since they were written by someone else, and also to verify that they were all valid tests that pass on the application they were written for.

2) Then the tests are executed on the updated version of their application, to determine which ones are broken and need to be repaired. A test is considered broken if its execution on the updated version of the application fails, or if it does not keep its intended behavior when it runs on the updated version of the application. The causes of failure for each test are carefully annotated.

3) Now we try to take advantage of the LLM to repair the broken tests. Each test is repaired individually. For each test the LLM is provided with the file that contains the current test under repair and the view hierarchy files in XML format of the GUI screens tested by the test, both for the base version and the updated version of the application. The prompt for the LLM uses the same format for every test. The prompt used in this stage is as follow:

```
"The file <file_name> contains an Appium test written for the older
version (referred as v1) of an Android application. Your goal is to
repair the test for the newer version of the application (referred as
v2). You are provided with view hierarchy files of the tested screen,
both for v1 and v2. Screens v1: <list_of_screens>. Screens v2:
<list_of_screens_v2>."
```

4) The LLM outputs a repaired version of the test. The new test is then executed on the updated version of the application. If the test passes, while maintaining its intended behavior, the test repair is considered successful. Otherwise, the LLM is asked again to repair the test. In this second interaction the LLM is provided with all the files of the first interaction, together with the failed repaired test generated by the LLM in the first interaction; optionally, the screenshots of the most relevant screens are also included if deemed necessary. The prompt includes the eventual error log from the execution of the failed repaired test and/or a description of the issue with the test, together with a brief request to repair the test. The LLM outputs another repaired version of the test. As before, the test is considered repaired if it passes while maintaining its intended behavior when executed on the updated version of the application. If the issue persists after this second interaction, the repair is considered failed. If the issue is solved, but another issue with the test arises, this step is repeated.

Figure 3.1 illustrates the process flowchart of our LLM-base mobile GUI tests repair process.

3.5 Repair Examples

In this section two examples are provided to better explain the repair process, but also to show the kind of issues that can be encountered. Both the examples refer to tests written for the application SuntimesWidget, an open-source application that displays sunlight and moonlight times for a given location, including sunrise and sunset, twilights, solstices and equinoxes, moonrise and moonset, moon phases and illumination; the application also provides configurable widgets for the device's home screen.

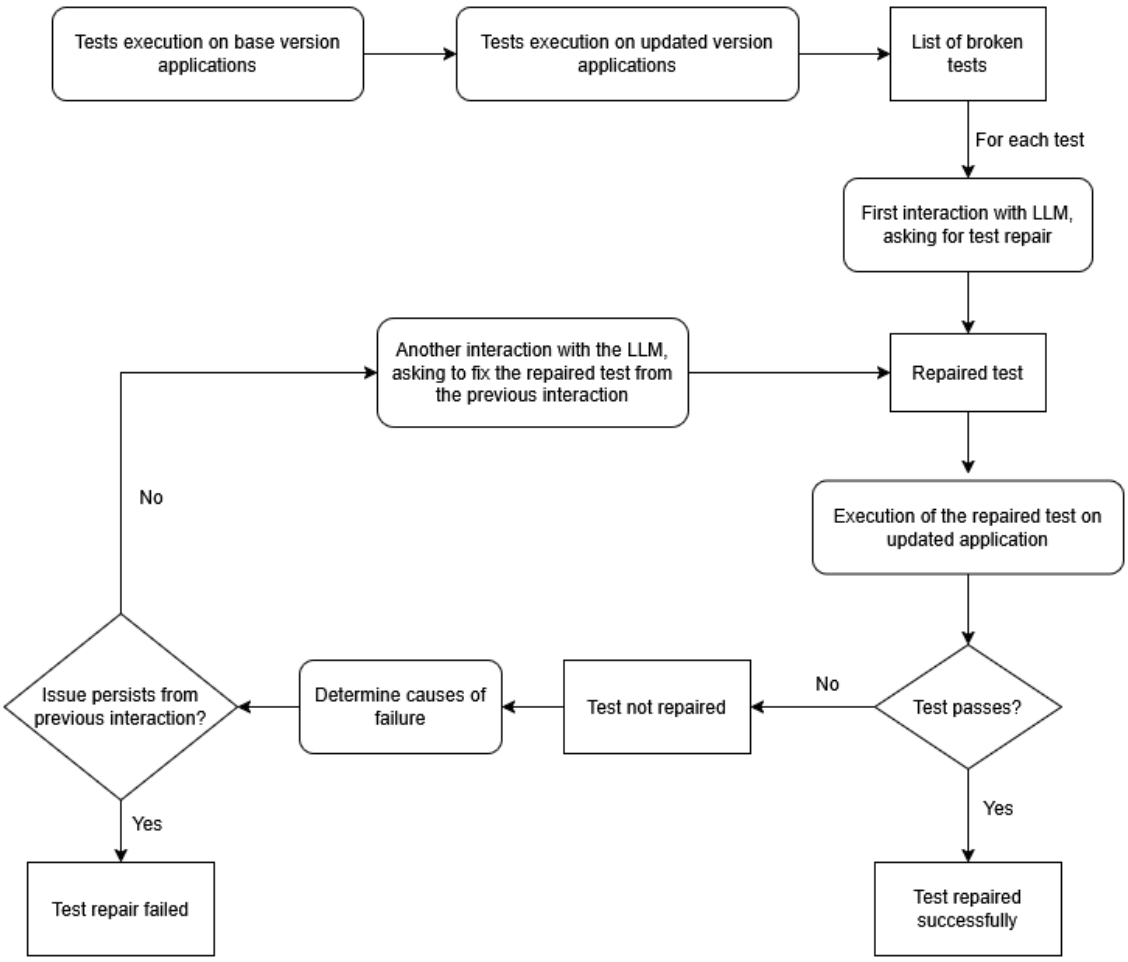

Figure 3.1. Process flowchart of our LLM-based tests repair process.

