

POLITECNICO DI TORINO

Master's Degree in Computer Engineering



Master Degree Thesis

Adaptive GUI test evolution and oracle maintenance in Android

Supervisors

Prof. Riccardo COPPOLA

Dott. Tommaso FULCINI

Candidate

Alessandro FEDRIGA

July 2025

Summary

Automated GUI testing is essential in contemporary Android development, yet it is notoriously fragile, particularly when utilizing dynamic frameworks like Jetpack Compose. Even minor changes to the UI frequently cause test failures, resulting in continuous integration pipelines being overwhelmed with false positives and developers facing expensive maintenance efforts. This thesis introduces a developer-in-the-loop approach leveraging Large Language Models (LLMs)—specifically GitHub Copilot combined with Claude 3.7 Sonnet—as a zero-shot test repair agent embedded within Android Studio. By examining IDE context, this strategy effectively repairs broken selectors, modifies test oracles, and preserves test logic after GUI modifications. An empirical assessment of this technique was conducted on the Bitwarden for Android mobile application, an open-source project with 1083 GUI tests. The analysis covered two recent versions of the application, identifying test failures and applying the LLM to address these issues. The study investigates test fragility (RQ1), the effectiveness of zero-shot LLM-based repairs (RQ2), additional benefits from retrying prompts (RQ3), and improvements resulting from brief developer interactions (RQ4). Findings demonstrate that a single zero-shot prompt can restore a notable number of failing tests, greatly decreasing manual repair efforts. Additional prompt retries yielded marginal further improvements, whereas limited developer intervention significantly boosted recovery effectiveness. These results highlight that incorporating LLM-driven methods substantially reduces the maintenance workload associated with GUI test suites, improving resilience to frequent UI changes in Jetpack Compose applications. Subsequent research will continue refining LLM integration and examining its broader practical applications.

Acknowledgements

Thanks to my parents for always being there and funding all of this.

Big shout-out to my brother Davide, my unofficial butler, for cooking and cleaning, and to my cousin Gabriele for listening to every one of my rants along the way.

To my supervisors, Riccardo Coppola and Tommaso Fulcini: thanks for the guidance and feedback that made this possible.

Finally, thanks to all my friends and extended family for the support and good times throughout this journey.

C'è sempre da tribolare

- Nonno Gianni

Table of Contents

List of Tables	7
List of Figures	8
1 Introduction	14
2 Background	18
2.1 GUI testing fragility	18
2.1.1 Core causes of fragile GUI tests	18
2.1.2 GUI testing fragility for Android apps using Jetpack Compose	19
2.1.3 Progress so far: addressing Android testing fragility	20
2.2 State-of-the-art in oracle testing	21
2.2.1 The oracle problem	21
2.2.2 Techniques for oracle improvement	21
2.2.3 Human-in-the-loop oracle creation and validation	21
2.2.4 External vs Internal oracles	22
2.3 Large Language Models	22
2.3.1 Architectural principles	23
2.3.2 Training methodologies	24
2.3.3 Prompt Engineering	24
2.4 LLMs for GUI Testing	25
2.4.1 A memory-driven LLM approach: GPTDroid	25
2.4.2 Mapping natural-language test descriptions to GUI actions	25
2.4.3 Context-aware text input generation: QTypist	26
2.4.4 LLM-driven GUI testing in engineering software: GERALLT	27
2.4.5 Intent-based testing for Android: DROIDAGENT	28
2.4.6 Beyond GUI: Broader LLM-based Software Testing Approaches	28
2.4.7 Summary and Outlook	29
3 Methodology	30
3.1 Goal-Question-Metric	30
3.2 Research Questions	30
3.3 Project Selection	31
3.4 Test Case Extraction	32
3.5 Operational Environment	33

4	Results	36
4.1	RQ1 - Baseline Suite Fragility	38
4.2	RQ2 - Zero-Shot LLM Repair Effectiveness	38
4.3	RQ3 - Zero-Shot Retry Gain	38
4.4	RQ4 - Chain-of-Thought Repair Effectiveness	39
5	Discussion	40
6	Conclusion	42
6.1	Recap of Purpose and Method	42
6.2	Main Findings	42
6.3	Implications for Practice	43
6.4	Contributions	43
7	Future Works	44
7.1	Broader and Deeper Empirical Replication	44
7.2	Fine-Grained, Commit-Level Evolution	44
7.3	Advanced Prompt Engineering and Retrieval	44
7.4	LLM Agent Architectures	45
7.5	Beyond Repair: Generation and Oracle Evolution	45
	Bibliography	46

List of Tables

3.1	GQM table for the study	30
4.1	Distribution of test case outcomes (Passing P , Failing F , Broken B) per research questions (only test classes with failing or broken test cases are present).	37
6.1	Evolution of test pass rates across repair stages.	42

List of Figures

2.1	Composable function sample	19
2.2	Simplified Transforer architecture	23
2.3	GPTDroid	26
2.4	QTypist	27
2.5	GERALLT	27
2.6	DROIDAGENT	28
3.1	LLM repair pipeline	33
4.1	Cumulative test outcomes per RQs	36
5.1	Sankey Chart of the test cases through the RQs	40

Listings

3.1	Example GUI Test case	32
3.2	Example GitHub Copilot response zero-shot	34
3.3	Example GitHub Copilot response CoT - Identify the issue	34
3.4	Example GitHub Copilot response CoT - Provide a fix	35

Glossary

Capture & Replay A testing technique that records (captures) user interactions with the GUI, then later replays these interactions automatically.

Composable Function A fundamental building block in Jetpack Compose that declares part of the user interface in Kotlin code. Each composable manages its own state and can be nested or combined to form complex UIs.

Compose Testing API A set of tools provided by Jetpack Compose for writing UI tests in Kotlin. It allows developers to simulate user interactions, assert UI properties, and verify behavior of composable functions in isolation or as part of a complete screen.

Failed Error Propagation A situation in which an internal error in a program does not manifest in the external output or behavior.

Flakiness A property of tests that yield inconsistent results across repeated runs, even when the tested code does not change.

Gamification The integration of game design elements—such as scoring systems, badges, and leaderboards—into non-game contexts to enhance user engagement, motivation, and participation.

Jetpack Compose A modern, declarative UI toolkit for Android that uses composable functions to define and render interface elements.

Mutation Testing A method for assessing test quality and improving oracles by introducing artificial faults (mutants) into the code.

ViewModel A lifecycle-aware component in Android’s architecture that stores and manages UI-related data, survives configuration changes, and helps separate business logic from UI controllers such as activities, fragments, or composables.

Acronyms

AI

Artificial Intelligence.

API

Application Programming Interface.

AUT

Application Under Test.

CI

Continuous Integration.

CoT

Chain-of-Thought.

FQN

Fully Qualified Name.

GPT

Generative Pre-training Transformer.

GQM

Goal-Question-Metric.

GUI

Graphical User Interface.

LLM

Large Language Model.

OS

Operative System.

Q&A

Question & Answer.

RQ

Research Question.

SUT

System Under Test.

UI

User Interface.

Chapter 1

Introduction

Software testing is the process of operating a system or component under specified conditions, observing or recording the results, and evaluating various aspects of the system or component [1]. It is widely recognized as a cornerstone of software development, playing a pivotal role in ensuring reliability, robustness, functionality, and user satisfaction throughout an application’s lifecycle [2, 3]. Effective testing not only catches defects early but also helps prevent issues from escalating, thereby reducing development costs, improving maintainability, and extending software longevity [2, 3].

Among the broad spectrum of testing methodologies, Graphical User Interface (GUI) testing is of particular importance because it directly evaluates how end-users interact with the visible elements of an application. GUI testing verifies that user-facing components function as intended, respond appropriately to user actions, and align with design and functional requirements [4]. In addition, GUI testing uncovers usability and accessibility concerns, facilitating iterative improvements in interface design and overall user experience.

Despite its importance, GUI testing for modern mobile platforms—and Android in particular—remains *notoriously fragile* [4–6]. Minor modifications—such as relocating a button, renaming a Composable Function, or shifting a ViewModel field—routinely break tests, flooding Continuous Integration (CI) pipelines with false alarms and forcing developers into an expensive repair cycle [5–7]. The challenge is aggravated by (i) Android’s fragmented device landscape [7], (ii) Jetpack Compose’s highly dynamic, state-driven rendering paradigm [8], and (iii) the *oracle problem*, whereby the expected outcome of a test must evolve in lock-step with a changing GUI [3, 9]. Together, these factors create a moving target in which even well-engineered suites quickly drift out of sync with the product they are meant to test and protect.

To tackle this long-standing problem, the software-testing community has proposed a wealth of techniques ranging from robust oracles to automatic locator updates and visual comparison heuristics [6, 10]. However, those techniques typically treat tests as static artifacts whose fragility must be anticipated at design time. In contrast, this thesis explores a *lightweight, developer-in-the-loop workflow* that leverages recent advances in Large Language Models (LLMs) to **heal fragile Android GUI tests *after* they break**, thus keeping their code, selectors, and oracles up-to-date with new versions of the Application Under Test (AUT).

Motivation and Challenges

Android’s Fragmented Ecosystem

Android [11]—Google’s mobile operating system—runs on an extraordinarily heterogeneous set of devices, including smartphones, tablets, wearables, televisions, and even automotive infotainment systems [12]. Released under the permissive Apache License 2.0 [13], Android allows manufacturers and developers to freely modify and extend the platform. While this flexibility fosters innovation, it also produces a highly fragmented ecosystem: different devices may run different Operative System (OS) versions, custom vendor skins, or bespoke hardware configurations. For GUI testing, such diversity translates into a combinatorial explosion of execution contexts in which fragile selectors, timing assumptions, and layout variations become primary sources of false positives and maintenance overhead [6, 7].

GUI Test Fragility

A key challenge in Android GUI testing is test fragility. Even minor changes to the interface—such as relocating a button or altering a Composable function—can break previously stable test scripts, causing false alarms and necessitating manual updates [5]. This challenge is further amplified by User Interface (UI) toolkits like Jetpack Compose, which employs declarative, state-driven rendering. Although Jetpack Compose streamlines UI development, its fluid approach to composable hierarchies can increase the volatility of test scripts that rely on rigid or outdated references. Mitigating fragility requires carefully designed oracles and testing strategies that can adapt to changing interfaces without requiring continuous human intervention.

Oracle Maintenance in GUI Testing

Oracle maintenance involves the continual evolution of the mechanism that determines whether an observed behavior matches the expected outcome. Oracles must be updated over time to reflect evolving requirements, design tweaks, or shifts in internal application logic—especially in a platform as dynamic as Android. Traditional oracles often rely on precise references (e.g., strict layout IDs or pixel-perfect comparisons) that can become invalid with even slight UI modifications [10]. Consequently, designers of GUI testing frameworks must address the so-called *oracle problem*, where achieving a balance between completeness (detecting all faults) and soundness (reporting only real faults) remains notoriously difficult.

Emergence of Large Language Models

Large Language Models—such as Generative Pre-training Transformer (GPT)-based architectures—have recently achieved remarkable performance in code completion, program synthesis, and natural-language reasoning [14]. Their ability to infer context, generate coherent code, and incorporate fresh knowledge positions them as promising allies for reducing fragility and automating oracle maintenance. By analysing source code, application state, and textual descriptions of desired behaviours, LLMs can potentially propose flexible, context-aware fixes, thereby reducing manual maintenance tasks while retaining developer oversight.

