

Automated Code Fix Suggestions for Accessibility Issues in Mobile Apps

Forough Mehralian
Apple
Seattle, USA
mehralian@apple.com

Titus Barik
Apple
Seattle, USA
tbarik@apple.com

Jeff Nichols
Apple
Seattle, USA
jwnichols@apple.com

Amanda Swearngin
Apple
Seattle, USA
aswearngin@apple.com

Abstract—Accessibility is crucial for inclusive app usability, yet developers often struggle to identify and fix app accessibility issues due to a lack of awareness, expertise, and inadequate tools. Current accessibility testing tools can identify accessibility issues but may not always provide guidance on how to address them. We introduce FixAlly, an automated tool designed to suggest source code fixes for accessibility issues detected by automated accessibility scanners. FixAlly employs a multi-agent LLM architecture to generate fix strategies, localize issues within the source code, and propose code modification suggestions to fix the accessibility issue. Our empirical study demonstrates FixAlly’s capability in suggesting fixes that resolve issues found by accessibility scanners—with an effectiveness of 77% in generating plausible fix suggestions—and our survey of 12 iOS developers finds they would be willing to accept 69.4% of evaluated fix suggestions.

Index Terms—accessibility, automated, repair, mobile, llm

I. INTRODUCTION

The increasing reliance on mobile apps for everyday tasks underscores the necessity of ensuring accessibility for all. Despite the existence of guidelines aimed at assisting developers in creating more accessible apps [1]–[3], research shows that many apps are still released with numerous accessibility issues [4]–[7]. Developers often struggle with building accessible apps because they lack awareness of accessibility requirements [4] or have limited knowledge and expertise in effectively addressing accessibility issues [8].

Existing accessibility scanning tools—such as Accessibility Scanner [9] for Android and Accessibility Inspector [10] for iOS—helpfully verify compliance of each app screen with rules derived from accessibility guidelines. In addition to these rule-based techniques, some automated tools dynamically examine apps using assistive technologies to detect issues that manifest during real-time interactions [11], [12]. However, current tools provide insufficient support for maintaining app accessibility [8] because fixing the large number of issues reported by these tools remains a significant challenge. While single-issue fix techniques address problems like color issues [13], missing labels [14], text scaling problems [15], and touch target size [16], these single-issue fix techniques have notable limitations.

According to documentation, Accessibility Inspector reports 7 categories of issues [10], and these single-purpose approaches can fix only a small subset.

To bridge this gap in tooling between single-purpose fix approaches and source code, we investigate an automated *plan-localize-fix* technique—implemented as a tool called FIXALLY—to fix various types of accessibility issues reported by scanners such as the Accessibility Inspector [10]. To understand the challenges of fixing issues detected by accessibility scanners in source code, we first conducted formative interviews with five developers. Our developers indicated that: 1) multiple strategies can address a single issue, 2) appropriate fixes must consider not only accessibility guidelines, but also the integrity of the app’s design and functionality, 3) implementing a fix frequently requires modifications beyond the problematic element, and 4) identifying these relevant locations in the code to apply fixes is the most time-consuming step.

To address these needs, FIXALLY employs a multi-agent LLM architecture capable of proposing *plausible* fix suggestions for issues reported by an accessibility scanner. In this context, a plausible fix is defined as a modification that passes the accessibility checks of the automated scanner for the target issue without introducing new ones or removing functionality. FIXALLY analyzes an open-source mobile app to detect various accessibility issues. FIXALLY localizes issues within the source code and proposes fix suggestions to resolve the issue using a suggestion generation engine. Each proposed fix suggestion aims to resolve the targeted accessibility issue without introducing new ones or compromising app functionalities. FIXALLY also assists the developer in the decision-making process to select the strategy that best aligns with the app’s design and requirements.

The contributions of this paper are:

- A novel plan-localize-fix technique—operationalized as an automated tool using a multi-agent LLM architecture—that generates code suggestions to fix accessibility issues in mobile apps.
- An empirical evaluation on 205 issues from 14 iOS apps built using SwiftUI, a declarative framework that allows developers to define the desired UI attributes and behavior [17]. Our evaluation demon-

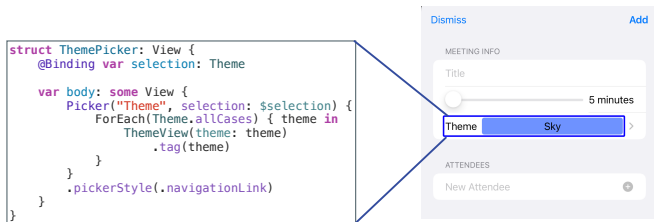


Fig. 1. Implementation of a dropdown list in SwiftUI.

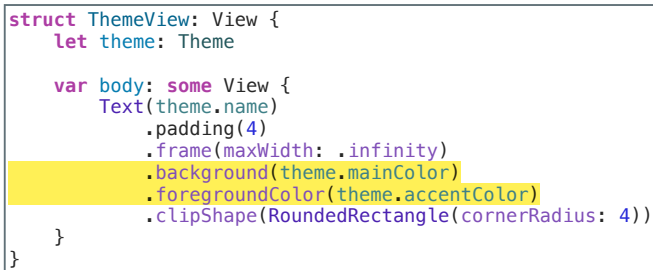


Fig. 2. Localization of the color contrast issue in the source code.

strates an effectiveness of 77% of FIXALLY in proposing plausible fix suggestions for accessibility issues.

- A survey of 12 iOS app developers, finding the tool was most helpful for less experienced developers in allowing them to explore multiple solutions when resolving accessibility bugs. Even experienced developers found it helpful that FIXALLY localized the issue in the code.

