ACM DIGITAL LIBRARY Association for Computing Machinery acm open

Latest updates: https://dl.acm.org/doi/10.1145/3728975

PDF Download
3728975.pdf
12 January 2026
Total Citations: 0
Total Downloads: 425

Published: 22 June 2025
Accepted: 31 March 2025
Received: 31 October 2024

Citation in BibTeX format

RESEARCH-ARTICLE

Automated Test Transfer across Android Apps using Large Language Models

BENYAMIN BEYZAEI, University of California, Irvine, Irvine, CA, United States

SAGHAR TALEBIPOUR, University of Southern California, Los Angeles, CA, United States

GHAZAL RAFIEI, University of Southern California, Los Angeles, CA, United States

NENAD MEDVIDOVIC, University of Southern California, Los Angeles, CA, United States

SAM MALEK, University of California, Irvine, Irvine, CA, United States

Open Access Support provided by:

University of California, Irvine

University of Southern California

Proceedings of the ACM on Software Engineering, Volume 2, Issue ISSTA (June 2025)
https://doi.org/10.1145/3728975
EISSN: 2994-970X

Automated Test Transfer across Android Apps using Large Language Models

BENYAMIN BEYZAEI, University of California at Irvine, USA
SAGHAR TALEBIPOUR*, University of Southern California, USA
GHAZAL RAFIEI*, University of Southern California, USA
NENAD MEDVIDOVIĆ, University of Southern California, USA
SAM MALEK, University of California at Irvine, USA

The pervasiveness of mobile apps in everyday life necessitates robust testing strategies to ensure quality and efficiency, especially through end-to-end usage-based tests for mobile apps' user interfaces (UIs). However, manually creating and maintaining such tests can be costly for developers. Since many apps share similar functionalities beneath diverse UIs, previous works have shown the possibility of transferring UI tests across different apps within the same domain, thereby eliminating the need for writing the tests manually. However, these methods have struggled to accommodate real-world variations, often facing limitations in scenarios where source and target apps are not very similar or fail to accurately transfer test oracles. This paper introduces an innovative technique, LLMigrate, which leverages Large Language Models (LLMs) to efficiently transfer usage-based UI tests across mobile apps. Our experimental evaluation shows LLMigrate can achieve a 97.5% success rate in automated test transfer, reducing the manual effort required to write tests from scratch by 91.1%. This represents an improvement of 9.1% in success rate and 38.2% in effort reduction compared to the best-performing prior technique, setting a new benchmark for automated test transfer.

CCS Concepts: • **Software and its engineering** → **Software notations and tools**.

Additional Key Words and Phrases: Mobile UI testing, Large language models, Test transfer

ACM Reference Format:
Benyamin Beyzaei, Saghar Talebipour, Ghazal Rafiei, Nenad Medvidović, and Sam Malek. 2025. Automated Test Transfer across Android Apps using Large Language Models. *Proc. ACM Softw. Eng.* 2, ISSTA, Article ISSTA098 (July 2025), 24 pages. https://doi.org/10.1145/3728975

1 Introduction

Testing mobile apps' user interfaces (UIs) is crucial and ensures a seamless user experience across different devices and platforms. Manually testing mobile UIs requires significant time and effort and is prone to human error. Many approaches have been proposed in recent years to address these issues and automate the testing process [23–25, 31, 35, 36, 39, 43, 44, 46, 49, 54, 55, 57, 61, 64, 70–72, 75, 81, 82, 86, 88, 92, 95, 96]. Many of these automated testing approaches focus on crash detection or maximizing certain criterion such as activity coverage [35, 39, 72, 77, 81, 88]. However, recent

*Both authors contributed equally to this research.

Authors' Contact Information: Benyamin Beyzaei, University of California at Irvine, Irvine, USA, bbeyzaei@uci.edu; Saghar Talebipour, University of Southern California, Los Angeles, USA, talebipo@usc.edu; Ghazal Rafiei, University of Southern California, Los Angeles, USA, grafiei@usc.edu; Nenad Medvidović, University of Southern California, Los Angeles, USA, neno@usc.edu; Sam Malek, University of California at Irvine, Irvine, USA, malek@uci.edu.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2025 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM 2994-970X/2025/7-ARTISSTA098
https://doi.org/10.1145/3728975

studies [42, 53, 62] have indicated that developers prefer UI tests that target specific functionalities of an app. For instance, in a shopping app, the tester wants to ensure that a user is able to successfully register, search for a product, and add the product to the shopping cart.

Test Transfer—also known as *Test Migration* —is a solution proposed in recent years for automatically creating new usage-based tests for a mobile app by using already existing tests from a similar app [28, 59]. The concept of test transfer is based on the idea that apps within a specific domain, such as shopping, mail clients, or web browsers, despite potential differences in appearance and the way they are programmed, share very similar functionalities. Therefore, it is possible to use the already existing usage-based tests from one app to automatically create tests that exercise analogous functionalities in another app within the same domain. For instance, use a test for validating the login functionality in *Geek* , a shopping app, to automatically create tests for validating the login functionality in other popular shopping apps such as *Yelp* or *Zalando* [5].

The existing test transfer solutions can be divided into two categories: (1) similarity-based approaches, which focus on matching events between the source and target apps using a distance metric [26, 27, 52, 58, 65, 66, 74, 76], and (2) classification approaches that define the test transfer problem as a classification task [44].

The similarity-based techniques have demonstrated high effectiveness in cases where the source and target apps are very similar. However, these approaches face significant limitations in more complex cases, especially where a straightforward one-to-one correspondence between the event sequences representing specific functionalities in the source and target apps does not exist [66, 102]. This limitation is mainly due to the fact that these approaches focus on directly mapping events between the source and target apps. As a result, they encounter limitations in adapting to cases where such direct mappings do not exist, which is often the case in real-world apps.

The only existing machine learning-based method, AppFlow [44], has shown to be dependent on the categories of apps that are used for its training, hampering its generalizability. This technique also requires considerable manual effort to adapt to a new app category, further limiting its usefulness.

Recent advances in AI have led to the development of Large Language Models (LLMs), which have shown significant promise in automating various software engineering tasks such as code generation [63], code summarization [22], and software testing [89]. These models, trained on a large amount of data, are capable of accurately understanding the semantics of user interfaces, including screens and widgets [68]. Intrigued by these results, we set out to investigate the degree to which the rich semantics embedded in LLMs can be applied to advance the state-of-the-art in test transfer.

This paper proposes LLMigrate, the first approach to employ multimodal LLMs for transferring UI tests across mobile apps, offering an end-to-end solution without requiring access to the app's source code. The proposed technique relies solely on dynamic analysis of the app, as static analysis typically makes the approach less practical for use with proprietary apps for which the source code is not available. This approach takes the binaries of two apps—the source and target apps—along with a UI test originally created for the source app as its inputs and generates a corresponding test that targets the same functionality in the target app.

LLMigrate's transfer process consists of two main steps: (1) *Source Abstraction*, and (2) *Test Migration*. In the source abstraction phase, the source test is converted to a version represented in natural language, which we call the *Abstract Source Test*. This is achieved by executing the source test on the source app to extract the required features and consulting with the LLM to understand the semantics of the source test. During the second step, test migration, the abstract source test in natural language, which was generated in the first phase, is adapted to the target app. This is achieved by dynamically exploring the target app, extracting features at each step, and selecting the optimal actions with the help of LLM.

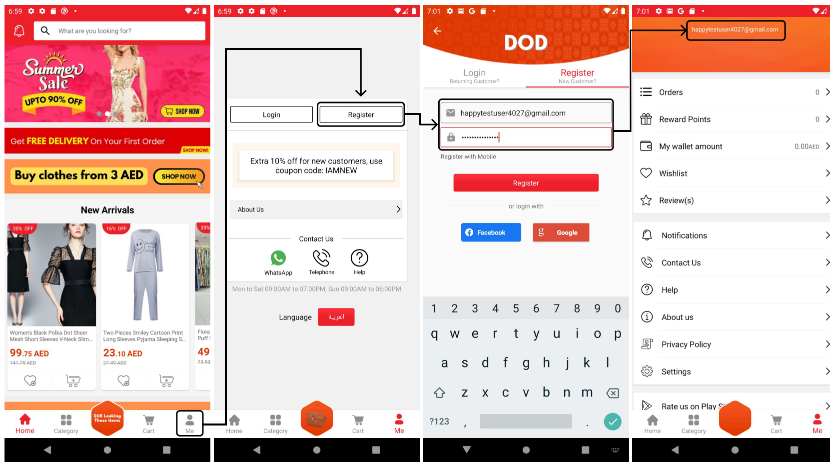

Fig. 1. Registration test in DODuae [1].

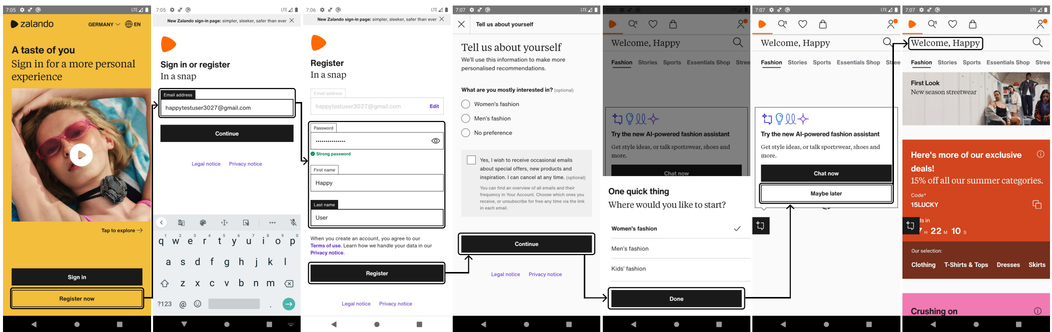

Fig. 2. Registration test in Zalando [5].

We evaluated LLMigrate by applying it to transfer UI tests for apps in five different app categories. In each category, tests were transferred across different apps, resulting in a total of 120 transfers. Both the tests and the apps are reused from the dataset introduced by CraftDroid [58]. The transfer outcomes are evaluated based on the precision, recall, reduction, and success rate metrics, and the results were compared to those obtained from the existing techniques evaluated on the same benchmark [58, 66, 97]. LLMigrate achieved a 97.5% success rate, reducing the manual effort required to write tests from scratch by 91.1%. This represents an improvement of 9.1% in success rate and 38.2% in effort reduction compared to the best-performing technique previously evaluated on the CraftDroid benchmark. Notably, LLMigrate attains these gains with an average transfer time of 247 seconds, surpassing the efficiency of prior methods.