The first example is a successful repair of the test *test-8* of the application SuntimesWidget. On the base version of the application, the test first clicks the 'more options' button and selects 'Set Alarm', making the Set Alarm pop-up appear on screen; after that, the dropdown menu is opened and the option 'solar noon' is selected; the test then ends by clicking the 'schedule' button to set the alarm for the selected time.

When the test is executed on the updated version of the application, two issues emerge: the first is that, after the Set Alarm pop-up is opened, the dropdown menu is not immediately accessible, but it is necessary to click on the 'event' tab first; the second issue is that the 'schedule' button, which was previously located using its *text* property, has become a green check button, no longer accessible with the older locator. Figure 3.2 shows the differences between the two versions of the application.

The LLM is provided with the XML files of the main screen, the main screen with 'more options' opened, the Set Alarm screen, and the Set Alarm screen with the dropdown menu opened for the base version, while for the updated version it is provided with the

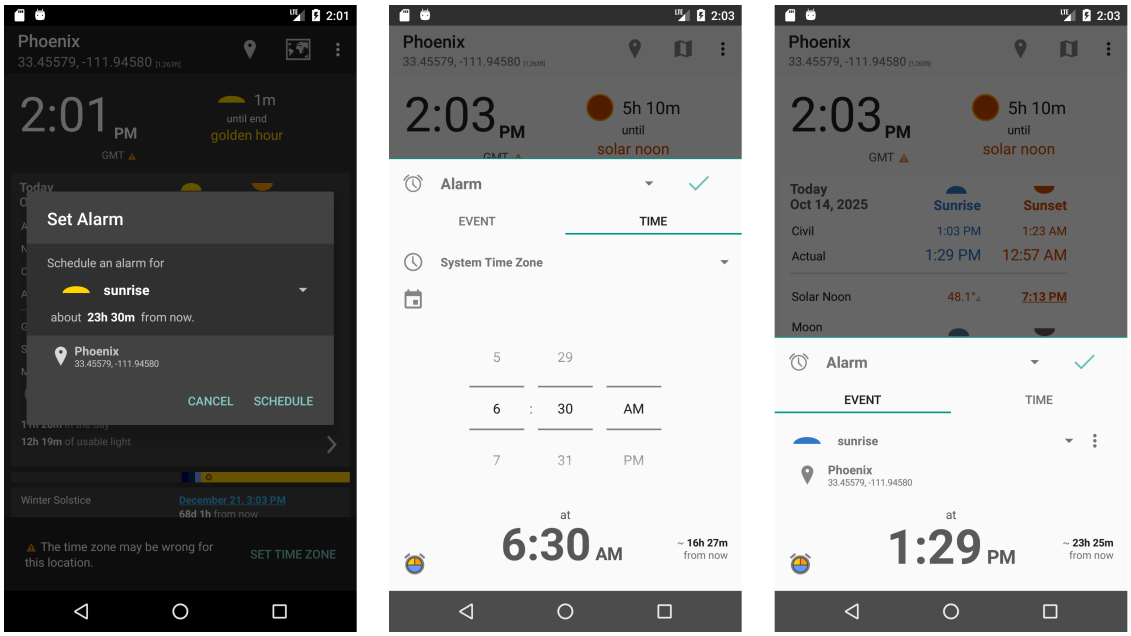

Figure 3.2. Differences between the Set Alarm screens on the base version (left screen) and the updated version (center and right screens).

main screen, the main screen with 'more options' opened, the Set Alarm screen with the Time tab selected, the Set Alarm screen with the Event tab selected, and the Set Alarm screen with the Event tab selected with the dropdown menu opened. The LLM is then asked to repair the test, using the prompt described in Section 3.4.

The LLM outputs a repaired test for the updated version of the application. This repaired test opens the 'more options' menu and selects 'Set Alarm' correctly, then tries to verify if the Event tab is selected with the intention of clicking it if it is not selected, but here the execution of the test fails and an error is thrown. The issue is the following: the repaired test is trying to verify if the Event tab is selected by using the statement `if 'selected="false"' in event_tab.get_attribute('outerHTML')`, which is not supported in Appium.

The LLM is asked to solve this issue. The error message is provided in the next prompt, together with all the XML files provided in the first prompt. A brief message explains the issue and asks to fix the test.

With the information provided, the LLM recognizes the issue immediately, and outputs a new repaired version of the test. This time the Event tab is checked correctly, and the execution of the test goes on by selecting the Event tab, opening the dropdown menu, and selecting 'solar noon'. The test ends by clicking the new 'schedule' button, this time using the *content-desc* instead of the outdated *text*. The test can now be considered repaired successfully.

The second example is the failed repair of the test *test-13-new* of the application SuntimesWidget. On the base version of the application, the test first clicks the 'more

options' button, and then it clicks the option 'World Map', opening the World Map pop-up. Now the test opens repetitively the Map Type dropdown menu, selecting all the three possible options. After that, the test selects, one by one, all the radial buttons under the map, ending by going back to the main screen.

On the updated version of the application, the test opens the 'more options' menu and the World Map pop-up in the same way as the base version. The first problem that emerges is that the *resource-id* of the button that opens the Map Type has changed, making the execution of the test fail. We also noticed that new map types are available now, and a new map menu is present in the updated version of the application, which includes many more options; the radial buttons under the map have been moved in the new map menu, under the option 'options'. The differences between the two versions are shown in Figure 3.3.