Developer-in-the-Loop LLM-Based Repair Workflow

Building on these observations, this thesis adopts a developer-in-the-loop workflow in which the engineer—assisted by GitHub Copilot and Claude 3.7 Sonnet inside Android Studio—incrementally

repairs GUI tests whenever they fail to compile or run. The process unfolds in four lightweight steps:

- (i) **context harvesting**, where error traces, neighbouring code, and relevant resources are gathered from the IDE;
- (ii) **selector healing**, which updates brittle locators (e.g., semantic tags or accessibility labels) so that they match the current Application Under Test (AUT);
- (iii) **oracle adjustment**, where expected outcomes are re-aligned with the live GUI hierarchy and assertions are patched accordingly;
- (iv) **intent preservation**, ensuring that identifiers, comments, and the original test semantics remain intact for future readability.

Applied to the Bitwarden Android application¹, this workflow restored 55% of the 1083 tests that were initially broken or failing, demonstrating that on-demand LLM assistance can be an effective, low-overhead complement to existing test-automation pipelines.

Thesis Goal and Approach

The overarching goal of this thesis is to investigate how a context-aware, LLM-based approach can mitigate fragmentation and fragility in Android GUI testing while reducing oracle-maintenance effort. Concretely, the thesis aims to:

- Design a developer-in-the-loop workflow that leverages LLMs to heal fragile selectors and oracles *post hoc*, rather than attempting to foresee every future change
- Integrate the workflow seamlessly into existing Android-Studio workflows using off-the-shelf tooling (GitHub Copilot and CLAUDE)
- Empirically assess effectiveness, stability, and human effort across multiple releases of a real-world Android application
- Provide actionable guidelines for practitioners and a roadmap for future research in LLM-driven GUI testing

The investigation is guided by the following Research Questions (RQs):

RQ1: Baseline Suite Fragility

RQ2: Zero-Shot LLM Repair Effectiveness

RQ3: Zero-Shot Retry Gain

RQ4: Chain-of-Thought Repair Effectiveness

¹<https://github.com/bitwarden/android>

Thesis Outline

Chapter 2 (Background) surveys the state of the art in Android GUI testing, the oracle problem, and recent advances in Large Language Models (LLMs). It establishes the theoretical foundations and highlights the open challenges that motivate the present work.

Chapter 3 (Methodology) details the research design. It introduces the Goal-Question-Metric (GQM) framework that structures the study, explains the selection of the Bitwarden Android app, describes the cross-version test-suite extraction process, and outlines the LLM-assisted repair pipeline evaluated in the experiments.

Chapter 4 (Results) presents the empirical findings for each of the four research questions. It reports baseline fragility, zero-shot repair effectiveness, the marginal gains from a retry strategy, and the impact of a brief chain-of-thought dialogue with the LLM.

Chapter 5 (Discussion) interprets the results, analysing their implications for practitioners and researchers. It examines factors that influence repair success—such as test-class size and context-window limits—and positions the contributions within the broader literature on automated GUI testing.

Chapter 6 (Conclusion) summarises the main contributions of the thesis, reflects on its limitations, and distils practical guidelines for integrating LLM-based repair into continuous-integration pipelines.

Finally, **Chapter 7— (Future Work)** outlines several avenues for further investigation, including broader empirical replication, commit-level longitudinal studies, advanced prompting strategies, autonomous repair agents, and oracle co-evolution.

Chapter 2

Background

This chapter outlines the related work and essential background knowledge required to understand this research on oracle maintenance for Android GUI testing leveraging LLMs.

2.1 GUI testing fragility

Modern software interfaces continuously evolve to address changing user expectations, introduce new functionalities, and align with emerging design trends. Nevertheless, the dynamic nature of such iteration introduces significant challenges to the stability and effectiveness of GUI testing efforts. Even minor modifications in GUI design, such as altering the placement of buttons or revising visual styling, can trigger failures in previously stable test scripts, increasing test fragility and the burden of maintenance [5]. Additionally, GUIs inherently support flexible user interactions, enabling users to perform numerous events in unpredictable sequences. This interactivity amplifies complexity during both test creation and maintenance, demanding consideration of various user paths that might break with small code or design updates.

2.1.1 Core causes of fragile GUI tests

GUI testing fragility often stems from the strong coupling between automated test scripts and specific visual or structural attributes of the interface. Traditional GUI testing methodologies frequently rely upon rigid, fixed locators such as hard-coded IDs, XPath expressions, coordinate-based selections, or strict visual comparisons, making test cases susceptible to interface changes. For instance, when interface components are renamed, moved, or removed during iterative development cycles, test scripts that directly reference these components immediately fail. Additionally, even subtle visual modifications—such as variations in color, iconography, or element dimensions—can invalidate tests based on pixel-perfect comparisons or rigid visual assertions [9, 15, 16].

Beyond visual elements, GUI test scripts must handle diverse interaction patterns used by end-users. Users may perform a combination of clicks, swipes, gestures, keystrokes, and multi-step workflows in varied orders, presenting difficulty for test case authors attempting to cover all possible user scenarios. Consequently, the complexity of modern interfaces, especially those presenting sophisticated interactive features, increases the probability of test fragility. For example, tests that presume a particular sequence of UI interactions or strict component hierarchies often fail after minor refactorings or enhancements. Introducing new interface components or state

transitions can produce cascading effects that inadvertently disrupt unrelated tests, significantly reducing the stability and reliability of the entire test suite [9].

2.1.2 GUI testing fragility for Android apps using Jetpack Compose

Jetpack Compose, Google’s modern and declarative UI toolkit designed explicitly for Android application development, transforms traditional interface-building practices (see Figure 2.1). By replacing XML-based layout definitions and imperative UI constructions with composable functions declared directly in Kotlin code, Jetpack Compose streamlines UI design processes and accelerates iterative development cycles [17]. While this declarative, state-driven architecture offers significant advantages, it also introduces new complexities for GUI testing strategies, particularly concerning test fragility.

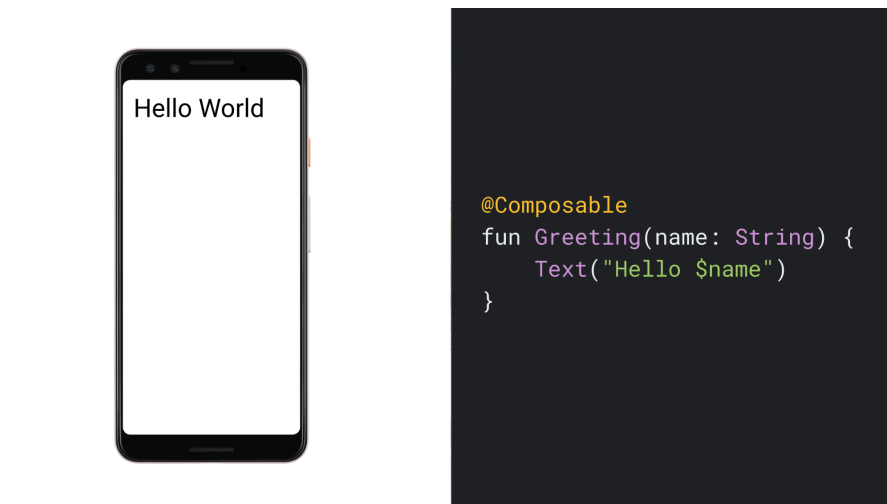


Figure 2.1: Composable function sample

Unlike traditional Android view hierarchies, Jetpack Compose renders UI dynamically based on application state, requiring testers to adopt different approaches when defining and maintaining test scripts. Test cases developed with Compose Testing API emphasize semantic properties (e.g., text labels, content descriptions) and logical node hierarchies rather than rigid visual or structural identifiers [18]. Although this semantics-based approach enhances test readability and reduces dependency on fragile visual attributes, even subtle alterations in composable function parameters or modifications of underlying state logic can result in unexpected shifts in the rendered UI hierarchy. Consequently, tests previously considered robust may suddenly become fragile, as they are tightly coupled to composable structures or semantic annotations that evolve with changing application requirements.

Moreover, Jetpack Compose encourages modularity and reusability through composable functions, which are frequently nested, combined, conditionally rendered, or rearranged during iterative refactoring [19]. While this modular composability significantly simplifies code reuse and increases productivity, it creates intricate and rapidly evolving UI structures that are challenging for testers to deal with. A minor refactor in a composable used across multiple screens can lead to cascading impacts on test scripts that depend upon the specific nesting or semantic descriptors initially established. This volatility demands that tests target stable, logically consistent properties and behaviors rather than structural details susceptible to frequent change.

As a result, while Jetpack Compose provides developers and designers with clear benefits, it can amplify GUI test fragility, especially if testing methodologies are not adapted to its paradigm. Ensuring test stability thus requires careful consideration of testing strategies aligned with Jetpack Compose’s declarative and state-driven nature. By managing semantic annotations, emphasizing stable composable behaviors, and structuring tests around resilient UI abstractions, teams can mitigate some—but not all—of the fragility risks associated with Jetpack Compose.

2.1.3 Progress so far: addressing Android testing fragility

Recent research (before Jetpack Compose became the endorsed framework by Google for developing Android apps) has made significant progress in addressing the fragility inherent to Android GUI testing. Coppola et al. [5] conducted a detailed empirical assessment of GUI test fragility for hybrid mobile applications, finding substantial maintenance burdens associated with both layout-based and visual GUI test scripts. Their study demonstrated that as much as 20% of layout-based and 30% of visual test methods required modifications at least once during application evolution, highlighting the high maintenance cost triggered by frequent, minor GUI alterations. To tackle these issues, the researchers suggested guidelines such as employing stable, semantically meaningful identifiers for UI components (which is similar to what Compose Testing API endorses), avoiding excessive reliance on visually precise locators, and regularly validating GUI elements against expected properties to minimize the sensitivity of tests to minor design modifications.

In a broader investigation into Android open-source apps, Coppola et al. [6] revealed that around 50% of changes in test code were directly attributable to modifications in GUI definitions and arrangements, emphasizing how GUI evolution significantly impacts test code maintenance and fragility. They employed a detailed taxonomy of modification causes, categorizing fragilities to better understand their sources. This taxonomy facilitated the identification of recurring patterns and specific types of GUI changes—such as adjustments in widget hierarchies, changes in the visibility or presence of UI elements, and modifications of textual content—that triggered test code maintenance. These insights are crucial for developers aiming to design robust test suites and avoid common pitfalls that lead to test fragility.

Furthermore, Coppola et al. [7] recently outlined a comprehensive conceptual framework aimed at enhancing robustness in Android GUI testing. This framework explicitly addresses critical challenges like fragmentation, fragility, and Flakiness by advocating automated maintenance, improved execution environments, and adaptive generation mechanisms. The proposed framework includes multiple integrated modules, such as automated generation and translation of test cases to adapt to GUI changes, advanced execution environments that handle various hardware and software configurations, and intelligent test maintenance and repair modules. These repair mechanisms rely on model-based and heuristic approaches to automatically adjust and update test scripts in response to GUI modifications, significantly reducing manual maintenance efforts.

The conceptual framework also proposes leveraging *Capture & Replay* techniques enhanced with Gamification elements to motivate testers, automated model-based test sequence generation that intelligently adapts to GUI structures, and Artificial Intelligence (AI)-driven translation of general test sequences into specific cases suitable for individual applications. The combination of visual and layout-based test cases allows cross-validation between structural and visual aspects of the GUI, increasing coverage and test resilience. Additionally, virtualized test environments are proposed to systematically handle software and hardware fragmentation, supporting repeated test execution across diverse configurations. The framework further suggests the inclusion of advanced analytics to detect and predict flaky tests, integrating fragility metrics directly into the continuous integration pipeline to optimize test prioritization and maintenance strategies.