2 Background and Terminology

Figure 1 depicts the required steps to register a new user on DODuae, a popular shopping app, while Figure 2 demonstrates the same on Zalando, another widely used shopping app. As outlined in Section 1, the core concept of test transfer is to leverage an existing test from one app, such as DODuae, to automatically generate a test that targets the same functionality in another app, in this specific example, Zalando. We use this example to introduce the concepts relevant to our approach.

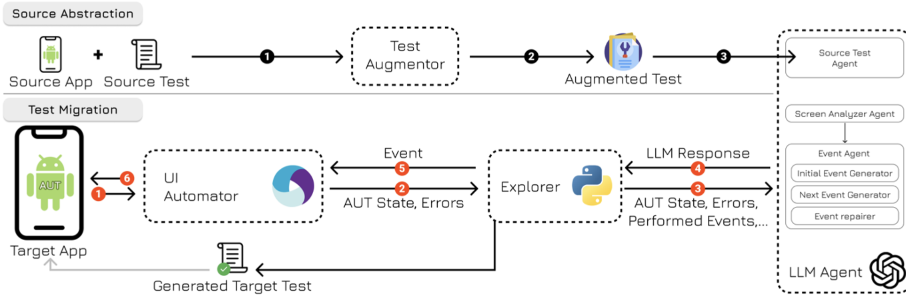

Fig. 3. LLMigrate's approach overview.

The *source app* is the app for which an existing UI test is available—in this case, DODuae. This test, called *source test* (e.g., the registration test on DODuae), will be transferred to the *target app*, an app in the same domain as the source app, which in this example is Zalando. The resulting test after the transfer, which is the final outcome of the transfer technique, is termed the *target test* (e.g., the registration test on Zalando). To evaluate the effectiveness of our approach, we compare the target test against a manually created test for the target app, referred to as the *ground truth test*. This ground truth test mirrors the same functionality as the source test and serves as an oracle for measuring the performance of our test transfer technique.

A *UI test* is structured as a sequence of events. The events can be classified into three categories: *UI events*, *system events*, and *oracle events*. UI events represent a user's actions in the app to interact with its user interface, such as taps, swipes, and text inputs. System events are generated by the underlying operating system, such as pressing the back button on Android devices. Oracles, also known as assertions in the relevant literature, are responsible for verifying the expected outcomes at various stages of a test. An oracle in the context of UI testing can be defined as a predicate function $F : (w, c) \mapsto \{True, False\}$, where w represents a widget of the app, and c is a condition to be evaluated pertaining to w. The function yields *True* if the widget w fulfills the criterion c; otherwise, the outcome is *False* resulting in the failure of the assertion.

We represent events in all of these categories as a triple *(action, event_type, widget)*. Here, the *action* denotes the type of action, such as click, as well as any necessary auxiliary inputs, such as the text input for keyboard events. The *event_type* can either be *"gui"*, *"oracle"* or *"system"*, specifying the nature of event. The *widget* denotes a target UI element in terms of its attributes. Some events may not be dependent on a specific widget, such as back or scroll, or may not require any auxiliary input.

3 Approach

Figure 3 shows a high-level overview of LLMigrate. It takes three primary inputs: 1) a source test, 2) the binary of the source app, and 3) the binary of the target app. Its output is the target test.

As shown in Figure 3, LLMigrate's approach involves two main phases: **1) Source Abstraction**, in which the source test is translated to a natural language representation called the abstract source test, and **2) Test Migration**, in which the core functionality of transferring the test to the target app happens by analyzing and understanding the semantics of the target app's UI. In the following subsections, we will discuss these phases in more detail, and the components involved in each of them.

3.1 Source Abstraction

During the source abstraction phase, the source test, which is dependent on a specific application, programming language, platform, and testing framework, is translated to an internal representation

called the abstract source test. This internal representation describes the source test in natural language and is independent of any testing framework and programming language. Also, since it is in the form of natural language, the abstract test is easily readable and comprehensible by both humans and LLMs. The translation is performed in two steps: 1) *Test Augmentation*, and 2) *Abstraction*.

3.1.1 Test Augmentation. The purpose of this step is two-fold. First, it aims to make the test independent of any specific programming language and testing framework. Second, it collects all the useful attributes from the elements that are interacted with in the source test, which might not necessarily be available in the test but can be helpful in understanding its semantics.

In order to achieve this, the source test is executed on a device running the source app event by event. This involves our *Test Augmentor* component communicating with the source app by utilizing the Appium testing framework [33]. After the execution of each of the events, this component extracts the UI layout hierarchy of the source app. This layout hierarchy is in the form of an XML tree and represents the UI elements in the current state of the source app. The Test Augmentor then locates the specific widget interacted with during that particular step by analyzing the XML hierarchy, using its locator, such as `resource-id`. The additional useful textual attributes are then captured from the layout hierarchy for the located widget, such as `content-desc`, `text`, or `class`.

Once this information is collected, each event is represented as a triple *(action, event_type, widget)*, as mentioned in Section 2. The final augmented test is a sequence of these events, which is independent of any specific programming language or testing framework. Figure 4 provides examples of GUI and oracle events after the augmentation process.

We support a limited set of UI interactions and primarily focus on two conditions for oracles: (1) the presence of an element, and (2) the invisibility of an element. The set of UI events and oracles supported in our work is consistent with those in the prior test transfer literature [58, 65, 66, 97]. Our augmentation module is available publicly on the project repository [17].

```
{
  "action": ["send_keys", "56.6"],
  "event_type": "gui",
  "class": "android.widget.EditText",
  "resource-id": "anti.tip:id/bill",
  "text": "0.00"
}
```

```
{
  "action": ["wait_until_element_presence", 10],
  "event_type": "oracle",
  "class": "android.widget.EditText",
  "resource-id": "anti.tip:id/total",
  "text": "65.09"
}
```

action event_type widget

Fig. 4. Examples of GUI and oracle events after the test augmentation step.

3.1.2 Abstraction. In this step, the augmented test, the output of the previous step, is translated to a representation in natural language, called the *Abstract Source Test,* which serves as a short, one-paragraph summary of the source test that describes each of its UI and oracle events.

To achieve this, we use the *LLM Agent* module, which is responsible for creating prompts and communicating with the off-the-shelf LLMs. To create the prompts for each of the tasks in our approach that require communication with LLM, we have manually created a prompt template that includes the task definition and the necessary structure to introduce each of the input features. This prompt template definition is a one-time manual effort needed for each task, such as test abstraction. Once the template is defined, for every query instance, the *LLM Agent* integrates the variable input features into the prompt template.

The utilized off-the-shelf LLM accepts two types of prompts: 1) system prompt, which is a general prompt that familiarizes the agent with the task and provides a general context, and 2) user prompt, which is task-specific. Prompt 1 represents the user prompt template for the abstraction task, defining the requirements for the abstract test. This includes specifications such as its length

and the information it should include. Note that, in all of the prompts demonstrated throughout the paper, the text presented in color green, represents the variable input features that are substituted in the prompt template for each specific query. For each task, we also provide a system prompt (not shown here) that specifies the context for the LLM. Due to space constraints, the complete prompt details are available in our publicly accessible repository [17].

Prompt 1: Source Abstraction Prompt

This is the test and each step is a JSON object. You should explain what this test does in a single paragraph. Don't add the details of the steps and keep it short but make sure to mention the exact values of inputs and texts. [...] Also explain which step is the final step and which step makes the test complete and emphasize on the functionality that is under test.
Augmented Test Steps (Generated with *Test Augmentor* module from a test script)

Once the required prompt is created, the *Source Test Agent* consults the off-the-shelf LLM using its provided API and collects the response. The following presents an example of the abstract source test for the registration task of the DODuae app, which was described earlier in Section 2.

This test contains three oracles and evaluates the registration functionality of the "mobilapp.opencart.doduae" app. It begins by verifying the presence of the dashboard element (oracle), then navigates to the "Me" section (GUI event), and confirms the presence of the register button (oracle). The test proceeds by clicking the "Register" button (GUI event), entering "sample@gmail.com" as the email address and "samplepassword" as the password (GUI events), and clicking the "Register" button again (GUI event). The final step, which completes the test, is an oracle that checks for the presence of the text "sample@gmail.com" to confirm successful registration.

3.2 Test Migration

In this phase, the abstract source test is transferred to the target app by dynamically exploring and analyzing it. It involves the following interconnected components as shown in Figure 3: 1) *UI Automator*, which is responsible for communicating with the device running the target app to collect information and execute commands on it, 2) *Explorer component*, which coordinates the main logic of the transfer process, such as tracking the executed events and determining the transfer completion, and 3) *LLM Agent*, which, as discussed in Section 3.1.2, is responsible for consulting with the LLM.

The workflow of the test migration phase is performed as a repetitive loop, executing steps 1 to 6 depicted in orange in Figure 3. In each iteration of the loop, the goal is to find the optimal next event executable in the target app such that the executed sequence in the target app is one step closer to replicating the source test.

The dynamic exploration of the target app always starts from the initial screen that appears when the app is first installed and launched. In the first step, the XML UI layout hierarchy representing the elements in the current screen of the target app is captured by the UI Automator component. Given that this layout hierarchy in its original form contains redundant information, which makes it considerably large, the UI Automator component processes it to clean the layout hierarchy, retaining only the necessary elements identified based on their type. The final set of retained elements includes 15 different widget types, including `android.widget.EditText` and `android.widget.ImageButton`, that are widely used across Android apps. This set of widget types is configurable in the tool.

The complete set can be found in our implementation [17], and if needed, extended.

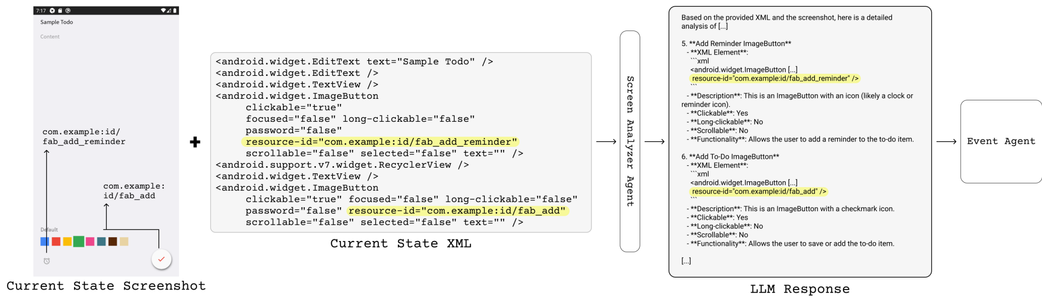

Fig. 5. An example of the screen analysis task performed by the screen analyzer sub-agent.