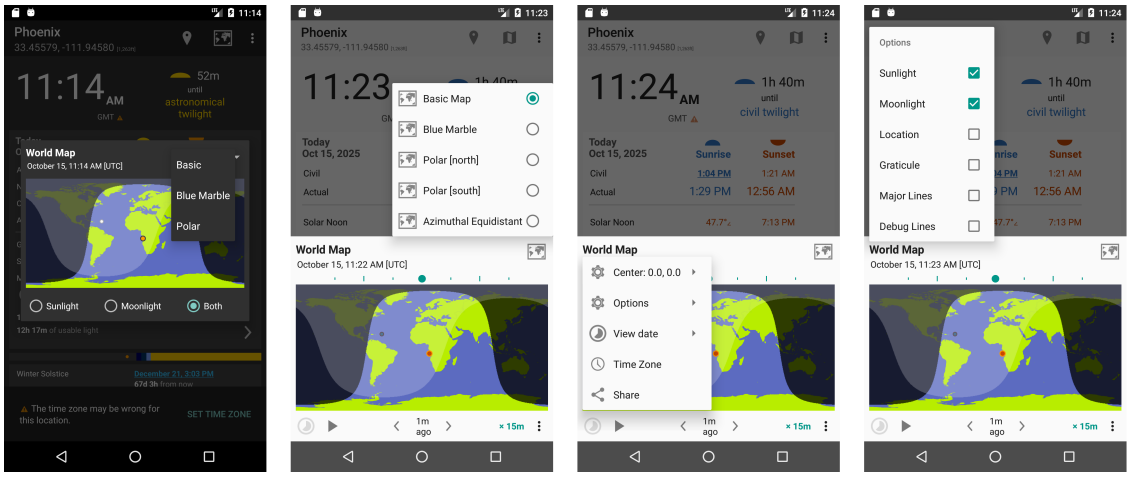

Figure 3.3. Differences between the World Map screens on the base version (left) and the updated version (the three on the right).

To repair this test, the LLM is provided XML files of the main screen, the main screen with 'more options' opened, and the World Map screen, with files for each menu opened, for both the base and the updated version of the application.

The repaired test generated by the LLM solves the problem of the *resource-id* for Map Type. The execution of the test on the updated version of the application goes on, selecting the different map types the same way as the base version. The test selects only the map types from the base version, ignoring the new ones. The LLM also understands that the radial buttons are no more under the map, but it is able to locate them in the new position by navigating the new map menu. Here a new problem appears: the repaired test tries to check and uncheck every item one after the other, without realizing that, after checking an item, the navigation closes the map menu and goes back to the World Map screen, making the test fail.

The LLM is asked to solve this issue. The error message from the execution of the repaired test is included in the prompt, together with all the XML files from the first interaction, and the screen of the application after an item is checked. The LLM outputs

a new repaired test, but the same issue persists. Since the repaired test keeps failing for the same reason, the repair is considered failed.

Chapter 4

Results and Discussion

4.1 RQ1: Incidence and Causes of Failure

While running the original tests on the updated versions of the applications, 102 causes of failure were determined (note that a test can have multiple causes of failure). Of all the causes of failure, 78 can be attributed to an outdated locator of the widget searched, 11 to a widget moved to another screen, and 13 to a functionality removed or that involves different steps.

Based on these data, outdated locators are the primary cause of broken tests across different versions of the same application. More specifically, 40 times the property *resource-id* of a widget was outdated, 24 times the *text*, 9 times the *class name*, and 3 times the *content-desc*, while in 2 occasions the tests were not able to identify a widget using the *xpath*. Despite the low number of failure because of it, the use of the *xpath* to locate a widget is usually discouraged for being fragile and harder to maintain. The low number that appears in this analysis is mainly due to the fact that it is rarely used.

The incidence of all the causes of failure, together with the repair results, is summarized in Figure 4.1.

In the analyzed tests, the *resource-id* was the most common property used to locate widgets. There are two main reasons for this locator to break: the first is that sometimes the *resource-id* of a widget changes from one version to another or disappears in the updated version of the application; the other reason has to do with the fact that the *resource-id* is not always unique among the widgets of a screen. In screens where there are lists of elements, it is common for the elements of the list to share the same *resource-id* and use indexes to distinguish one element from the other. This is a cause of fragility, since it is enough to change the order of items or have a new element in the list to make the old indexes obsolete.

The same issue happens with the *class name* locator. This locator is by nature not unique inside a screen, making the use of indexes necessary to identify a certain element. This is a cause of fragility, and it is quite common for this kind of locator to break whenever an updated brings some changes to the screen.

An example of this kind of issue can be observed in the application DIDA. In the tests written for the base version, the button that opens the sidebar is accessed to from