II. BACKGROUND: COMPARING ANDROID AND IOS

Many of the existing approaches focus on Android applications which require different techniques than iOS applications. Declarative programming languages, such as SwiftUI, represent a contemporary paradigm for building GUIs in mobile apps by enabling developers to define the desired UI and its behavior using concise syntax. Figure 1 illustrates how a dropdown list is implemented in SwiftUI. This approach contrasts with traditional imperative methods, where developers must meticulously specify attributes of each UI element and manage its state. For example, in Android, the UI specification is defined using an XML file, with Java classes binding behavior to each element. Developers can also modify UI attributes dynamically through specified behavior in declarative languages. Therefore, properly locating GUI problems in the project cannot be effectively achieved through analysis or modification of the static GUI specification alone. The dynamic specification of UI attributes, which is the core of declarative programming languages, introduces additional challenges in identifying GUI problems, such as accessibility issues, within the source code.

Consider the color contrast failure issue for the annotated dropdown list in Figure 1, as reported by Accessibility Inspector. Developers typically employ various

strategies to address such issues, such as adjusting background colors, modifying text colors, or increasing font sizes. However, these strategies cannot be applied in the provided code snippet that demonstrates the element’s implementation. Instead, the appropriate place to adjust the color for this element is within the ThemeView called from the ThemePicker (Figure 2). Locating this correct position necessitates navigating through the UI hierarchy, understanding the semantics of the GUI, and comprehending the structure of the source code required for implementing these elements. This complexity has made localization a challenging task.

III. FORMATIVE STUDY

To elicit the process developers follow to fix accessibility bugs found by an accessibility scanner in iOS apps, we conducted formative interviews with five iOS app developers in our company. The developers had at least 1 year of experience in developing SwiftUI apps and median of intermediate accessibility familiarity (1 – No Experience to 5 – Expert). During the study, the developers used the Accessibility Inspector and Xcode to detect issues in the Landmarks app¹ running on an iOS simulator. We asked them to think-aloud while they detected and fixed as many issues as they could within 1 hour. At the end, we asked them follow-up questions about their experiences and their ideas on any tools that could improve their process in finding and fixing the accessibility issues. All sessions took place virtually over Webex. Some developers built and tested the app locally while sharing their screen and some developers remotely controlled the screen of the lead researcher who also had the app and simulator built and running in Xcode. A second researcher observed and took notes for four out of five sessions. We recorded audio, video, and notes for each session.

To analyze the data, we annotated the transcripts and built an affinity diagram [18] where one paper author led the annotation and initial grouping, and another author read and also validated the themes.

1) *Accessibility Bug Fixing Phases*: We examined the developers’ overall process in fixing the accessibility bugs with two goals: first, understanding the varied activities (e.g., localizing an issue, fixing it in the code) within the whole process, and second, understanding inefficiencies within each of those activities where an automated tool could help. We found that developers accessibility bug fixing workflow can be grouped into the following phases: *hypothesis formation & fix planning*, *localization*, and *code editing and validation*. We ultimately designed the architecture of FIXALLY around these three phases.

Hypothesis formation and fix planning: All developers in our study had at least intermediate accessibility knowledge and could understand the issues reported by Accessibility Inspector. They were all able to propose hypotheses to

¹<https://github.com/pd95/SwiftUI-Landmarks>

diagnose one or more issues, and come up with a high level plan to fix one or more of them. In some cases, developers directly proposed a fix plan for some issues because they were already highly familiar with the issue and could quickly come up with the fix. In other cases, developers mentioned multiple hypotheses that the issue could be related to, and validated which to test after localizing the impacted UI element in the source code.

Localization: The output of Accessibility Inspector, and typically other accessibility scanners, is a screenshot highlighting the impacted UI element and the ability to highlight the element with the issue on a live device. While these tools provide some metadata and inspectors to examine the running app’s hierarchy, they provide little help with localizing UI elements in code. In our study, developers used a variety of methods to localize UI elements in the code including searching for specific text strings found in the interface. Others tried to match the visual hierarchy in the interface with the hierarchy of views in the source files by looking at them one by one. Sometimes they also looked for specific SwiftUI modifiers in the code based on their hypothesis, or examined the view hierarchy in the accessibility inspector for class names or metadata they could search for. Localization was where they predominantly spent the most time during the study.

Code editing and validation: After fix planning and localization, developers applied their fix to the code. They then re-ran the Accessibility Inspector audit on the same screen again to confirm the issue was resolved. If the issue was not resolved, they could repeat the overall process as long as time allowed.

Developers did not necessarily complete the bug fixing phases in the same order. In all cases, developers either localized or came up with a hypothesis or fix plan first. Some developers first attempted to localize a UI element in the source code before coming up with a hypothesis or fix plan, while others first described a hypothesis or fix plan before attempting to localize the UI element. Some developers repeated this process multiple times before confirming the fix by validating it no longer appeared in the Accessibility Inspector.

2) *Design Goals:* At the closing of the session, we asked the developers to describe their overall process for fixing the accessibility issues, and to provide feedback on any tools that could expedite their process. We then formulated list of design goals that we incorporated into our system. The design goals were motivated by the challenges developers faced throughout the accessibility bug fixing process and suggestions they provided to address these challenges.

Design Goal 1 – Localize impacted UI elements in source, and automatically apply fix suggestions: Developers struggled the most during our interviews with finding impacted UI elements in source code, which is often the starting point for making a fix. This added tedious, and unnecessary friction to their bug fixing process. Developers utilized

different heuristics to localize the issue, including searching for textual elements, mapping their mental model of elements in the screenshots to the UI structure in the code, looking for specific accessibility attributes they were planning to modify, or combinations of these approaches. Thus a key goal of FIXALLY is to automatically localize potential fix locations in code for detected issues to remove this inefficiency and also generate fix suggestions in the form of patches so they can be applied automatically.