We refer to the processed layout hierarchy and a screenshot captured from the app screen as the *Current State* of the app. The current state is the main source of information about the target app at each step of the transfer process.

Once the app state is collected, it is passed to the Explorer component in the second step. If the Explorer component detects that no previous steps have been executed on the target app, it simply passes the app state to the LLM Agent to identify the optimal initial event to be executed on the current state of the target app in step 3.

In order to detect the optimal event on the target app, the initial step is to understand the semantics of the current app state, including the processed XML UI layout hierarchy and the app screenshot. To achieve this, we have utilized the capabilities of LLMs, as they have demonstrated a strong understanding of structured texts such as XML and HTML [34, 40, 41, 83] and are capable of effectively analyzing screenshots of web and mobile apps [103]. Specifically, this task is managed by the *Screen Analyzer* sub-agent, as illustrated in Figure 5. The screen analyzer consults the LLM to analyze both the XML layout hierarchy and the screenshot, matching the widgets within the hierarchy to the captured screenshot. Prompt 2 represents the specific prompt template used for the screen analysis task.

The outcome of the screen analysis task is an analysis report, which, as shown in Figure 5, includes the key elements of the screen along with their locators, natural language functionalities, and the potential actions they can support.

The rationale for using multimodal inputs for the screen understanding task is that relying solely on the textual info obtained from the layout hierarchy can sometimes be insufficient, which may lead to suboptimal event selection. For instance, consider the example of adding an entry in a Todo app, demonstrated in Figure 5. In this case, two highlighted widgets exist with textual attributes `fab_add` and `fab_add_reminder`, respectively. By only relying on textual information, it can be ambiguous to determine which of the two widgets should be used to add a new task, since the term "reminder" is often used interchangeably with "task" or "to-do" in the domain of to-do apps. However, by incorporating the visual information from the captured screenshot, the screen analyzer agent is able to use the images and icons and conclude that the functionality of the second widget is to allow the user to add a reminder to a to-do item.

Prompt 2: Screen Analysis Prompt

The attached image is a screenshot of an Android application. Below is the XML representation of the same screenshot:
Current State (XML)

Your task is to analyze both the image and the XML file. Describe the key widgets you observe in the screenshot and match them to their corresponding XML elements. For each widget, indicate whether:
- It is clickable or long-clickable.
- It is an EditText field, allowing for text input (send_keys).
- Scrolling is possible within it.
And also explain what is the possible functionality of the widget. Be detailed in your descriptions and clearly highlight the key widget behaviors.

Next, the LLM Agent, specifically the *Initial Event Generator* sub-agent, defines the initial action selection task as a question-answering task and utilizes a specific prompt template for this task. The variable input features for this task include: 1) The abstract source test, which guides the agent in identifying the next best event; 2) the XML UI layout hierarchy, which provides information about the existing elements in the target app; and 3) the screen analysis report generated in the previous step. Prompt 3 provides the corresponding prompt template for the initial event selection task. Note that in all the prompts presented in the paper, the sections of the prompts written in gray serve as section headings and descriptive summaries. These elements do not exist in the original prompt text but are included in the paper to provide the purpose and content of specific sections of the prompts.

Prompt 3: Initial Event Selection Prompt

Abstract Source Test: This is a sequence of events intended to perform a user interface (UI) test on the source app Source App Package Name application. We define it as a source test: Abstract Source Test

Task Definition: This part defines the role of a mobile test engineering assistant responsible for migrating automated tests between Android applications. The focus is on ensuring compatibility with the target app, initializing properly, and adapting test actions based on the target app's current state.

You have the initial screen of the target application along with the source test definition above. Now, you are migrating the source test to another application. Given this scenario, which event do you believe is the best to perform next? [...]

Event Definition: This section defines the types of events that can be used when migrating tests: GUI events interact with the app's interface, Oracle events check the state of the app, and System events handle system-level actions.

You can choose between three types of events: 1. GUI events 2. oracle events 3. system events
1. GUI events include event_type as GUI, class of the element, resource-id of the element, action and other necessary fields if they exist like text, hint, content-desc and naf. You should generate this event based on the current state of the target application [...]
2. oracle events include event_type as oracle and the action which can be one of the "wait_until_element_presence" or "wait_until_element_invisible". If you return an oracle event you just need to include event_type and action fields not anything else. [...]
3. system events include event_type as system [...]

Application State: Here is the initial screen of the current application in XML format: Current State (XML)

Here is a description of the current state widgets: Screen Analysis Report (Generated from *Screen Analyzer Agent*)

Possible Actions: Instruction for Generating an Action
Note: Every action should be an array.
1. key_back: Press the system back button. Choose this action when test goal contains back press or back navigation. example: ["key_back"]
[...]
9. wait_until_element_invisible: Use this oracle to validate that a particular element is not visible on the screen. The selector type could be "xpath", "content-desc", "id" (resource-id) and "text". you should just include an array inside the

action field of the event, alongside with the event_type. First item inside the array is the action type, second one is the wait time, third is the selector type and the last one is the selector value. Example: ["wait_until_element_invisible", 10, "text", "a sample text"]: Check if "a sample text" is not visible on the screen for 10 seconds.

The LLM Agent consults the LLM using Prompt 3 to create the initial optimal event, which is expected to be in the JSON format, including the event triple *(action, event_type, widget)* defined in Section 2. Examples for LLM responses that contain an event are shown in Figure 4. In order to mitigate the challenge imposed by the non-determinism of LLMs, we took two specific actions for all the action selection prompts. Firstly, we set the temperature parameter of the LLM to zero to reduce its exploration abilities and make its responses more predictable. Second, to reduce the impact of LLM hallucination, we used a majority algorithm to determine the final generated event. This involves querying the LLM `n` times repeatedly for each task as a zero-shot prompt, leading to the receipt of `n` responses in the form of JSON objects. In a JSON object, a key is a unique identifier for a piece of data, and its corresponding value is the data itself. To generate the final event, the LLM Agent includes only the keys that appear more than `m` times among the `n` received responses and sets the value of these keys to the most frequently occurring value for each included key. Note that `m` and `n` are configurable parameters that are set to 2 and 3, respectively, in our current evaluation.

Upon determining the initial event, the LLM Agent sends the generated event to the Explorer component in step 4. The Explorer then records it as an executed event to keep track of it and sends it to the UI Automator component (step 5), so that it can be executed on the target app, enabling it to proceed to the next state (step 6).

It is important to note that not all the events generated by the LLM Agent are necessarily executable or valid events. Consequently, executing the generated event on the target app may result in one of four different scenarios: ***1)*** The UI event is successfully executed on the target app, causing the target device to proceed to a new state, or the condition for the generated oracle event is met on the target app. ***2)*** The execution of a UI event or system event leads to an exception, indicating it is not a valid event. This can have reasons such as invalid widget locators created by LLM. ***3)*** The condition for the generated oracle is not met in the app, which results in an exception. ***4)*** The execution of a UI event will not raise any exceptions, however it will not result in any changes in the target app, indicating it was not a useful step to be included in the target test. Next, we discuss in detail the subsequent steps required to address each of these four scenarios.

Scenario 1 indicates a successful iteration of the migration loop. The migration loop continues to iterate until all the source oracles are transferred to the target app or until a configurable threshold of the number of target events is reached. This threshold is set to three times the number of source test events in our current implementation. Note that, this end condition has the assumption that tests typically conclude with an oracle event, which is often the case in practice. If the end condition is not met, another iteration of the migration loop is initiated. In this scenario, the key difference between the initial and the subsequent iteration is in the event selection prompt. For the subsequent steps, the event selection prompt introduces an additional input feature to be integrated into each query alongside the abstract source test, XML layout hierarchy, and screen analysis report, which is the previously executed steps recorded by the Explorer component to guide the LLM Agent in identifying the next optimal event.

One of the limitations of the existing test transfer techniques is their failure to handle input value differences between the source and target applications. In real-world scenarios, The required parameters might differ between the source and target apps. For example, some of the required parameters for the source app may not be used in the target app, while some required parameters in the target app may be missing in the source test. For instance, a source test exercising the registration functionality in a shopping app might include a "ZIP code" field not needed in the target

app, or the target app might require additional fields like "name" and "family name". Our approach addresses these differences in input requirements by guiding the LLM to generate appropriate values for input fields in case they are not available in the source test while skipping the unnecessary existing input values. The LLM's ability to create appropriate inputs that are not present in the source test based on its understanding of the application's context makes it highly useful for test transfer in contrast to the existing similarity-based techniques, which rely solely on the existing input values in the source test and face shortcomings when additional input values are required.

Additionally, during the event selection process, we incorporate a set of general guidelines, referred to as hints, into the prompts. These hints are designed to help the LLM agent interact more effectively with the application by using a set of specific rules. For instance, one of these hints instructs the agent to ensure that all required fields in a form are filled in before submission. Another hint instructs against selecting repeated actions. Furthermore, the "possible actions" section of Prompt 4, provides additional instructions regarding action types, such as the possibility to interchange `swipe_right` and `long_click`, across Android applications. We have ensured that all the provided hints are general and contain no app-specific or scenario-specific details.

Prompt 4 demonstrates the prompt template for the next steps event selection task. Note that the sections that do not contain detailed descriptions are identical to the sections explained in Prompt 3.

Prompt 4: Next Event Selection Prompt

Abstract Source Test

Performed Events: You have successfully performed these events in the target application so far (performed events array): Performed Events (Captured by *Explorer* module)

Application State

Task Definition: This section outlines that during test migration, it is important to prioritize transferring oracle events to verify the correct application state. While exact steps from the abstract source test don't need to be followed, actions should be adapted to the current state of the target app.

You are not required to transfer the exact steps in the test goal, just transfer the suitable ones based on the current state. [...]

Event Definition

Values: In the target app, you may need to use specific values such as name, email, password, etc. Fill out all required fields, but leave optional fields empty. Use the same values in confirmation fields (e.g., password and confirm password) when required. If no specific values are provided in the test goal, generate random valid data based on the field names in the target app. Always prioritize using values from the test goal if they match the field in the target app.
If there are no values to use inside the abstract source test generate random correct values based on the field name in the target app but always prefer using values from the abstract source test for related fields.