Collectively, these works underscore ongoing efforts to mitigate Android GUI test fragility through empirical assessment, test design guidelines, automated test repair strategies, and comprehensive conceptual frameworks designed to support continuous integration environments. Such advancements promise to streamline GUI testing practices, reduce maintenance overheads, and improve the reliability of automated testing suites, making Android application development more sustainable and efficient.

2.2 State-of-the-art in oracle testing

Test oracles serve as the reference mechanism for verifying that the System Under Test (SUT) behaves as intended. They can be derived from formal specifications, program documentation, or inferred dynamically from system behavior. Key properties of a robust oracle include completeness, that is detecting all faults, and soundness, namely reporting only actual faults. However, achieving these properties is notoriously difficult in practice. For instance, a significant challenge arises from the phenomenon known as Failed Error Propagation [10].

2.2.1 The oracle problem

The oracle problem remains one of the most challenging bottlenecks in automated software testing. As noted by Gunel Jahangirova [10], a high-quality test oracle must strike a delicate balance between generality and accuracy. In other words, an ideal oracle is one that can be applied broadly across many test cases while reliably distinguishing correct behavior from faults. Poorly designed oracles can lead to two critical issues: false negatives, where faults go undetected, and false positives, where correct behavior is mistakenly flagged as erroneous—both of which undermine the efficiency and cost-effectiveness of automated testing.

2.2.2 Techniques for oracle improvement

Recent research has focused on addressing oracle deficiencies through a variety of innovative approaches. One prominent method involves using search-based test case generation combined with Mutation Testing to systematically expose weaknesses in existing oracles. This approach transforms the problem into one where the generation of counterexamples (i.e., test cases that either trigger false positives or false negatives) guides the iterative improvement of the oracle. By incorporating a human in the loop, where developers review the counterexamples and adjust the oracle accordingly, this technique has been shown to significantly enhance fault detection capabilities [10].

2.2.3 Human-in-the-loop oracle creation and validation

Despite advances in automation, human insight remains crucial in defining and validating test oracles, especially for complex GUI behaviors. A growing body of work advocates human-in-the-loop approaches, where automated tools assist in oracle creation but a human tester or developer provides guidance or confirmation to ensure correctness [20–22]. This stems from the recognition that many oracle decisions require domain understanding or subjective judgment that humans excel at. For instance, Zamprogno et al. [20] present *AutoAssert*, a tool that automatically suggests assertions for unit tests and lets the developer review and accept or modify them. This demonstrates how human-in-the-loop oracle generation can reduce manual effort while leveraging human expertise to vet the machine’s output.

The incorporation of human feedback is even more vital when using LLMs for oracle generation. LLMs can propose expected responses or validation rules, but they may occasionally produce assertions that capture the program’s actual behavior rather than the intended behavior. Therefore, having a human in the loop to judge the validity of LLM-proposed oracles is important to avoid false oracles and missed bugs [22].

Another human-in-the-loop strategy in testing is crowdsourced oracle validation, where crowd workers review automatically generated test cases by evaluating individual assertions against the software’s documentation. They decide whether each assertion correctly represents the intended behavior, mark it as indeterminate when documentation is insufficient, or identify it as wrong. If an assertion is found to be incorrect, workers are also tasked with providing a corrected version. This process leverages human judgment to bridge the gap left by automated test generation, effectively addressing the oracle problem in software testing [21].

2.2.4 External vs Internal oracles

Another key aspect of oracle maintenance is the strategic placement of oracles within the code.

External oracles, often implemented as test case oracles, operate by comparing only the externally observable outputs and side effects of a method. Because they rely solely on visible states—such as return values, changes to global variables, or modifications in persistent storage—these oracles may fail to detect faults that only affect the internal state of a program. For instance, if an error causes a subtle corruption within a method’s intermediate variables yet the externally observable behavior remains unchanged, an external oracle might not detect the fault. In such cases, the issue is said to involve external failed error propagation, where the internal discrepancy does not manifest itself in the output. This limitation often necessitates a large number of test cases to eventually expose the fault, especially when the probability of the error propagating outward is low [10].

In contrast, internal oracles—such as those placed at return points or other critical internal checkpoints—are given direct access to the program’s complete execution state at strategic moments. This enables them to assess not only the externally observable effects but also the nuances of the internal state. By examining all aspects of the program’s execution, internal oracles can detect subtle faults that might be missed by their external counterparts. However, the very advantage of internal oracles comes with the caveat of increased complexity. Creating these oracles manually requires a deep understanding of the intended internal state transitions and the interplay between different components of the system. Additionally, even internal oracles can sometimes be subject to a phenomenon known as internal failed error propagation—where differences in the internal state are “squeezed out” as execution progresses—thus complicating their effective design [10].

Ultimately, the challenge lies in striking the optimal balance: achieving maximal fault detection by placing internal oracles while minimizing the manual effort required for their precise definition. This trade-off is a key focus of ongoing research, as it addresses the dual goals of enhancing testing effectiveness and maintaining practical development efforts [10].

2.3 Large Language Models

LLMs have emerged as a transformative advancement within artificial intelligence and natural language processing, capable of performing complex linguistic and cognitive tasks through deep learning techniques [14]. These models leverage deep neural networks trained on massive and diverse datasets containing billions of words, enabling them to capture linguistic patterns, contextual nuances, and semantic relationships. Their ability to understand context, infer intent,

and generate coherent text has resulted in numerous successful applications. In particular, code generation stands out as an increasingly prominent application of LLMs, which is important to the scope and goals of this thesis.

The significance of code generation arises from its potential to automate and streamline software development tasks, thereby reducing manual coding efforts, accelerating prototyping phases, and facilitating rapid iterative software delivery. Given the previous discussion about challenges associated with traditional GUI testing and maintenance, the prospect of exploiting LLMs to generate adaptable, context-aware testing scripts presents a clear opportunity.

2.3.1 Architectural principles

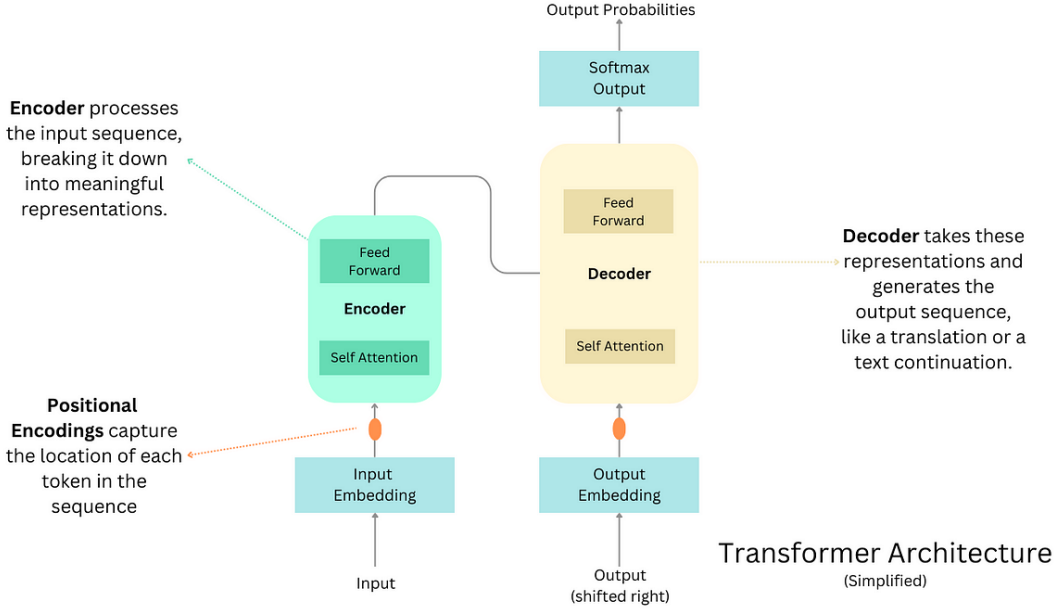


Figure 2.2: Simplified Transformer architecture

At the core of contemporary LLMs is the Transformer architecture (see Figure 2.2, an innovative deep learning structure proposed by Vaswani et al. [23]). This architecture fundamentally departs from traditional recurrent neural networks [24], which processed input sequences sequentially and faced inherent limitations when modeling long-range dependencies. Transformers, on the other hand, employ self-attention mechanisms—allowing each token (e.g., word, character, or sub-word fragment) to directly reference and weigh the relevance of every other token in the input sequence simultaneously. This mechanism enables the model to efficiently capture long-range contextual relationships and dependencies between words, significantly enhancing its understanding of complex sentence structures, semantics, and nuances in meaning.

Moreover, Transformer-based models exhibit remarkable scalability, benefiting from increased computational resources and model parameter counts. As parameter counts grow, the models become better in their tasks. This scalability has driven recent advancements in all the most popular LLMs such as the GPT-series models, Gemini, Claude, and other state-of-the-art generative models [25].

2.3.2 Training methodologies

The training of large language models primarily revolves around self-supervised learning methodologies [26]. Unlike traditional supervised learning, self-supervised approaches do not rely on extensive manual annotations; instead, these models derive linguistic and conceptual knowledge directly from massive amounts of unstructured textual data publicly available online. The predominant strategy involves training the model to predict missing tokens (words or phrases) or subsequent tokens in a given sequence of text [27]. Through this iterative process of context prediction, LLMs internalize grammatical structures, lexical relationships, semantic coherence, and contextual information, thus building comprehensive representations of language and meaning.

Following this general pre-training phase, models typically undergo task-specific fine-tuning to specialize their learned capabilities for particular application domains. In domains requiring precise outputs—such as medical diagnosis, legal interpretation, or, notably, software development—fine-tuning significantly enhances the accuracy and contextual appropriateness of model-generated outputs. In particular, fine-tuning LLMs on software repositories, source code datasets, or programming tasks has proven highly effective for automated code generation tasks [28], allowing the models to internalize syntax, semantic logic, coding patterns, best practices, and even project-specific conventions.

2.3.3 Prompt Engineering

Prompt engineering has emerged as a key technique for effectively querying LLMs. At its core, prompt engineering involves designing and refining natural language prompts to guide model behavior, maximizing the accuracy and appropriateness of outputs produced by the model [29]. Therefore, the inherent flexibility of natural language also implies sensitivity; minor changes in wording, phrasing, context, or structure can significantly influence the generated outputs.

Several main approaches to prompt engineering currently exist:

- **Zero-shot prompting** [30]: In zero-shot prompting, tasks are specified explicitly in the prompt without providing examples. The model thus relies entirely upon its pre-trained knowledge and comprehension to perform the task. While powerful, zero-shot prompting requires clarity and precision in the wording to ensure accurate interpretations by the model.
- **One-shot prompting** [31]: One-shot prompting supplements the task instruction with one example within the prompt itself. This example acts as a reference, clarifying user's expectations and enabling the model to generate output that closely aligns with the provided context. One-shot prompting often reduces ambiguity and can enhance the quality and consistency of outputs compared to zero-shot scenarios.
- **Few-shot prompting** [32]: Few-shot prompting involves providing the model with multiple examples within the prompt. This approach is particularly effective for complex tasks requiring nuanced understanding, pattern recognition, or creativity. By examining multiple contextual examples, the model can better infer patterns, conventions, or styles, improving the likelihood of generating more accurate outputs.
- **Chain-of-thought prompting** [33]: Chain-of-thought prompting makes the model detail intermediate steps and reasoning processes before arriving at a final answer, effectively following a "think aloud" process. This method enhances transparency, logical coherence, and interpretability, providing insights into how the model arrives at its conclusions. Chain-of-thought prompting has proven valuable for tasks involving logical reasoning or multi-step solutions such as code generation and debugging.