Design Goal 2 – Provide multiple fix suggestions for an issue: Some developers struggled to come up with hypothesis for fixes, and even if they had a plausible hypothesis, some did not know how to make the corresponding code changes to test it. Furthermore, some issues often can be fixed in multiple ways. For example, contrast issues can be fixed by increasing text font size, or changing the color of UI elements or background. Developers noted other considerations and potential side effects in choosing a correct fix including consulting with design teams, support for other languages, and impact on the layout of other areas of the screen or the application. Some developers also struggled to understand some issues reported by Accessibility Inspector, and requested better suggestions more contextualized and specific to their code. To help developers in understanding each suggested fix, FIXALLY also includes information about the model’s fix plan and reasoning along with each generated fix suggestion.

IV. APPROACH

Figure 3 shows an overview of FIXALLY, with its three main modules: Data Processing, Suggestion Generation Engine, and Suggestion Assessment.

FIXALLY takes as input an app with GUI tests, each designed to navigate to different screens of the app. The XCTest framework [19], integrated into Xcode IDE, allows iOS developers to create automated test scenarios that navigate the app to the screens targeted for accessibility assessment. Additionally, existing automated app crawlers can help developers generate these test scenarios automatically. These crawlers either perform random actions on the screen [20] or analyze UI elements to systematically explore all possible actions on those elements [21].

Automated testing frameworks recommend that developers use unique identifiers for each element, serving as a bridge between testing frameworks and apps. To ensure every UI element is associated with an identifier, the Data Processing module statically analyzes the source code, restores the UI hierarchy, and instruments the app to insert a unique accessibility identifier for each UI element [22] in the input SwiftUI app. To find accessibility issues, the Data Processing module runs GUI tests to navigate to varied screens in the app. It then uses Accessibility Inspector [10] to obtain a report of accessibility issues on an app screen navigated to by a GUI test. The output of the scanner includes a description of each issue, the identifier of the impacted UI element or its parent elements

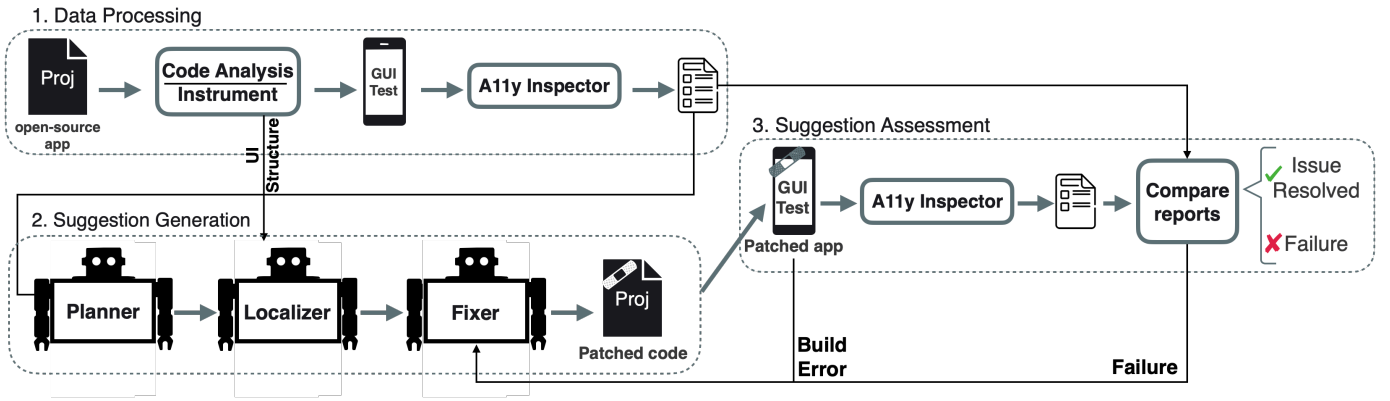


Fig. 3. FIXALLY’s approach, consisting of 1) *Data Processing* which instruments the application and navigates to various screens via GUI tests to capture accessibility scans and screenshots, 2) *Suggestion Generation* which uses a multi-agent LLM architecture to generate fix suggestions for each detected issue from the input screenshot, source code, and issue descriptions, and 3) *Suggestion Assessment* which captures a new accessibility scan of the patched app and GUI screen and compares it to the prior report to determine if the fix suggestion resolved the issue.

in the UI hierarchy, and a screenshot of the app with the location of the problematic element. The Suggestion Generation Engine processes the information for each issue to generate multiple fix suggestions for addressing the issue, and patched project to verify whether suggestions resolve the issue (Details in Section V). Inspired by *Design Goal 1*, the Suggestion Generation Engine is capable of localizing the source of issues among the code for an app across multiple files, and generating patches that developers can automatically apply to fix the issue.

To verify that generated fix suggestions can plausibly fix each issue, FIXALLY evaluates each code modification in its Suggestion Assessment module. This module takes the modified code snippet to generate a patched version of the project, attempts to build the app, and runs the same GUI test to navigate to the target screen where the issue was detected. It then uses the Accessibility Inspector to audit the screen and compare the report with the initial report.