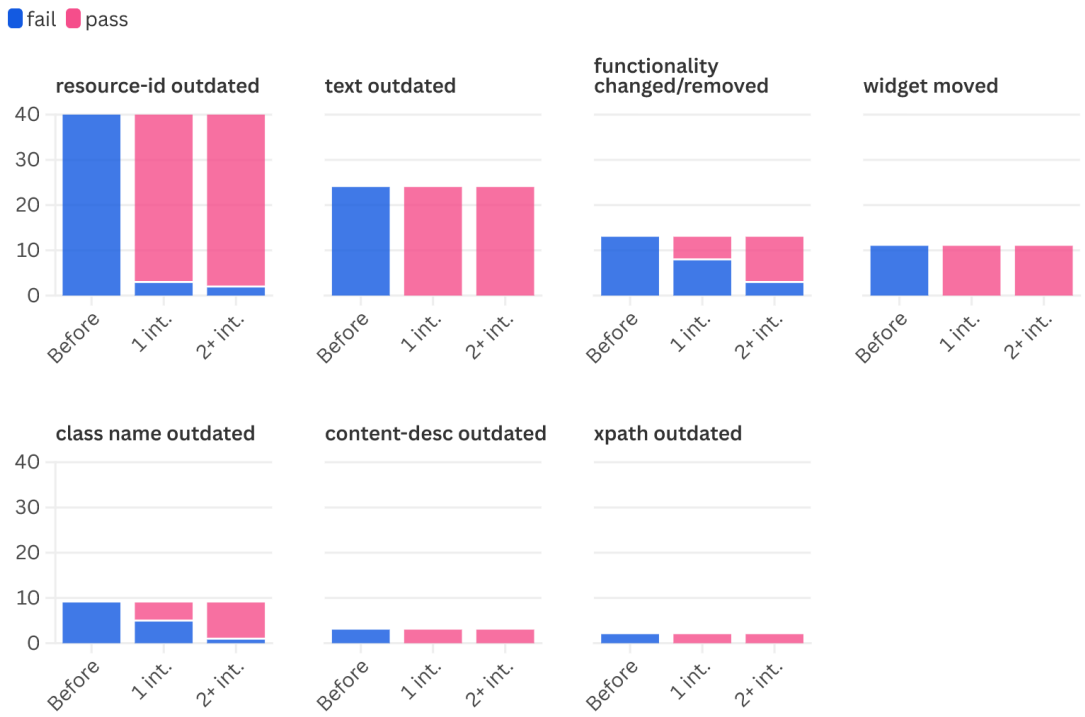

Figure 4.1. Causes of failure of the tests when executed on the updated version of the applications and repair results.

the "Inbox" screen using the locator class name "android.widget.ImageButton"[0], while the FAB button in the same screen in accessed to with the locator class name "android.widget.ImageButton[1]". We can see how the same locator is used to access both the widgets, with only the index to distinguish between the two. The use of this kind of locator causes problems when the same tests are executed on the updated version: despite the screen not receiving any significant visual change in the updated version (as shown in Figure 4.2), some internal changes made the sidebar button to be identified with the locator class name "android.widget.ImageButton[1]", and the FAB button by the locator class name "android.widget.ImageButton[0]", making the test fail when executed on the updated version of DIDA.

The *text* locator also proved to be prone to fragility. It is not rare for the text inside a widget to receive some small or big changes when the application is updated, or to just disappear in favor of other visual cues, making the old locator fail.

We can see an example of broken *text* locator in the application AppLaunch. As show in Figure 4.3, in the base version, the themes of the application where called "Light theme" and "Dark theme", while in the updated version they were abbreviated in "Light" and "Dark", making the old locators fail.

From these results, a couple of mitigation strategies can be derived to help reduce future fragility problems.

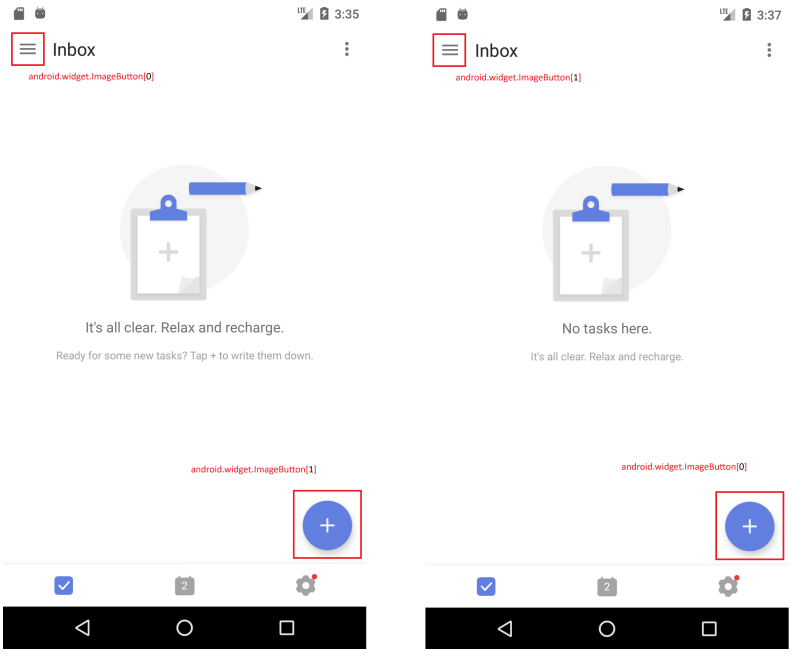

Figure 4.2. DIDA "Inbox" screen, base version (left) and updated version (right).

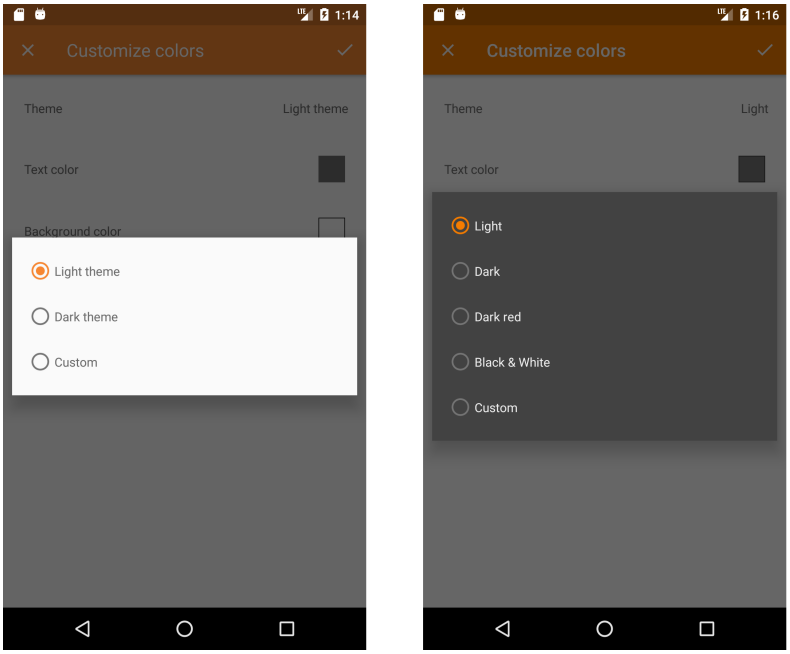

Figure 4.3. AppLaunch themes screen, base version (left) and updated version (right).