Nevertheless, prompt engineering remains an active and rapidly evolving research area. Recent studies indicate varying degrees of effectiveness depending on task complexity, domain specificity, and model size. Interestingly, some research, such as that conducted by Reynolds and McDonell [34], indicates that zero-shot prompting can sometimes yield superior results compared to few-shot or one-shot prompting.

2.4 LLMs for GUI Testing

Recent advances in LLMs have given rise to a new generation of GUI testing frameworks that leverage deep neural networks’ powerful language and code-generation capabilities. These approaches promise to address many of the shortcomings inherent in conventional GUI testing, including fragility and maintenance overhead. Unlike rigid test scripts that rely on brittle locators or simplistic model-based exploration, LLM-based frameworks can reason about a richer contextual understanding of both code (e.g., composable elements) and higher-level descriptions of user interfaces.

2.4.1 A memory-driven LLM approach: GPTDroid

Liu et al. [35] introduce *GPTDroid* (see Figure 2.3), which reframes mobile GUI testing as a dynamic, multi-turn Question & Answer (Q&A) exchange between an LLM and the application under test. Crucially, *GPTDroid* provides the LLM with detailed, evolving prompts containing local screen context (e.g., widget types, textual hints, and layout positions) as well as a “functionality-aware memory” that tracks broader testing progress, such as which features have already been explored.

This richer prompt design enables *GPTDroid* to navigate multi-step operations—like adding, editing, and deleting items in a sequence—without losing track of high-level objectives. At each step, *GPTDroid* executes the model’s recommended action in the app, then updates the prompt with observed changes (e.g., newly appeared widgets or screens), allowing it to adapt to real-time GUI variations (such as renamed or repositioned widgets). Evaluations on 93 diverse Android apps show that *GPTDroid* achieves substantially higher activity coverage (on average, 75%) compared to conventional approaches and exposes more bugs (up to 31% more), underscoring the power of a memory-driven LLM framework to handle complex, evolving interfaces.

2.4.2 Mapping natural-language test descriptions to GUI actions

Zimmermann and Koziolok [36] demonstrate a method for interpreting natural-language test cases with *GPT-3* and automatically mapping them to executable GUI actions. Their approach continually re-supplies the entire GUI state at each step—rather than relying on persistent hidden memory—so that the LLM can decide the next interaction with full contextual awareness.

In an experiment with a small desktop application (a 25-button grid), the authors constructed a dataset of natural-language test descriptions alongside corresponding GUI states and then fine-tuned *GPT-3* to execute these tests by clicking buttons in the specified sequences. Despite the relatively constrained scenario, the results showcased the feasibility of using prompt-based techniques to convert plain-English instructions into real-time GUI actions with minimal error. This proof-of-concept also points to how LLM-driven testing can lower the barrier for non-technical test writers while supporting iterative, stateful execution across diverse interface states.



Figure 2.3: GPTDroid

2.4.3 Context-aware text input generation: QTypist

Another line of work by Liu et al. [37] addresses the challenge of generating valid text inputs for mobile GUI testing. They propose a tool called *QTypist* (see Figure 2.4), which dynamically constructs prompts from the GUI's local and global context—such as widget hints, parent nodes, and activity names—and feeds them into *GPT-3* to produce realistic text inputs.

This prompt-based approach is tuned on automatically collected examples of user-provided text inputs, refining the model's ability to generate domain-specific strings (e.g., login credentials, addresses, or search terms). In evaluations with 106 real-world Android apps, *QTypist* reported an 87% pass rate on pages requiring text inputs and significantly outperformed existing baselines. When integrated into popular automated GUI-testing frameworks, *QTypist* substantially increased coverage (by up to 52% more pages, on average) and exposed nearly twice as many defects. These findings highlight how context-aware input generation can unlock deeper exploration of app states and lead to more robust, comprehensive GUI tests.

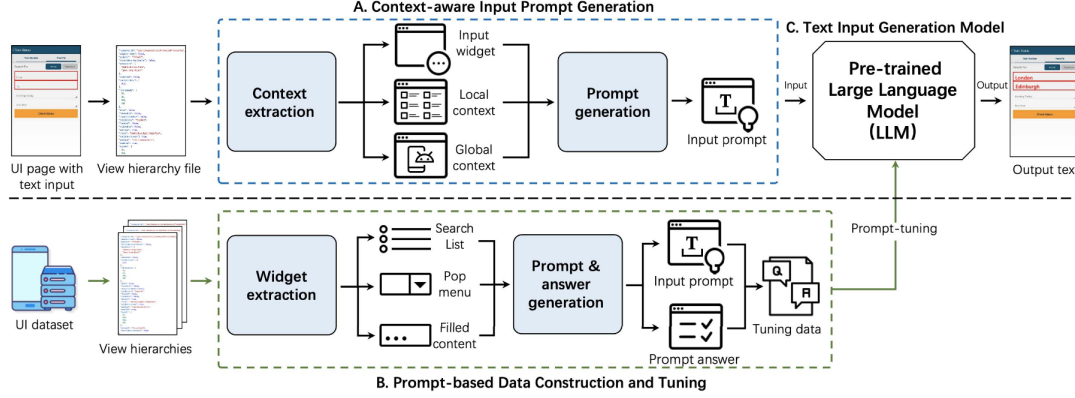


Figure 2.4: QTypist

2.4.4 LLM-driven GUI testing in engineering software: GERALLT

Moving beyond mobile environments, Rosenbach et al. [38] explore LLM-driven GUI testing in desktop-based engineering applications. Their system, *GERALLT* (see Figure 2.5), targets a real-world tool-integration wizard in a large-scale research application and orchestrates two LLM-based components—one that autonomously interacts with the UI and another that evaluates potential usability problems after each action.

Through prompts containing localized GUI context (e.g., screenshots, labeled metadata of widgets) and relevant textual documentation, *GERALLT* forms test actions aligned with domain-specific requirements. In evaluations, it discovered issues overlooked by manual testing efforts, such as misaligned error messages or inconsistent UI states. This approach further underscores how LLM-based methods can extend beyond mobile scenarios and automate exploratory tests for more specialized desktop software.

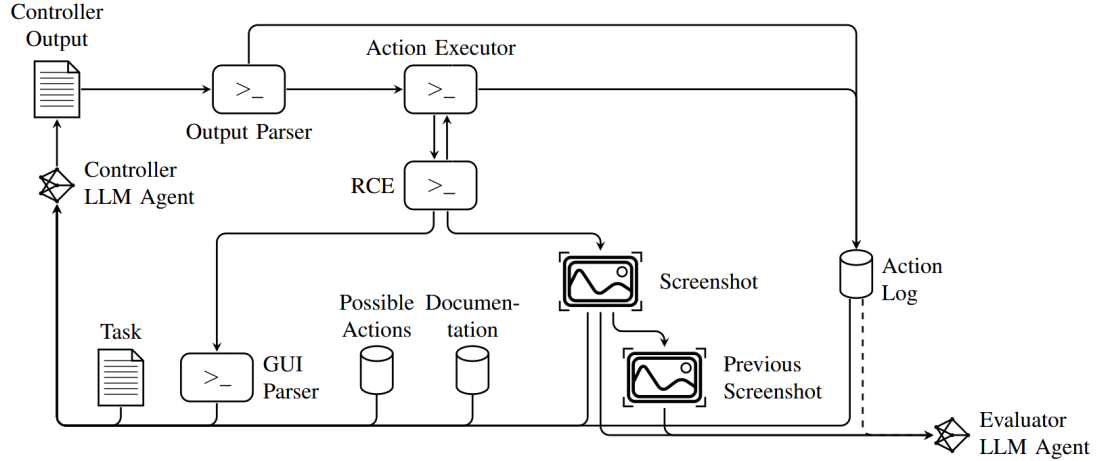


Figure 2.5: GERALLT

2.4.5 Intent-based testing for Android: DROIDAGENT

Yoon et al. [39] propose an LLM-based technique named *DROIDAGENT* (see Figure 2.6) for Android GUI testing, focusing explicitly on intent-driven scenarios rather than simple coverage of visual elements. *DROIDAGENT* leverages multiple cooperating LLM agents with long- and short-term memory to identify meaningful tasks—such as sending messages or configuring user settings—that mirror realistic, high-level user goals.

Once a task is defined, one LLM agent executes it step by step, while another reviews the actions in real time to detect unexpected outcomes. The system maintains both a memory of previously visited interface states and textual descriptions of each screen, driving deeper, semantically aware exploration. Empirical studies show that *DROIDAGENT* can achieve high activity coverage and generate realistic test flows spanning advanced app features—proving that LLM-based test generation can be guided by user-centric objectives rather than purely structural metrics.

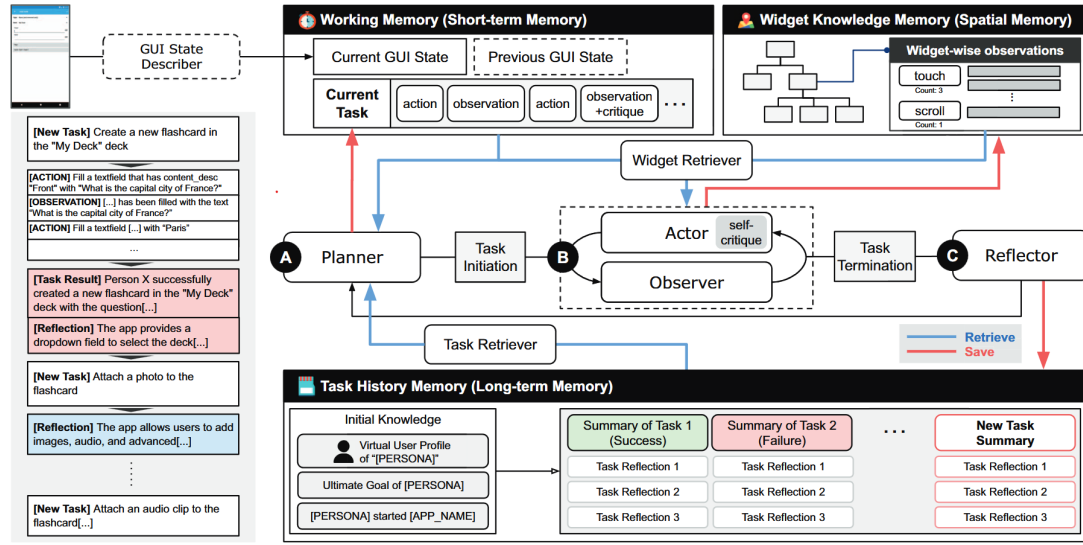


Figure 2.6: DROIDAGENT

2.4.6 Beyond GUI: Broader LLM-based Software Testing Approaches

Beyond these GUI-specific efforts, broader evaluations of LLMs in software testing reveal the general promise of neural approaches for oracle creation, code generation, and automated debugging [40]. In fact, the comprehensive survey by Wang et al. [40] underscores how LLMs can be employed to tackle unit test case generation, bug localization, and program repair, suggesting that GUI testing could similarly benefit from neural approaches to produce more robust and flexible test oracles.

Nevertheless, adapting these ideas from code-centric testing to GUI scenarios requires careful consideration of graphical nuances. Common LLM-based approaches for oracle generation (e.g., [41–43]) frequently assume a well-defined functional specification—such as a method signature or developer comment—which might not exist for GUI elements. Hence, GUI-oriented solutions must design prompts that integrate high-level UI semantics (e.g., composable hierarchies, state

transitions) and textual descriptions to capture the intended behavior of graphical interfaces. By incorporating information about layout structures, user workflows, and interface states, researchers can harness the full potential of LLMs to create testing strategies that are both flexible and aligned with actual end-user interactions.