Suggestion Generation Engine fails when the generated fix has build errors, was not able to resolve the accessibility issue, or introduced new issues. The model may also inadvertently comment out sections of code or remove necessary screen elements. FIXALLY feeds these failure messages back to the Suggestion Generation Engine to self-reflect and let it revise the modified code snippet. This iterative process mirrors a developer’s approach of assessing and revising modifications in response to issues. In our work, we configure the number of iterations for this feedback loop, setting it to 3 to allow sufficient opportunities for the model to refine its fix suggestions. Finally, if the fix suggestion successfully resolves the issue without introducing other issues or removing functionalities, the Suggestion Assessment module marks it as a fix suggestion that can be shown to developers. FIXALLY provides developers with multiple fix suggestions for an issue inspired by *Design Goal 2*.

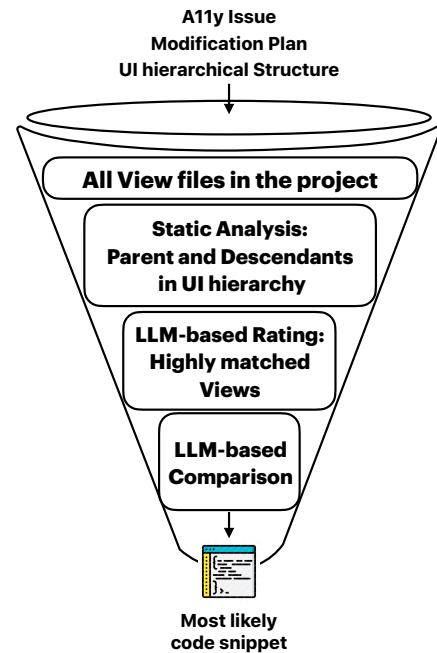


Fig. 4. Multi-level, hybrid localization architecture: Static analysis of the UI hierarchy identifies parent and descendant views. LLM-based rating evaluates the match of each individual code snippet to the screenshot. Finally, LLM-based comparison examines the highly matched views to determine the most likely code snippet for applying the fix.

V. SUGGESTION GENERATION ENGINE

The Suggestion Generation Engine (Figure 3.2) suggests fix strategies for each reported accessibility issue. Figure 1 shows the Suggestion Generation Engine, illustrating three agents in this module: Planner, Localizer, Fixer. These agents are responsible for performing specific steps that developers take in fixing an accessibility bug, as we observed in our Formative Study (Section III). The details of each agent are below. For each agent,

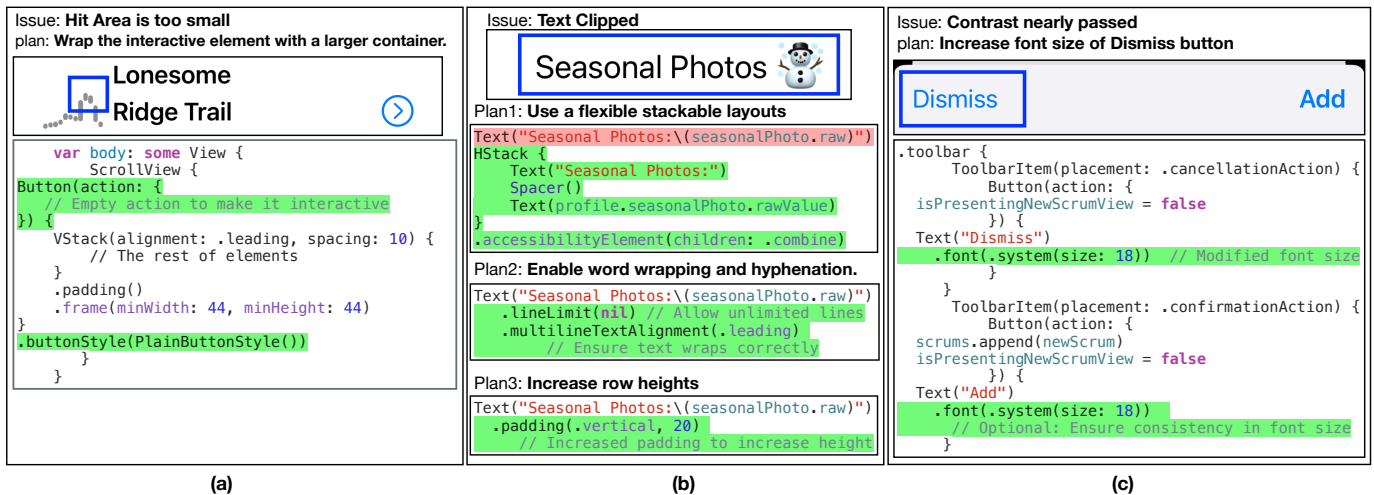


Fig. 5. Three different types of issues that were fixed by FIXALLY. (a) The tool addresses the issue of small hit target size by identifying a group of semantically related elements and merging them into an interactive container. (b) Shows three generated fix suggestions for Text Clipped issue. The first plan involves replacing a single element with a group of elements while preserving the functionality (c) FIXALLY addresses the contrast issue of the “Dismiss” button, while also maintaining design integrity by applying the font change to another similar button.

we currently use GPT-4o [23] for the LLM model. The specifics of the prompt, along with the source code, are available in the supplementary material submitted with the paper.

A. Planner: Suggesting fix plans

Each accessibility issue may be resolved in various ways. According to our Formative Study III, developers prefer solutions that not only fix the issue without introducing new issues, but also align with the design decisions of the app and work well in different modes, such as dark mode or horizontal mode. Providing different options for fixing an issue allows the developer to choose the one that best fits the overall design and functionality. The Planner agent facilitates this by generating natural language suggestions of strategies to fix each issue. For example, agent may suggest a plan of “adjust background color for better contrast” to fix a “contrast failed” issue for a UI element.