Hints: This part provides general tips for handling events during test migration. Key guidelines include: Base your actions on the "current_state". Only add oracle events when indicated in the "current_state". Avoid unnecessary actions like sending keys to already-filled fields. Use attributes like text and class if resource-id is unavailable

Some steps and rules that you should follow:
- If you have already interacted with an element and have not reached the correct oracle, try another action on that element or generate a completely new event. Do not repeat the same events in the already performed events array. [...]

Possible Actions

Scenarios 2 and ***3*** occur when an invalid event is created by LLM, which are easily identifiable by the UI Automator component. This is because executing these events on the target app results in raising an exception. In these cases, again, another iteration of the migration loop begins. However, since the previous iteration was unsuccessful, additional measures need to be taken by Explorer, our primary coordinator, to address the failure. These measures are fourfold. First, the Explorer component removes the previously generated event that resulted in an exception from the list of executed events, as it should not be included in the final transferred test. Second, the Explorer records this event as a dead-end event to ensure that it will not be mistakenly considered a valid event in future iterations. It is important to note that dead-end events are saved based on the current transfer step, as an event might be a dead-end in one step but not in another one. During step 5 of the migration loop, the Explorer always checks to ensure that a suggested event by the LLM Agent has not been previously identified as a dead-end event. Third, the Explorer backtracks one step in the execution to undo the consequences of the last invalid generated event. The backtrack is achieved by restarting the app, clearing the cache, and re-executing all recorded events except the last invalid one. Finally, the Explorer needs to inform the LLM Agent that the generated event in the previous iteration was invalid. This results in the LLM Agent utilizing a specific prompt, asking the off-the-shelf LLM to repair its previous attempt.

Prompt 5 shows the sections of the repair event selection prompt that are different from Prompt 4. This prompt specifically includes a feedback section that consists of the previously generated incorrect event and the exception that was raised as a result of executing that event. The rationale behind including this feedback is based on prior research in program repair and analysis, which has shown that LLMs have a strong ability to understand bugs and exceptions when they are provided with detailed feedback [29]. Therefore, to enhance the handling of cases in which an exception is raised, we use a Chain-of-Thoughts (CoT) approach by including the exception description and prior responses within the prompt. This method has been shown to improve the reasoning capabilities of LLMs [90].

Prompt 5: Repair Event Prompt

Abstract Source Test + **Performed Events** + **Application State**

Feedback: You have been asked for generating an event, this is the last generated event and it has already been attempted and failed, throwing an exception. This event should not be recommended again in your response: Last Wrong Event
This event is not correct because of this exception: Last Exception

Task Definition + **Event Definition** + **Values**

Hints: - If you have already interacted with an element and have not reached the correct oracle, try another action on that element or generate a completely new event. Do not repeat the same events from the already performed events array.
- When your last wrong event is a send key action but the exception indicates that you cannot set the element and you are interacting with the wrong element, you should try clicking on that element first before sending keys. Fix the last wrong event by changing the action to click. [...]

Possible Actions

Scenario 4, where the execution of the generated event does not result in any changes in the target app's state, is not identifiable by the UI Automator as it does not raise an exception. To detect this scenario, at the beginning of each migration loop iteration, the Explorer component compares the target app state received from the UI Automator with its previous state. If the two states are identical, indicating that the generated event was not useful, the Explorer takes the same steps

required for scenarios 2 and 3 mentioned above, including backtracking and utilizing the repair event selection prompt (Prompt 5) in the iteration. To prevent the migration process from becoming stuck on a specific screen and continuously generating invalid events, a threshold is established for the number of incorrect events generated in each step of the transfer process. Upon reaching the threshold of unsuccessful attempts, which is set to three attempts in the current implementation, the Explorer initiates a backtrack. The last performed event is also removed from the executed events in Explorer's record, and it is marked as a dead-end event.

As previously mentioned, the migration loop continues until all the source oracles are transferred to the target app. At this point, the Explorer creates the final output from the recorded executed events, resulting in the generated target test. Note that the generated target test is presented as an augmented test, which is in the form of a triple *(action, event_type, widget)*, as discussed in Section 2. In this representation, the widget attribute may contain different selectors, such as `resource_id` or `text`, any of which can be used for widget interaction. As previously mentioned, the UI Automator component is responsible for executing the events of the transferred test on the device which includes prioritizing the widget selectors to be utilized for the event execution.

4 Evaluation

In this section, we detail the evaluation of LLMigrate, focusing on how effective it is in transferring UI tests across mobile apps. Our evaluation aims to answer the following four research questions:

- **RQ1.** How effective is LLMigrate in accurately transferring UI tests across real-world mobile apps?
- **RQ2.** How useful are the tests transferred using LLMigrate?
- **RQ3.** What are the time and cost implications of using LLMigrate for transferring tests?
- **RQ4.** How practical is LLMigrate for transferring tests on today's popular apps?
- **RQ5.** How well does LLMigrate perform in transferring tests across apps that are not previously seen by LLMs?

4.1 Experimental Setup

LLMigrate is designed to be platform-agnostic and, therefore, capable of transferring tests across various devices and platforms. Our current implementation focuses on transferring tests across Android apps. For our evaluation, we utilized the publicly available dataset introduced by the authors of CraftDroid [58]. As shown in Table 1, this dataset contains tests from five different app categories: Browser, To Do List, Shopping, Mail Client, and Tip Calculator. The advantages of utilizing this dataset are two-fold. First, it enables the evaluation of our developed approach within the context of real-world apps. Second, since this dataset has also been utilized as the benchmark for evaluating the other existing test transfer techniques, such as CraftDroid [58], TEMdroid [97], TRASM [65], and TREADROID [66], it enables us to effectively compare our technique against the state-of-the-art approaches across various dimensions such as accuracy, usefulness, and performance.

It is important to note that we were unable to utilize the CraftDroid dataset in its original form because, for some of the apps, the versions that were used in this dataset are no longer functional. In these cases, we used an updated version of the apps, provided the following two constraints hold: 1) a functional and supported version of the app is available, and 2) the update does not alter the test flow from the original version utilized in the CraftDroid dataset, such as requiring additional steps like CAPTCHA. In the cases where these conditions were not met, we removed the non-functional app from the dataset. This resulted in reusing 19 out of the 25 apps from the original CraftDroid dataset. Table 1 demonstrates the final set of the subject apps as well as the versions used in our evaluation.

Table 1. Subject apps.

Category	c1-Browser	c2-To Do List	c3-Shopping	c4-Mail Client	c5-Tip Calculator
Apps (Versions)	Lightning (5.1)	Minimal (1.2)	Geek (2.3.7)	K-9 Mail (6.603)	Tip Calculator (1.1)
	Browser for Android (6.0)	Clear List (1.5.6)	Yelp (10.21.1)	Mail.Ru (14.117.0)	Tip Calc (1.11)
	Privacy Browser (2.1)	To-Do List (2.1)		myMail (14.97.0)	Simple Tip Calculator (1.2)
	FOSS Browser (5.8)	Shopping List (0.10.1)			Tip Calculator Plus (2.0)
	Firefox Focus (6.0)				Free Tip Calculator (1.0.0.9)

Table 2. Test cases for the proposed functionalities.

Functionality	#Test Cases	Average# Total Events	Average# Oracle Events
c1/t1-Access website by URL	5	3.6	1.0
c1/t2-Website navigation involving back button	5	6.6	3.0
c2/t1-Add task	4	4.25	1.0
c2/t2-Add then remove task	4	6.75	2.0
c3/t1-Registration	2	14.5	5.0
c3/t2-Login with valid credentials	2	7.0	3.0
c4/t1-Search email by keywords	3	5.0	3.0
c4/t2-Send email with valid data	3	9.3	3.0
c5/t1-Calculate total bill with tip	5	3.8	1.0
c5/t2-Split bill	5	4.8	1.0
Total	38	5.9	2.0

In the resulting dataset, there are tests for validating at least two of the main functionalities provided by apps in each category, as shown in Table 2. In the presented table, categories are represented by *c*, and functionalities under tests are represented by *t*. For example, *c1/t1* represents the first tested functionality for the first app category, Browser, which is *Access website by URL*. We conducted evaluations on a total of 120 transfers and manually evaluated the transfer results.

Our experiments were conducted using a Nexus 5X emulator running Android 6.0 (API 23), aligning with CraftDroid's evaluation for apps where the original versions were functional. For the updated apps that are incompatible with this older version, we employed Nexus 6a emulators running Android 10.0 (API 29). For our evaluation, we used GPT-4o as an off-the-shelf LLM provided by OpenAI, which can perform reasoning across both visual and textual inputs. We selected this model due to its reasonable cost and superior performance on reasoning and coding benchmarks [78]. All tests were conducted through OpenAI's API, and to accurately measure the transfer time, we tested across various internet connections and VPNs. The transfers were executed on a Mac machine with an 8-core CPU, 10-core GPU, and 16GB of unified memory.

As discussed in Section 3, our approach has three adjustable parameters: 1) maximum wrong tries at the Same Step, 2) total number of runs for majority voting (n), and 3) threshold for majority voting to include a field (m). We empirically observed the best-performing values for all these parameters and set them to 3, 3, and 2, respectively, in our evaluation.

4.2 RQ1. Efficacy of LLMigrate

For evaluating the efficacy of LLMigrate, we utilize the precision and recall metrics introduced and utilized by the existing research targeting test transfer [58, 66, 97]. Similar to the definition utilized by the existing research, true positives (TP) are events in the transferred test that exist in the ground truth. False positives (FP) are events in the transferred test that do not exist in the ground truth. Finally, false negatives (FN) are events that exist in the ground truth but are not present in the transferred test.

Table 3 presents a comparative analysis of the precision and recall metrics obtained by LLMigrate, CraftDroid [58] and TREADROID [66]. Consistent with prior research, we categorized all events

into two main types: GUI events and oracle events, with system events classified under GUI events. To ensure a fair and accurate comparison, we aimed to capture results achieved by the other techniques that were derived solely from the 19 subject apps that are currently functional from the CraftDroid dataset, as shown in Table 1. This required the detailed analysis of each transferred test for other techniques, similar to the process we used to analyze tests transferred by LLMigrate. This was possible for CraftDroid and TREADROID, as the artifacts containing the transferred tests are publicly available for CraftDroid, and we obtained the corresponding artifacts for TREADROID upon communicating with the authors. This was not possible for TEMdroid as the final transferred tests are not publicly available, and we were unable to obtain the research artifacts that would allow us to process it for a fair comparison, even after contacting the authors. Furthermore, we were unable to successfully run TEMdroid as the implementation of certain components was not publicly available. Consequently, for TEMdroid, we used the average metrics reported in their paper. In this case, although the datasets are not identical, the comparison remains informative since our dataset shares 82.6% (19 out of 23) of the apps. We have not included the metrics obtained by TRASM [65] and ATM [27], as TREADROID previously benchmarked its obtained results against these techniques and demonstrated superior performance across all metrics [66].