2.4.7 Summary and Outlook

Overall, these efforts illustrate the emerging potential of neural approaches, backed by adaptive prompt engineering and domain-specific constraints, to tackle challenges in GUI testing. While promising, challenges remain—such as managing token-length limits for large-scale apps or ensuring prompts remain valid under significant interface restructuring. Nonetheless, the success of these frameworks in various domains (mobile, desktop engineering, and general Android) suggests that LLM-driven testing strategies can significantly lower test fragility by adapting to interface changes in real time, ultimately producing scripts that are more maintainable and aligned with genuine application behavior.

Chapter 3

Methodology

3.1 Goal-Question-Metric

Table 3.1: GQM table for the study

Analyze:	LLM-assisted, developer-in-the-loop GUI test maintenance
For the purpose of:	Easing maintenance burden of Android GUI suites
With respect to:	Test fragility
From the viewpoint of:	Android developers and CI engineers
In the context of:	Rapidly evolving Jetpack Compose apps

This thesis employs the GQM paradigm, detailed in Table 3.1, which highlights five essential dimensions that guide and structure the investigation. Firstly, the objective is to examine the maintenance workflow, specifically focusing on how developer-in-the-loop, LLM-assisted technique can alleviate the challenges associated with maintaining Android GUI test suites. Secondly, the intent is evaluative: the research aims to gather empirical evidence demonstrating the degree to which these techniques simplify the process of maintaining Android GUI test suites. Thirdly, the central quality attribute is test fragility—specifically, the capability to swiftly and effectively repair failing tests, ensuring their continued alignment with the evolving GUI and its test oracles. Fourthly, the perspective considered is that of Android developers and CI engineers, who directly encounter and manage the consequences of failed pipelines. Finally, the study is situated within the context of rapidly changing Jetpack Compose applications. These applications follow frequent release schedules and depend on runtime recomposition, where minor adjustments can significantly reorganize the underlying semantics tree.

3.2 Research Questions

This investigation is structured over these four research questions:

RQ1—Baseline Suite Fragility. *Given a GUI test suite authored for version v_1 of the AUT, how many tests (i) still **compile and pass**, (ii) **compile but fail**, and (iii) **fail to compile** when executed unchanged on version v_2 ?*

RQ2—Zero-Shot LLM Repair Effectiveness. *For the tests that either do not compile or compile but fail on v_2 , what proportion can be **recovered** after a single zero-shot GitHub Copilot prompt that leverages contextual information from the IDE?*

RQ3—Zero-Shot Retry Gain. *If the first zero-shot repair attempt is unsuccessful, what additional proportion of tests can be recovered by issuing another **one zero-shot prompt**—a retry informed only by the updated compilation or runtime feedback?*

RQ4—Chain-of-Thought Repair Effectiveness. *For the tests still failing after the two zero-shot attempts, how many can be **salvaged** through a brief, developer-in-the-loop conversation with Copilot?*

To answer these RQs, the focus is on test cases originally written for a previous v_1 version of the AUT which are then run against a later v_2 version. The test case naming convention for this thesis is: a test case is classified as *passing* if it compiles successfully and the test passes, *failing* if it compiles but fails during execution due to assertion failures or runtime exceptions, and *broken* if it fails to compile.

Importantly, compilation errors may not always stem from the code within the test case itself; they can also arise from broader contextual elements, such as setup routines executed prior to each test or shared companion objects referenced across multiple tests. Once a test successfully passes at a given stage of the repair process, it is excluded from any further maintenance attempts. To address this, a structured methodology is introduced to evaluate the effectiveness of LLM-based approaches in repairing broken tests, track progressive enhancements achieved through successive repair efforts, and assess the role of developer-guided interventions in facilitating test maintenance.

3.3 Project Selection

The choice of the BITWARDEN mobile application for Android as the subject under test (AUT) in this study was informed by several key considerations.

First, the selected system needed to be *(i)* fully open-source, ensuring complete accessibility to all components, including Gradle build configurations and test suites; *(ii)* implemented using Jetpack Compose; and *(iii)* equipped with a comprehensive GUI regression testing framework. To identify a suitable candidate, existing literature on Android application analysis was reviewed. Based on these criteria and its prior use in related research [44], the Bitwarden app was deemed an appropriate choice.

Bitwarden Android serves as the Android client for the open-source Bitwarden password management platform, offering secure, encrypted storage and retrieval of user credentials on mobile devices. It supports features such as system-wide autofill, biometric authentication, offline access, and password generation. As part of a vibrant open-source ecosystem, the app benefits from community-driven development, independent security audits, and the flexibility of cloud-based or self-hosted deployment options.

For this cross-version analysis, when the study began the two most recent stable tags were: $v_{2025.1.2}$ ¹, dated 10 February 2025, that will be referred to as v_1 and $v_{2025.2.0}$ ², dated 7 March 2025, which it will be called v_2 from now on. The four-week gap is a threshold that is intended to verify that the core business logic remains unchanged. However, it is also long enough

¹<https://github.com/bitwarden/android/releases/tag/v2025.1.2>

²<https://github.com/bitwarden/android/releases/tag/v2025.2.0>

for routine UI refinements, such as tweaks to the vault list, the bottom navigation bar, and the accessibility semantics – all modifications mentioned in the changelog – to come to light. After an inspection in the changelog and diff files between the two releases of the application, no major changes to the business logic were found. This snapshot comparison gives a clear baseline for assessing how well LLM-assisted techniques can repair and maintain fragile GUI tests.

3.4 Test Case Extraction

To systematically identify and synchronize GUI test cases across the two versions, two Python scripts were developed: `FIND_UI_TESTS.PY`³ and `SYNC_UI_TESTS.PY`⁴.

`FIND_UI_TESTS.PY` inspects the code-base and produces a list of GUI test cases. Starting at the project root, it visits every file that resides inside the `src/**/test` subtree and ends in `.kt`. A file is considered a test class only when two conditions hold: it imports Jetpack Compose or Espresso test libraries, and it contains at least one method annotated with `@Test`. Each such method is then examined; if its body calls a recognised Compose helper like `onNode()` or `performClick()`, the method is classified as a UI test case (see Listing 3.1). Finally, the script writes two plain-text artifacts as output files. `UI_TEST_REPORT.TXT`⁵ is a human-orientated index that shows, per source file, the names of its UI-test methods and a Google Sheet-compatible hyperlink to the file in the on-line repository. `UI_TEST_FQNS.TXT`⁶ is machine-orientated: it lists one Fully Qualified Name (FQN) per line, ready to be supplied to a Gradle script.

`SYNC_UI_TESTS.PY`⁷ consumes the `UI_TEST_FQNS.TXT` generated above and ensures that precisely this set of tests is present on a newer Git reference. With the repository still checked out at the commit of version v_1 , it reads `UI_TEST_FQNS.TXT`, maps every class to the subset of its methods that must be retained, and loads the corresponding Kotlin source file from that commit. The script then checks out the commit of version v_2 , injects each source file into that version of the tree, and removes any `@Test` method whose simple name is not on the keep-list. Removal is implemented by balanced-brace counting so that helper code and non-test logic remain untouched. Each transformed file is staged with git add, and on completion the script reports the number of files copied and trimmed, leaving the index ready for commit.

```

1  @Test
2  fun 'Turning off autofill should send AutofillServiceChanged with value of false'
3      () {
4      mutableStateFlow.update {
5          it.copy(autofillEnabled = true)
6      }
7      composeTestRule
8          .onNodeWithText("Autofill services", ignoreCase = true)
9          .performScrollTo()
10         .performClick()
11     verify {
12         viewModel.trySendAction(SetupAutoFillAction.AutofillServiceChanged(false))
13     }

```

³https://github.com/BlakeDake/android/blob/main/find_ui_tests.py

⁴https://github.com/BlakeDake/android/blob/main/sync_ui_tests.py

⁵https://github.com/BlakeDake/android/blob/MYv2025.1.2/ui_test_report.txt

⁶https://github.com/BlakeDake/android/blob/MYv2025.1.2/app/ui_test_fqns.txt

⁷https://github.com/BlakeDake/android/blob/main/sync_ui_tests.py

Listing 3.1: Example GUI Test case

3.5 Operational Environment

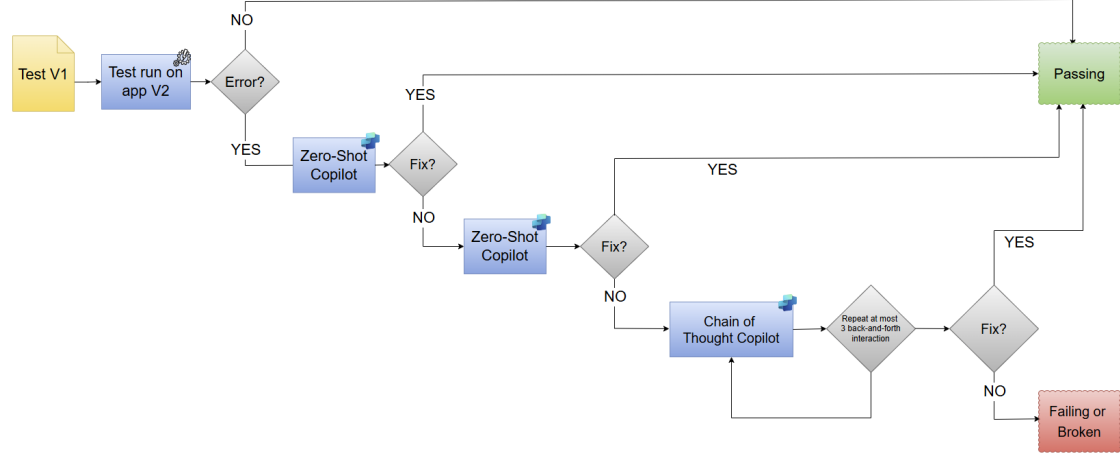


Figure 3.1: LLM repair pipeline

All experiments were carried out on a consumer-grade laptop running Windows 11 (version 24H2). Development and testing were conducted using Android Studio Meerkat Feature Drop, with JetBrains Runtime 21 configured as the Gradle JDK. Versions for Kotlin, Jetpack Compose, and other relevant dependencies are defined in the centralized configuration file `LIBS.VERSIONS.TOML`⁸. During the experimental phase (May 2025), GitHub Copilot Pro and Claude 3.7 Sonnet (Thinking) were employed as LLM-based tools for test repair tasks.

The complete testing pipeline is illustrated in Figure 3.1. For each test originating from version v_1 , it is executed against the updated codebase at version v_2 . This initial run addresses RQ1 by categorizing the test outcome using a three-tier classification. If the test compiles and passes, it is marked as *Passing* and excluded from further intervention. Otherwise, the corresponding error log is supplied to GitHub Copilot chat along with a minimal prompt: *"Fix the test"*. Although straightforward, this zero-shot strategy is supported by existing literature demonstrating its effectiveness in GUI testing scenarios [45, 46].

The output of this first repair attempt informs RQ2. If the generated fix results in a passing test, the case is again labeled as *Passing*. If errors persist, RQ3 is addressed by reiterating the same prompt but updating the context with the new error log.