The first possible mitigation is using unique identifiers for the widgets. In particular,

this mitigation refers to the general use of the *class name* locator and to the use of the *resource-id* locator in situations where it is not unique, usually is lists of items. As noted previously in this section, the use of this kind of locators makes the use of indexes necessary, and this leads to locators that breaks very easily even with small changes to the GUI. The use of more solid locators, like the *content-desc* (when it is present) or the *resource-id* (when it is unique), should be recommended to write more future-proof GUI tests.

Another mitigation is to avoid using the *text* when possible, in favor of more solid locators. Again, there are quite a few instances in the analyzed tests where the *text* inside widgets changes from one version to the other, and the use of locators like the *content-desc* (when it is present) or the *resource-id* (when it is unique) can make for more solid tests.

As noted before, the use of the *xpath* should also be avoided. This locator is very flexible, being able to traverse up, down, or across the view hierarchy, and can locate elements based on multiple attribute. On the other hand, the *xpath* is quite slow, since it requires to traverse the entire UI tree, fragile, due to the fact that *xpath* expressions often breaks when the GUI changes, and harder to maintain, because of how long *xpath* strings can be.

4.2 RQ2: LLM Repair Performance

To determine the performance in mobile GUI tests repair of LLMs, we measured the repair success rate after only the first interaction with the LLM, and then the repair success rate with two or more interactions. Following our approach, 45 of the 61 broken tests were repaired successfully after the first interaction with the LLM, while using two or more interaction brings the number of successfully repaired tests to 56 out of 61, corresponding to a repair success rate of 73.8% and 91.8% respectively. The results of tests execution on the updated versions of the applications are shown in Table 4.1, and visualized in Figure 4.4 and Figure 4.5.

The LLM proved to be very effective in repairing tests when there are no major structural changes in the screens from one application version to another and widgets are matched one to one. In these cases the problem is usually an outdated locator of a widget, and the LLM is capable of identifying the corresponding widget in the updated screen and fix its locator during the first interaction using the XML view hierarchy files. Even when a widget is moved to another screen, the LLM proved to be capable of identifying it correctly; the generated repaired test will navigate to the widget and interact with it, eventually going back to the previous screen to continue with the rest of the test.

Taking a look at Figure 4.1, we can see that our LLM-based approach has excellent results in repairing *resource-id*, *text*, and *content-desc*, and also has perfect repair results when a widget is moved to another screen.

The LLM begins to struggle a bit when there are no meaningful locators to identify the widgets. In the analyzed applications, it is common for lists of options to have all the same *resource-id*, or to be identified just by their *class name*. In these cases it is necessary to use indexes to distinguish one option from another. The LLM showed to have some

Application	# of broken tests	# of repairs 1 int.	# of repairs 2+ int.
AnkiDroid	4	2	3
AntennaPod	3	1	3
AppLaunch	2	2	2
BookDash	1	1	1
Counter	2	1	1
Currency	2	1	2
DIDA	8	4	7
DNS66	5	5	5
DumbphoneAssistant	2	2	2
Fwknop	1	1	1
Mgit	2	2	2
Now in Android	4	4	4
OpenBikeSharing	5	5	5
RedReader	3	2	3
Sgit	2	1	2
SimpleCalculator	1	1	1
SimpleWeather	2	2	2
SoundRecorder	1	1	1
SunTimesWidget	11	7	9

Table 4.1. LLM-based approach repair results.

issues with this kind of indexes, and using the wrong index to identify an element during the first interactions (usually it keeps using the old index, without realizing it is changed), making a second interaction sometimes needed to repair the tests successfully. It is not uncommon for the LLM to try to use a more solid locator, like the *text*, in this kind of situations, when possible.

Going back again to Figure 4.1, we can see the impact that this kind of locators has on the repair results, especially on the *class name* locator, which needs a second interaction to be repaired more than half of the times.

Multiple interaction are usually necessary when there are bigger changes to the screen structure, or when a functionality includes different steps that are not easy or possible to infer from the view hierarchy files alone. As an example of the first case, when testing the "Episodes" screen of the AntennaPod application, the "more options" button is not visible in the updated version when the search bar is active; the LLM is able to fix this issue only in the second interaction, with the error log and relevant screenshots provided. For the second case, an example comes from testing the "add note" screen of the application AnkiDroid; in the updated version, when creating a cloze note, the user has to add a cloze deletion, otherwise the note will not be added and an error message will appear on the screen; the LLM has no way to know this new behavior from the view hierarchy files alone, so a second interaction is needed, in which a brief description of the problem and the screenshot of the screen with the error message are provided.

This kind of situation was the most problematic of our LLM-based approach. As

Figure 4.4. Test results LLM-based approach on the updated version of the ExtRep applications.

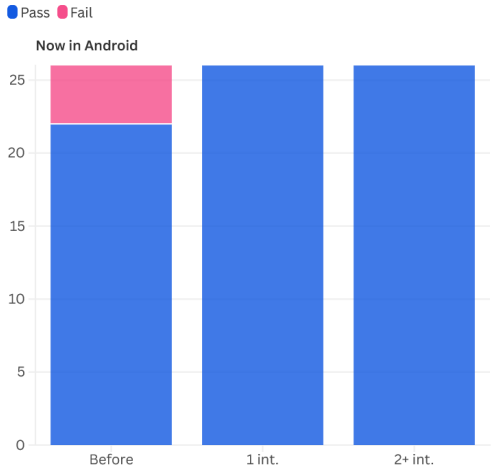

Figure 4.5. Test results LLM-based approach on the updated version of Now in Android.

highlighted in Figure 4.1, most of the times the LLM was not able to repair tests in which the tested functionality changed or was removed on the first interaction, and in three cases multiple interaction were not enough either.