Note that there may be more than one correct ground truth in transferring a test from a source to a target app. To this end, we manually inspected the transferred tests generated by all the transfer techniques rather than automating the process to ensure a fair comparison between different techniques.

As demonstrated in Table 3, LLMigrate was able to achieve a total average precision of 98% for GUI and 94% for oracle events. Similarly, LLMigrate achieved an average recall of 99% for GUI events and 94% for oracle events. These results indicate that LLMigrate outperforms all the existing techniques in both average precision and recall metrics for both GUI and oracle events in total. Note that, due to the varying lengths of different scenarios, all reported average metrics are calculated based on the total number of events within each category or across all transfers rather than based on the achieved metric for each individual migration. Furthermore, a detailed analysis of the results for each individual migration is presented in our publicly available repository [17].

Table 3. Comparative analysis of precision and recall score metrics achieved by LLMigrate, CraftDroid, and TREADROID across different app categories and TEMdroid's average.

App Category	Approach	Precision		Recall	
		GUI Event	**Oracle Event**	**GUI Event**	**Oracle Event**
Browser	LLMigrate	100 (— %)	100 (— %)	100 (— %)	100 (— %)
	CraftDroid	90.54	100	98.52	97.50
	TREADROID	100	100	100	100
To Do List	LLMigrate	93.26 (↑7.9%)	97.22 (↑4.9%)	98.98 (— %)	97.22 (↑5.8%)
	CraftDroid	83.48	92.30	78.44	80
	TREADROID	85.39	91.42	98.70	91.42
Shopping	LLMigrate	100 (↑55.2%)	56.25 (↑27.7%)	96.87 (↑32.1%)	56.25 (↑23.4%)
	CraftDroid	44	28.57	64.70	35.29
	TREADROID	44.73	27.78	51.52	32.81
Mail Client	LLMigrate	100 (↑6.8%)	91.67 (↓8.3%)	100 (— %)	91.67 (↓8.3%)
	CraftDroid	64.70	83.33	68.75	83.33
	TREADROID	93.18	100	95.34	100
Tip Calculator	LLMigrate	100 (↑5.0%)	100 (— %)	100 (↑4.0%)	100 (— %)
	CraftDroid	77.78	75	88.15	73.17
	TREADROID	94.94	100	95.91	89.74
Total Average	LLMigrate	98.39 (↑9.6%)	94.71 (↑3.1%)	99.53 (↑6.0%)	94.71 (↑5.7%)
	CraftDroid	78.54	83.59	85.65	82.29
	TREADROID	88.77	91.58	93.52	87.44
	TEMdroid	71	90	93	89

For the app categories containing less complicated apps and test flows, such as the browser category, the baseline method demonstrated successful transfer of all events and oracles, and LLMIGRATE was able to achieve similar performance. However, in more complex scenarios, we observe a notable improvement over the baseline. This improvement is largely due to the LLMIGRATE's more advanced comprehension of screen elements, enabling it to surpass the limitations of one-to-one event transfer.

Unlike GUI events for which we observed improvement in both precision and recall across all app categories, we saw degradation in one category in these metrics for oracle events. Upon further analysis of these cases, we realized that in these cases, many oracles are specifically designed to confirm page accuracy prior to executing an action. When LLM can detect a widget on a page and interact with it directly, sometimes it omits the necessary oracle transfers for verification purposes. This may account for a decrease in oracle-related metrics in categories such as mail applications. Earlier methods, which transfer events sequentially, transfer oracles at particular steps, thus benefiting from consistent application flows and yielding marginally higher precision and recall in oracle events. But this is not always the case, and LLMIGRATE design principles allow it to achieve better performance on apps which have different flows.

4.3 RQ2. Usefulness of the Tests Transferred by LLMIGRATE

The usefulness of a test is defined by how helpful it is in reducing the manual effort for a human tester. To measure usefulness, we utilized the reduction metric, defined by previous research in this area [102]. This metric compares the manual effort required to write the ground truth test from scratch to the effort required to manually transform the transferred test to be identical to the ground truth test. The manual effort is defined as the Levenshtein distance [56] between the sequence of events of the transferred and ground truth tests. The reduction metric is calculated using the following equation: (# Ground Truth Events - Manual Effort) / (# Ground Truth Events).

Figure 6 presents the average reduction metric achieved by LLMIGRATE, CRAFTDROID, and TREADROID across different app categories and in total. Similar to the analysis performed for answering RQ1, we used the subset of the subject apps common across all approaches to ensure a fair comparison and evaluated all the transferred tests manually. Again, we were unable to include the reduction metric achieved by TEMDROID due to the unavailability of their artifacts and not reporting the reduction metric on CRAFTDROID dataset in the corresponding publication. On average, LLMIGRATE achieved 91% reduction, demonstrating that LLMIGRATE was able to eliminate more than 91% of the manual effort required for writing tests, outperforming all the prior techniques by almost 40% in total average.

In categories such as shopping, which involve more complex test cases to transfer, previous research [58, 66] has shown a negative reduction metric, indicating that it can be more efficient to write tests from scratch rather than rely on transfer methods followed by extensive manual edits to the generated tests. This insight underscores the significance of metrics like reduction, which directly reflect the decrease in manual effort, as opposed to focusing solely on precision and recall of the transferred events. Since tools in this domain are fundamentally intended to transfer complete tests to minimize manual work for test engineers, reduction serves as a more relevant measure of a tool's practical effectiveness.

In our evaluation of LLMIGRATE's usefulness, we utilized another metric called the successful transfer rate. This is a binary metric, assigned a value of 1 (100%) if the objective of the source test is met in the transferred test and 0 (0%) if it is not. Detecting if the objective of the test is met is done through manual inspection.

Figure 7 presents the successful transfer rate achieved by various techniques across different app categories and in total. On average, LLMIGRATE was able to achieve a total of 97.5% successful

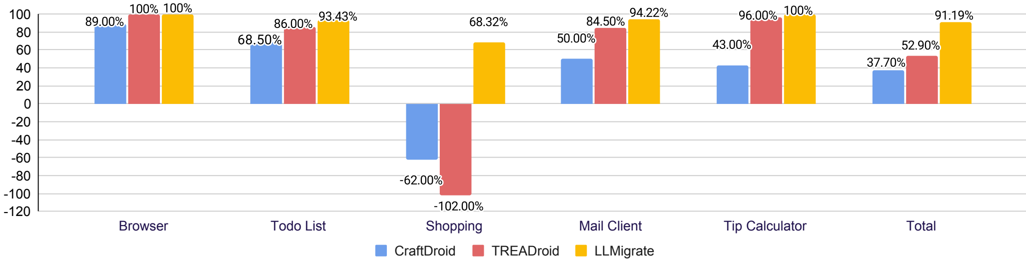

Fig. 6. The reduction achieved by LLMigrate, CraftDroid, and TREADROID across various app categories.

transfer rate across all the 120 transfers. This means that, although the transferred test may have additional extra steps not present in the ground truth or reach the objective via a path that was not identical to our specific manually defined ground truth, 97.5% of the transferred tests successfully met their objectives, including executing the required functionality and asserting the appropriate conditions using the transferred oracles. This shows that LLMigrate was able to outperform the existing technique with the highest successful transfer rate, TREADROID, by almost 10%. Again, for TEMdroid, we were unable to obtain the exact value for this metric, but the total average success rate reported in a subsequent work by the same authors [98] indicates a far inferior successful transfer rate of 53%, which is 44% lower than LLMigrate.

4.4 RQ3. LLMigrate's Performance and Cost Effectiveness

With respect to the required time to transfer a test from the source to the target app, LLMigrate has significantly better performance than previous methods and transfers each test in 247 seconds on average, which is 290 seconds and 5,120 seconds better than TEMdroid and CraftDroid, respectively. While TREADROID does not report an exact average transfer time, an analysis of the results reported in the paper suggests that it performs faster than CraftDroid but slower than TEMdroid. Therefore, LLMigrate outperforms TREADROID on the performance metric as well.

The main cost of LLMigrate is due to the usage of LLMs such as GPT-4o. To calculate the cost for each transfer, we tracked the tokens in each query during the transfer and computed the accumulated cost of all of the queries as the total cost of one transfer. On average, each of the transfers requires 118,600 input tokens and 5,180 output tokens, costing USD $0.70, which depends on the length of the transferred test. On average, each of the transferred tests has 5.5 steps, and each step costs USD $0.12.

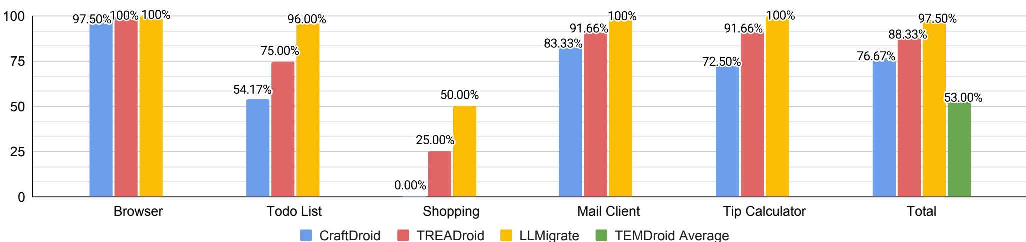

Fig. 7. The successful transfer rate achieved by LLMigrate, CraftDroid, TEMdroid, and TREADROID across various app categories.

4.5 RQ4. Practical Usage of LLMigrate on Today's Popular Apps

As presented in Section 4, LLMigrate achieved strong results on the CraftDroid dataset, which is the primary benchmark used by existing test transfer techniques. However, since many apps in this dataset are outdated or no longer supported, and given the rapid evolution of mobile app development and workflows, we conducted a follow-up study to evaluate LLMigrate on more recent, widely used real-world apps. This study aimed to broaden the evaluation scope and address the dataset's limitations, particularly in categories with fewer functional apps. We selected additional up-to-date apps in the Browser, To Do List, Mail Client and Shopping categories. In each case, we transferred existing CraftDroid tests to the new apps and measured precision, recall, success rate, and reduction, as shown in Table 4. LLMigrate demonstrated strong performance on this new set, achieving an average success rate of 98%, an average reduction score of 97%, and an average transfer time of 238 seconds. These findings validate LLMigrate's effectiveness in adapting to modern app workflows and highlight the value of transferring tests from older apps to reduce manual testing effort in current app development.