FIXALLY provides the Planner agent with an annotated screenshot of the app along with the issue description from the Accessibility Inspector. Its task is to identify the most relevant accessibility guideline related to the reported issue and list techniques to resolve it in natural language. FIXALLY aims to leverage the language model’s knowledge of accessibility fixes across various platforms, such as the web, to provide suggestions that can be adapted for mobile apps on specific platforms like iOS. FIXALLY instructs the agent to avoid suggesting solutions that are not applicable to the source code, filtering out recommendations such as using third-party tools to test the app. In our evaluation, FIXALLY instructs the Planner to return three alternative plans for each issue, though this number can be configured.

B. Localizer: Finding the relevant code snippet

Localizing an issue involves identifying the precise location in the source code where the fix plan should be applied. Source code projects often span numerous files with thousands of lines, potentially exceeding the token limits that language models can effectively query in one request or retain in their context window. To manage this complexity, we employ a multi-level localization approach using static code analysis and LLM code analysis. Figure 4 illustrates the different levels of localization, including one static code analysis step and two LLM-based steps for issue-to-code mapping.

First, the Localizer identifies all of the View files of the project, identifiable by the ‘import SwiftUI’ statement and structures extending the View class. Then, it extracts candidate code snippets from these view files through static code analysis. The key insight that the Localizer uses to optimize this process is that the behavior and design of UI elements are predominantly influenced by their descendants or ascendants. The Static Analysis module takes the accessibility identifier of the problematic element and traverses the pre-analyzed UI structure to return all descendant views and the parent of that element to form a set of candidate code snippets. FIXALLY adds these accessibility identifiers to each view in the code in the Data Processing phase, where it also captures the UI hierarchy (Recall Section IV).

Next, the Localizer generates an LLM-based rating for the code snippets filtered by the static analysis module. It matches each snippet to the modification plan for the issue and the screenshot highlighting the problem. Due to the limited context window of LLMs, it may not be feasible to consider and compare all candidate snippets simulta-

neously to find the correct one to apply the fix. Instead, the Localizer agent first assesses each snippet individually based on detailed issue information, the screenshot, and the proposed fix plan to determine its suitability. FIXALLY instructs the agent to map the problematic element and other elements in its vicinity to the source code, consider the fix plan, and rate the likelihood that the snippet is the correct location. This approach mirrors various techniques developers use to localize accessibility issues in the source code, as mentioned in Section III. The language model’s code comprehension capabilities, combined with its knowledge of accessibility, enable it to rate the alignment of the code with the highlighted element in the screenshot and the fix strategy. FIXALLY selects the code snippets with the highest match rates for further comparison by the Localizer agent. The Localizer agent ranks the highly rated snippets and selects the one most likely to be the correct location for applying the fix.

C. Fixer: Modifying the code

The goal of the Fixer agent is to apply the fix strategy to the selected code snippet. The Fixer agent receives the issue details, the fix strategy for the issue, and the most relevant code snippet to be modified based on the plan. Instead of relying on the agent to generate a diff, we instruct the agent to directly apply the modifications and return the updated code. This approach avoids potential inaccuracies in diff generation by language models, which may struggle with accurately calculating changes across lines of code and managing the required number of tokens. Finally, the Suggestion Generation Engine module applies the diff and creates a new copy of the project.

D. Output

It also stores a diff of the updated code and the original buggy code to provide a visualization of the code suggestion for developers to examine. Figure 5 shows examples of fix suggestions generated by FIXALLY, demonstrating its ability to address different types of issues and apply different strategies to fix an issue. The output of FIXALLY is currently a diff visualization of the code for each fix suggestion along with the fix plan and the model’s explanation of the changes.

As shown in Figure 5(a), FIXALLY correctly identifies a group of semantically related elements and merges them into an interactive container to address the issue of small hit target size. In contrast, Figure 5(b) demonstrates the capability of FIXALLY in splitting a single element into a group of elements to address the Text Clipped issue. Figure 5(c) shows that when fixing the contrast issue of the “Dismiss” button, the tool also maintains the app’s design integrity by applying the proposed fix to a similar “Add” button in the toolbar, demonstrating its ability to fix accessibility issues while maintaining consistency between similar UI elements.

VI. EVALUATION

We evaluated FIXALLY through the following research questions:

- RQ1.** (Effectiveness) How effective is FIXALLY in generating code fixes for accessibility issues detected by an accessibility scanner?
- RQ2.** (Efficiency) What is the efficiency of FIXALLY in terms of time, the number of attempts, and the cost?
- RQ3.** (Helpfulness) How helpful are the proposed fixes for developers?

A. Experimental Setup

We evaluated our approach using 14 open-source apps sourced from GitHub. Specifically, we randomly selected apps from two GitHub repositories that catalog open-source iOS apps [24], [25], excluding apps not built using SwiftUI. For each app, one of the authors attempted to build the app successfully within a 30 minute window. We also excluded apps with build errors due to dependencies, external packages, or very old iOS versions from the dataset. Table I provides a list of the apps included in our study. The list of apps with their corresponding GitHub links is also available in our supplementary materials. FIXALLY’s implementation leverages LLM agents based on GPT-4o, which features a 128K context window and has a knowledge cut-off date of October 2023. We conducted the experiments on a MacBook M1 Pro equipped with 32GB of RAM, a typical computer setup for development. We used Xcode 15.0, the latest available version, to build the apps, and we installed and tested them on an iPhone 12.

B. RQ1. FIXALLY’s effectiveness

We assessed the efficacy of FIXALLY by evaluating its ability to propose fix suggestions for 204 issues across 22 screens of SwiftUI iOS apps. Table I presents the outcomes of FIXALLY in generating fix suggestions. We use the term *plausible* to indicate that the generated fix resolved the targeted issue while maintaining app functionality and without introducing new issues.