Regarding the failed repaired, there are 5 cases in which the LLM was not able to repair the test successfully. One case is a test of a functionality that has been removed in the updated version of the application, so the test was not possible to be repaired and should just be eliminated. Three cases are tests where the screens have received major changes, and the navigation between them changed as well. These tests reflect the most common cause of repair failure, and highlight the main limitation of this approach: the LLM is very good at repairing outdated locators and adapt tests to small navigation changes, but it struggles when it comes to big changes that are not easily inferred from the files provided. The last failed repair is a bit of a unique case: the LLM managed to repair the test correctly, but in the end it tries to go back to the previous screen using *touch_outside* instead of the back button of the device, that was already used in the base version of the test; this new method of going back did not work, but when asked to fix it, the LLM used the same method with more checks, ending up failing again. Note that this last test is the only one that was repaired successfully by the baseline Healenium-Appium, but not by the LLM-based approach; the reason for this is that Healenium-Appium is limited to find a new locator when the old one is outdated, and does nothing when the execution of the old test proceed without problems, while sometimes the LLM tries to introduce some changes to the code to improve the test, which ended up making the test fail in this case.

11 of the 19 applications had all their tests repaired by our LLM-based approach after the first interaction. We can also add the application Counter to this list, since the only test that was not repaired refers to a functionality that has been removed from the application, bringing the count to 12 applications. These applications can be seen as the optimal cases for our repair approach, as the repair mainly consisted of updating some locators or finding a widget in another screen. The LLM was able to repair this kind of tests immediately thanks to its ability to understand the XML view hierarchy files and map the old widgets of the base version to the new widgets of the updated version of the application.

4 applications needed multiple interaction, but all of their tests were repaired successfully. Of these applications, AntennaPod was the one that received the most significant changes in the updated version, with multiple widgets moved and changes to the behavior of some functionalities, resulting in 2 of 3 tests that needed multiple interactions with the LLM to be repaired. The other applications of this list (Currency, RedReader, and Sgit), only had a test each for which a second interaction was needed: Currency and RedReader had a problem with the index of a *class name* locator and a *resource-id* locator respectively, while Sgit had an issue regarding the *content-desc* of a back button from a screen that was not visited in the older test.

For the last 3 applications, the LLM was not able to repair all the tests. These were the most complex applications of the dataset, and also the ones that received the most radical changes from the base to the updated version. The complexity of the applications, together with the many functionality changes from one version to the other are the main reasons for the mixed results in tests repair for these applications.

4.3 RQ3: Comparison with the Healenium-Appium Baseline

Finally, the repair success rate of the LLM-based approach obtain in RQ2 is compared with the repair success rate of the Healenium-Appium baseline. Healenium-Appium was able to repair successfully 14 of the 61 broken tests (success rate of 23.0%). This means that the LLM-based approach outperformed Healenium-Appium by repairing 50.8% more tests after the first interaction, and 68.8% more tests after two or more interactions. Furthermore, only one test was repaired successfully by Healenium-Appium and not by the LLM-based approach, as noted in the previous section. The results of tests execution on the updated versions of the applications are shown in Table 4.2, and visualized in Figure 4.6 and Figure 4.7.

Application	**# of broken tests**	**# of repairs**
AnkiDroid	4	2
AntennaPod	3	0
AppLaunch	2	2
BookDash	1	0
Counter	2	0
Currency	2	0
DIDA	8	0
DNS66	5	0
DumbphoneAssistant	2	2
Fwknop	1	1
Mgit	2	0
Now in Android	4	3
OpenBikeSharing	5	0
RedReader	3	0
Sgit	2	0
SimpleCalculator	1	1
SimpleWeather	2	0
SoundRecorder	1	0
SunTimesWidget	11	3

Table 4.2. Healenium-Appium repair results.

The discrepancy in results between our approach and the baseline can be explained by the following observations. Healenium-Appium was able to repair the locators of widgets that are on the same screen in the updated version with mixed results; when a test cannot find a widget on the screen, Healenium tries to find the corresponding widget in the updated screen, but the results are not always as expected, especially if the correspondence is not trivial. The LLM, on the other hand, is able to identify corresponding widgets from one version to another with excellent results. Healenium in not able to repair other kinds of tests failure (moved widgets and changed functionalities), since it can only replace a broken locator with a new one, and cannot change the test

Figure 4.6. Test results Healenium-Appium on the updated version of the ExtRep applications.

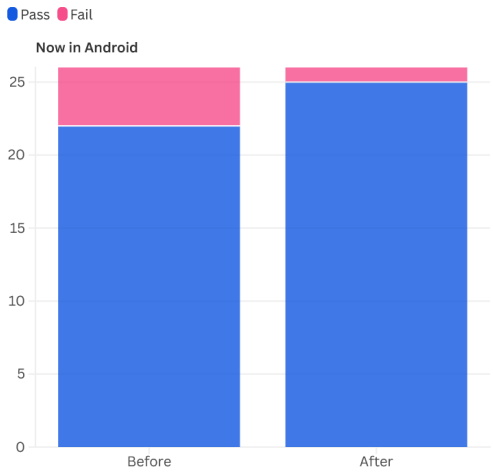

Figure 4.7. Test results Healenium-Appium on the updated version of Now in Android.

structure in any way, like interacting with another widget to reach a new screen or changing order of operations. Despite some difficulties when the changes are significant, the LLM was often able to deal with this kind of repair.

While the LLM-based approach proved to be way more effective in generating successful test repairs, time-wise it was quite a bit slower compared to the baseline. The time spent to repair a test with Healenium-Appium can essentially be reduced to the execution of two test cases, one on the base version and one on the updated version of the application, with a few seconds of slowdown when a broken locator in encountered; after running the test on the two versions of the application, the repairs are visible in the Healenium report. On the other hand, for each test the LLM-based approach needs the time to collect the XML view hierarchy files of the base version and the updated version of the application, and, for each interaction with the LLM, writing the prompt, providing the files, and execute the repaired test on the updated version of the application to determine whether the repair was successful or not. As an estimate, we can say that a test repair with the LLM-based takes approximately two to three times more time compared to the Healenium-Appium baseline, with multiple interactions increasing this difference even more.

