Automated Program Repair for UI-centric Android Bugs: How Far Are We?

JUNAYED MAHMUD, University of Central Florida, USA
SPARSH PANDEY, University of Central Florida, USA
NADEESHAN DE SILVA, William & Mary, USA
ATISH KUMAR DIPONGKOR, University of Central Florida, USA
JINGJING WU, University of Minnesota, USA
OSCAR CHAPARRO, William & Mary, USA
MATTIA FAZZINI, University of Minnesota, USA
KEVIN MORAN, University of Central Florida, USA

Substantial research effort has been devoted to developing techniques for automated program repair (APR) that suggest patches for localized buggy code – and more recent techniques have begun to leverage the capabilities of code-centric large language models (LLMs). However, the scope and diversity of bugs to which these techniques have historically been applied are limited. In particular, the research community currently lacks a comprehensive understanding of the performance of APR techniques on bugs that arise in UI-centric programs, such as mobile apps. Bugs in UI centric programs carry with them unique challenges, including (i) the need to reason across interconnected subroutines that connect presentation and program logic, (ii) event-driven programming paradigms, and (iii) the need to reason about program state through cues in the UI.

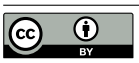
In this paper, we investigate the effectiveness of existing APR techniques when applied to fix bugs in UI-centric programs – specifically Android applications. To explore this phenomenon, we conduct a comprehensive empirical study with five existing program repair techniques (including those that utilize LLMs) on a hybrid dataset including 46 synthetic bugs, generated via MDroid+, an Android-specific mutation tool, and 50 real bugs systematically mined from issue reports of 23 popular Android applications. Our findings illustrate important current limitations in resolving UI-related issues in mobile apps. We synthesize these results to form a taxonomy of the limitations of existing program repair techniques. This taxonomy outlines key limitations and can inform future research efforts in designing automated program repair tools for UI-centric bugs in mobile applications.

CCS Concepts: • **Software and its engineering** → **Software testing and debugging**; *Software maintenance tools*; • **Human-centered computing** → *User interface programming*.

Additional Key Words and Phrases: Automated Program Repair, Mobile Apps, User Interfaces, Android, Large Language Models, Empirical Study

ACM Reference Format:
Junayed Mahmud, Sparsh Pandey, Nadeeshan De Silva, Atish Kumar Dipongkor, Jingjing Wu, Oscar Chaparro, Mattia Fazzini, and Kevin Moran. 2026. Automated Program Repair for UI-centric Android Bugs: How Far Are We?. *Proc. ACM Softw. Eng.* 3, ISSTA, Article ISSTA181 (October 2026), 25 pages. https://doi.org/10.1145/3832272

Authors' Contact Information: Junayed Mahmud, University of Central Florida, Orlando, FL, USA, junayed.mahmud@ucf.edu; Sparsh Pandey, University of Central Florida, Orlando, FL, USA, sparsh.pandey@ucf.edu; Nadeeshan De Silva, William & Mary, Williamsburg, VA, USA, kgdesilva@wm.edu; Atish Kumar Dipongkor, University of Central Florida, Orlando, FL, USA, atish.kumardipongkor@ucf.edu; Jingjing Wu, University of Minnesota, Minneapolis, MN, USA, wu000295@umn.edu; Oscar Chaparro, William & Mary, Williamsburg, VA, USA, oscarch@wm.edu; Mattia Fazzini, University of Minnesota, Minneapolis, MN, USA, mfazzini@umn.edu; Kevin Moran, University of Central Florida, Orlando, FL, USA, kpmoran@ucf.edu.



ACM 2994-970X/2026/10-ARTISSTA181
https://doi.org/10.1145/3832272

1 Introduction

Given the financial cost and engineering challenges associated with testing and fixing defects, researchers have invested considerable effort into the topic of automated program repair (APR). In its traditional formulation by Rinard *et al.* [56] and refined by Le Goues *et al.* [30], APR techniques generate patches for known, localized bugs, *"consisting of one or more code changes, that cause [the code] to pass a set of test cases"*, wherein passing tests indicates a successful repair. As such, APR techniques typically require both robust bug-revealing test suites and fully localized buggy code to operate. APR techniques often operate in conjunction with automated fault localization [48].

Because the evaluation of APR techniques requires several artifacts (*e.g.,* known bugs, localized code, tests that fail for the bug, existing patches), measuring the effectiveness of these approaches has traditionally been carried out on carefully crafted benchmarks such as Defects4J [28]. Such benchmarks have been an invaluable resource for the research community; however, they carry limitations related to diversity and the domain of the bugs contained in the datasets.

One area where current datasets tend to lack coverage is in bugs derived from *UI-centric* programs. In this paper, we define UI-centric programs as primarily user-facing software whose execution is driven by interaction events on a graphical user interface. UI-centric mobile and web applications are ubiquitous in modern society, and carry with them long documented challenges related to their construction [49]. UI-related code, and the effort devoted to it, is substantial. Past studies report that a program's UI often comprises nearly half of the code size, implementation time, and maintenance effort of user-facing software [50]. It is no surprise, then, that software bugs are prevalent across UI-centric software in web [53] and mobile apps [27, 34, 76]. As such, it is critical to understand the capabilities of APR techniques and their limitations in this software domain.

Defects in UI-centric programs carry with them unique characteristics that may make them particularly challenging for APR tools. UI programming tends to follow non-traditional paradigms. For instance, as described by Myers, one difference in UI programs compared to traditional programs is that they must be written "inside-out" [49]. That is, instead of writing code such that the application is in control, applications must be structured as a set of interconnected subroutines that react appropriately when the user performs an action (*e.g.,* clicking on a menu item). This style of programming has been illustrated to be more difficult for developers to reason about and organize [58]. Past studies have also documented the difficulties that developers often have when working with UI-related code, including understanding (i) UI event callbacks, (ii) graphical properties of the UI, and (iii) interactions across UI objects [62]. Such challenges stem from the need to reason across semantically diverse abstractions – graphical (*i.e.,* visual UI) information, and more traditional code. Given the well-documented difficulties related to comprehending and debugging UI-related code, it is critical to understand how APR tools navigate these challenges.

Given the bug localization and testing requirements of APR techniques, they have traditionally been challenging to practically apply to UI-centric software, given the effort required to create a benchmark in the face of known difficulties in testing such programs [35]. However, we have recently seen marked improvements, driven partially by LLMs, to automated bug reproduction [69, 85], UI test scenario generation [13], bug localization [44, 60], and assertion generation [26] that stand to make APR for Android UI-centric programs feasible in practice.

Given the importance of APR for UI-centric programs, the unique challenges that UI-centric bugs carry with them, and the advancement of automated tools that make APR possible in this domain, there is a pressing need to understand how well current APR techniques function on bugs from UI-centric programs, and any potential limitations. As such, in this work we conduct the first comprehensive empirical study on how different types of modern APR techniques function on UI-centric bugs derived from Android applications. To carry out this study, we derive a novel

benchmark, called DroidFixBench (Defects for Android Apps), that includes 50 real bugs systematically mined from issue reports of popular open source Android applications, and 46 synthetic bugs (injected using an Android-specific mutation tool). This mix of simpler, synthetic bugs and more complex real bugs helps us to assess the range of capability of APR tools on bugs of different difficulty/complexity levels. Each of the 96 bugs in this dataset includes (i) fully-localized fault information to the statement level, (ii) correct patches, (iii) a reusable set of test cases for each bug that apply to the buggy behavior and surrounding functionality, (iv) compilable buggy versions of the program, and (v) bug descriptions derived from the original bug reports or multi-author generated descriptions for synthetic bugs. This is the first dataset of its kind for UI-centric mobile apps. To understand how APR techniques function on DroidFixBench, we selected five recent APR tools of varying characteristics: D4C [77], ChatRepair [73], OpenHands [70], NTR [22], and RewardRepair [81]. These techniques span template-based repair, neural machine translation-based repair, LLM-based repair, and LLM-powered software engineering agents.

Our findings reveal that APR techniques generate correct or plausible patches for 4% to 87% of the synthetic bugs, with OpenHands performing best. In contrast, the best performing APR technique, ChatRepair, is only capable of generating correct or plausible patches for 34% of the real bugs. The failure cases across both types of bugs reveal important limitations across our studied tools. As such, we conducted a comprehensive open coding of a statistically significant sample of 379 of the Non-Plausible patches across all datasets and tools, and synthesized this into a taxonomy of APR tool limitations related to UI-centric bugs. Of this sample, nearly 1/3 of the issues are directly related to UI-specific program constructs, whereas other categories of limitations are likely exacerbated by UI-programming paradigms.

In summary, this paper makes the following main contributions:

- We create DroidFixBench, a dataset of diverse, generalized UI-centric bugs for Android apps, operationalized for APR. This dataset contains 50 real and 46 synthetic bugs, with statement-level fault localization information, a GUI-level test suite covering the target bug and related functionality, correct or original patch information, compilable buggy versions, and descriptions.
- We conduct a comprehensive empirical study applying 5 recent APR techniques to DroidFixBench, and examine the rate of correct, plausible, and non-plausible patches.
- We conduct a thorough, qualitative open coding analysis on a statistically significant sample of 379 failed (*i.e.,* non-plausible and plausible but not correct) patches generated by the APR tools. Our results identify 9 key categories of failures, representing limitations, that can inform future work on UI-centric program repair.
- A comprehensive replication package that contains the DroidFixBench dataset, the experimental infrastructure supporting the application of APR techniques, and all of the data and patches generated as part of this study [41, 42].

2 Background & Motivation

In this section, we provide background information related to bugs that manifest in UI-centric programs (and more specifically mobile apps) and illustrate why fixing these bugs is challenging.

2.1 A Brief Overview of UI Programming Paradigms

As discussed earlier, UI programming follows a form that differs from many forms of more traditional programming. While the exact form that UI programming patterns take tend to differ across software domains and UI programming frameworks, they tend to share a core set of principles, which we describe below, using the Android platform as a means of illustrating these concepts:

- **Declarative UI Layouts & Components:** Most UI-centric programming frameworks offer a means to define the various components, and their corresponding layouts on a screen. This UI

definition is typically hierarchical in nature and information can often flow between components at different levels of the hierarchy. In Android, UI layouts are declared in `.xml` resource files that define component attributes and hierarchical relationships.

- **Event Handlers to Respond to User Interactions:** Given that UI-centric programs are *user-driven* they typically require a means of responding to user interactions on the screen (taps, clicks, drags, etc.). This is typically achieved through event handlers, which are special functions bound to specific UI components, that respond to specific types of interactions when triggered for those components. In Android, this is implemented through `EventListeners` for different UI actions – such as a tap or long-tap. These constructs trigger the execution of program logic when the corresponding actions are detected.
- **State Management & Data Binding:** In order to keep track of data related to specific UI components (e.g., the text within a text box), or that is unique to the scope of a particular screen, most UI frameworks offer some form of state management and a means to bind data to specific UI components. In Android, this is accomplished in multiple ways, but it typically stored in a `ViewModel`, and the Android View Framework automatically updates the UI according to the state stored in the model.
- **Theming & Styling:** In addition to being functional, UIs require theming or styling to help emphasize the features afforded by a given screen or component, and to create aesthetically pleasing interfaces. In Android, base theming can be created statically through style-oriented `.xml` resource files, and updated dynamically through composable UI elements.

The interconnectedness of these various components, and the complex manner in which state and user actions are handled in UI programming make it challenging to comprehend and reason about bugs [62]. Furthermore, given the event-driven nature of UI-centric apps, the potential state space of the program grows exponentially, adding further complexity to understanding observed UI behavior.

2.2 The Unique Types and Attributes of Bugs in UI-centric Programs

Bugs in UI-centric programs generally manifest as erroneous behavior reflected visually through the UI of an app. Because most of the program output for these apps is drawn to the UI, debugging and fixing issues requires reasoning across both graphical and textual representations of a program. The gap between these abstractions contributes to the difficulty of fixing UI-centric bugs.