Table 4. Analysis of precision, recall, success, and reduction metrics achieved by LLMigrate, on new apps.

App	Download	Test	Precision		Recall		Success	Reduction
			GUI Event	Oracle Event	GUI Event	Oracle Event		
Mozilla Firefox [3]	100M+	c1-t1	100	100	100	100	100	100
		c1-t2	100	100	100	100	100	100
ToDo [4]	10M+	c2-t1	100	100	100	100	100	100
		c2-t2	91.66	88.88	91.66	88.88	100	90.63
DODuae [1]	1M+	c3-t1	100	30	100	37.5	100	50
		c3-t2	100	66.67	100	66.67	100	77.78
Zalando [5]	50M+	c3-t1	94.73	40	100	57.14	50	62.99
		c3-t2	81.81	66.67	100	66.67	100	64.58
Email [2]	10M+	c4-t1	100	100	100	100	100	100
		c4-t2	100	77.78	100	77.78	100	84.26

4.6 RQ5. Evaluating LLMigrate on Data Unseen by LLMs

Since LLMigrate leverages existing off-the-shelf LLMs like GPT-4o to transfer tests, it is important to evaluate its effectiveness on apps and test cases that the LLM has not previously encountered. This helps ensure that the promising results discussed earlier are not simply due to the LLM's prior familiarity with the subjects, and it also validates LLMigrate 's applicability to future, unseen apps. To do this, we took two key steps: selecting apps that were released after GPT-4o's knowledge cutoff date of October 2023 [78], and manually writing new test cases for these apps rather than relying on publicly available ones. We introduced five new app categories and, within each, selected three recently released apps. For each category, we manually designed a representative usage scenario and created corresponding tests, resulting in 30 total test transfers across these new apps.

Table 5 outlines the app categories, selected apps, and usage scenarios. Table 6 presents the precision, recall, success rate, reduction, and transfer time metrics for these unseen test transfers. On average, LLMigrate achieved 97% and 89% precision and 97% and 92% recall for GUI and oracle events, respectively. Additionally, it reached a 93% average successful transfer rate and 88% reduction score, consistent with the findings in earlier research questions. These outcomes confirm that LLMigrate performs reliably on data that was not part of the LLM's training set, supporting the robustness of the approach.

Table 5. Evaluation subjects used to verify LLMigrate's applicability on data previously unseen to GPT-4o.

App Category	Subject Apps	Release Date	Functionality	# Total Events	# Oracle Events
c6-EMI Calculator	a1-EMI Calculator & Financial [14]	9 Aug 2024	t1-Add 3 entries and calculate the EMI	5.4	1
	a2-Cash Loan EMI Calculator [8]	20 Jan 2025			
	a3-EMI Calculator : Loan Planner [15]	1 Feb 2025			
c7-AI Chatbots	a1-Google Gemini [16]	4 Jun 2024	t1-Start a conversation asking a Yes/No question and verify the answer	3	2
	a2-Deep Search - AI Chatbot [13]	30 Jan 2025			
	a3-Chatbot - AI Smart Assistant [9]	16 Feb 2024			
c8-Movies	a1-Ava Assistant - Movies & Shows [7]	29 Jan 2025	t1-Search for a movie and share the details	4.5	1
	a2-200TV - Live TV Movies App [6]	16 Jan 2025			
	a3-ClipFix: Movie Shazam [10]	29 Aug 2024			
c9-Note	a1-Daily Notes - Easy Notebook [12]	1 Feb 2025	t1-Add a note and search for it by its title	6.5	2
	a2-Notes - QuickNotes [20]	14 Jan 2024			
	a3-Personal notes and tasks [21]	5 Jan 2025			
c10-Messenger	a1-Messages for SMS - DUAL SIM [18]	26 Dec 2024	t1-Search for a phone number and send a message to the found recipient	5	2
	a2-Messages: Text SMS [19]	29 Jan 2025			
	a3-Color SMS: Message & Messenger [11]	26 Oct 2023			

Table 6. Analysis of precision, recall, success, reduction, and transfer time metrics achieved by LLMigrate, on unseen apps.

Category Test	Precision		Recall		Successful Transfer Rate	Reduction	Transfer Time
	GUI Event	Oracle Event	GUI Event	Oracle Event			
c6-t1	100	100	100	100	100	100	294.5
c7-t1	100	66.7	100	72.7	100	60	265.44
c8-t1	100	30	100	100	100	100	273.69
c9-t1	91.7	91.7	91.7	91.7	66.7	83.3	333.77
c10-t1	100	100	100	100	100	100	424.63
Average	**97.8**	**89.6**	**97.8**	**92.8**	**93.3**	**88.6**	**318.41**

5 Discussion

Our evaluation demonstrates that LLMigrate achieves high accuracy in transferring tests across Android apps, outperforming existing solutions. However, we observed certain failure cases that highlight limitations of the current approach. These include incorrect action selection by the LLM, incomplete flow transfer where critical steps are omitted, and the insufficient oracles generation, particularly for transitions. While these issues do not always prevent test execution, they can compromise the correctness or completeness of the transferred tests. Readers can find specific examples of these cases on the project repository [17].

Additionally, LLMigrate shares some common limitations with prior test transfer techniques. Its performance is influenced by the accessibility and quality of UI metadata in the target app, particularly when meaningful attributes like `resource-id` or `content-desc` are missing. The tool also struggles with transient UI elements such as toast messages, which may disappear before they can be processed. Furthermore, certain events, like user registration, can be irreversible, making it difficult to recover from partial failures. These challenges suggest directions for future enhancements, such as incorporating visual analysis [30] or improving app state management.

6 Threats to Validity

An important threat to the validity of our work stems from our reliance on off-the-shelf LLMs such as GPT-4o. These models produce responses that are not fully deterministic. Consequently, our approach may also produce varying results for the same test, impacting the findings' reproducibility. We tried to address this threat by querying the LLM multiple times, as discussed in Section 3.2.

Another threat to the validity of our evaluation comes from our inability to run the previous test transfer tools. This is due to reasons such as the unavailability of parts of the source code [97] and technical problems that are mainly due to changes and the lack of maintenance of existing dependencies [58, 66]. As detailed in Section 4, to navigate this issue and offer a comparative analysis, we utilized the publicly available CraftDroid dataset [58] previously used by existing tools. A related threat arises from a small number of apps from the CraftDroid dataset [58] that are deprecated. To mitigate this threat and ensure a fair comparison, for CraftDroid [58] and TREADROID [66], which provided detailed evaluation results for individual transfers, we restricted our comparison to the subset of apps from the dataset that are still functional. As discussed in Section 4, for TEMdroid [97], we could not obtain detailed experimental results even after contacting the authors. Due to the lack of available data, with respect to TEMdroid, we compared the average numbers for each category, which can be a threat to validity.

Our tool leverages Appium and the Appium UiAutomator2 Driver [85] to interact with UI elements. This driver follows the WebDriver standard [32], enabling a wide range of interactions with the app UI. However, since our tool builds on earlier approaches, we adopt a similarly limited yet well-curated set of interactions. While we support all actions covered by previous approaches—enhancing selector accuracy in the process—we do not provide full support for every possible action.

7 Related Works

The most relevant group of works targets transferring tests from one Android app to another, similar to LLMigrate. Behrang et al. [26, 27] and Lin et al. [58] proposed ATM and CraftDroid, which rely on app analysis and NLP techniques to transfer tests across different Android apps within the same domain. AppFlow [44] is a machine learning-based approach that utilizes screen and widget classification to generate UI tests for an app using a library of existing tests. Liu et al. [65, 66] proposed adaptive semantic matching strategies for test transfer. Yu et al. [97] have recently proposed TEMdroid, a semantic matching-based approach for test transfer that leverages dynamic analysis and Siamese networks. Zhao et al. [102] proposed FrUITeR, a framework for automatically evaluating the previous test transfer approaches. Mariani et al. [74] presented a study on techniques for semantic matching of GUI events used by existing test reuse approaches. Khalili et al. proposed SemFinder [52], an approach that assesses different configurations for mapping UI events across apps based on their textual information but does not focus on the test transfer problem as a whole. Mishra et al. [76] extended SemFinder by incorporating visual information into the event mapping technique.

Most of the techniques mentioned above that directly target test transfer rely on similarity-based matching between the events of source and target apps. These approaches can be highly effective in cases where the source and target apps have similar workflows for the functionality under test. However, as research work showed [102], in practice, due to the inherent differences between apps even within the same domain, relying solely on event-by-event similarity-based matching may not result in useful transferred tests. In contrast with these techniques, LLMigrate addresses the test transfer holistically and does not depend on event-by-event similarity-based matching.

Another group of recent works targets a slightly different problem, focusing on many-to-one UI test transfers. MigratePro [99] is a technique that aims to improve UI test transfer by generating a new test from multiple tests that have already been migrated to the target app from various source apps. MigratePro is not a transfer technique itself, and it improves tests transferred by an existing transfer technique. Future research can explore using tests transferred by LLMigrate as input for MigratePro to assess potential improvements. Another recent work, MACdroid [98], uses LLMs to create tests for a new target app using an abstract test logic created from multiple

source tests targeting the same functionality on different apps. Note that compared to one-to-one transfer, many-to-one transfer is a less challenging problem due to the availability of more data, such as multiple different flows in various apps that can be more similar to the intended flow in the target app. However, in practice, there often are not multiple compatible tests for the same functionality available that can be used for transfer, which limits the practicality of these techniques. Furthermore, unlike LLMigrate, which employs multimodal LLMs and utilizes both visual and textual information for UI understanding tasks, MACdroid relies solely on textual data, which can lead to certain limitations, as discussed in Section 3. A more detailed comparison of the two approaches is not possible due to the unavailability of the implementation and artifacts relevant to this technique at the time of publication.

There exists another group of relevant research work that targets transferring UI tests across different platforms. TestMig [79] and MAPIT [84] have targeted test transfer across Android and iOS apps. Ji et al. [47] conducted a comprehensive study on vision-based widget mapping for cross-platform GUI test migration. In the context of web apps, Rau et al. [80] proposed an approach for efficiently generating UI tests by learning from the existing tests of other apps. Mariani et al. [73] proposed an approach that automatically exploits the common functionalities of Java apps to generate UI tests. TransDroid [60] has transferred tests from a web app to its Android version by making use of a navigation graph and the textual data of the events and widgets involved in them. MUT [38] is a technique for transferring GUI tests of one web app to another using NLP methods.