To access the model Claude Sonnet 3.7 Thinking for GitHub Copilot, a subscription to GitHub Copilot PRO is needed, which comes at the price of $10 USD per month, or $100 USD per year. On the other hand Healenium-Appium is open source and free to use, with a price only for support and service plans. It is worth noting that, while in our experience the model Claude Sonnet 3.7 Thinking gave the best results, similar results are expected from free-to-use models as well.

4.4 Discussion

The results of the experiment show the great potential of the integration of LLMs in mobile GUI tests repair processes. Our LLM-based approach was able to repair the vast majority of the tests in the experiment, vastly outperforming the baseline.

The analysis on the causes of failure for the tests revealed three main reasons of failure: the locator of a widget is outdated, a widget was moved to another screen, the tested functionality has changed or is removed. Outdated locators were the most common cause of failure, accounting for the 76.5% of the failures found, while moved widgets and changes in functionalities accounted for the 10.8% and 12.7% respectively.

The LLM demonstrated outstanding capabilities in handling the repair of outdated locators and identifying widgets moved to other screens. Usually, this kind of issue only needed one interaction, in which the LLM uses the XML view hierarchy files of the tested screens to identify the correct widget in the updated version of the application.

While the LLM was always able to handle moved widgets at the first interaction, some issues arose when the outdated locator was not unique and the test used indexes to access it. It was not uncommon for this type of issue to require a second interaction with the LLM, in which the LLM is also provided with the description of the issue, together with relevant screenshots when deemed necessary.

The worst results for our approach were for the tests where the tested functionality changed. The main reason for this is that it is difficult for the LLM to infer how a functionality has changed having only access to view hierarchy files and screenshots. This led to multiple tests that needed more than one interaction, and some cases in which the

repair failed.

Despite some difficulties, our LLM-based approach produced much better results compared to the Healenium-Appium baseline, being able to repair the 73.8% of tests after one interaction and the 91.8% of tests after multiple interactions, while the Healenium-Appium repaired the 23.0% of tests. These results demonstrate that LLMs represent a promising solution for mobile GUI tests repair.

Chapter 5

Conclusion and Future works

5.1 Threats to validity

The discussion of the threats to validity for this research refers to the categorization provided by Feldt et al. [2].

Conclusion validity concerns on whether the treatment we used in the experiment have a statistically significant effect on the outcome we measure. In this research, a potential threat comes from the use of an incomplete or biased data collection to draw incorrect conclusions. For example, the tests of some applications used in the experiment may not test their application exhaustively, leading to an incomplete representation of the test fragility. To mitigate this problem, we carefully analyzed all the application and their tests, adding new tests written by us when the testing was incomplete.

Internal validity concerns on whether the treatment we used in the experiment have actually caused the outcome rather than other factors. In our case, the internal validity refers on whether the LLM-based repair approach actually produced repaired tests based on the provided information without external interference, such as previous conversations with the LLM or information extracted from other files opened in the IDE. To mitigate this problem, our approach started a new conversation with the LLM for each test, to avoid any interference from other exchanges with the LLM. Also, we used the web version of the LLM tool, and provided it with only the related files, to avoid any interference from other files in the project. Another concern regarding internal validity is whether the new tests we wrote may have introduced some errors. To mitigate this issue, we ran all the tests on the base version of their application, to ensure that they pass on the base version and adhere to the functionality of the application.

Construct validity concerns on whether the treatment and the outcome actually correspond to the cause and effect we are interested in. In this research, this is mainly related on the accurate with which we measure the effectiveness of the LLM in repairing mobile GUI tests. As a mitigation, we carefully analyzed all the tests to determine which ones were to be considered broken, by running them on both the versions of their application and annotating the causes of failure. After undergoing the repair process, each test was executed on the updated version of the application; the execution of each test was carefully inspected to make sure that the test not only passes, but also maintains its intended

behavior.

External validity concerns on whether the results can be generalized outside of the scope of our study. In this research, this is related to whether our findings can be generalized to other Android applications outside of the ones used in the experiment. To mitigate this issue we selected multiple real-world applications for our experiment, each one with exhaustive testing (either from the beginning or after adding the tests written by us). While there is a huge amount of different applications out there, we are quite confident that the ones used in our experiment should cover the majority of cases regarding GUI test repair.

5.2 Conclusion

In this research, we carried out an analysis on mobile GUI tests fragility and explored the potential of LLMs in addressing the challenges of mobile GUI tests repair. In recent years many studies have been proposed on the use of LLMs in software testing and tests repair. The background section covers some of the most promising of them.

We gathered 19 real-world Android applications as the dataset of our research. Each application came with a base version, a series of GUI tests written for it, and an updated version. Our first step was to run all the tests on the updated applications, to determine the causes of test failure and their incidence when an application evolves. From this analysis, we found three main causes of failure, which are outdated widget locators, widgets moved from one screen to another, and tested functionality changed or removed. Of the three, outdated locators were the most common cause of failure.

After that, we developed an LLM-based mobile GUI tests repair approach to be used to repair the failed tests. This approach leverages on multiple interactions with the LLM to generate a repaired test. In the first interaction the LLM tries to generate a repaired test based on the failed test and the XML view hierarchy files of the tested screens. If the first interaction in not enough to generate a successful repair, other interactions are used, in which the LLM also receives the description of the issues and eventual screenshots.

Our approach was able to repair the majority of the tests successfully. Specifically, it was able to repair the 73.8% of tests after one interaction, and the 91.8% of tests after multiple interactions.