Prior work that mined and examined a sizable set of 180 real bug reports from one type of UI-centric app, namely Android applications, broadly categorized the defects into four main groups [27]:

- **Output/Content Errors:** These defects represent errors to information that is computed/retrieved by program logic and shown via the UI. For instance, an incorrect BMI calculation in a weight loss app being shown on the screen.
- **Cosmetic Issues:** Cosmetic issues primarily affect the theming or styling of an application, but typically do not interfere directly with program functionality. Examples of cosmetic issues include partially occluded components or internationalization issues.
- **Navigation Issues:** This defect type affects the ability of a user to move between the screens of a program and exercise features.
- **Crashes:** These defects result in the unexpected termination of a program, and can have many different underlying causes.

As we discuss in the next section, while some prior work has attempted to fix small subsets of some of these bugs types, this is the first work that examines APR techniques applied to a range of different bugs that manifest in UI-centric programs.

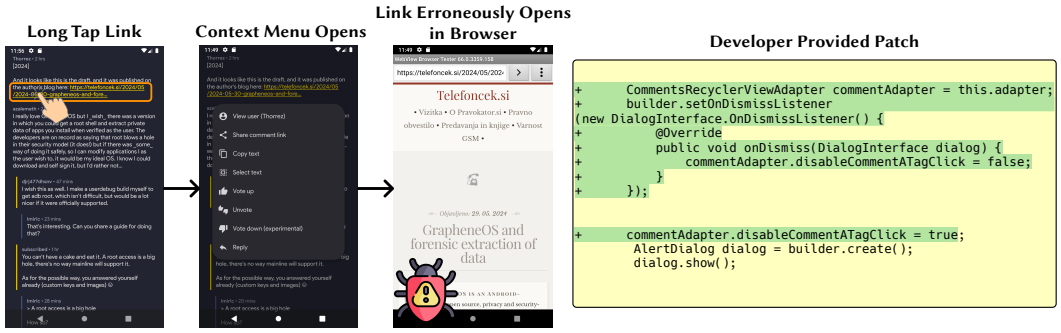


Fig. 1. Example of a UI-centric bug from Issue # 114 of the Harmonic App [4]. The bug describes an issue where long pressing a link erroneously opens up the link instead of opening a modal menu with share options.

2.3 Challenges of Bug Repair in UI-centric Programs

Figure 1 illustrates a Navigation bug resulting from an erroneous implementation of an event-handler specific to the long-tap action on a link in the Harmonic news reader application. This is one of the real bugs included in DroidFixBench, and is one for which *none* of our studied APR tools was successfully able to generate a correct or plausible patch.

In this bug, a long-tap action on a link in the comments of a given news story *should* open a modal context window that allows the user to share the link to a number of different channels. Instead the bug causes the user to be redirected to a web browser where the link is opened, causing an unexpected navigation flow. The fix illustrated in Figure 1 is part of a larger method that builds the context menu shown in the second screen using an `ArrayAdapter` to render each row of the modal menu. The added code, representing the developer provided patch, creates a boolean that temporarily ignores the `OnClick()` event handler related to the links themselves, which are inadvertently triggered during the long-touch to open the context menu. This boolean resets when the context menu is dismissed to restore normal link behavior.

This bug is challenging as it cross-cuts several of the UI programming paradigms discussed above. Here, event handlers for *different types of user actions* were interfering with one another, and as such, program state was created and tracked to help avoid this. To properly address this bug, the APR tool needs to infer and correctly reason about the potential program state, and how the different event handlers are connected with one another – a challenging proposition. This example illustrates how various UI programming paradigms can combine to contribute to patch generation difficulty.

3 Related Work

3.1 Automated Program Repair Techniques.

Automated Program Repair (APR) [17, 31] techniques have emerged to shift costly, manual software maintenance work of fixing bugs to precise and efficient automated assistants. Over the past decade, APR techniques have developed rapidly and received much attention from researchers in both academia and industry. As per Huang et al. [21], existing program repair techniques can be categorized into four main types: (i) search-based, (ii) constraint-based, (iii) template-based, and (iv) learning-based approaches.

Search-based methods [25, 30] navigate a predefined search space to identify suitable patches to fix a bug. However, identifying the correct patches from a large search space and validating the correctness with the existing test suite remains challenging. Constraint-based methods [51, 78] aim to mitigate the above limitations by utilizing constraint solvers and formal specifications to prune incompatible patches, reducing the search space. However, these techniques heavily rely on specification constraints, thus limiting flexibility. Template-based approaches [10, 36] rely

on predefined patch templates to suggest fixes, providing more effectiveness than search- and constraint-based methods. However, these methods do not have any continuous learning capability and are often tightly coupled to specific defect types. Learning-based techniques leverage machine or deep learning models trained on historical faults/fixes to suggest patches [37, 45].

In our research, we focus on template-based and learning-based approaches for bug fixing, as they are the more commonly the focus of recent APR studies.

LLMs in Program repair. With the advent of Deep Learning, several researchers proposed using sequence-to-sequence (Seq2Seq) models to perform program repair. For instance, Prenner et al. [57] utilized the CodeX model (12B parameters) [12] for program repair in both Java and Python, achieving state-of-the-art results on Python while showing poor performance on Java. Xia et al. proposed AlphaRepair [75], which combines CodeBERT [15] with relevant contexts for program repair, with limited capacity (125M parameters), restricting its effectiveness in real-world projects.

With the emergence of Large Language Models (LLMs), sequence-to-sequence models have been largely superseded by more powerful transformer-based architectures for program repair tasks. Kolak et al. [29] conducted a comprehensive evaluation of four language models, including Codex [12], for patch generation in program repair. Their results demonstrate that LLM-generated patches exhibit greater semantic similarity and naturalness [19] compared to original patches. Subsequently, Xia et al. [74] leveraged ChatGPT [54] to generate correct code from buggy implementations using few-shot learning approaches across three distinct experimental settings. Hossain et al. [20] introduced Toggle, which employs CodeGPT [40] and CodeGen [52] for automated program repair. Toggle achieved state-of-the-art performance on the CodeXGlue [40] benchmark and demonstrates strong results across multiple evaluation datasets.

However, none of these approaches specifically target UI-centric bugs, and account for the unique properties of these bugs as illustrated earlier. In this paper, we aim to investigate the successes and limitations in applying existing APRs to UI-centric bugs.

3.2 Agent-based Program Repair Techniques

With the recent rapid advancement of LLMs and agentic coding frameworks, autonomous software agents are increasingly being utilized for the task of program repair. Yang et al. [79] introduced SWE-agent, which is composed of a language model and an agent-computer harness. This agent can view and explore different files in software repositories and generate and update patches to fix software defects. Zhang et al. [84] proposed AutoCodeRover, which utilizes one LLM agent to search through a codebase to retrieve information about the project and source code snippets, and another LLM agent to generate patches to fix bugs. Bouzenia et al. [11] presented RepairAgent, which consists of three key components. They utilized an LLM to query with a dynamic prompt, 14 tools covering different steps a human might take to resolve software bugs, and a middleware to orchestrate communication between those tools and the LLM. Ye et al. [82] introduced AdverIntent-Agent, which consists of three different agents. Each agent performs the following tasks separately: reason about program intent, generate test cases, and produce patches for program repair. In addition to this academic work, the efficacy of agents from frontier labs, such as Claude Code from Anthropic [9], Codex from OpenAI [55], and the Gemini coding CLI from Google [18], has continued to improve to the point where they are gaining significant industrial traction.

Our research primarily aims to understand how to adapt and measure the efficacy of *traditional* APR techniques to repair only the buggy Java code in Android apps. Rather than exploring the entire source code, these APR techniques generate code to fix a bug for a given buggy code. We replace the buggy code with the tool-generated code to see whether fixing the code without making any other code changes is sufficient to fix bugs. In essence, before exploring how well agents

perform on UI-centric program repair, it is important to understand how well these more traditional techniques function and the limitations that they currently have. Given the rising cost of using autonomous agents for tasks such as bug fixing, it is important to understand how simpler techniques function, and their limitations. While these agent-based program repair techniques show effectiveness in repairing various software bugs, recent evidence suggests that current SWE agents still face important limitations [32]. Specifically, they often over-retrieve noisy context, under-utilize retrieved evidence, and struggle with cross-file and repository-wide information. As such, we believe that agentic repair of UI related issues deserve their own, separate study, where agent trajectories, failure modes, prompting templates, etc. are studied in detail.

3.3 Existing Datasets of Mobile App and UI-centric Bugs

Given the importance of bug report management for mobile apps, there are existing datasets that are targeted for bug reproduction, localization, and testing. The AndroR2 [72] and AndroR2+ [27] datasets contain a combined 180 reproducible bugs mined from open source Android applications. These datasets have supported work on bug reproduction [69, 85] and bug reporting [64, 65]. The Themis dataset [66] contains 52 reproducible crash bugs mined from open source Android applications for the purpose of benchmarking automated testing tools for Android. Both of these datasets lack the localized code, test cases, and correct patches required for APR tools to properly function. The DroixBench [67] dataset contains 24 crash bugs for applying a search-based program repair technique, as described in Section 3.1. However, as illustrated in Section 2.2, in practice, there can be many other types of UI bugs, such as output/content errors, cosmetic bugs, and navigation bugs [27, 72]. If we evaluate APR techniques on only one type of bug, we cannot understand the capabilities of APR tools, because different types of UI bugs require different reasoning strategies, contexts, and underlying program analysis. Gao *et al.* [16] introduced the SITAR tool and dataset for Android bugs, but this dataset is centered on repair of *test scripts* and not bugs in applications, making its focus different from our work. Xiong *et al.* [76] introduced a large dataset of non-crashing, functional bugs in Android apps, for the purpose of better understanding their properties and assessing the effectiveness of automated testing tools. This dataset contains 399 bugs, of which 265 were reproducible at the time the dataset was published. However, this dataset does not include statement level localization information or the test cases required for APR techniques to function – further, it is not clear how many of the buggy commit snapshots would be compilable – a key requirement for APR. Finally, a recent addition to the SWE-Bench suite of benchmarks, called SWEBench Multimodal [80], aimed to evaluate emerging SWE agents on issues that contain a visual component. This benchmark contains four different task definitions that contain issues related to (i) diagramming, (ii) interactive mapping, (iii) syntax highlighting, and (iv) javascript web frameworks. While this benchmark does include javascript web frameworks, the issues are related to the *frameworks themselves*, and not to client apps that make use of the framework to implement user-facing apps. As such, SWE-multimodal does not capture many of the challenges related to the UI programming paradigms discussed earlier.

3.4 Approaches for Detecting and Resolving Cosmetic UI Issues

Most existing research on the detection and resolution of UI-related issues [14, 61] has concentrated on addressing a subset of the cosmetic type of UI-centric bugs. Liu et al. [39] proposed OwlEyes, which employs deep learning techniques to detect and mitigate UI display issues. Woodpecker [38] further advances such research by detecting display issues in screenshots, localizing the corresponding bugs in source code, and automatically fixing UI-related defects. This tool utilizes computer vision techniques, view hierarchy file analysis, and predefined repair templates to fix UI bugs. Coala, which is introduced by Mehralian *et al.* [45], also leverages deep learning approaches to address

problems with text labels and icons in graphical user interfaces. Zhang et al. [83] proposed Iris, a context-aware framework specifically designed for repairing color-related UI issues.

The main limitation of these prior techniques lies in the limited scope of the bugs they detect and/or resolve, as they typically target only a narrow slice of the cosmetic bug category defined earlier. In contrast, DroidFixBench, as described in Section 4.1 includes not only visual UI display issues but also bugs across all discussed UI-centric bug categories, and these bugs involve functional and interaction issues, such as event handling, backend API interactions, application logic, and GUI state management. In consequence, DroidFixBench's bugs are challenging to resolve solely through visual analysis or with a single program repair technique, such as predefined template-based or learning-based approaches, because they require reasoning about program information, runtime behavior, inter-component communication, and event-driven execution flows.