FIXALLY demonstrated a 77% effectiveness in automatically generating fix suggestions for accessibility issues, where effectiveness means it successfully produced at least one plausible suggestion out of three suggestions for 157 out of 204 issues. Furthermore, for 129 (63%) of these issues, FIXALLY generated two or three plausible fix suggestions, providing developers multiple options to consider.

We also assessed the categories of issues that FIXALLY can generate fix suggestions for. Our dataset contains nine different types of issues as shown in Table II. According to the documentation for Accessibility Inspector [26], these issues encompass the categories of Element description, Element detection, Hit region, Contrast, Clipped text, Traits, and Dynamic type. However, our dataset does not contain trait issues: we found that even when modifying

TABLE I
FIXALLY’S EFFECTIVENESS IN GENERATING FIXES
FOR ACCESSIBILITY ISSUES

App ¹	Screens	<i>n</i>	Plausible Fix (PF)
ARPlasticOcean	1	5	4
Calculator	1	24	22
DeTeXt	1	1	1
DesignRemakes	1	1	0
ExpenseTracker	1	5	4
Fingerspelling	1	4	3
Instagram	2	5	3
Landmarks	3	55	48
Ratio	1	15	10
Scrumdinger	2	7	6
Go Cycling	4	74	52
DesignCode	1	8	7
GradeCalc	2	7	3
Sunshine	1	1	1
Total	22	205	158

¹ Open-source apps from GitHub.

TABLE II
FIXALLY’S EFFECTIVENESS IN FIXING DIFFERENT TYPES OF ISSUES

Category	Issue type	<i>n</i>	PF ¹
Clipped text	Text clipped	15	11
Contrast	Contrast failed	27	21
Contrast	Contrast nearly passed	21	17
Dynamic type	Dynamic Type font sizes are partially unsupported	16	11
Dynamic type	Dynamic Type font sizes are unsupported	58	39
Element description	Element has no description	43	38
Element description	Label not human-readable	3	3
Element detection	Potentially inaccessible text	8	7
Hit region	Hit area is too small	21	17

¹ Indicates the number of issues with at least one plausible fix.

some apps to purposefully contain these issues, Accessibility Inspector detected these only on the iOS simulator and not on the physical device used for our experiments. Excluding Traits, FIXALLY could successfully resolve at least one issue from each type.

To understand the failures of FIXALLY, the first author manually inspected a subset of the generated fix suggestions that did not resolve the accessibility issues. Their analysis suggests that these failures may stem from shortcomings in the planning, localization, or fixing phases. For example, for the issue “Text Clipped” only one of the fix suggestions was plausible. Two out of three plans generated by the planner were irrelevant, indicating that not all issues may have multiple plausible solutions. Additionally, in some cases, the tool may select incorrect code snippets for fixing. When there is insufficient information about the problematic element, the model might fail to identify the relevant code snippet, leading to ineffective fix suggestions. Even if localization is accurate, the generated code may contain build errors or be ineffective. Despite these issues,

the proposed plans, related code snippets, and SwiftUI accessibility attributes generated by the tool can still help developers devise a fix more quickly. Furthermore, some reported failures were due to false positives from the Accessibility Inspector. For instance, after testing the app with different font sizes, we found that the issue “Dynamic Type font sizes are unsupported” was incorrectly reported in two cases for Landmarks app. Given these factors, the tool’s effectiveness in practice may be higher than what is reflected in our current report.

We also hypothesized that the capabilities of FIXALLY extend beyond the issues reported by the Inspector. In one experiment on a reported issue on GitHub for an iOS app [27], we attempted to fix the “incorrect focus order” issue using FIXALLY. Due to the lack of accessibility identifiers and GUI tests, we manually localized the code snippet and allowed the tool to perform the planning and fixing phases. We provided the screenshot and the title of the report to the model. Given the related code snippet, the model’s first plan was to “set the accessibilityElements property of the parent view.” The model generated a fix, adding `‘.accessibilityElement(children:.contain)’` to the parent view, which was very similar to the fix submitted by developers. To ensure there was no data leakage to the LLM, we confirmed that the commit date for fixing that issue was after the knowledge cut-off date of GPT-4. While this experiment demonstrates that FIXALLY’s capabilities can extend beyond the issues reported by the Accessibility Inspector, further experiments are needed.

C. RQ2. FIXALLY’s efficiency

In this research question, we assessed FIXALLY’s efficiency in terms of the number of attempts, time, and number of tokens required to fix issues.

In terms of the number of attempts, our experiments indicate that out of 363 plausible fix suggestions, FIXALLY generated 157 of them on the first attempt, while FIXALLY generated 124 and 82 on the second and third attempts, respectively. For the majority of the plausible fix suggestions (57%), the feedback loop design helped FIXALLY resolve the reported failures and generate a plausible fix suggestion, positively impacting the model’s effectiveness. However, this improvement in effectiveness comes at the cost of efficiency. By making this feedback loop a configurable parameter, users can adjust it to match their specific resource constraints, thereby balancing efficiency and effectiveness.

We also evaluated the time required for the tool to fix the issues. The mean time to propose alternative fix suggestions (averaged across 10 randomly selected issues) was 54 seconds. Therefore, for a screen with 10 issues, the tool can provide solutions in less than 10 minutes, demonstrating its practical usability. The breakdown of this time is as follows:

The time required for the Data Processing module to statically analyze the app depends on the project’s com-

plexity and the number of views it contains. For the apps in our test set, it takes an average of 6 ms to extract the UI hierarchy and instrument the code. Extracting accessibility issues involves building the app, running the GUI test, and using the Accessibility Inspector to dynamically assess the app. These steps, performed by both the Data Processing and Suggestion Assessment modules, take an average of 130 ms per screen.