Finally the results of our approach were compared to the state-of-the-art approach of Healenium-Appium, to understand how our LLM-based approach performs compared to a traditional method. Healenium-Appium was able to repair the 23.0% of tests, meaning that our approach vastly outperformed the baseline in generating repaired tests successfully. There results demonstrate that the integration of LLMs is a promising solution to enhance the process of mobile GUI tests repair.

5.3 Future works

We can see a couple of different direction that could be taken to improve this work in the future.

The first improvement could be a more refined prompt engineering. The prompt is a crucial aspect of the interaction with LLMs, and a well-refined prompt can help improve the repair accuracy and efficiency, producing correct test fixes that reflect the intended behavior in fewer interactions.

In our experiment, the situation in which the LLM struggled the most was when the tested functionality received significant changes in the updated version of the application. The main reason is that the LLMs was not always able to understand the new behavior of the application just from the XML view hierarchy files and the screenshots. Being able to provide the LLM with the information to more easily understand the behavior of the application could help a lot in solving this issue.

Another possible improvement is the automation of the process. At the moment our approach is done manually, from the test execution and the collection of the XML view hierarchy files and the screenshots, to the prompting to the LLM and the analysis on the repaired tests. The consequence is that our approach was about two to three times slower compared to the Healenium-Appium baseline. Introducing some level of automation would make our approach much more efficient and closer, time-wise, to the baseline.

Bibliography

[1] Shaoheng Cao, Minxue Pan, Yu Pei, Wenhua Yang, Tian Zhang, Linzhang Wang, and Xuandong Li. Comprehensive semantic repair of obsolete gui test scripts for mobile applications. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ICSE '24, New York, NY, USA, 2024. Association for Computing Machinery.

[2] Robert Feldt and Ana Magazinius. Validity threats in empirical software engineering research - an initial survey. pages 374–379, 01 2010.

[3] Tommaso Fulcini, Riccardo Coppola, Marco Torchiano, and Luca Ardito. An analysis of widget layout attributes to support android gui-based testing. In *2023 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 117–125, 2023.

[4] Difei Gao, Lei Ji, Zechen Bai, Mingyu Ouyang, Peiran Li, Dongxing Mao, Qinchen Wu, Weichen Zhang, Peiyi Wang, Xiangwu Guo, Hengxu Wang, Luowei Zhou, and Mike Zheng Shou. Assistgui: Task-oriented desktop graphical user interface automation, 2024.

[5] Zhe Liu, Chunyang Chen, Junjie Wang, Mengzhuo Chen, Boyu Wu, Xing Che, Dandan Wang, and Qing Wang. Make llm a testing expert: Bringing human-like interaction to mobile gui testing via functionality-aware decisions. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ICSE '24, New York, NY, USA, 2024. Association for Computing Machinery.

[6] Zhe Liu, Cheng Li, Chunyang Chen, Junjie Wang, Mengzhuo Chen, Boyu Wu, Yawen Wang, Jun Hu, and Qing Wang. Seeing is believing: Vision-driven non-crash functional bug detection for mobile apps, 2024.

[7] Yonghao Long, Yuanyuan Chen, Chu Zeng, Xiangping Chen, Xing Chen, Xiaocong Zhou, Jingru Yang, Gang Huang, and Zibin Zheng. Extrep: a gui test repair method for mobile applications based on test-extension. *Automated Software Engineering*, 32, 04 2025.

[8] Minxue Pan, Tongtong Xu, Yu Pei, Zhong Li, Tian Zhang, and Xuandong Li. Gui-guided test script repair for mobile apps. *IEEE Transactions on Software Engineering*, 48(3):910–929, 2022.

[9] Tim Rosenbach, David Heidrich, and Alexander Weinert. Automated testing of the gui of a real-life engineering software using large language models. In *2025 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 103–110. IEEE, March 2025.

[10] Hao Wen, Yuanchun Li, Guohong Liu, Shanhui Zhao, Tao Yu, Toby Jia-Jun Li, Shiqi

Jiang, Yunhao Liu, Yaqin Zhang, and Yunxin Liu. Autodroid: Llm-powered task automation in android. In *Proceedings of the 30th Annual International Conference on Mobile Computing and Networking*, ACM MobiCom '24, pages 543–557, New York, NY, USA, 2024. Association for Computing Machinery.

[11] Chunqiu Steven Xia and Lingming Zhang. Less training, more repairing please: revisiting automated program repair via zero-shot learning. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2022, pages 959–971, New York, NY, USA, 2022. Association for Computing Machinery.

[12] Tongtong Xu, Minxue Pan, Yu Pei, Guiyin Li, Xia Zeng, Tian Zhang, Yuetang Deng, and Xuandong Li. Guider: Gui structure and vision co-guided test script repair for android apps. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2021, pages 191–203, New York, NY, USA, 2021. Association for Computing Machinery.

[13] Juyeon Yoon, Robert Feldt, and Shin Yoo. Autonomous large language model agents enabling intent-driven mobile gui testing, 2023.

[14] Shengcheng Yu, Yuchen Ling, Chunrong Fang, Quan Zhou, Chunyang Chen, Shaomin Zhu, and Zhenyu Chen. Llm-guided scenario-based gui testing, 2025.

[15] Yakun Zhang, Chen Liu, Xiaofei Xie, Yun Lin, Jin Song Dong, Dan Hao, and Lu Zhang. Gui test migration via abstraction and concretization. *ACM Transactions on Software Engineering and Methodology*, April 2025.

[16] Yakun Zhang, Wenjie Zhang, Dezhi Ran, Qihao Zhu, Chengfeng Dou, Dan Hao, Tao Xie, and Lu Zhang. Learning-based widget matching for migrating gui test cases. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ICSE '24, New York, NY, USA, 2024. Association for Computing Machinery.

[17] Daniel Zimmermann and Anne Koziolek. Gui-based software testing: An automated approach using gpt-4 and selenium webdriver. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering Workshops (ASEW)*, pages 171–174, 2023.