4 Design of Empirical Study

The goal of this study is twofold: (i) to investigate whether we can use existing APR techniques to fix UI-related Android bugs, and (ii) to identify the challenges that must be addressed for future techniques that target UI-centric bugs. With this in mind, we formulate the following research questions:

- **RQ**$_1$: *How do different types of program repair approaches perform on UI synthetic bugs?*
- **RQ**$_2$: *How do different types of program repair approaches perform on UI real bugs?*
- **RQ**$_3$: *What types of errors are commonly observed in the generated bug-fixing patches?*

To answer these research questions, we created the DroidFixBench dataset consisting of 50 real and 46 synthetic UI-centric bugs in Android apps. The main reason for exploring ***both*** synthetic and real bugs is to explore how APR techniques fare when applied to bugs of varying *complexity*, hence deriving a more complete picture of their overall effectiveness at repairing UI-centric Android bugs. We explore five baselines described in Section 4.2. We would also like to stress the significant manual effort required for sourcing our dataset of real bugs – from compiling buggy snapshots, to reproducing and localizing the bugs, to writing test cases that properly expose the bug – *several person months of effort were required to source this dataset.* Many historical snapshots are also generally not compilable without significant effort. Our dataset spans multiple popular open source apps and bug types, making it reasonably representative of the variety of bugs found in the wild.

4.1 Dataset Construction

As no prior dataset supporting APR for *generalized* UI-centric bugs exists, we constructed our own, ensuring that each bug is reproducible and manifests in the corresponding app's UI. Our dataset consists of two main parts: (i) a set of 50 real bugs (Section 4.1.1) mined from popular open source Android apps, and (ii) a set of 46 synthetic bugs (Section 4.1.2), created using an Android specific mutation analysis tool, to help expand the diversity of our dataset while reducing the manual collection effort.

4.1.1 ***Real Bugs.*** Our real bugs portion of DroidFixBench was constructed from three sources: (i) a set of 80 bug reports primarily developed for bug localization proposed by Mahmud et al. [43, 44], (ii) the real bug reports found in Github repositories for five additional applications that were also used for creating the synthetic dataset as described in Section 4.1.2, and (iii) a set of 152 bug reports primarily developed for generating failure-based oracles by Johnson et al. [26].

Selecting Bugs from Mahmud *et al.'s* Dataset - To begin the construction of our dataset, we first turned to the set of 80 bug reports provided by Mahmud et al. [43, 44], which were primarily developed for file-level bug localization, and were derived from AndroR2+ [27]. Note that in this dataset, all bug reports had a predetermined commit ID where a bug was found and another commit ID that fixed the bug. We reviewed all the bug reports in descending order based on their assigned

Table 1. Real Bugs Dataset Code Complexity. This includes the number of files, methods, and lines added or subtracted for each bug as a proxy to examine bug complexity.

Bug IDs	#Files	#Methods	#Edits$^-$	#Edits$^+$	#Total Edits	Bug IDs	#Files	#Methods	#Edits$^-$	#Edits$^+$	#Total Edits
harmonic-114	2	2	0	9	9	andror2-159	1	1	1	1	2
harmonic-136	1	1	2	2	4	andror2-160	1	1	0	2	2
harmonic-171	1	1	1	1	2	andror2-191	2	2	12	15	27
harmonic-181	1	2	6	28	34	andror2-192	1	3	0	13	13
urlcheck-358	1	1	3	4	7	andror2-271	1	1	2	2	4
urlcheck-365	1	2	8	4	12	andror2-275	2	4	8	9	17
urlcheck-382	1	1	6	5	11	andror2-1130	2	2	18	27	45
urlcheck-447	1	1	4	4	8	andror2-1146	1	2	1	2	3
skytube-1255	1	1	6	1	7	andror2-1151	2	3	5	9	14
pslab-2587	1	1	8	9	17	andror2-1198	1	1	6	16	22
pslab-2677	1	5	11	10	21	andror2-1202	1	4	16	9	25
pslab-2678	1	1	3	2	5	andror2-1205	1	1	4	6	10
gpslogger-1138	1	2	2	2	4	andror2-1207	1	1	1	5	6
fbo4a-273	1	2	0	10	10	andror2-1213	4	5	8	6	14
fbo4a-123451	1	1	4	1	5	andror2-1214	1	1	36	56	92
andror2-8	1	1	0	1	1	andror2-1215	1	1	12	3	15
andror2-18	3	6	7	7	14	andror2-1299	1	1	0	3	3
andror2-44	1	1	2	3	5	andror2-1441	1	1	5	6	11
andror2-53	1	1	1	3	4	andror2-1445	1	1	1	3	4
andror2-54	1	1	1	1	2	andror2-1446	1	4	11	7	18
andror2-71	1	3	0	10	10	andror2-1563	1	1	2	8	10
andror2-84	1	1	1	1	2	andror2-1568	2	3	3	3	6
andror2-92	1	3	3	7	10	andror2-1640	2	2	2	3	5
andror2-129	1	1	39	44	83	andror2-1641	2	3	11	16	27
andror2-130	1	5	6	14	20	andror2-1645	1	1	1	1	2

bug IDs. For each bug report, we attempted to build the source code of the buggy application version by checking the buggy commit ID reported in their released dataset, given that APR tools will need to make changes to and build the buggy version of the program to function. We were able to initially build the source code with no or minor modifications in project dependencies for only 15 bug reports (compiling historical versions of Java-based software has been shown difficult [68]). Later, by handling .gradle version mismatches, resolving deprecated dependency issues, and using newer sources for dependencies, we built the source code for 27 additional bug reports. In total, we were able to build 42 out of 80 bug reports. Of these 42 reports, we exclude seven due to an inability to reproduce the bug. For the remaining 35 bug reports, two authors compared the buggy commit IDs with the fixed commit IDs and identified the code changes required to fix the bugs.

Writing Test Cases - Then two authors constructed two test cases for each bug verifying that the test cases fail on the buggy code and pass on the fixed code. The first test case is designed to exercise the steps of bug reproduction and verify if the expected behavior is present. Then, a second test case is written to ensure that the related UI functionalities associated with the bug reproduction steps continue to function correctly – typically with multiple assertions. The main reason for having multiple test cases is that during patch generation, it is possible that an APR technique fixes the target bug, but introduces another one in related UI functionality, and we aimed to observe such a phenomenon. Bugs were divided among two paper authors. Half of the test cases were written by each author, and the author who did not write the original test case reviewed it for correctness.

Sourcing Additional Recent Real Bugs - Given the difficulty in finding reproducible bugs that also contained a compilable buggy snapshot, we aimed to increase the diversity of DroidFixBench, while targeting more recently updated apps containing recent snapshots that were more likely to compile. As such, we selected five applications from F-droid [2] by applying filtering criteria. We only considered the applications implemented in Java, were cross listed on Google Play, and where the latest update was in 2025. All the applications were then ordered by Google Play star rating from highest to lowest. Then, we divided the applications by app category (e.g., productivity), and examined the highest rated app in each category. From the first app category, we selected the top-ranked application and attempted to compile the latest released source code available in its GitHub repositories. If the project build was unsuccessful, we proceeded to the next application in this

category. If we were unable to build any applications in a category after five attempts, we excluded that category from our study and proceeded to the next app category. If the project build was successful within the first five attempts, we selected the application from this category in our dataset. Following this procedure, we selected five open-source applications from five different Google Play categories: (i) GPSlogger [3] from navigation, (ii) URLCheck [8] from Connectivity, (iii) SkyTube [7] from Internet, (iv) HarmonicHN [5] from Reading, and (v) PSLab [6] from Science and Education.

From the five applications we selected, we systematically reviewed the closed issues reported in their GitHub repositories. We sorted the issues from newest to oldest and went through each bug individually. Specifically, we checked all bugs labeled "bug" and, in cases where no specific label was present, we analyzed the bug report to determine if it might refer to a bug. Following this process, we selected six bug reports per application, resulting in a total of 30 bug reports.

Filtering the Recent Bugs - From these 30 bug reports, we selected a subset that fit our study criteria. We excluded two bug reports because they were pull requests incorrectly identified as bugs. One key characteristic of a bug report used for APR is that it must contain a fixed commit ID to track the code changes that address the bug. However, one bug report did not have a fixed commit ID, and we excluded it from our study. From the remaining 27 bug reports, we attempted to build the application with the version of a commit ID that was released immediately before the bug was reported. In this step, we excluded four bug reports because we were unable to build the source code. Then, we checked if we could reproduce the bug on a Pixel 2 emulator running Android 9. In this step, we excluded two bug reports because we were unable to reproduce one bug, and the other required an external Android device for reproduction. For the remaining bug reports, we compared the source of the buggy version of the project with the commit ID that fixed the bug. Two annotators reviewed the code changes between the buggy and fixed commit IDs and identified the necessary changes to fix the bug. Note that we only considered the .java file changes that were necessary to fix the bug. Two bug reports only had modifications in .gradle or .xml files; therefore, we excluded those bug reports from our study. From the remaining 19 bug reports, we constructed two test cases as outlined above. However, six bug reports required complicated bug reproduction steps, and it was difficult to write test cases. Therefore, we excluded those six reports. This process resulted in 13 bug reports across five applications, which have all the necessary APR components.

Sourcing Further Additional Real Bugs - To further expand the size of our dataset, we sourced another dataset named FBO4A, consisting of 152 bug reports provided by Johnson et al. [26]. We explored each bug report sequentially from the lowest to the highest bug IDs. For each bug ID, we first checked whether the bug report was already included in the 48 bug reports we had collected so far from the previous two sources. If the bug report already existed in our collected dataset, we excluded it from further consideration and moved to the next bug report. Next, we checked whether the bug report includes a buggy and a fixed commit ID, and whether the fixed commit ID is not tangled (i.e., commits with many method changes). The next selection criterion is to check if at least one .java file change exists in the fixed commit IDs. Then we checked whether the project could build successfully, and whether or not we were able to reproduce the bug on the Pixel 2 Android 9 emulator. If the bug reproduction is successful, we write the test cases for that specific bug. Following these steps, we reviewed 22 bug reports and added two more bugs to our dataset. We stopped exploring additional bugs once we collected 50 bugs from all three sources.

In total, we selected 50 real bug reports across 23 applications from those three sets of bug reports. Two authors categorized the bug reports into three different bug types, with a Cohen's Kappa score of 0.72. In those bug reports, 12 were crash failures, 25 were output failures, 9 were cosmetic failures, and 4 were navigation failures. This entire process for creating the real dataset took multiple person months. In this dataset, 27 bug reports have bugs in a single Java method, and 23 bug reports have bugs in multiple Java methods. The Cohen's Kappa score for creating the test cases was 0.84 and

Table 2. Distribution of Mutant Types for Synthetic Dataset. Mutant Descriptions derived from [34].

Mutant Types	Description	Counts
NullIntent	Replace an Intent instantiation with null	5
DifferentActivityIntentDefinition	Replace the Activity.class argument in an Intent instantiation	5
LengthyGUICreation	Insert a long delay (i.e., Thread.sleep(..)) in the GUI creation thread	5
NullValueIntentPutExtra	Replace the value argument in an Intent.putExtra(key, value) call with new Parcelable[0]	5
LengthyGUIListener	Insert a long delay (i.e., Thread.sleep(..)) in the GUI listener thread	3
ViewComponentNotVisible	Set visible attribute (from a View) to false	5
InvalidIDFindView	Replace the id argument in an Activity.findViewById call	5
FindViewByIdReturnsNull	Assign a variable (returned by Activity.findViewById) to null	5
InvalidKeyIntentPutExtra	Randomly generate a different key in an Intent.putExtra(key, value) call	5
BuggyGUIListener	Delete action implemented in a GUI listener	3
	Total	**46**

disagreements were resolved through discussion and the involvement of a third author, if necessary. Among the 50 real bugs, 290 lines are removed, and 414 lines are added to fix the bugs. These fixes required 97 method changes in 63 Java files. Table 1 illustrates the complexity of all 50 real bugs.