FIXALLY’s performance is also closely tied to the LLM response time. We use a publicly available API to communicate with the agents, and various factors, such as online traffic and token constraints per minute, can impact the model’s response time. In our experiments, the average response times (across 10 randomly sampled issues) for the Planner, Localizer, and Fixer were 7s, 29.5s, and 8.2s, respectively. The Localizer’s longer response time is due to its need to evaluate multiple candidate snippets, requiring more than one inquiry to the model.

In addition to time considerations, the cost of using LLMs is closely related to the number of tokens processed. The average number of tokens per inquiry to the Planner, Localizer, and Fixer is 10K, 15K, and 20K, respectively. The GPT-4o model used in this study costs \$5 per 1M tokens. For a screen with about 10 issues, the total cost of using the tool is less than \$10.

D. RQ3. FIXALLY’s helpfulness

To evaluate the helpfulness of FIXALLY in assisting developers with fixing accessibility issues, we conducted a survey of 12 iOS developers within our company. We recruited them from a participant pool from prior studies with around 60 candidates. In the survey, developers rated suggestions produced by FIXALLY and gave feedback on the overall usefulness of the tool. The developers self-rated their SwiftUI iOS development experience on a scale from 1 (No Experience) to 5 (Expert). The developers’ median self-rated expertise in SwiftUI app development was 5 (■) and in accessibility testing was 5 (■). We excluded developers who self-rated as 1 (No Experience) for either of these questions from answering the survey.

We randomly selected 9 issues and generated plausible fixes for them using FIXALLY. Figure 6 illustrates one of these issues with three alternative code suggestions to fix the issue. For each issue, we showed developers an annotated screenshot with the issue reported by Accessibility Inspector along with a diff visualization for each of the three suggestions and the fix plan and explanation of changes from the LLM.

1) *Acceptance of Fix suggestions:* Initially, the survey asked the developers to describe what the issue meant and if they had a hypothesis and a plan for how to fix the issue before showing them the fix suggestions. The developers overall accessibility testing expertise was relatively high. We presented the developers three fix suggestions from the tool and asked whether they would accept any of the proposed suggestions. Overall, developers accepted a

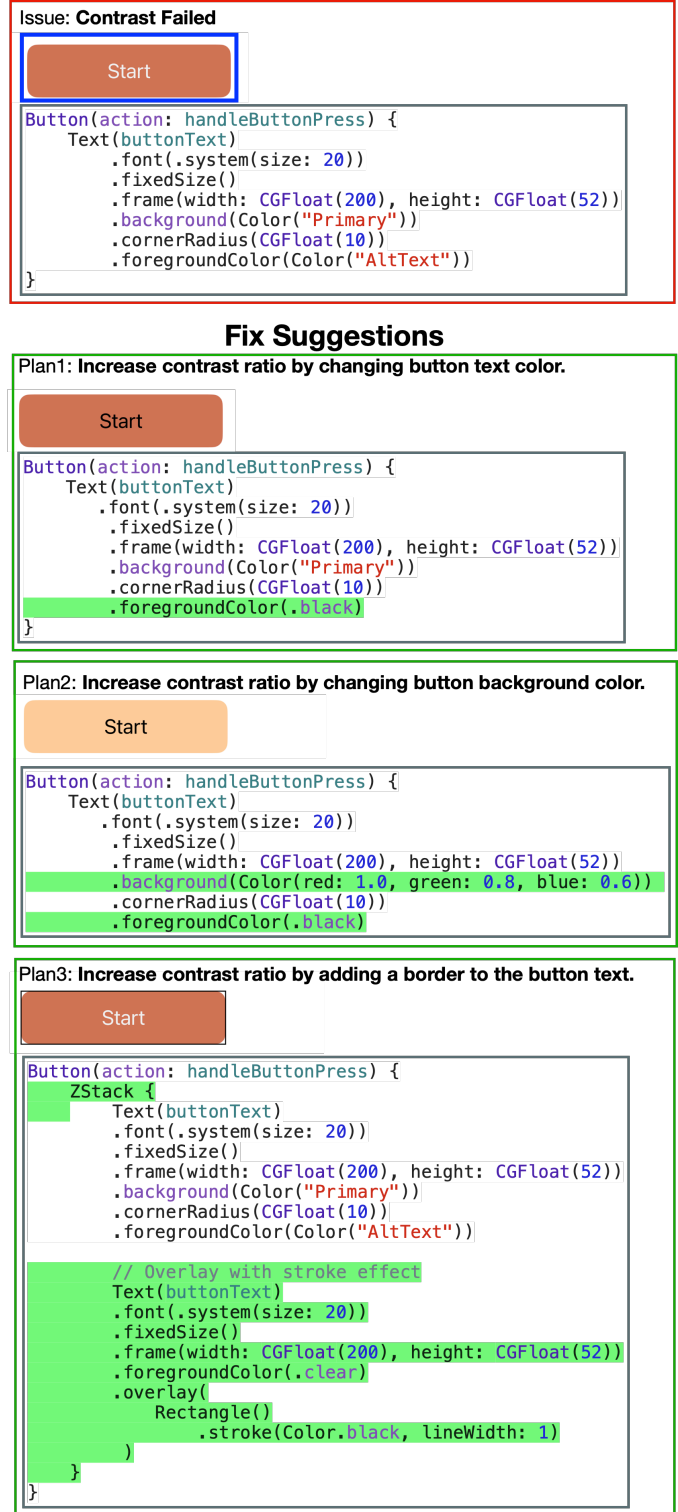


Fig. 6. A sample issue with three fix suggestions generated by FIXALLY. The first image shows the “Contrast Failed” issue and its original code snippet in Ratio app. The following three boxes represent different plans, each with the modified code snippet and the corresponding screenshot updates.

suggestion as is or with some modifications for 70% of the issues.