Should the second zero-shot attempt still fail, RQ4 is examined through a brief chain-of-thought interaction with Copilot. This guided dialogue begins with identifying the error, followed by an explicit reasoning step and a proposed fix (*"Please identify the issue in the error log above"* followed by *"Please provide a fix for that issue"*). If the initial fix fails to resolve the issue, a

⁸<https://github.com/BlakeDake/android/blob/main/gradle/libs.versions.toml>

follow-up prompt requests an alternative solution based on the latest error feedback (*"That did not work, provide another fix given the new error log above"*). If both guided attempts fail to produce a compiling and passing test, the case is ultimately labeled as either *Failing* or *Broken*, depending on the nature of the unresolved issue.

Below, two example replies: one for a zero-shot round (see Listing 3.2) and the other for a CoT interaction (see Listing 3.3 for the identify the issue step and see Listing 3.4 for the provide a fix step) .

```

1 The test is failing because it can't find the node with text "By activating this
  switch you agree" as a substring. Looking at the strings.xml file, I can see
  the actual string includes a comma after "switch" that isn't being matched:
2
3 <string name="
  by_activating_this_switch_you_agree_to_the_terms_of_service_and_privacy_policy
  ">
4   By activating this switch, you agree to the
5   <annotation link="termsOfService">Terms of Service</annotation>
6   and <annotation link="privacyPolicy">Privacy Policy</annotation>
7 </string>
8
9 Also, looking at the TermsAndPrivacySwitch composable in CreateAccountScreen.kt, I
  see that the switch has a content description "AcceptPoliciesToggle" which
  would be more reliable to use for testing.
10
11 Here's how to fix the test:
12
13 @Test
14 fun accept policies should be toggled on or off according to the state() {
15     composeTestRule
16         .onNodeWithContentDescription("AcceptPoliciesToggle")
17         .assertIsOff()
18
19     mutableStateFlow.update { it.copy(isAcceptPoliciesToggled = true) }
20
21     composeTestRule
22         .onNodeWithContentDescription("AcceptPoliciesToggle")
23         .assertIsOn()
24 }
25
26 This uses the content description to reliably find the switch rather than
27 trying to match the text, which can be tricky especially with annotated
28 strings that contain formatting.

```

Listing 3.2: Example GitHub Copilot response zero-shot

```

1 The error in your test is happening because the test is trying to find a UI
  element with the text "Master password hint (optional)" but no such element
  exists in the UI.
2
3 Looking at the string resources in your application, there is:
4 - 'master_password_hint_not_specified' -> "Master password hint"
5 - 'new_master_password_hint' -> "New master password hint"
6
7 But there's no string for "Master password hint (optional)".
8
9 To fix this issue, you need to update your test to look for the actual text that
  appears in the UI. You should change the test to use the exact label that's
  displayed in your reset password screen, which is likely either "Master
  password hint" or "New master password hint" based on the string resources
  available.

```

```

10
11 Check your ResetPasswordScreen implementation to see which string resource it's
    actually using, and update your test accordingly.

```

Listing 3.3: Example GitHub Copilot response CoT - Identify the issue

```

1 Looking at the test file, it seems you need to use the correct string resources
  for testing password hint input. Based on the strings.xml file, there are two
  relevant strings for password hints:
2
3 1. 'new_master_password_hint' - "New master password hint"
4 2. 'master_password_hint_not_specified' - "Master password hint"
5
6 Here's a fix for the test method that uses these string resources:
7
8 @Test
9 fun 'password hint input change should send PasswordHintInputChanged action'() {
10     // Setup
11     val hintText = "My password hint"
12     val hintLabel = composeTestRule.activity.getString(R.string.
        new_master_password_hint)
13
14     // Execute
15     composeTestRule
16         .onNodeWithText(hintLabel)
17         .performTextInput(hintText)
18
19     // Verify
20     verify {
21         mockActionHandler.invoke(
22             match { action ->
23                 action is ResetPasswordAction.PasswordHintInputChanged && action.
                hint == hintText
24             }
25         )
26     }
27 }
28
29 This fix retrieves the "New master password hint" string from resources and uses
    it to find the correct input field, then verifies the correct action is sent
    when text is entered.

```

Listing 3.4: Example GitHub Copilot response CoT - Provide a fix

Chapter 4

Results

This section outlines the results corresponding to each research question.

Figure 4.1 illustrates the distribution of test outcomes, categorized by their association with the respective research questions. In parallel, Table 4.1 provides a detailed view of how test cases evolve across the different RQ stages, organized by test class.

Overall, the data in Table 4.1 indicates that little to no improvement is observed during the RQ3 phase. Additionally, it becomes evident that larger test classes tend to pose greater challenges for the LLM. Among the three research question phases, the initial output generated in response to RQ2 emerges as the most impactful in determining whether a test can be successfully repaired.

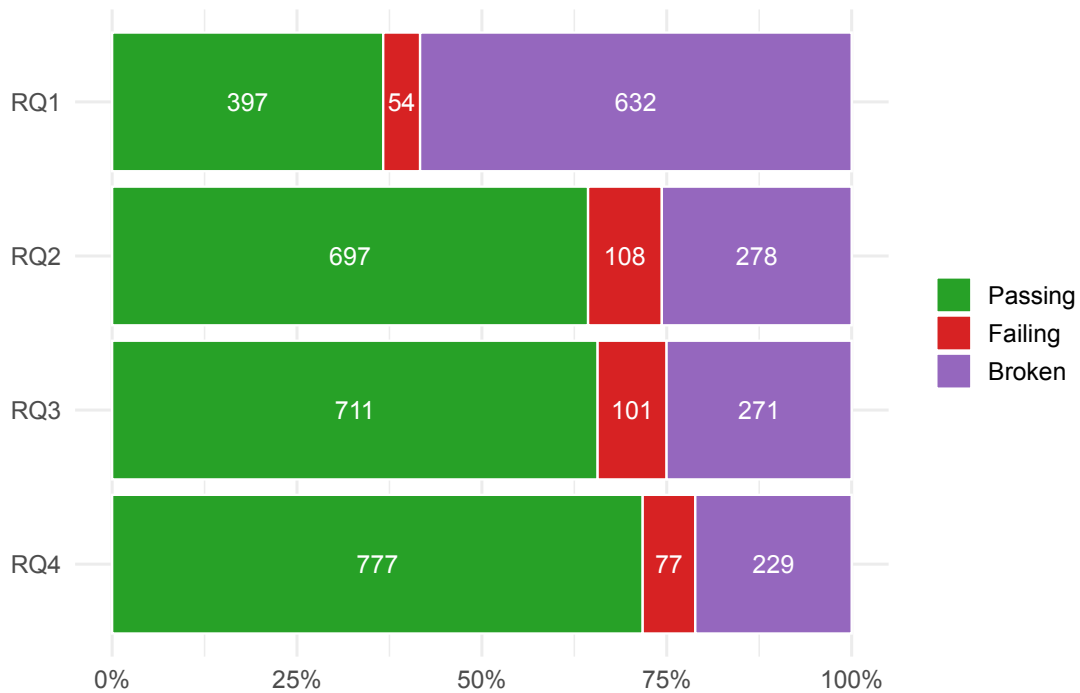


Figure 4.1: Cumulative test outcomes per RQs

Table 4.1: Distribution of test case outcomes (Passing P , Failing F , Broken B) per research questions (only test classes with failing or broken test cases are present).

Test Class	Tot	RQ1			RQ2			RQ3			RQ4		
		P	F	B	P	F	B	P	F	B	P	F	B
SetupUnlockScreenTest.kt	27	26	1	0	1	0	0	0	0	0	0	0	0
CheckEmailScreenTest.kt	5	0	0	5	4	1	0	0	1	0	0	1	0
CreateAccountScreenTest.kt	17	13	0	4	1	0	3	1	1	1	0	2	0
EnterpriseSignOnScreenTest.kt	7	5	0	2	1	0	1	0	0	1	0	1	0
EnvironmentScreenTest.kt	14	0	0	14	0	0	14	0	0	14	0	12	2
LandingScreenTest.kt	22	0	0	22	19	3	0	0	3	0	0	0	3
LoginScreenTest.kt	16	0	0	16	16	0	0	0	0	0	0	0	0
ResetPasswordScreenTest.kt	12	0	0	12	0	0	12	5	6	1	2	1	4
StartRegistrationScreenTest.kt	12	10	1	1	1	0	1	0	0	1	1	0	0
TwoFactorLoginScreenTest.kt	16	0	0	16	12	4	0	1	1	2	1	2	0
SearchScreenTest.kt	27	0	0	27	0	0	27	0	0	27	26	1	0
AccountSecurityScreenTest.kt	60	0	0	60	45	15	0	3	9	3	2	10	0
AppearanceScreenTest.kt	8	2	6	0	4	1	1	2	0	0	0	0	0
AutoFillScreenTest.kt	28	24	4	0	2	1	1	2	0	0	0	0	0
ExportVaultScreenTest.kt	16	0	0	16	16	0	0	0	0	0	0	0	0
FolderAddEditScreenTest.kt	8	0	0	8	8	0	0	0	0	0	0	0	0
OtherScreenTest.kt	10	7	3	0	3	0	0	0	0	0	0	0	0
VaultUnlockedNavBarScreenTest.kt	13	0	0	13	13	0	0	0	0	0	0	0	0
GeneratorScreenTest.kt	55	0	0	55	0	54	1	0	55	0	34	21	0
SendScreenTest.kt	28	0	28	0	28	0	0	0	0	0	0	0	0
VaultAddEditScreenTest.kt	143	0	0	143	0	0	143	0	0	143	0	0	143
VaultItemScreenTest.kt	98	0	0	98	75	23	0	0	5	18	0	5	18
VaultItemListingScreenTest.kt	57	0	0	57	0	0	57	0	0	57	0	0	57
ManualCodeEntryScreenTests.kt	8	5	3	0	0	3	0	0	3	0	0	3	0
VaultMoveToOrganizationScreenTest.kt	12	8	4	0	0	3	1	0	3	1	0	4	0
VaultScreenTest.kt	53	0	0	53	51	0	2	0	0	2	0	0	2
VerificationCodeScreenTest.kt	14	0	0	14	0	0	14	0	14	0	0	14	0
Test classes requiring maintenance	786	100	54	632	300	108	278	14	101	271	66	77	229
Total test classes	1083	397	54	632	697	108	278	711	101	271	777	77	229

4.1 RQ1 - Baseline Suite Fragility

To address RQ1, it was assessed how many test cases and test classes written for version v_1 of the Bitwarden application could be executed successfully—i.e., compiling and passing without modification—on the subsequent version v_2 .

As noted in Section 3.2, a compilation failure in a single test case can prevent its entire class from compiling, potentially skewing the number of reported non-compiling test cases. To mitigate this, the pipeline employs a selective exclusion mechanism: if a test case fails to compile due to an issue isolated to its own code (i.e., not related to shared setup logic or companion objects), it is excluded from the suite and the remaining tests are recompiled. This iterative approach allows the rest of the test class to be evaluated. In contrast, if compilation errors stem from shared structures—such as `@Before` methods or companion objects—the entire test class is treated as non-compilable.

From a total of 1083 GUI test cases extracted from version v_1 , execution on version v_2 yielded the following results: 397 test cases passed successfully, 54 failed at runtime, and 632 could not be compiled. These outcomes are visualized in Figure 4.1, alongside results for subsequent research questions. While more than 36% of the test cases remain functional across versions, the large number of failures and compilation issues highlights the significant maintenance overhead introduced by even short-term UI changes. This pronounced disparity underscores the inherent fragility of the test suite under study.

4.2 RQ2 - Zero-Shot LLM Repair Effectiveness

RQ2 evaluates the effectiveness of automated maintenance through the use of zero-shot prompting with a LLM. Specifically, this phase examines how well such an approach can repair test cases that were either failing or non-compilable in the initial run.