4.1.2 ***Synthetic Bugs***. Due to the difficulties and manual effort required to collect fully localized, compilable, bugs with test cases, we opted to include an additional set of *synthetic bugs*, generated via an Android-specific mutation analysis tool. To accomplish this, we used the 5 more recent popular apps, described above, and we applied the MDroid+ mutation testing framework, MDroid+ [34, 47]. This tool supports 38 different mutation operators corresponding to ten different Android fault categories empirically derived from a large scale study. We applied MDroid+ to the source code of our 5 selected applications, and collected all the mutants applied to Activities or Fragments, or the classes representing UI screens, resulting in 16-22 mutants generated for each of the 5 apps. We only chose mutants that appear in Activities and Fragments, as we wanted to ensure the bugs manifest through the UI, and cover the UI programming paradigms discussed in Section 2.

For each of the mutation operators, we randomly selected one method change in a .java file of an Activity or Fragment. Then, we applied the change to the actual source code and attempted to build the project. Then, we attempted to reproduce the bug in a Pixel 2 Android emulator with Android 9. If the project did not build or the bug reproduction was unsuccessful, we randomly selected another injected mutant. If we could not build and reproduce any bugs from a mutation operator category, we excluded this category in constructing the dataset.

After completing all these steps, we created **46 bug reports** by applying ten different mutation operators across five applications. Specifically, we created ten bug reports each from GPSLogger, HarmonicHN, and PSLab, and 8 bug reports each from SkyTube and URLCheck. These 46 bug reports are categorized into the four categories discussed in Section 2 and one additional type – delay – signifying sluggish UI responsiveness. Two authors independently reproduced the bugs, ensured that they manifested through the app UI, and categorized these bug reports into different bug types with a Cohen's Kappa score of 1.0. In addition, one author created a bug report based on how the buggy behavior manifests in the UI screen(s), capturing observed behavior (OB), expected behavior (EB), and reproduction steps (S2Rs). A second author reviewed each bug report for accuracy. In total, our synthetic dataset contains 19 crash failures, eight output failures, two navigation failures, nine cosmetic failures, and eight delay failures. This entire process of collecting this synthetic dataset again took multiple person months, and the Cohen's Kappa score for the test case construction was 0.98. In total, among the 46 synthetic bugs, 76 lines are removed, and 49 lines are added to fix the bugs. Each fix required a single method change in a single Java file. We provide a breakdown and description of the different mutant types in Table 2.

4.2 Program Repair Techniques

We selected a diverse set of *traditional* APR techniques spanning template-based, LLM-based, and neural machine translation (NMT)-based approaches, following the categories identified in recent APR surveys [21]. Selection was constrained by the availability of reproducible research

artifacts that could be operationalized in our study. Our goal was to compare the strengths and limitations of representative repair paradigms on UI-centric Android bugs. We used each technique and underlying model as described in its original paper.

4.2.1 ***D4C.*** D4C [77] is a one-shot prompting framework that repairs entire functions rather than buggy hunks by aligning repair with the LLM pre-training objective. It takes a buggy program, bug report, and failed test information as input and generates a repaired program without fine-tuning. We used the authors' prompt template with GPT-4 [54], although the framework also supports Mixtral-MoE [24].

4.2.2 ***ChatRepair.*** ChatRepair [73] is a conversational APR approach that iteratively refines patches using execution feedback. Given buggy code and test failure information, GPT-3.5 [54] generates a candidate patch, which is validated against the test suite. Failed test results are incorporated into subsequent prompts while avoiding previously generated patches until either a passing patch is found or the maximum number of attempts is reached. When a passing patch is found, ChatRepair also searches for alternative valid patches.

4.2.3 ***OpenHands.*** OpenHands [70] is an AI agent framework that performs developer-like actions (e.g., coding, command-line interaction, and web browsing) using an event stream and a Docker-based execution environment. Although designed for general software engineering tasks, we used it for program repair by prompting it with the bug report, buggy code, and failed test information. We used Qwen2.5-Coder [23] as the underlying model.

4.2.4 ***NTR.*** Neural Template Repair (NTR) [22] combines template guidance with LLM-based repair in two stages. It first predicts a repair template via classification, then guides an LLM to generate a patch using the selected template or a special *OtherTemplate* for repairs outside the predefined template set. Following the original approach, we selected a template for each bug before patch generation. We used CodeLlama [59] instead of StarCoder [33], as prior work reports better repair performance [59]. For inputs exceeding the model's context window, we applied *OtherTemplate* directly.

4.2.5 ***RewardRepair.*** RewardRepair [81] is an NMT-based repair technique that incorporates compilation and test execution feedback into training, rewarding patches that compile and pass tests rather than optimizing only token-level similarity. The model requires a buggy line and surrounding context; whereas the original work used ten surrounding lines, we provided the buggy line (when available) together with the entire buggy method. We generated candidate patches using the authors' released pre-trained models.

4.3 Methodology to Answer Research Questions

4.3.1 ***Metrics.*** We use two metrics commonly used in prior program repair literature [71, 73, 75, 77]. The first metric is the *plausible patch* rate, which measures whether the generated patches pass all test cases. The second metric, *correct patch* rate, investigates if a plausible patch semantically matches the ground-truth developer provided or mutation supplied patch. The third metric is *overfitting patch* rate: if a generated patch passes the test cases but only due to program behavior or output that is specifically targeted at passing the test case, instead of fixing the bug in a more general manner, we label it an *overfitting patch* [63]. Patches that do not build or do not pass the test cases are labeled as *not plausible patches*. To calculate these metrics, for each plausible patch, we validated if the patch is either (i) correct (matches the ground truth), (ii) does not match the ground truth (*i.e.,* plausible, but not correct), or (iii) overfits to the test cases. Note that the patches that are overfit to the test cases are a subset of plausible patches that *do not match the ground truth.*

Given the qualitative nature of this validation process we iteratively designed a set of guidelines with examples for labelers to follow in order to standardize the definitions of the three categories mentioned above. These guidelines and examples are available in our online replication package [42]. Then, following these guidelines, each patch was reviewed by two authors to identify the categories it falls into, and in the event of disagreements, the two authors discuss them to reach a consensus. In total, we checked 457 plausible patches for the synthetic bugs and 169 plausible patches for the real bugs. The Cohen's Kappa score for the synthetic bugs was 0.86, and for the real bugs, it was 0.96, exhibiting a high level of agreement.

4.3.2 ***RQ1 & RQ2: Automated Program Repair approaches on Synthetic and Real Dataset.*** We investigate five existing automated program repair approaches for both real and synthetic bugs, as described in Section 4.2. (i) D4C [77] generates ten patches for each buggy method associated with a bug report. So, we investigated 460 patches for synthetic bugs and 500 patches for real bugs. (ii) ChatRepair [73] investigates multiple candidate patches in its internal pipeline in searching to generate the correct patches. We consider the final patches generated by ChatRepair. For synthetic bugs, we utilize the single hunk patch generation strategy, as it is the best-performing setting described in the original paper in their dataset. This strategy attempts 200 times to generate patches to fix bugs. For real bugs, buggy lines exist in multiple places in the code; therefore, we utilized a single-function patch generation strategy instead of a single-hunk strategy for this dataset. This strategy attempts 100 times to generate patches to fix bugs. These numbers of attempts were taken from the experiments in the original paper [73]. (iii) NTR [22] generates patches prioritizing the order that is more likely to contain a bug. If NTR cannot generate any patches based on the given templates, it generates patches using the OtherTemplate option. In cases where more than ten patches are generated for a single bug, we utilize the top ten patches. In total, we studied 445 generated patches for the synthetic bugs and 492 patches for the real bugs. (iv) OpenHands [70], generates one patch per bug. So, we investigated 46 patches for the synthetic bugs and 50 patches for the real bugs. (v) For RewardRepair [81], we generate ten patches per bug and investigate those in our study.

4.3.3 ***RQ3: Qualitative Analysis of the Generated Unsuccessful Patches.*** In our evaluation, we not only calculate plausible and correct patches, but also qualitatively investigate the issues with the generated patches. Sections 5.1 and 5.2 present the results for plausible and correct patches. The goal of our qualitative analysis is to identify where APR techniques generate unsuccessful patches rather than the successful patches. From those results, we find 184 plausible but not correct patches and 2,423 not plausible patches, which could not fix the bugs. With a 90% confidence interval and 5% margin of error, we select 117 patches from the plausible but not correct set and 262 patches from the not plausible patches. In total, our qualitative study includes 379 sample patches.

To categorize our sample patches, we adapt an open-coding [46] process commonly practiced in software engineering research. We initially select a subset containing 41 samples of diverse types, which we define as the initial set of data to derive an initial codebook. Three annotators mutually discuss and categorize those 41 sample patches. The remaining 338 samples are divided among the annotators, ensuring that two annotators categorize each patch. Then, the patches are labeled based on whether the issues with the patches are present in the initial codebook; the annotators would label them in that category. In case the category is not present in the codebook, the annotators define new categories to identify the issues. Finally, once the categorization of the patch issues is complete, the annotators mutually discuss their opinions on the categorization of the issues with the patches. Note that even if there are slight differences in the naming of the categories, if the annotators identify the same problems, we consider those as agreements. In total, we find agreements in categories with a Cohen's Kappa score of 0.70, indicating strong agreement. When a disagreement occurs, the two annotators discuss to resolve disagreements and finalize the categories.

4.4 Implementation

For our implementation, we utilize the open-source code available for each of our 5 studied APR techniques. We adapt the default experimental settings for each of those approaches. For D4C, ChatRepair, and OpenHands, we utilized GPT-4, GPT-3.5-turbo-0301, and Qwen2.5-Coder LLMs, respectively, to generate responses, as these were the best performing underlying models from the original papers. D4C with GPT-4 generates ten patches per bug and per API call costs $0.03. In the case of ChatRepair, the maximum attempts for generating correct patches are 200 for the bugs from the synthetic dataset using the single hunk patch generation strategy and 100 attempts for the bugs from the real dataset using the single-function patch generation strategy from the original paper [73]. This approach uses GPT-3.5-turbo-0301 and, per API call, costs $0.0005. We generate one patch per bug for Qwen2.5-Coder for OpenHands and per API call costs $0.287. For these three approaches, we generate patches using macOS Sequoia 15.6 and 16 GB RAM and an M1 Pro chip.

We utilized CodeLLaMa to generate responses for NTR. We evaluate the top ten ranked patches for NTR, which were generated using a Linux system with dual NVIDIA H100 GPUs with a total of 160GB RAM. For generating patches using RewardRepair, we utilized the pre-trained models released by the original authors. We generated ten patches per bug using a Rocky Linux 9.2 Cluster with 1 NVIDIA A40 24GB GPU and 512 GB of RAM. In the evaluation, all generated patches are tested with two separate test cases for each bug on MacOS Sequoia 15.6 and 16 GB RAM. We use Pixel 2 Android 9 emulator. Running per test case takes approximately two minutes and 30 seconds.