Another group of related research focuses on bug reproduction in Android apps [87, 100, 101]. Note that although these works and test migration efforts both aim to execute a sequence of events in an Android app, the problems differ in two important aspects: (1) One of the biggest challenges of test migration comes from the differences in the flow of executed steps for a scenario between the source and target app. However, this issue does not exist in bug reproduction since the bug report belongs to the same app. (2) Test transfer also involves the challenge of accurately transferring and creating oracle events, which is a complexity that is not a part of the bug reproduction task.

Finally, several recent publications [37, 45, 48, 50, 51, 67–69, 91, 93, 94] have explored the application of LLMs to advance mobile testing. However, none of these techniques address the automated transfer of existing usage-based tests across apps with similar functionality.

8 Conclusion and Future Work

This paper has presented LLMigrate, a technique that relies on multimodal LLMs for transferring usage-based UI tests across Android apps. In our extensive evaluation covering five app categories, LLMigrate was able to successfully transfer 97% of tests and reduce more than 90% of the total manual work required for writing UI tests.

Potential future areas of work can target expanding our technique, particularly for development across various platforms such as Web and iOS, which promises to yield significant time savings in end-to-end test development and maintenance. Another possible research direction is the study of how integrating the approaches [30] that enhance app accessibility can improve the applicability of techniques such as LLMigrate on non-accessible apps.

9 Data Availability

LLMigrate's implementation and all of our research artifacts are available publicly [17].

Acknowledgments

This work has been supported, in part, by award numbers 2106871, 2106306, and 2211790 from the U.S. National Science Foundation.

References

[1] 2024. DODuae - Women's Online Store. https://tinyurl.com/mu5zkenz.
[2] 2024. Email - Fast & Secure Mail. https://tinyurl.com/53ensprk.
[3] 2024. Firefox Fast & Private Browser. https://tinyurl.com/yc5t5tkh.
[4] 2024. To Do List. https://tinyurl.com/c8chz4fb.
[5] 2024. Zalando – Online Fashion Store. https://tinyurl.com/mpee366u.
[6] 2025. 200TV - Live TV Movies App. https://tinyurl.com/3xf6wb8v.
[7] 2025. Ava Assistant - Movies & Shows. https://tinyurl.com/4e2pzxry.
[8] 2025. Cash Loan EMI Calcualtor. https://tinyurl.com/2tb59nr7.
[9] 2025. Chatbot - AI Smart Assistant. https://tinyurl.com/vk4t739r.
[10] 2025. ClipFix: Movie Shazam. https://tinyurl.com/avjp9bzv.
[11] 2025. Color SMS: Message & Messenger. https://tinyurl.com/428xy74b.
[12] 2025. Daily Notes - Easy Notebook. https://tinyurl.com/rrr9j5ee.
[13] 2025. Deep Search - AI Chatbot. https://tinyurl.com/5c68xzdt.
[14] 2025. EMI Calculator & Financial. https://tinyurl.com/2kbtpd25.
[15] 2025. EMI Calculator : Loan Planner. https://tinyurl.com/2vsb3h3y.
[16] 2025. Google Gemini. https://tinyurl.com/nhe8hpty.
[17] 2025. LLMigrate open-source repository. https://github.com/seal-hub/llmigrate.
[18] 2025. Messages for SMS - DUAL SIM. https://tinyurl.com/mrd2rdus.
[19] 2025. Messages: Text SMS. https://tinyurl.com/bdfhk5xk.
[20] 2025. Notes - QuickNotes. https://tinyurl.com/mue2jyau.
[21] 2025. Personal notes and tasks. https://tinyurl.com/393fn3ww.
[22] Toufique Ahmed and Premkumar Devanbu. 2022. Few-shot training LLMs for project-specific code-summarization. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. 1–5. https://doi.org/10.1145/3551349.3559555
[23] Domenico Amalfitano et al. 2012. Using GUI ripping for automated testing of Android applications. In *2012 Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 258–261.
[24] D. Amalfitano, A. R. Fasolino, P. Tramontana, B. D. Ta, and A. M. Memon. 2015. MobiGUITAR: Automated Model-Based Testing of Mobile Apps. *IEEE Software* 32, 5 (Sept 2015), 53–59. https://doi.org/10.1109/MS.2014.55
[25] Saswat Anand et al. 2012. Automated Concolic Testing of Smartphone Apps. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering* (Cary, North Carolina) *(FSE '12)*. ACM, New York, NY, USA, Article 59, 11 pages. https://doi.org/10.1145/2393596.2393666
[26] Farnaz Behrang and Alessandro Orso. 2018. Test migration for efficient large-scale assessment of mobile app coding assignments. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. https://doi.org/10.1145/3213846.3213854
[27] Farnaz Behrang and Alessandro Orso. 2019. Test Migration Between Mobile Apps with Similar Functionality. In *34th International Conference on Automated Software Engineering (ASE 2019)*.
[28] Farnaz Behrang and Alessandro Orso. 2019. To appear.. Test Migration Between Mobile Apps with Similar Functionality. In *Proceedings of the The 34th IEEE/ACM International Conference on Automated Software Engineering* (San Diego, USA) *(ASE '19)*. https://doi.org/10.1109/ASE.2019.00016
[29] Islem Bouzenia et al. 2024. Repairagent: An autonomous, llm-based agent for program repair. *arXiv preprint arXiv:2403.17134* (2024).
[30] Jieshan Chen et al. 2020. Unblind your apps: Predicting natural-language labels for mobile gui components by deep learning. In *Proceedings of the ACM/IEEE 42nd international conference on software engineering*. 322–334. https://doi.org/10.1145/3377811.3380327
[31] Wontae Choi, George Necula, and Koushik Sen. 2013. Guided GUI Testing of Android Apps with Minimal Restart and Approximate Learning. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications* (Indianapolis, Indiana, USA) *(OOPSLA '13)*. ACM, New York, NY, USA, 623–640. https://doi.org/10.1145/2509136.2509552
[32] World Wide Web Consortium. 2025. https://www.w3.org/TR/webdriver/.
[33] Appium Contributors. [n. d.]. Appium. https://github.com/appium/appium.
[34] Xiang Deng et al. 2024. Mind2web: Towards a generalist agent for the web. *Advances in Neural Information Processing Systems* 36 (2024). https://doi.org/10.48550/arXiv.2306.06070
[35] Zhen Dong et al. 2020. Time-travel testing of Android apps. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE, 481–492. https://doi.org/10.1145/3377811.3380402
[36] Markus Ermuth and Michael Pradel. 2016. Monkey See, Monkey Do: Effective Generation of GUI Tests with Inferred Macro Events. In *Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA 2016)*. ACM,

New York, NY, USA, 82–93. https://doi.org/10.1145/2931037.2931053
[37] Sidong Feng and Chunyang Chen. 2024. Prompting is all you need: Automated android bug replay with large language models. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*. 1–13. https://doi.org/10.1145/3597503.3608137
[38] Yi Gao et al. 2024. MUT: Human-in-the-Loop Unit Test Migration. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–12. https://doi.org/10.1145/3597503.3639124
[39] Tianxiao Gu et al. 2019. Practical GUI testing of Android applications via model abstraction and refinement. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 269–280. https://doi.org/10.1109/icse.2019.00042
[40] Izzeddin Gur et al. 2022. Understanding html with large language models. *arXiv preprint arXiv:2210.03945* (2022). https://doi.org/10.18653/v1/2023.findings-emnlp.185
[41] Izzeddin Gur et al. 2023. A real-world webagent with planning, long context understanding, and program synthesis. *arXiv preprint arXiv:2307.12856* (2023).
[42] Roman Haas et al. 2021. How can manual testing processes be optimized? developer survey, optimization guidelines, and case studies. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1281–1291. https://doi.org/10.1145/3468264.3473922
[43] Shuai Hao, Bin Liu, Suman Nath, William G.J. Halfond, and Ramesh Govindan. 2014. PUMA: Programmable UI-automation for Large-scale Dynamic Analysis of Mobile Apps. In *Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services* (Bretton Woods, New Hampshire, USA) *(MobiSys '14)*. ACM, New York, NY, USA, 204–217. https://doi.org/10.1145/2594368.2594390
[44] Gang Hu et al. 2018. AppFlow: using machine learning to synthesize robust, reusable UI tests. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 269–282. https://doi.org/10.1145/3236024.3236055
[45] Yuchao Huang et al. 2024. Crashtranslator: Automatically reproducing mobile application crashes directly from stack trace. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*. 1–13. https://doi.org/10.1145/3597503.3623298
[46] Casper S. Jensen, Mukul R. Prasad, and Anders Møller. 2013. Automated Testing with Targeted Event Sequence Generation. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis* (Lugano, Switzerland) *(ISSTA 2013)*. ACM, New York, NY, USA, 67–77. https://doi.org/10.1145/2483760.2483777
[47] Ruihua Ji et al. 2023. Vision-Based Widget Mapping for Test Migration Across Mobile Platforms: Are We There Yet?. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1416–1428. https://doi.org/10.1109/ASE56229.2023.00068
[48] Bangyan Ju et al. 2024. A Study of Using Multimodal LLMs for Non-Crash Functional Bug Detection in Android Apps. *arXiv preprint arXiv:2407.19053* (2024).
[49] Jouko Kaasila et al. 2012. Testdroid: automated remote UI testing on Android. In *Proceedings of the 11th International Conference on Mobile and Ubiquitous Multimedia*. 1–4. https://doi.org/10.1145/2406367.2406402
[50] Sungmin Kang et al. 2023. Evaluating Diverse Large Language Models for Automatic and General Bug Reproduction. *arXiv preprint arXiv:2311.04532* (2023).
[51] Sungmin Kang et al. 2023. Large language models are few-shot testers: Exploring llm-based general bug reproduction. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2312–2323. https://doi.org/ICSE48619.2023.00194
[52] Farideh Khalili et al. 2024. Semantic matching in GUI test reuse. *Empirical Software Engineering* 29, 3 (2024), 1–58. https://doi.org/10.1007/s10664-023-10406-8
[53] Pavneet Singh Kochhar et al. 2015. Understanding the Test Automation Culture of App Developers. In *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*. 1–10. https://doi.org/10.1109/ICST.2015.7102609
[54] Yavuz Koroglu et al. 2018. QBE: QLearning-based exploration of android applications. In *Software Testing, Verification and Validation (ICST), 2018 IEEE 11th International Conference on*. IEEE, 105–115. https://doi.org/10.1109/ICST.2018.00020
[55] Firebase Test Lab. 2024. Robo test (Android). https://firebase.google.com/docs/test-lab/android/robo-ux-test.
[56] Vladimir I Levenshtein. 1966. Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet physics doklady*, Vol. 10. 707–710.
[57] Kanglin Li and Mengqi Wu. 2006. *Effective GUI testing automation: Developing an automated GUI testing tool*. John Wiley & Sons.
[58] Jun-Wei Lin et al. 2019. Test Transfer Across Mobile Apps Through Semantic Mapping. In *34th International Conference on Automated Software Engineering (ASE 2019)*. https://doi.org/10.1109/ASE.2019.00015