In a realistic development setting, the developer would apply the LLM’s proposed fix—whether it takes the form of a direct code snippet, a structured series of edits, or step-by-step instructions referencing specific patterns or lines. Any fix that led to a compiling and passing test case was considered a successful repair.

A total of 686 test cases were analyzed under RQ2. Following a single zero-shot prompt based solely on the error message, 300 test cases (approximately 43.7%) were successfully repaired. This subset includes 40 test cases that had previously failed at runtime and 260 that had originally failed to compile. At this stage, 108 test cases remained in a failed state—100 of which had been broken from the outset. Of the 278 broken test cases that were not fixed, 272 remained non-compilable.

These findings suggest that LLM-assisted, developer-guided repair workflows can meaningfully reduce the volume of broken or failing tests, successfully resolving nearly one-third of the cases initially deemed fragile. However, the proportion of unresolved tests remains considerable, warranting further investigation through RQ3 and RQ4.

4.3 RQ3 - Zero-Shot Retry Gain

RQ3 investigates the potential added benefit of issuing a second zero-shot prompt to those test cases that remained failing or broken after the initial LLM-based repair attempt. In this second iteration, the LLM was provided with the most recent error logs, alongside the same minimal prompt used previously.

The observed improvement from this follow-up attempt was considerably lower than that of the first. Only a small number of additional test cases were successfully repaired: 14 in total, comprising 6 that had previously failed and 8 that had been broken. The number of failed test cases remained high at 101—of which 79 were carried over from prior failures and 22 originated from the previously broken set. As for broken tests, the count stood at 271, including 23 cases that now failed and 248 that continued to be non-compilable.

These findings indicate a limited return from simple repetition of the zero-shot prompt. The marginal recovery rate in this second round suggests that a purely iterative prompting strategy, without additional context or reasoning support, is not an effective approach for test repair in this setting.

4.4 RQ4 - Chain-of-Thought Repair Effectiveness

The final set of test cases evaluated under RQ4 includes 372 instances, of which 271 were non-compilable. In this phase, the developer engaged in a brief but targeted interaction with GitHub Copilot, following a chain-of-thought (Chain-of-Thought (CoT)) prompting strategy.

This interactive approach enabled the LLM to generate more nuanced and potentially creative fixes. As a result, 66 test cases were successfully repaired—38 of which had previously failed at runtime and 28 that were initially broken. Despite these gains, 77 test cases remained in a failed state, with 57 carried over from earlier failures and 20 originating from the broken set. Additionally, 229 test cases continued to be non-compilable. Of these, 223 remained consistently broken across phases, while 6 had transitioned from a failing state.

These results highlight the potential of minimal developer intervention combined with structured prompting to enhance repair effectiveness, particularly for cases that resist simpler zero-shot approaches.

Table 4.1 presents the distribution of test outcomes across the different research questions. To clearly highlight the impact of each repair phase, the table focuses exclusively on the subset of test files that contained at least one failed test case during the initial execution. Files unaffected by GUI changes—i.e., those in which all tests passed from the outset—are excluded, allowing for a more concise and meaningful depiction of the test maintenance process.

Chapter 5

Discussion

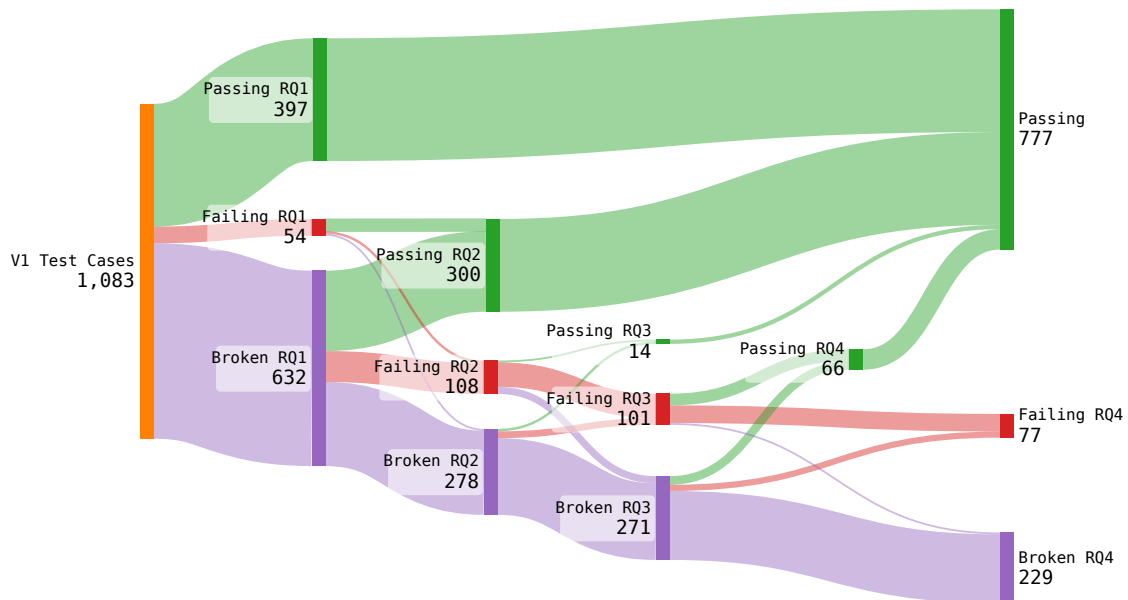


Figure 5.1: Sankey Chart of the test cases through the RQs

This chapter discusses the empirical findings obtained across the four research questions, emphasizing their implications for test maintenance in Bitwarden, an Android GUI application developed using Jetpack Compose. The analysis offers both quantitative and qualitative insights into the strengths and limitations of LLM-assisted test repair strategies.

The results for RQ1 reveal that 63.3% of test cases originally authored for version *v1* fail when executed without modification on version *v2*. Of these, 58.4% fail to compile, and 5% fail at runtime. These figures underscore the high fragility of GUI tests in the face of even minor application evolution. Structural UI changes—such as altered composable function signatures, modifications to layout hierarchies, or renamed resource identifiers—are key contributors to this degradation. These findings are consistent with prior work highlighting the brittleness of GUI

tests in reactive or declarative rendering frameworks, affirming that short release cycles can introduce substantial test maintenance challenges.

RQ2 investigates the effectiveness of a single zero-shot repair prompt using GitHub Copilot, relying on error messages and IDE context. Out of 686 broken or failing test cases, 300 were successfully repaired, yielding a recovery rate of 43.7%. These successful fixes included both direct code edits and step-by-step modifications that allowed the tests to compile and pass. The results demonstrate that LLMs can serve as practical assistants in test maintenance workflows, producing syntactically valid and semantically meaningful patches even without task-specific fine-tuning or elaborate prompting. However, a large portion of test cases remained unresolved, reflecting the limitations of single-step automated interventions—especially in the presence of intricate dependencies or unclear failure messages.

The findings for RQ3 show that a second zero-shot repair attempt, using updated error feedback, produced only a modest improvement. An additional 14 test cases were repaired, corresponding to a marginal gain of 2.0%. This minimal increase suggests that iterative prompting without additional context or developer feedback leads to diminishing returns. The ineffectiveness is particularly pronounced in structurally complex test classes, where deeply nested components or lengthy test bodies may confuse the model and hinder accurate repair.

RQ4 explores the role of minimal developer guidance through a brief COT interaction with the LLM. This strategy resulted in the repair of 66 additional test cases—a recovery rate of 17.7% over the unresolved cases from the previous stage. These results underscore the benefits of interactive repair: guided prompting enables the model to better reason about the failure context, refine its suggestions, and align more closely with the developer’s intent. Such interactions are especially valuable when automated fixes fail due to incomplete context or ambiguous test logic. The findings support the integration of lightweight, focused developer input into the repair loop to enhance effectiveness.

A further crosscutting observation concerns the influence of test class size on repair success. Larger test files, particularly those containing numerous test methods, proved systematically more difficult to repair. In such cases, the overall complexity likely exceeded the LLM’s effective context window, making it harder to localize compilation errors or generate coherent fixes. This was especially evident in test classes spanning thousands of lines, where shared setup logic and tightly coupled test bodies obscured the fault structure. These observations suggest that future repair workflows should incorporate modularization strategies or targeted prompting techniques to improve scalability in large or monolithic test suites.

In summary, the results demonstrate that LLM-based test repair can significantly reduce the manual effort required to maintain GUI test suites. Zero-shot prompting offers an effective entry point, and developer-guided interactions add value for cases that resist straightforward repair. Among the strategies evaluated, context-informed zero-shot prompting emerges as the most efficient compromise between repair effectiveness and maintenance effort, making it particularly suitable for time-sensitive development cycles.

Chapter 6

Conclusion

This thesis set out to answer a recurring question in modern Android development: *How can we keep large GUI-regression suites alive while the application and its Jetpack Compose interface change almost daily?*

To explore this, we designed and evaluated a *developer-in-the-loop* workflow that turns recent Large Language Models (LLMs) into just-in-time repair agents for broken tests.

6.1 Recap of Purpose and Method

Guided by the Goal–Question–Metric framework, we examined four research questions that progress from observing baseline fragility (RQ1) to measuring successive layers of LLM assistance (RQ2–RQ4):

- **Application under test.** Bitwarden Android—an open-source, production-grade app implemented with Jetpack Compose.
- **Experimental setup.** 1 083 GUI tests originally written for tag `v2025.1.2` were replayed on tag `v2025.2.0`; failures triggered one or more LLM-driven repair rounds inside Android Studio (Github Copilot + Claude 3.7 Sonnet).
- **Classification.** After every round each test was labelled *passing*, *failing*, or *broken*

6.2 Main Findings

Table 6.1: Evolution of test pass rates across repair stages.

Stage	Passing	Failing	Broken	Δ Passing
Baseline (RQ1)	397	54	632	—
Zero-shot repair (RQ2)	697	108	278	+300 (+43.7%)
Zero-shot retry (RQ3)	711	101	271	+14 (+2.0%)
Chain-of-thought (RQ4)	777	77	229	+66 (+17.7%)

Baseline fragility is severe. Nearly two-thirds (63.3 %) of tests written just four weeks earlier either failed or could not compile.

Single-shot LLM fixes go a long way. A single error-centric prompt lifted the pass rate from 36.7 % to 64.3 %, repairing 300 tests without bespoke tuning.

Blind repetition has little benefit. Re-prompting with updated logs rescued only 14 additional cases (+2 %), confirming diminishing returns for «retry-only» strategies.

Lightweight human guidance pays off. A brief chain-of-thought exchange added 66 further repairs, pushing the final pass rate to 71.8 %.

Class size matters. Large test files (>100 methods) resisted repair more often, hinting at context-window limits and tangled setups.

6.3 Implications for Practice

1. **Embed LLM repair into CI.** Even vanilla zero-shot prompting can more than halve the breakage avalanche after a release. Integrating a repair step before the build fails can spare developers hundreds of manual edits.
2. **Keep prompts concise yet contextual.** Error traces plus a one-line intent (*“Fix the test”*) proved sufficient; surplus instructions did not improve outcomes.
3. **Reserve developer attention for hard cases.** A focused two-or-three-turn conversation unlocks another ~18 % of stubborn failures—excellent leverage for minutes of effort.
4. **Modularise tests.** Splitting monolithic classes lowers both the cognitive and token load for the LLM, boosting repair success and future maintainability.