5 Results

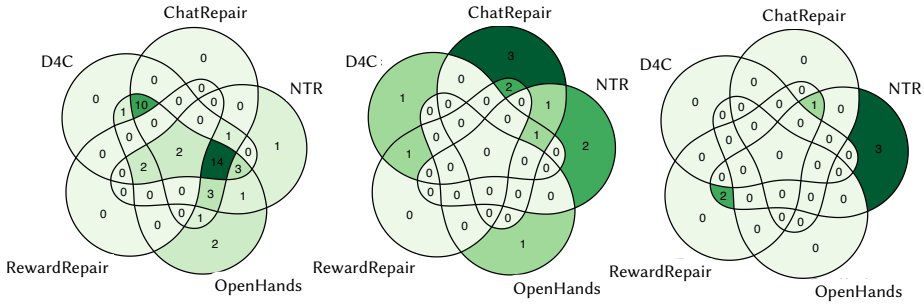
We present the results for each of the research questions in the order described in Section 4. Of the 46 *synthetic* bugs from five different Android apps, 43 bugs are fixed by at least one program repair approach. Moreover, of the 50 *real* bugs from 23 different Android applications, 30 bugs are fixed by at least one of the program repair approaches, with the best technique only fixing 19 real bugs. We explore 379 non-plausible and plausible but not correct patches, and identify that for 159 patches, they often fail because of the GUI-specific code.

5.1 RQ1: Synthetic Dataset Results

Table 3 shows the results for all generated patches for five APR approaches and whether these patches are (i) correct (plausible and match the ground truth patch), (ii) plausible but not correct (do not match the ground truth), (iii) overfit to the test cases (a subset of plausible but not correct patches) [63] (iv) failed in running test cases, (v) failed in building the application, and (vi) failed in replacing or generating patches. The last three cases correspond to subcategories of not plausible patches. Note that not all program repair approaches generate the same number of patches; however, we evaluate all patches generated by each approach. The highest percentage of correct patches is generated by OpenHands, where 39 out of 46 (≈84.78%) generated patches are correct. In addition, more than half of the generated patches are correct for D4C (234/460≈50.87%) and ChatRepair (31/46≈ 67.39%). One common reason these three APR approaches generate more correct patches than the others is that they utilize LLMs in their internal architecture that use a large number of parameters. In the case of NTR, 77 out of 445 generated patches are correct because this approach uses program repair templates, where most of the templates are designed for non-UI coding paradigms; no templates are designed specifically for the UI-specific code. The least successful patches we receive for RewardRepair, which generates only 18 out of 460 plausible patches. The reason for this small number of correct patches is that this approach uses a pre-trained model, which is not fine-tuned on any UI-related bugs. In addition, this approach only uses buggy methods with specific buggy lines but does not use additional contexts from bug reports and test cases. We also

Table 3. APR Effectiveness on the **Synthetic** Bugs portion of DroidFixBench.

	Plausible Patches				Not Plausible Patches				
Approaches	**# (%) Plausible Patches (All)**	**# (%) Correct Patches**	**# (%) Not Match Ground Truth**	**# (%) Overfit Patches**	**# (%) Not Plausible Patches (All)**	**# (%) Output Issues**	**# (%)Tests Failures**	**# (%) Build Failures**	**# Total Patches**
D4C	258 (56.09%)	234 (50.87%)	24 (5.22%)	0 (0.00%)	202 (43.91%)	2 (0.43%)	61 (13.26%)	139 (30.22%)	460
ChatRepair	38 (82.61%)	31 (67.39%)	7 (15.22%)	1 (2.17%)	8 (17.39%)	1 (2.17%)	2 (4.35%)	5 (10.87%)	46
OpenHands	40 (86.96%)	39 (84.78%)	1 (2.17%)	0 (0.00%)	6 (13.04%)	0 (0.00%)	3 (6.52%)	3 (6.52%)	46
NTR	103 (23.15%)	77 (17.30%)	26 (5.84%)	9 (2.02%)	342 (76.85%)	0 (0.00%)	201 (45.17%)	141 (31.69%)	445
RewardRepair	18 (3.91%)	8 (1.74%)	10 (2.17%)	6 (1.30%)	442 (96.09%)	0 (0.00%)	93 (20.22%)	349 (75.87%)	460
Overall	**457 (31.37%)**	**389 (26.70%)**	**68 (4.67%)**	**16 (1.10%)**	**1,000 (68.63%)**	**3 (0.21%)**	**360 (24.71%)**	**637 (43.72%)**	**1,457**



(a) Correct Patches (b) Plausible but Not Correct Patches (c) Overfit Patches

Fig. 2. Breakdown of Different Types of APR-generated Patches on the **Synthetic** Portion of DroidFixBench.

observe that mutants involving Null Values or Intents are comparatively more difficult to fix, with a subset of bugs generated with the NullIntent, NullValueIntentPutExtra, InvalidKeyIntentPutExtra, FindViewByIdReturnsNull, and BuggyGUIListener operators not being fixed by any technique.

Moreover, Figure 2c shows the cases where the generated patches overfit to the test cases. We observed 16 patches from the synthetic dataset that overfit to the test cases. All those patches are for 6 bugs related to the delay bug type (i.e., sluggish UI responsiveness). In our test cases for these bugs, to check whether the UI screen is displayed within 3 seconds, the measured time between the last event and the UI display must be within 3 seconds (i.e., `assertTrue(elapsedTime <= 3000)`). To fix such bugs, an ideal patch would remove the thread-sleep delay. However, these 16 patches only reduce the sleep delay rather than removing the try-catch block to pass the test cases.

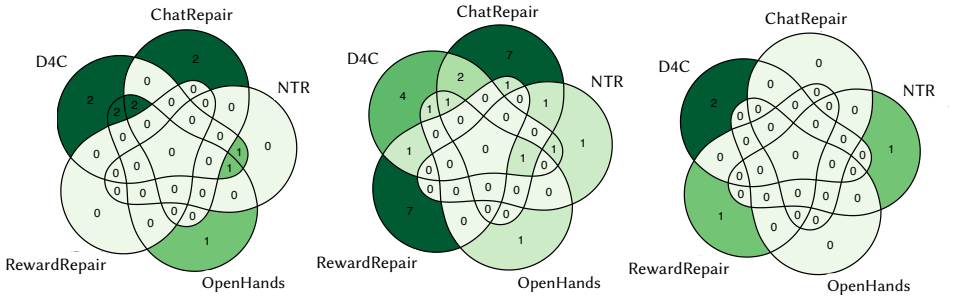
While the APR techniques generate a different number of patches, we also investigate *the number of bugs fixed* i.e., based on the number of plausible patches by each of these approaches. Among all the patches generated by an APR for a bug, if we find at least one plausible patch, we consider that the bug is fixed with that particular patch. Figure 2a and 2b show the results of the *plausible* patches generated by different program repair approaches, and in total, those approaches fixed 43 out of 46 bugs. This means that most APR techniques can fix bugs that require small modifications.

We also investigate where the generated patches are syntactically correct but *fail to fix* bugs, i.e., they compile but fail the test cases. D4C, NTR, and Reward Repair have the highest rate of these types of patches, whereas ChatRepair and OpenHands have lower rates of such patches. This is likely due to the fact that OpenHands is using a more recent LLM and ChatRepair has a test case feedback mechanism that allows it to refine test cases based on result feedback.

Summary of Findings for RQ1: Our study reveals that more than half of the generated patches are correct for D4C, ChatRepair, and OpenHands. RewardRepair generates the least number of successful patches because it uses a pre-trained model and is not fine-tuned on UI bugs. It only uses the buggy method and marked buggy lines, without leveraging contexts from bug reports or test cases. Overall, 43 out of 46 synthetic bugs are fixed by at least one of the program repair approaches.

Table 4. APR Effectiveness on the **Real Bugs** portion of DroidFixBench.

	Plausible Patches				Not Plausible Patches				
Approaches	**# (%) Plausible Patches (All)**	**# (%) Correct Patches**	**# (%) Not Match Ground Truth**	**# (%) Overfit Patches**	**# (%) Not Plausible Patches (All)**	**# (%) Output Issues**	**# (%) Tests Failures**	**# (%) Build Failures**	**# Total Patches**
D4C	100 (20.00%)	40 (8.00%)	60 (12.00%)	17 (3.40%)	400 (80.00%)	18 (3.60%)	65 (13.00%)	317 (63.40%)	500
ChatRepair	17 (34.00%)	4 (8.00%)	13 (26.00%)	0 (0.00%)	33 (66.00%)	21 (42.00%)	5 (10.00%)	7 (14.00%)	50
OpenHands	10 (20.00%)	6 (12.00%)	4 (8.00%)	0 (0.00%)	40 (80.00%)	2 (4.00%)	10 (20.00%)	28 (56.00%)	50
NTR	27 (5.49%)	3 (0.61%)	24 (4.88%)	1 (0.20%)	465 (94.51%)	48 (9.76%)	123 (25.00%)	294 (59.76%)	492
RewardRepair	15 (3.00%)	0 (0.00%)	15 (3.00%)	1 (0.20%)	485 (97.00%)	0 (0.00%)	57 (11.40%)	428 (85.60%)	500
Overall	**169 (10.62%)**	**53 (3.33%)**	**116 (7.29%)**	**19 (1.19%)**	**1423 (89.38%)**	**89 (5.59%)**	**260 (16.33%)**	**1074 (67.46%)**	**1592**



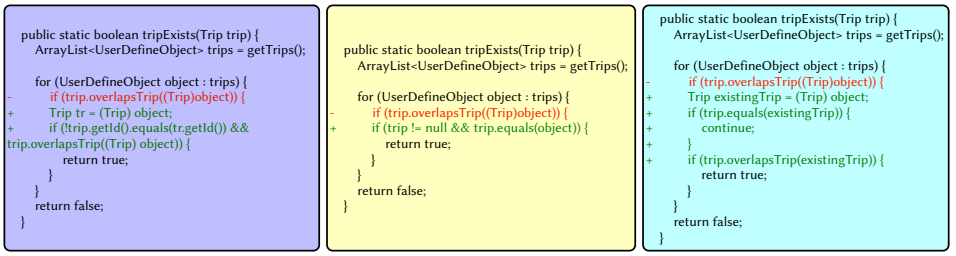
(a) Correct Patches (b) Plausible but Not Correct Patches (c) Overfit Patches

Fig. 3. Breakdown of Different Types of APR-generated Patches on the **Real Bugs** Portion of DroidFixBench.

5.2 RQ2: Real Dataset Results

Table 4 shows the results for five APR techniques on the real dataset. Among all APR techniques, OpenHands produces the highest percentage of correct patches (6/50≈12%). Both D4C and ChatRepair generate 8% correct patches. Although generating correct patches for real bugs can be challenging, these three APR techniques generate at least 8% correct patches where the other techniques fail. The reason can be attributed to the fact that these techniques use LLMs with a large number of parameters. RewardRepair generates the least successful number of patches. Although this approach generates 15 plausible patches out of 500 patches, none of them were correct. Moreover, we have observed 19 patches for 4 bugs in the real dataset that overfit to the test cases. From these bugs, one patch deletes characters from a text entry field to meet a specific test condition, nine patches always return the same boolean values expected by test cases, and nine patches assign hard-coded values specific to the test cases. For example, for bug 2678 in the pslab application [6], the correct patch requires rounding a value and assigning it to the `degree` variable (i.e., `degree = Math.round(seekArc.getProgress() * 3.6f)`). However, the generated patch assigns zero to the `degree` variable (i.e., `degree=0`) because the test case assertion contains that value (i.e., `assertEquals("0", inputBox2.getText())`).

We also investigate *the number of bugs fixed* in the real dataset using the five automated program repair approaches, i.e., those bugs that are plausible. Figure 3-a & 3-b shows the bugs that are fixed by the approaches used in our study. We have seen that only 30 out of 50 bugs are collectively fixed by at least one automated program repair approach, i.e., they generate plausible patches. D4C fixed 19 unique bugs (*i.e.,* under 50%), which is the highest among all these program repair approaches. This finding indicates that existing program repair approaches have clear limitations when fixing UI bugs, given the differences in GUI-specific code compared to more traditional programs. When examining the complexity of the subset of bugs that were fixed, we find that they tend to be the ones with fewer required changes in the developer patch, with an average of 1.09 methods changed and an average of 7.91 lines changed. This indicates that current APR techniques tend to struggle with more complicated bugs that have larger patch sizes. Figure 3-c illustrates the breakdown of the overfit patches across techniques.