[59] Jun-Wei Lin, Reyhaneh Jabbarvand, and Sam Malek. 2019. Test Transfer Across Mobile Apps Through Semantic Mapping. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 42–53. https://doi.org/10.1109/ASE.2019.00015
[60] Jun-Wei Lin and Sam Malek. 2022. Gui test transfer from web to android. In *2022 IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 1–11. https://doi.org/10.1109/ICST53961.2022.00011
[61] Mario Linares-Vásquez et al. 2015. Mining Android App Usages for Generating Actionable GUI-based Execution Scenarios. In *Proceedings of the 12th Working Conference on Mining Software Repositories* (Florence, Italy) *(MSR '15)*. IEEE Press, Piscataway, NJ, USA, 111–122. http://dl.acm.org/citation.cfm?id=2820518.2820534
[62] Mario Linares-Vásquez et al. 2017. How do developers test android applications?. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. https://doi.org/10.48550/arXiv.1801.06268
[63] Jiawei Liu et al. 2024. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *Advances in Neural Information Processing Systems* 36 (2024). https://doi.org/10.5555/3666122.3667065
[64] Peng Liu et al. 2017. Automatic Text Input Generation for Mobile Testing. In *Proceedings of the 39th International Conference on Software Engineering* (Buenos Aires, Argentina) *(ICSE '17)*. IEEE Press, Piscataway, NJ, USA, 643–653. https://doi.org/10.1109/ICSE.2017.65
[65] Shuqi Liu et al. 2022. Test reuse based on adaptive semantic matching across android mobile applications. In *2022 IEEE 22nd International Conference on Software Quality, Reliability and Security (QRS)*. IEEE, 703–709. https://doi.org/10.1109/QRS57517.2022.00076
[66] Shuqi Liu et al. 2024. Enhancing test reuse with GUI events deduplication and adaptive semantic matching. *Science of Computer Programming* 232 (2024), 103052. https://doi.org/10.1016/j.scico.2023.103052
[67] Zhe Liu et al. 2023. Fill in the blank: Context-aware automated text input generation for mobile gui testing. In 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE). *IEEE, 1355ś1367* (2023). https://doi.org/10.1109/ICSE48619.2023.00119
[68] Zhe Liu et al. 2024. Make llm a testing expert: Bringing human-like interaction to mobile gui testing via functionality-aware decisions. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–13. https://doi.org/10.1145/3597503.3639180
[69] Zhe Liu et al. 2024. Vision-driven Automated Mobile GUI Testing via Multimodal Large Language Model. *arXiv preprint arXiv:2407.03037* (2024).
[70] Aravind Machiry, Rohan Tahiliani, and Mayur Naik. 2013. Dynodroid: An Input Generation System for Android Apps. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering* (Saint Petersburg, Russia) *(ESEC/FSE 2013)*. ACM, New York, NY, USA, 224–234. https://doi.org/10.1145/2491411.2491450
[71] Riyadh Mahmood et al. 2014. EvoDroid: Segmented Evolutionary Testing of Android Apps. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Hong Kong, China) *(FSE 2014)*. ACM, New York, NY, USA, 599–609. https://doi.org/10.1145/2635868.2635896
[72] Ke Mao et al. 2016. Sapienz: Multi-objective automated testing for android applications. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*. 94–105. https://doi.org/10.1145/2931037.2931054
[73] Leonardo Mariani et al. 2018. Augusto: Exploiting popular functionalities for the generation of semantic gui tests with oracles. In *Proceedings of the 40th International Conference on Software Engineering*. 280–290. https://doi.org/10.1145/3180155.3180162
[74] Leonardo Mariani et al. 2021. An Evolutionary Approach to Adapt Tests Across Mobile Apps. In *The 2nd ACM/IEEE International Conference on Automation of Software Test (AST 2021)*. https://doi.org/10.1109/AST52587.2021.00016
[75] N. Mirzaei et al. 2015. SIG-Droid: Automated system input generation for Android applications. In *2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE)*. 461–471. https://doi.org/10.1109/ISSRE.2015.7381839
[76] Yash Mishra et al. 2023. Image Understanding of GUI Widgets for Test Reuse. In *2023 3rd International Conference on Pervasive Computing and Social Networking (ICPCSN)*. IEEE, 572–579. https://doi.org/10.1109/icpcsn58827.2023.00100
[77] Kevin Moran et al. 2016. Automatically Discovering, Reporting and Reproducing Android Application Crashes. In *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. 33–44. https://doi.org/10.1109/ICST.2016.34
[78] OpenAI. [n. d.]. GPT-4o. https://platform.openai.com/docs/models/gpt-4o/.
[79] Xue Qin, Hao Zhong, and Xiaoyin Wang. 2019. TestMig: Migrating GUI Test Cases from iOS to Android. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Beijing, China) *(ISSTA 2019)*. ACM, New York, NY, USA, 284–295. https://doi.org/10.1145/3293882.3330575
[80] Andreas Rau et al. 2018. Transferring Tests Across Web Applications. In *Web Engineering*, Tommi Mikkonen, Ralf Klamma, and Juan Hernández (Eds.). Springer International Publishing, Cham, 50–64. https://doi.org/10.1007/978-3-319-91662-0_4

[81] Ting Su et al. 2017. Guided, stochastic model-based GUI testing of Android apps. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. 245–256. https://doi.org/10.1145/3106237.3106298

[82] Ting Su et al. 2017. Guided, Stochastic Model-based GUI Testing of Android Apps. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering* (Paderborn, Germany) *(ESEC/FSE 2017)*. ACM, New York, NY, USA, 245–256. https://doi.org/10.1145/3106237.3106298

[83] Haotian Sun et al. 2024. Adaplanner: Adaptive planning from feedback with language models. *Advances in Neural Information Processing Systems* 36 (2024).

[84] Saghar Talebipour et al. 2022. UI test migration across mobile platforms. In *Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering* (Melbourne, Australia) *(ASE '21)*. IEEE Press, 756–767. https://doi.org/10.1109/ASE51524.2021.9678643

[85] Appium Team. 2024. Appium UiAutomator2 Driver. https://github.com/appium/appium-uiautomator2-driver.

[86] Android Studio Team. 2023. UI/Application Exerciser Monkey. https://developer.android.com/studio/test/monkey.

[87] Dingbang Wang et al. 2024. Feedback-driven automated whole bug report reproduction for android apps. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 1048–1060. https://doi.org/10.1145/3650212.3680341

[88] Jue Wang et al. 2020. ComboDroid: generating high-quality test inputs for Android apps via use case combinations. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 469–480. https://doi.org/10.1145/3377811.3380382

[89] Junjie Wang et al. 2024. Software testing with large language models: Survey, landscape, and vision. *IEEE Transactions on Software Engineering* (2024). https://doi.org/10.1109/TSE.2024.3368208

[90] Jason Wei et al. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems* 35 (2022), 24824–24837. https://doi.org/10.5555/3600270.3602070

[91] Hao Wen et al. 2023. Droidbot-gpt: Gpt-powered ui automation for android. *arXiv preprint arXiv:2304.07061* (2023).

[92] Wei Yang et al. 2013. A Grey-Box Approach for Automated GUI-Model Generation of Mobile Applications. In *Fundamental Approaches to Software Engineering*. Springer Berlin Heidelberg, 250–265.

[93] Juyeon Yoon et al. 2023. Autonomous Large Language Model Agents Enabling Intent-Driven Mobile GUI Testing. *arXiv preprint arXiv:2311.08649* (2023).

[94] Shengcheng Yu et al. 2023. Llm for test script generation and migration: Challenges, capabilities, and opportunities. In *2023 IEEE 23rd International Conference on Software Quality, Reliability, and Security (QRS)*. IEEE, 206–217. https://doi.org/10.1109/QRS60937.2023.00029

[95] Razieh Nokhbeh Zaeem et al. 2014. Automated Generation of Oracles for Testing User-Interaction Features of Mobile Apps. In *Proceedings of the 2014 IEEE International Conference on Software Testing, Verification, and Validation (ICST '14)*. IEEE Computer Society, Washington, DC, USA, 183–192. https://doi.org/10.1109/ICST.2014.31

[96] Hailong Zhang and Atanas Rountev. 2017. Analysis and Testing of Notifications in Android Wear Applications. In *Proceedings of the 39th International Conference on Software Engineering* (Buenos Aires, Argentina) *(ICSE '17)*. IEEE Press, Piscataway, NJ, USA, 347–357. https://doi.org/10.1109/ICSE.2017.39

[97] Yakun Zhang et al. 2024. Learning-based Widget Matching for Migrating GUI Test Cases. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering* (Lisbon, Portugal) *(ICSE '24)*. Association for Computing Machinery, New York, NY, USA, Article 69, 13 pages. https://doi.org/10.1145/3597503.3623322

[98] Yakun Zhang et al. 2024. LLM-based Abstraction and Concretization for GUI Test Migration. arXiv:2409.05028 [cs.SE] https://arxiv.org/abs/2409.05028

[99] Yakun Zhang et al. 2024. Synthesis-Based Enhancement for GUI Test Case Migration. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 869–881. https://doi.org/10.1145/3650212.3680327

[100] Yu Zhao et al. 2019. Automatically extracting bug reproducing steps from android bug reports. In *Reuse in the Big Data Era: 18th International Conference on Software and Systems Reuse, ICSR 2019, Cincinnati, OH, USA, June 26–28, 2019, Proceedings 18*. Springer, 100–111. https://doi.org/10.1007/978-3-030-22888-0_8

[101] Yu Zhao et al. 2019. Recdroid: automatically reproducing android application crashes from bug reports. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 128–139. https://doi.org/10.1109/ICSE.2019.00030

[102] Yixue Zhao et al. 2020. FrUITeR: a framework for evaluating UI test reuse. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '20)*. ACM. https://doi.org/10.1145/3368089.3409708

[103] Boyuan Zheng et al. 2024. Gpt-4v (ision) is a generalist web agent, if grounded. *arXiv preprint arXiv:2401.01614* (2024).

Received 2024-10-31; accepted 2025-03-31