6.4 Contributions

- **Empirical evidence** that LLM-assisted maintenance can raise the survivability of a real-world Compose suite from 37 % to 72 % within three short cycles.
- **A reproducible workflow** combining automated scripts, IDE context harvesting, and progressive prompting patterns.
- **Actionable guidelines** on when to rely on zero-shot fixes, when to involve a human, and how to structure tests for LLM compatibility.

In summary, this thesis demonstrates that recent LLMs—paired with minimal but strategic developer oversight—can *transform brittle GUI regression suites into self-healing assets*. By folding language-model intelligence directly into everyday workflows, Android teams can keep pace with rapid UI evolution, trading the drudgery of manual repairs for a collaborative conversation with machines.

Chapter 7

Future Works

Building on the findings of the present study, several promising research directions can be pursued to deepen our understanding of LLM-assisted GUI testing and to push the boundaries of automated repair.

7.1 Broader and Deeper Empirical Replication

- **Multiple case studies.** Replicating the experiment on a diverse set of Android, iOS, and multiplatform projects (e.g. Flutter, React Native) would clarify which benefits are universal and which depend on framework specifics, codebase size, or team culture.
- **Extended temporal scope.** Observing the same suites across months or years—capturing major refactors, design overhauls, and library upgrades—can reveal how LLM performance decays (or improves) as the divergence between training data and code reality widens.
- **Meta-analysis of historical releases.** Mining public repositories that tag every store release (e.g. F-Droid) would enable a retrospective view of GUI test fragility at scale, complementing prospective studies.

7.2 Fine-Grained, Commit-Level Evolution

A pipeline that replays the entire test suite *at every commit* could produce an invaluable longitudinal dataset: pass/fail labels, repair attempts, and contextual signals such as diff size or author experience. Such granularity would let future researchers pinpoint the exact kinds of code changes (UI layout vs. business logic) that trigger breakage, the threshold at which LLMs “lose the plot” due to accumulating drift and early-warning heuristics for proactive repair suggestions.

7.3 Advanced Prompt Engineering and Retrieval

- **Adaptive prompts.** Mechanisms that condition the prompt on static analysis (e.g. dependency graphs) or dynamic coverage could supply richer context without bloating token budgets.
- **Curriculum strategies.** Starting with coarse fixes and progressively exposing deeper failure traces may help the model converge to correct patches with fewer hallucinations.

- **Retrieval-augmented prompting.** Coupling an LLM with a vector store of past fixes or code snippets can ground suggestions in project-local idioms and Application Programming Interfaces (APIs).

7.4 LLM Agent Architectures

Beyond single-shot or conversational prompting, future work could explore full-fledged *autonomous repair agents* that iterate through *plan* \rightarrow *edit* \rightarrow *compile* \rightarrow *test* loops. Key research questions may include:

1. How many reflection or self-critique cycles maximise benefit before diminishing returns?
2. Can reinforcement learning from compilation/test rewards surpass heuristic retry policies?
3. What governance mechanisms (e. g. static analysis filters, policy-based “safety rails”) are required to keep agents efficient and secure?

7.5 Beyond Repair: Generation and Oracle Evolution

Finally, the long-standing oracle problem in GUI testing remains only partly addressed. Future studies could (*i*) investigate LLM-generated assertions that evolve alongside the UI or (*ii*) integrate crowdsourced validation or differential execution (reference-vs-candidate runs) to vet the semantic correctness of repaired tests or (*iii*) compare test generation vs. test repair pipelines to evaluate cost–benefit trade-offs in green-field versus legacy scenarios.

In summary, extending the experimental canvas—across projects, time, and methodological sophistication—will not only solidify the empirical grounding of LLM-assisted maintenance but also chart a path toward truly resilient, self-evolving GUI test suites.

Bibliography

- [1] IEEE. *IEEE Standard Glossary of Software Engineering Terminology*. Tech. rep. 610.12-1990. IEEE Standard. IEEE, 1990.
- [2] Glenford Myers, Corey Sandler, and Tom Badgett. *The Art of Software Testing*. John Wiley & Sons Inc, 2011.
- [3] Antonia Bertolino. «Software testing research: Achievements, challenges, dreams». In: *Future of Software Engineering (FOSE'07)*. IEEE. 2007, pp. 85–103.
- [4] Kabir S Said et al. «GUI testing for mobile applications: objectives, approaches and challenges». In: *Proceedings of the 12th Asia-Pacific Symposium on Internetware*. 2020, pp. 51–60.
- [5] Riccardo Coppola, Luca Ardito, and Marco Torchiano. «Fragility of layout-based and visual GUI test scripts: an assessment study on a hybrid mobile application». In: *Proceedings of the 10th acm sigsoft international workshop on automating test case design, selection, and evaluation*. 2019, pp. 28–34.
- [6] Riccardo Coppola et al. «Scripted GUI testing of Android open-source apps: evolution of test code and fragility causes». In: *Empirical Software Engineering* 24 (2019), pp. 3205–3248.
- [7] Riccardo Coppola, Luca Ardito, and Marco Torchiano. «Multi-device, Robust, and Integrated Android GUI Testing: A Conceptual Framework». In: *IFIP International Conference on Testing Software and Systems*. Springer. 2023, pp. 115–125.
- [8] Google. *Thinking in Compose*. <https://developer.android.com/develop/ui/compose/mental-model>. Accessed: 2025-05-22.
- [9] Atif Memon. *GUI Testing: Pitfalls and Process*. <https://www.cs.umd.edu/~atif/papers/MemonIEEEComputer2002.pdf>. Accessed: 2025-03-20.
- [10] Gunel Jahangirova. «Oracle problem in software testing». In: *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 2017, pp. 444–447.
- [11] Google. *What is Android*. <https://www.android.com/what-is-android/>. Accessed: 2025-03-28.
- [12] Android Wiki. *List of Android devices*. https://android.fandom.com/wiki/List_of_Android_devices. Accessed: 2025-03-24.
- [13] Google. *Content license*. <https://source.android.com/license>. Accessed: 2025-03-31.
- [14] IBM. *What are large language models (LLMs)?* <https://www.ibm.com/think/topics/large-language-models>. Accessed: 2025-03-20.
- [15] Selenium. *Selenium Locator Strategies*. <https://www.selenium.dev/documentation/webdriver/elements/locators/>. Accessed: 2025-03-20.

- [16] Rafael AP Oliveira et al. «Definition and evaluation of mutation operators for GUI-level mutation analysis». In: *2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE. 2015, pp. 1–10.
- [17] Google. *Get started with Jetpack Compose*. <https://developer.android.com/develop/ui/compose/documentation>. Accessed: 2025-03-20.
- [18] Google. *Testing APIs*. <https://developer.android.com/develop/ui/compose/testing/apis>. Accessed: 2025-03-20.
- [19] Google. *Jetpack Compose architectural layering*. <https://developer.android.com/develop/ui/compose/architecture>. Accessed: 2025-03-20.
- [20] Lucas Zamprogno et al. «Dynamic human-in-the-loop assertion generation». In: *IEEE Transactions on Software Engineering* 49.4 (2022), pp. 2337–2351.
- [21] Fabrizio Pastore, Leonardo Mariani, and Gordon Fraser. «Crowdoracles: Can the crowd solve the oracle problem?» In: *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*. IEEE. 2013, pp. 342–351.
- [22] Michael Konstantinou, Renzo Degiovanni, and Mike Papadakis. «Do LLMs generate test oracles that capture the actual or the expected program behaviour?» In: *arXiv preprint arXiv:2410.21136* (2024).
- [23] Ashish Vaswani et al. «Attention is all you need». In: *Advances in neural information processing systems* 30 (2017).
- [24] Scholarpedia. *Recurrent Neural Networks*. http://scholarpedia.org/article/Recurrent_neural_network. Accessed: 2025-03-20.
- [25] Jared Kaplan et al. «Scaling laws for neural language models». In: *arXiv preprint arXiv:2001.08361* (2020).
- [26] IBM. *What is self-supervised learning?* <https://www.ibm.com/think/topics/self-supervised-learning>. Accessed: 2025-03-20.
- [27] Jacob Devlin et al. «Bert: Pre-training of deep bidirectional transformers for language understanding». In: *Proceedings of the 2019 conference of the North American chapter of the association for computational linguistics: human language technologies, volume 1 (long and short papers)*. 2019, pp. 4171–4186.
- [28] Mark Chen et al. «Evaluating large language models trained on code». In: *arXiv preprint arXiv:2107.03374* (2021).
- [29] Jules White et al. «A prompt pattern catalog to enhance prompt engineering with chatgpt». In: *arXiv preprint arXiv:2302.11382* (2023).
- [30] Meredith Syed and Vrunda Gadesha. *What is zero-shot prompting?* <https://www.ibm.com/think/topics/zero-shot-prompting>. Accessed: 2025-03-20.
- [31] Vrunda Gadesha. *What is one shot prompting?* <https://www.ibm.com/think/topics/one-shot-prompting>. Accessed: 2025-03-20.
- [32] Vrunda Gadesha. *What is few shot prompting?* <https://www.ibm.com/think/topics/few-shot-prompting>. Accessed: 2025-03-20.
- [33] Vrunda Gadesha. *What is chain of thoughts (CoT)?* <https://www.ibm.com/think/topics/chain-of-thoughts>. Accessed: 2025-03-20.
- [34] Laria Reynolds and Kyle McDonell. «Prompt programming for large language models: Beyond the few-shot paradigm». In: *Extended abstracts of the 2021 CHI conference on human factors in computing systems*. 2021, pp. 1–7.

- [35] Zhe Liu et al. «Make llm a testing expert: Bringing human-like interaction to mobile gui testing via functionality-aware decisions». In: *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 2024, pp. 1–13.
- [36] Daniel Zimmermann and Anne Koziolok. «Automating gui-based software testing with gpt-3». In: *2023 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE. 2023, pp. 62–65.
- [37] Zhe Liu et al. «Fill in the blank: Context-aware automated text input generation for mobile gui testing». In: *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE. 2023, pp. 1355–1367.
- [38] Tim Rosenbach, David Heidrich, and Alexander Weinert. «Automated Testing of the GUI of a Real-Life Engineering Software using Large Language Models». In: *Proceedings of the IEEE ICST Workshops 2025*. Naples, Italy: IEEE, 2025, pp. 103–110.
- [39] Juyeon Yoon, Robert Feldt, and Shin Yoo. «Intent-driven mobile gui testing with autonomous large language model agents». In: *2024 IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE. 2024, pp. 129–139.
- [40] Junjie Wang et al. «Software testing with large language models: Survey, landscape, and vision». In: *IEEE Transactions on Software Engineering* (2024).
- [41] Elizabeth Dinella et al. «Toga: A neural method for test oracle generation». In: *Proceedings of the 44th International Conference on Software Engineering*. 2022, pp. 2130–2141.
- [42] Soneya Binta Hossain and Matthew Dwyer. «Togll: Correct and strong test oracle generation with llms». In: *arXiv preprint arXiv:2405.03786* (2024).
- [43] Facundo Molina, Alessandra Gorla, and Marcelo d’Amorim. «Test Oracle Automation in the era of LLMs». In: *ACM Transactions on Software Engineering and Methodology* (2024).
- [44] Abhyudaya Sharma and Sweta Mishra. «A Security Analysis of Password Managers on Android». In: *International Conference on Information Systems Security*. Springer. 2023, pp. 3–22.
- [45] Zhe Liu et al. «Chatting with gpt-3 for zero-shot human-like mobile automated gui testing». In: *arXiv preprint arXiv:2305.09434* (2023).
- [46] Kristian Kolthoff et al. «Zero-Shot Prompting Approaches for LLM-based Graphical User Interface Generation». In: *arXiv preprint arXiv:2412.11328* (2024).