(a) Ground Truth (b) Plausible but not Correct (c) Tests Failed

Fig. 4. Patch Examples

Moreover, we investigate the cases where the generated patches are not *correct*. We observe this scenario most frequently for NTR (for 16 bugs), as this template-based approach generates patches without utilizing a proper template for UI bugs. To show the limitations of different APR techniques on UI bugs, we provide examples of the diverse types of issues in generated patches. Figure 4a shows the code changes between the buggy version (highlighted in red) and the fixed version (highlighted in green) for bug 347 from Anglers-log application [1], which is a fishing-based application. In this bug, a user is unable to edit the date after creating a trip. Specifically, when a user creates a trip and then attempts to edit the date, the `overlapTrip` method should check if any overlap exists with other trips. However, this bug checks the overlap with the current trip, leading to errors in editing the date. The ideal fix would require checking overlap with the other trips.

We illustrate the errors with two patches generated by two other approaches. Figure 4b shows a patch generated by D4C. Although this patch passes the test cases, it is not correct. The reason is that `equals()` method compares object references instead of trip ID and `trip.equals(object)` returns False, which causes the buggy method, `tripExists` to return False. This bug overfits to the test case by simply manipulating the code to return a False boolean. Figure 4c shows another example patch generated by OpenHands, which fails in the evaluation with test cases. In this example, `trip.equals(existingTrip)` returns False; so, the buggy method, `tripExists` returns True. Although this method attempts to generate the correct conditional logic in code, the patch is incorrect because the approach lacks knowledge of the `getId()` method inside the Trip class in the project. To fix such issues, the program repair approaches should extract relevant contexts from the GUI and its underlying source code structure.

Summary of Findings for RQ2: OpenHands produces the highest percentage, *i.e.*,12% correct patches compared to any other program repair approaches. 30 out of 50 bugs are fixed by at least one of the program repair approaches. Notably, D4C fixed 19 bugs, which is the highest number of bug fixes among all approaches. These findings suggest that existing APR techniques need improvement to fix UI-specific code issues.

5.3 RQ3: A Taxonomy of Errors in the Generated Patches

Figure 5 presents a taxonomy of the errors with the generated patches identified during our qualitative study. These errors are found in the patches generated by five different program repair approaches applied to two different datasets. Section 4.3.3 describes the design of our qualitative study. We manually categorize the patches for 379 sample bugs into nine distinct categories. Each category is annotated at the top of each rectangle with the number of instances found in our study. All rectangles contain inner rectangles that represent the specific errors observed, alongside the number of samples. Note that these errors can be related to specific patterns in Android UI programming, Java-specific coding, or fundamental object-oriented programming. The particular errors in Android UIs are marked with the Android logo at the top right of each rectangle.

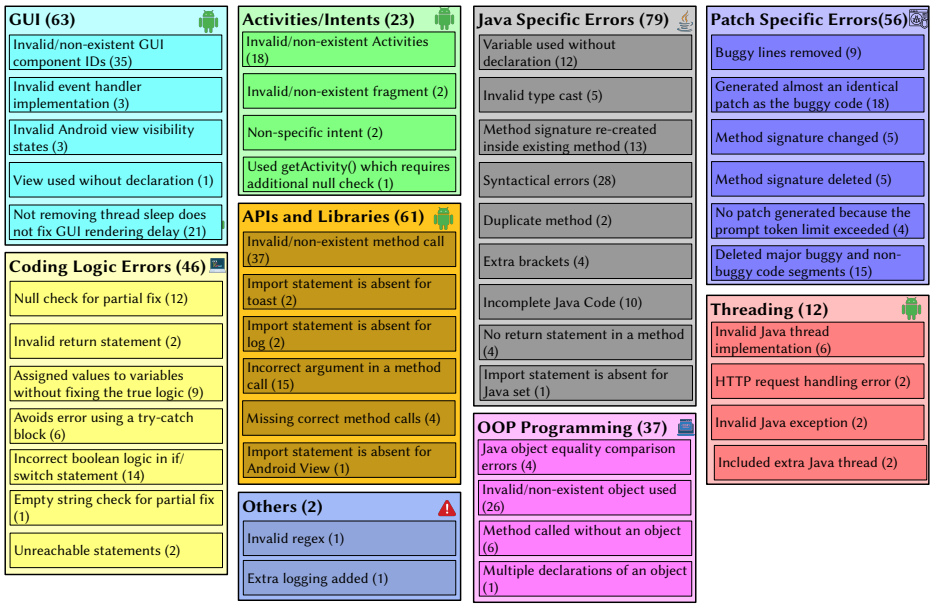


Fig. 5. **The Taxonomy of Errors in the Generated Patches.**

Takeaway 1: Among all the errors with the generated patches, 159 out of 379 patches (≈41.95%) relate to UI specific code. One of the most common patch-related errors that we observe is specific to the internal GUI structure and internal GUI-specific backend implementation. In our analysis, we categorize them into specific areas, including GUI and underlying code, activities or intents, invalid APIs and libraries, and other backend implementation errors. One of the most prevalent types of errors we find in our study is invalid GUI component ID usage. Specifically, we observe this type of error for 35 bugs, where the generated patches include component IDs that are either invalid or do not exist. These component IDs are generated based on the relevant contexts provided in the inputs to automated program repair approaches. As many GUI component IDs in the generated patches are invalid or non-existent, these patches are inadequate to fix bugs or lead to build failures.

We observe another error where the generated patches introduce activities or fragments based on the relevant contexts provided as input to the APR approaches. Specifically, we observe this error for 18 bugs, where the generated patches include invalid or non-existent activities, and for one bug, the patches include invalid or non-existent fragments. The automated program repair approaches lack knowledge of internal source code architectures and GUI screens, leading to the generation of patches with non-existent or invalid activities or fragments, which are inadequate to fix bugs.

To implement UI-specific functionalities, it is often necessary to call certain UI specific APIs. However, we find that in 37 bugs, patches include calls to particular methods that do not exist. In addition, for 15 bugs, methods are called with invalid method signatures. To generate patches for such bugs, the program repair approaches should have specific knowledge about the APIs available to handle the underlying GUI backend implementation for executing UI functionalities. We also observe errors in handling threads and exceptions for 12 bugs, which may impact the UI responsiveness after executing GUI actions such as clicks, swipes, typing, etc.

Takeaway 2: Automated program repair approaches may generate patches based on avoiding the observed behavior described in bug reports; however, in many cases, such patches are insufficient to fix bugs effectively. We observe minor modifications to buggy lines in an attempt to generate patches for 46 bugs. However, the buggy lines alone may not provide

enough context for the models to fix the bugs. For example, consider this buggy line from bug 347 of the Anglers-Log application [1]: `if (trip.overlapsTrip((Trip)object))`. D4C generates this patch to fix the bug: `if (object instanceof Trip && trip.overlapsTrip((Trip)object))`, which adds a boolean statement to the buggy code. However, this invalid code change is insufficient to fix this bug. Without knowing more about the context of the UI codebase or specific GUI functionalities, it is not possible to generate the correct patches for such bugs.

Takeaway 3: The program repair approaches may not make any changes in the buggy code or may remove buggy code segments instead of addressing the bugs. One common issue we observe with automated program repair approaches is that they do not make any modifications to the buggy code. We observe such cases in the patches for 18 bugs. Moreover, instead of fixing the buggy lines in the patches, the generated patches tend to remove the buggy lines of the method entirely. These types of changes are inadequate to fix bugs. We also observe a fair amount of hallucinating when referencing UI program constructs.

Summary of Findings for RQ3: We categorize 379 not plausible and plausible but not correct patches into nine different categories. Of these patches, we find that 159 patches (≈41.96%) fail to fix bugs due to the issues with GUI-specific code. Moreover, the generated patches may include syntactical errors that lead to build failures or may not make significant code changes to fix bugs.

6 Discussion & Future Research Directions

1. Instead of relying on automated program repair approaches to generate component IDs or activities/fragments, we should provide available GUI information to an automated program repair approach. We observe that automated program repair approaches tend to hallucinate component IDs and activities/fragments. However, these component IDs and activity/fragment names are often not validated to determine whether they exist in the project source code or not. Among the 379 incorrect patch samples in our study, 54 samples generate invalid or non-existent GUI component IDs and activities/fragments. To avoid compilation issues due to these types of errors and generate correct patches, the program repair approaches should incorporate those GUI components and activities/fragments contexts into their input. These contexts will likely increase the number of successful patches to fix bugs.

2. Automated program repair approaches should utilize the available APIs present in a project's source code. One common issue with the existing program repair approaches is that the generated patches call non-existent APIs or invoke certain APIs with incorrect arguments. We observed such errors in 52 samples in our qualitative study. This situation arises because the generated patch tends to hallucinate and calls these APIs without validating their existence. Rather than generating such APIs, we should provide available APIs and offer suggestions based on the relevant contexts. This information will help to avoid build errors and properly implement the GUI backend, leading to more buildable and correct patches.

3. Instead of providing GUI context in text format, future techniques should explore incorporating multimodal information, such as GUI screenshots and recordings, to provide additional context. Existing program repair approaches utilize bug reports, failure information, and test cases as inputs in their internal architecture. This is because these approaches are designed to fix traditional programs rather than UI code. The UI itself also contains additional information that can aid in generating accurate patches to fix UI bugs. In addition to providing textual GUI components and activities/fragments, we can also provide multimodal information such as GUI screenshots and recordings of bugs and related features. With this multimodal information, automated program repair approaches will likely generate more correct patches. Although GUI-related information is present in its internal source code architecture in textual format, we can utilize

the additional visual information to enhance the accuracy of the generated patches. Additionally, incorporating sequential bug reproduction GUI screens may provide more context.

4. Automated program repair approaches should validate the syntactical correctness of the generated patches. We observe 637 build failures for our synthetic dataset and 1074 build failures for our real dataset. To mitigate build failures, we must validate whether the generated patches are syntactically correct. We should develop a system that builds projects with the generated patches before generating the final patch. In case of build failures, the approach should attempt to fix them using the feedback messages received. The approach should attempt multiple times until a patch is generated with no build issues or until a certain number of iterations have been completed.

7 Threats to Validity

Internal Validity. Our initial internal threat to validity arises from our manual validation of the generated patches. However, each patch is validated by two authors to evaluate its semantic equivalence to the ground truth, which mitigates these patch validation issues. Another internal threat to validity arises in identifying the errors with the problematic patches, which we used to generate the taxonomy in Section 5.3. There may be situations where we cannot identify the precise errors in the patches. To address these limitations, we performed a systematic process of categorization to identify the errors in the generated patches. This categorization was ultimately performed with multiple annotators with a Cohen's Kappa score of 0.70.

External Validity. In our study, we examined five different APR approaches and analyzed the results. However, our findings may not generalize to all other existing program repair approaches. Our APR techniques come from a diverse set of categories and generally represent the state of the art. Our experiments are conducted using both a synthetic dataset of 46 bugs and a real dataset of 50 bugs. Collecting UI related fix-specific patches and writing test cases is a time-consuming process for UI bugs. To mitigate these shortcomings, we experimented with diverse types of UI bugs, divided into five different categories, which bolsters generalizability.

Construct Validity. Our primary construct validity stems from the construction of our dataset. A ground truth patch might require expert knowledge from the developers regarding the specific changes needed for a bug. In addition, writing test cases should be conducted by multiple expert developers across different features. To mitigate such limitations, each patch and test case was verified by multiple authors.

8 Conclusion

This paper aims to investigate whether we can apply existing APR techniques to Android UI bugs and the efficacy and limitations of such an application. With this goal, we conduct a comprehensive empirical study with five APRs on a new dataset called DroidFixBench, that includes 46 synthetic and 50 real bugs sourced from popular open-source Android applications. Our study identifies existing limitations of APR techniques and provides suggestions for future research directions.

Data availability

We release our replication package, including all of the data, source code, and generated patches by APRs at our online Zenodo archive and GitHub repository [41, 42].

Acknowledgments

This work was partially supported by NSF grants CCF-2441355, CCF-2423813, CCF-2132285, CCF-1955853, IIS-2541053, CCF-2239107, and a gift from Apple's Human-centered Machine Learning Team. Any opinions, findings, and conclusions expressed herein are the authors' and do not necessarily reflect those of the sponsors.

References

[1] 2025. AnglersLog - 347. https://github.com/cohenadair/anglers-log/issues/347.
[2] 2025. F-Droid apps. https://fdroid.tabler.dev.
[3] 2025. GPSLogger. https://f-droid.org/en/packages/com.mendhak.gpslogger/.
[4] 2025. Harmonic App Bug 114. https://github.com/SimonHalvdansson/Harmonic-HN/issues/114.
[5] 2025. Harmonic for Hacker News. https://play.google.com/store/apps/details?id=com.simon.harmonichackernews.
[6] 2025. PSLab. https://f-droid.org/packages/io.pslab/.
[7] 2025. SkyTube. https://github.com/SkyTubeTeam/SkyTube.
[8] 2025. URLCheck. https://f-droid.org/packages/com.trianguloy.urlchecker/.
[9] Anthropic. 2026. Claude Code – Anthropic's Agentic Coding System. https://www.anthropic.com/product/claude-code.
[10] Tien-Duy B. Le, David Lo, Claire Le Goues, and Lars Grunske. 2016. A Learning-to-Rank Based Fault Localization Approach Using Likely Invariants. In *Proceedings of the 25th International Symposium on Software Testing and Analysis* (Saarbrücken, Germany) *(ISSTA 2016)*. Association for Computing Machinery, New York, NY, USA, 177–188. doi:10.1145/2931037.2931049
[11] Islem Bouzenia, Premkumar Devanbu, and Michael Pradel. 2025. RepairAgent: An Autonomous, LLM-Based Agent for Program Repair. In *Proceedings of the IEEE/ACM 47th International Conference on Software Engineering* (Ottawa, Ontario, Canada) *(ICSE '25)*. IEEE Press, 2188–2200. doi:10.1109/ICSE55347.2025.00157
[12] Mark Chen et al. 2021. Evaluating Large Language Models Trained on Code. arXiv:2107.03374 [cs.LG] https://arxiv.org/abs/2107.03374
[13] Mengzhuo Chen, Zhe Liu, Chunyang Chen, Junjie Wang, Boyu Wu, Jun Hu, and Qing Wang. 2025. Standing on the Shoulders of Giants: Bug-Aware Automated GUI Testing via Retrieval Augmentation. *Proc. ACM Softw. Eng.* 2, FSE, Article FSE038 (June 2025), 22 pages. doi:10.1145/3715755
[14] Sen Chen, Chunyang Chen, Lingling Fan, Mingming Fan, Xian Zhan, and Yang Liu. 2022. Accessible or Not? An Empirical Investigation of Android App Accessibility. *IEEE Transactions on Software Engineering* 48, 10 (2022), 3954–3968. doi:10.1109/TSE.2021.3108162
[15] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. arXiv:2002.08155 [cs.CL] https://arxiv.org/abs/2002.08155
[16] Zebao Gao, Zhenyu Chen, Yunxiao Zou, and Atif M. Memon. 2016. SITAR: GUI Test Script Repair. *IEEE Transactions on Software Engineering* 42, 2 (2016), 170–186. doi:10.1109/TSE.2015.2454510
[17] Luca Gazzola, Daniela Micucci, and Leonardo Mariani. 2019. Automatic Software Repair: A Survey . *IEEE Transactions on Software Engineering* 45, 01 (Jan. 2019), 34–67. doi:10.1109/TSE.2017.2755013
[18] Google. 2026. Gemini CLI – Build, Debug and Deploy with AI. https://geminicli.com.
[19] Abram Hindle, Earl T. Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. 2012. On the naturalness of software. In *Proceedings of the 34th International Conference on Software Engineering* (Zurich, Switzerland) *(ICSE '12)*. IEEE Press, 837–847.
[20] Soneya Binta Hossain, Nan Jiang, Qiang Zhou, Xiaopeng Li, Wen-Hao Chiang, Yingjun Lyu, Hoan Nguyen, and Omer Tripp. 2024. A Deep Dive into Large Language Models for Automated Bug Localization and Repair. arXiv:2404.11595 [cs.SE] https://arxiv.org/abs/2404.11595
[21] Kai Huang, Zhengzi Xu, Su Yang, Hongyu Sun, Xuejun Li, Zheng Yan, and Yuqing Zhang. 2023. A Survey on Automated Program Repair Techniques. *ArXiv* abs/2303.18184 (2023). https://api.semanticscholar.org/CorpusID:257900782
[22] Kai Huang, Jian Zhang, Xiangxin Meng, and Yang Liu. 2024. Template-guided program repair in the era of large language models. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE) IEEE Computer Society*. 367–379.
[23] Binyuan Hui et al. 2024. Qwen2.5-Coder Technical Report. *ArXiv* abs/2409.12186 (2024). https://api.semanticscholar.org/CorpusID:272707390
[24] Albert Q. Jiang et al. 2024. Mixtral of Experts. arXiv:2401.04088 [cs.LG] https://arxiv.org/abs/2401.04088
[25] Jiajun Jiang, Yingfei Xiong, Hongyu Zhang, Qing Gao, and Xiangqun Chen. 2018. Shaping program repair space with existing patches and similar code. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Amsterdam, Netherlands) *(ISSTA 2018)*. Association for Computing Machinery, New York, NY, USA, 298–309. doi:10.1145/3213846.3213871
[26] Jack Johnson, Junayed Mahmud, Oscar Chaparro, Kevin Moran, and Mattia Fazzini. 2025. Generating Failure-based Oracles to Support Testing of Reported Bugs in Mobile Apps. In *Proceedings of the 40th International Conference on Automated Software Engineering (ASE'25)*. 262–273.
[27] Jack Johnson, Junayed Mahmud, Tyler Wendland, Kevin Moran, Julia Rubin, and Mattia Fazzini. 2022. An Empirical Investigation into the Reproduction of Bug Reports for Android Apps. *Proceedings of the IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER'22)* (2022).

[28] René Just, Darioush Jalali, and Michael D. Ernst. 2014. Defects4J: a database of existing faults to enable controlled testing studies for Java programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis* (San Jose, CA, USA) *(ISSTA 2014)*. Association for Computing Machinery, New York, NY, USA, 437–440. doi:10.1145/2610384.2628055

[29] Sophia D. Kolak, Ruben Martins, Claire Le Goues, and Vincent Josua Hellendoorn. 2022. Patch Generation with Language Models: Feasibility and Scaling Behavior. *Deep Learning for Code Workshop* (2022). https://par.nsf.gov/biblio/10340618

[30] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. 2012. GenProg: A Generic Method for Automatic Software Repair. *IEEE Transactions on Software Engineering* 38, 1 (2012), 54–72. doi:10.1109/TSE.2011.104

[31] Claire Le Goues, Michael Pradel, Abhik Roychoudhury, and Satish Chandra. 2021. Automatic Program Repair . *IEEE Software* 38, 04 (July 2021), 22–27. doi:10.1109/MS.2021.3072577

[32] Han Li, Letian Zhu, Bohan Zhang, Rili Feng, Jiaming Wang, Yue Pan, Earl T. Barr, Federica Sarro, Zhaoyang Chu, and He Ye. 2026. ContextBench: A Benchmark for Context Retrieval in Coding Agents. *ArXiv* abs/2602.05892 (2026). https://api.semanticscholar.org/CorpusID:285303471

[33] Raymond Li et al. 2023. StarCoder: may the source be with you! *Trans. Mach. Learn. Res.* 2023 (2023). https://api.semanticscholar.org/CorpusID:258588247

[34] Mario Linares-Vásquez, Gabriele Bavota, Michele Tufano, Kevin Moran, Massimiliano Di Penta, Christopher Vendome, Carlos Bernal-Cárdenas, and Denys Poshyvanyk. 2017. Enabling mutation testing for Android apps. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering* (Paderborn, Germany) *(ESEC/FSE 2017)*. Association for Computing Machinery, New York, NY, USA, 233–244. doi:10.1145/3106237.3106275

[35] Mario Linares-Vásquez, Cárlos Bernal-Cardenas, Kevin Moran, and Denys Poshyvanyk. 2017. How do Developers Test Android Applications?. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 613–622. doi:10.1109/ICSME.2017.47

[36] Kui Liu, Anil Koyuncu, Tegawendé F. Bissyandé, Dongsun Kim, Jacques Klein, and Yves Le Traon. 2019. You Cannot Fix What You Cannot Find! An Investigation of Fault Localization Bias in Benchmarking Automated Program Repair Systems. In *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*. 102–113. doi:10.1109/ICST.2019.00020

[37] Kui Liu, Shangwen Wang, Anil Koyuncu, Kisub Kim, Tegawendé F. Bissyandé, Dongsun Kim, Peng Wu, Jacques Klein, Xiaoguang Mao, and Yves Le Traon. 2020. On the efficiency of test suite based program repair: A Systematic Assessment of 16 Automated Repair Systems for Java Programs. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 615–627. doi:10.1145/3377811.3380338

[38] Zhe Liu. 2022. Woodpecker: identifying and fixing Android UI display issues. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 334–336. doi:10.1145/3510454.3522681

[39] Zhe Liu, Chunyang Chen, Junjie Wang, Yuekai Huang, Jun Hu, and Qing Wang. 2021. Owl eyes: spotting UI display issues via visual understanding. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering* (Virtual Event, Australia) *(ASE '20)*. Association for Computing Machinery, New York, NY, USA, 398–409. doi:10.1145/3324884.3416547

[40] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, et al. 2021. Codexglue: A machine learning benchmark dataset for code understanding and generation. *arXiv preprint arXiv:2102.04664* (2021).

[41] Junayed Mahmud et al. 2026. DroidFixBench GitHub Repository. https://github.com/SageSELab/UI-Program-Repair.

[42] Junayed Mahmud et al. 2026. DroidFixBench Zenodo Repository. https://doi.org/10.5281/zenodo.20338470.

[43] Junayed Mahmud, James Chen, Terry Achille, Camilo Alvarez-Velez, Darren Dean Bansil, Patrick Ijieh, Samar Karanch, Nadeeshan De Silva, Oscar Chaparro, Andrian Marcus, and Kevin Moran. 2025. LadyBug: A GitHub Bot for UI-Enhanced Bug Localization in Mobile Apps. In *Proceedings of the 41st International Conference on Software Maintenance and Evolution* (Auckland, New Zealand) *(ICSME 2025)*.

[44] Junayed Mahmud, Nadeeshan De Silva, Safwat Ali Khan, Seyed Hooman Mostafavi, S M Hasan Mansur, Oscar Chaparro, Andrian (Andi) Marcus, and Kevin Moran. 2024. On Using GUI Interaction Data to Improve Text Retrieval-based Bug Localization. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering* (Lisbon, Portugal) *(ICSE 2024)*.

[45] Forough Mehralian, Navid Salehnamadi, and Sam Malek. 2021. Data-driven accessibility repair revisited: on the effectiveness of generating labels for icons in Android apps. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Athens, Greece) *(ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 107–118. doi:10.1145/3468264.3468604

[46] Matthew B. Miles, A. Michael Huberman, and Johnny Saldaña. 2013. *Qualitative data analysis: a methods sourcebook*. Thousand Oaks, Californinia: SAGE Publications, Inc., [2014] ©2014.

[47] Kevin Moran, Michele Tufano, Carlos Bernal-Cárdenas, Mario Linares-Vásquez, Gabriele Bavota, Christopher Vendome, Massimiliano Di Penta, and Denys Poshyvanyk. 2018. MDroid+: a mutation testing framework for android. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings* (Gothenburg, Sweden) *(ICSE '18)*. Association for Computing Machinery, New York, NY, USA, 33–36. doi:10.1145/3183440.3183492

[48] Manish Motwani and Yuriy Brun. 2023. Better Automatic Program Repair by Using Bug Reports and Tests Together. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. 1225–1237. doi:10.1109/ICSE48619.2023.00109

[49] Brad Myers. 1994. Challenges of HCI Design and Implementation. *Interactions* 1, 1 (Jan. 1994), 73–83. doi:10.1145/174800.174808

[50] Brad A. Myers and Mary Beth Rosson. 1992. Survey on user interface programming. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (Monterey, California, USA) *(CHI '92)*. Association for Computing Machinery, New York, NY, USA, 195–202. doi:10.1145/142750.142789

[51] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. 2013. SemFix: Program repair via semantic analysis. In *2013 35th International Conference on Software Engineering (ICSE)*. 772–781. doi:10.1109/ICSE.2013.6606623

[52] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Haiquan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022. CodeGen: An Open Large Language Model for Code with Multi-Turn Program Synthesis. In *International Conference on Learning Representations*. https://api.semanticscholar.org/CorpusID:252668917

[53] Frolin Ocariza, Kartik Bajaj, Karthik Pattabiraman, and Ali Mesbah. 2013. An Empirical Study of Client-Side JavaScript Bugs. In *2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement*. 55–64. doi:10.1109/ESEM.2013.18

[54] OpenAI. 2025. ChatGPT. https://chat.openai.com.

[55] OpenAI. 2026. Codex – A Coding Agent that Helps You Build and Ship with AI. https://openai.com/codex/.

[56] Jeff H. Perkins, Sunghun Kim, Sam Larsen, Saman Amarasinghe, Jonathan Bachrach, Michael Carbin, Carlos Pacheco, Frank Sherwood, Stelios Sidiroglou, Greg Sullivan, Weng-Fai Wong, Yoav Zibin, Michael D. Ernst, and Martin Rinard. 2009. Automatically patching errors in deployed software. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles* (Big Sky, Montana, USA) *(SOSP '09)*. Association for Computing Machinery, New York, NY, USA, 87–102. doi:10.1145/1629575.1629585

[57] Julian Aron Prenner, Hlib Babii, and Romain Robbes. 2022. Can OpenAI's codex fix bugs? an evaluation on QuixBugs. In *Proceedings of the Third International Workshop on Automated Program Repair* (Pittsburgh, Pennsylvania) *(APR '22)*. Association for Computing Machinery, New York, NY, USA, 69–75. doi:10.1145/3524459.3527351

[58] Mary Beth Rosson, Susanne Maass, and Wendy A. Kellogg. 1986. Designing for designers: an analysis of design practice in the real world. In *Proceedings of the SIGCHI/GI Conference on Human Factors in Computing Systems and Graphics Interface* (Toronto, Ontario, Canada) *(CHI '87)*. Association for Computing Machinery, New York, NY, USA, 137–142. doi:10.1145/29933.30873

[59] Baptiste Rozière et al. 2024. Code Llama: Open Foundation Models for Code. arXiv:2308.12950 [cs.CL] https://arxiv.org/abs/2308.12950

[60] Antu Saha, Yang Song, Junayed Mahmud, Ying Zhou, Kevin Moran, and Oscar Chaparro. 2024. Toward the Automated Localization of Buggy Mobile App UIs from Bug Descriptions. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 1249–1261.

[61] Navid Salehnamadi, Forough Mehralian, and Sam Malek. 2023. Groundhog: An Automated Accessibility Crawler for Mobile Apps. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering* (Rochester, MI, USA) *(ASE '22)*. Association for Computing Machinery, New York, NY, USA, Article 50, 12 pages. doi:10.1145/3551349.3556905

[62] David I. Samudio and Thomas D. LaToza. 2022. Barriers in Front-End Web Development. In *2022 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 1–11. doi:10.1109/VL/HCC53370.2022.9833127

[63] Edward K. Smith, Earl T. Barr, Claire Le Goues, and Yuriy Brun. 2015. Is the cure worse than the disease? overfitting in automated program repair. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering* (Bergamo, Italy) *(ESEC/FSE 2015)*. Association for Computing Machinery, New York, NY, USA, 532–543. doi:10.1145/2786805.2786825

[64] Yang Song, Junayed Mahmud, Nadeeshan De Silva, Ying Zhou, Oscar Chaparro, Kevin Moran, Andrian Marcus, and Denys Poshyvanyk. 2023. BURT: A Chatbot for Interactive Bug Reporting. In *Proceedings of the 45th IEEE/ACM International Conference on Software Engineering (ICSE'23)*. 170–174.

[65] Yang Song, Junayed Mahmud, Ying Zhou, Oscar Chaparro, Kevin Moran, Andrian Marcus, and Denys Poshyvanyk. 2022. Toward interactive bug reporting for (android app) end-users. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 344–356.

[66] Ting Su, Jue Wang, and Zhendong Su. 2021. Benchmarking automated GUI testing for Android against real-world bugs. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Athens, Greece) *(ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 119–130. doi:10.1145/3468264.3468620

[67] Shin Hwei Tan, Zhen Dong, Xiang Gao, and Abhik Roychoudhury. 2018. Repairing crashes in Android apps. In *Proceedings of the 40th International Conference on Software Engineering* (Gothenburg, Sweden) *(ICSE '18)*. Association for Computing Machinery, New York, NY, USA, 187–198. doi:10.1145/3180155.3180243

[68] Michele Tufano, Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Andrea De Lucia, and Denys Poshyvanyk. 2017. There and back again: Can you compile that snapshot? *Journal of Software: Evolution and Process* 29, 4 (2017), e1838. arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/smr.1838 doi:10.1002/smr.1838 e1838 smr.1838.

[69] Dingbang Wang, Yu Zhao, Sidong Feng, Zhaoxu Zhang, William G. J. Halfond, Chunyang Chen, Xiaoxia Sun, Jiangfan Shi, and Tingting Yu. 2024. Feedback-Driven Automated Whole Bug Report Reproduction for Android Apps. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis* (Vienna, Austria) *(ISSTA 2024)*. Association for Computing Machinery, New York, NY, USA, 1048–1060. doi:10.1145/3650212.3680341

[70] Xingyao Wang et al. 2025. OpenHands: An Open Platform for AI Software Developers as Generalist Agents. In *The Thirteenth International Conference on Learning Representations (ICLR'25)*. https://openreview.net/forum?id=OJd3ayDDoF

[71] Yuxiang Wei, Chunqiu Steven Xia, and Lingming Zhang. 2023. Copiloting the Copilots: Fusing Large Language Models with Completion Engines for Automated Program Repair. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (San Francisco, CA, USA) *(ESEC/FSE 2023)*. Association for Computing Machinery, New York, NY, USA, 172–184. doi:10.1145/3611643.3616271

[72] Tyler Wendland, Jingyang Sun, Junayed Mahmud, Syeda Mansur, Steven Huang, Kevin Moran, Julia Rubin, and Mattia Fazzini. 2021. Andror2: A Dataset of Manually-Reproduced Bug Reports for Android apps. *Proceedings of the 18th IEEE/ACM International Conference on Mining Software Repositories (MSR'21)* (2021), 600–604.

[73] Chun Xia and Lingming Zhang. 2024. Keep the Conversation Going: Fixing 162 out of 337 bugs for $0.42 each using ChatGPT. In *Proceedings of the 2024 International Symposium on Software Testing and Analysis (ISSTA'24)*. Association for Computing Machinery, Vienna, Austria, USA, 12 pages.

[74] Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. 2023. Automated Program Repair in the Era of Large Pre-Trained Language Models. In *Proceedings of the 45th International Conference on Software Engineering* (Melbourne, Victoria, Australia) *(ICSE '23)*. IEEE Press, 1482–1494. doi:10.1109/ICSE48619.2023.00129

[75] Chunqiu Steven Xia and Lingming Zhang. 2022. Less training, more repairing please: revisiting automated program repair via zero-shot learning. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Singapore, Singapore) *(ESEC/FSE 2022)*. Association for Computing Machinery, New York, NY, USA, 959–971. doi:10.1145/3540250.3549101

[76] Yiheng Xiong, Mengqian Xu, Ting Su, Jingling Sun, Jue Wang, He Wen, Geguang Pu, Jifeng He, and Zhendong Su. 2023. An Empirical Study of Functional Bugs in Android Apps. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis* (Seattle, WA, USA) *(ISSTA 2023)*. Association for Computing Machinery, New York, NY, USA, 1319–1331. doi:10.1145/3597926.3598138

[77] Junjielong Xu, Ying Fu, Shin Hwei Tan, and Pinjia He. 2024. Aligning the Objective of LLM-Based Program Repair. *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)* (2024), 2548–2560. https://api.semanticscholar.org/CorpusID:269148552

[78] Jifeng Xuan, Matias Martinez, Favio DeMarco, Maxime Clement, Sebastian Lamelas Marcote, Thomas Durieux, Daniel Le Berre, and Martin Monperrus. 2017. Nopol: Automatic Repair of Conditional Statement Bugs in Java Programs. *IEEE Trans. Softw. Eng.* 43, 1 (jan 2017), 34–55. doi:10.1109/TSE.2016.2560811

[79] John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. 2024. SWE-agent: agent-computer interfaces enable automated software engineering. In *Proceedings of the 38th International Conference on Neural Information Processing Systems* (Vancouver, BC, Canada) *(NIPS '24)*. Curran Associates Inc., Red Hook, NY, USA, Article 1601, 125 pages.

[80] John Yang, Carlos E Jimenez, Alex L Zhang, Kilian Lieret, Joyce Yang, Xindi Wu, Ori Press, Niklas Muennighoff, Gabriel Synnaeve, Karthik R Narasimhan, Diyi Yang, Sida Wang, and Ofir Press. 2025. SWE-bench Multimodal: Do AI Systems Generalize to Visual Software Domains?. In *The Thirteenth International Conference on Learning Representations*. https://openreview.net/forum?id=riTiq3i21b

[81] He Ye, Matias Martinez, and Martin Monperrus. 2022. Neural program repair with execution-based backpropagation. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE'22)*. Association for Computing Machinery, New York, NY, USA, 1506–1518. doi:10.1145/3510003.3510222

[82] He Ye, Aidan Z.H. Yang, Chang Hu, Yanlin Wang, Tao Zhang, and Claire Le Goues. 2025. AdverIntent-Agent: Adversarial Reasoning for Repair Based on Inferred Program Intent. *Proc. ACM Softw. Eng.* 2, ISSTA, Article ISSTA062 (June 2025),

23 pages. doi:10.1145/3728939

[83] Yuxin Zhang, Sen Chen, Lingling Fan, Chunyang Chen, and Xiaohong Li. 2023. Automated and Context-Aware Repair of Color-Related Accessibility Issues for Android Apps. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (San Francisco, CA, USA) *(ESEC/FSE 2023)*. Association for Computing Machinery, New York, NY, USA, 1255–1267. doi:10.1145/3611643.3616329

[84] Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. 2024. AutoCodeRover: Autonomous Program Improvement. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis* (Vienna, Austria) *(ISSTA 2024)*. Association for Computing Machinery, New York, NY, USA, 1592–1604. doi:10.1145/3650212.3680384

[85] Zhaoxu Zhang, Fazle Mohammed Tawsif, Komei Ryu, Tingting Yu, and William G. J. Halfond. 2024. Mobile Bug Report Reproduction via Global Search on the App UI Model. *Proc. ACM Softw. Eng.* 1, FSE, Article 117 (July 2024), 21 pages.

Received 2026-01-30; accepted 2026-06-25