Developers did not always accept a fix suggestion that aligned with their initial hypothesis and fix plan. An accessibility testing expert (also an author) assessed “alignment” of developers’ hypothesis and fix plan matched the corresponding fix suggestion using a scoring rubric of 0 – did not match, 1 – matched with some conceptual difference or different level of specificity, and 2 – perfect match. The mean score for “alignment” was 1.1 (Med: 1, $\sigma = 1.31$) and 68% of the accepted fixes at least partially aligned with the developers’ initial fix plan and hypothesis.

2) *Helpfulness*: Developers also rated how helpful (from 1 - least helpful to 5 - most helpful) they would find a tool that proposed suggestions for fixing accessibility issues. Developers median rating for helpfulness was 3 (■...). Less experienced developers in accessibility testing rated the tool as more helpful.

Developers rating the tool a 5 or 4 (6 developers) found it useful the tool gave multiple suggestions, and noted it was especially useful to help less experienced developers in accessibility testing and implementation to understand different fix strategies.

While developers who rated the tool a 2 (Slightly Helpful) or 3 (Moderately Helpful) (6 developers) found the FIXALLY’s localization of the issue helpful, and liked that it provided several options, they critiqued some solutions for not addressing the root of the issue or for being sub-optimal. All developers rating FIXALLY’s helpfulness as 2 or 3 predominantly focus on accessibility engineering and testing for their job, while the remaining developers (rating helpfulness as 4 or 5) primarily focus on software engineering and occasionally perform accessibility testing.

The developers’ opinions were also mixed on whether it would be useful for the tool to also provide sub-optimal fix suggestions or suggestions which do not fix the issue. Some thought it would be useful to see alternative suggestions to stoke the developer’s imagination for finding the right solution (6 developers), while some (3 developers) thought providing these suggestions which were known to not fix the issue could be confusing or distracting, especially to developers less experienced with accessibility testing.

3) *Plausible fix vs correct fix.*: With FIXALLY, our goal was to generate plausible fixes—code modifications that pass the accessibility checks of an automated scanner without introducing new issues or removing any functionality. Our survey shows that out of 36 developer assessments of plausible fix suggestions, only 11 cases did not receive approval for any of the suggested fixes, resulting in a 69.4% developer acceptance of fixes. However, further studies are needed to understand the various considerations app development teams take into account when determining a correct fix, before automating that process.

VII. THREATS TO VALIDITY

External Validity: One limitation of FIXALLY is that it assesses issues individually, even though many issues may be interconnected. For example, fixing one issue might resolve others, or altering the appearance of one element might require adjustments to other related elements. To address this, we have designed the Fixer prompt to cascade design changes across elements to maintain design integrity. As Figure 5(c) shows, the model can consider this aspect in some cases. However, without a clear definition of relevant elements and design integrity, the tool’s limitations and capabilities are unknown. Future work could focus on developing metrics to assess design integrity or grouping related issues to propose unified solutions and enhance the performance of the tool.

Additionally, FIXALLY focuses on single-view files for localizing and generating fixes for issues, which means it cannot address problems that require changes across multiple files. Although the issues in our test set could be resolved within single files, studying more complex, cross-file issues—albeit less common—remains a compelling area for future research. This limitation also impacts the consideration of overall app design integrity beyond a single screen. While providing multiple suggestions allows developers to choose the most suitable one based on app design decisions, incorporating techniques to group issues across different app screens would enhance the tool.

Lastly, the tool has been implemented and tested on iOS apps. We believe that the benefits of FIXALLY can be generalized to other platforms by using appropriate tools to build, instrument, and audit apps on those platforms. The system definitions for agents can also be adjusted according to the platform, such as specifying expertise in SwiftUI for iOS or in Android development for Android apps. However, this generalizability needs to be further evaluated.

Internal Validity: We implemented FIXALLY using various tools and libraries, including XCTest, Accessibility Inspector, and the Tree-sitter library for code parsing [28]. These external tools may introduce defects into the system, and the prototype itself may contain implementation bugs. To mitigate these issues, we tested the tool on a variety of apps at different stages and ensured that we used the latest updates of the external tools.

VIII. RELATED WORK

Our work fits into the space of automated accessibility testing and repair tools, which have advanced the state-of-the-art for automated detection and reported of accessibility issue. Our work is among the first to localize and suggest fixes in code for these issues. We also review work LLM-based program repair and fault localization which use similar multi-agent architecture, but do not address GUI or accessibility issues.

A. Automated accessibility testing and repair

Many automated tools have been released and proposed in research over the years to detect accessibility issues. Static tools [29] examine code directly to find potential issues. One limitation with these tools is that they do not have access to the run-time interface that can be created programmatically or injected with data at runtime. Test-time tools can detect issues from the runtime UI [30]–[32] but are limited by the coverage of the input UI tests which prior work suggests may not exist or have very incomplete coverage over UI states [33]. Accessibility scanners that examine a run-time interface can detect different classes of issues that surface at run-time [9], [10], [30] and do not rely on pre-existing UI tests. However, accessibility scanners have two main limitations: 1) they do not assist developers in effectively localizing the impacted UI element with an issue, and 2) they provide developers with little assistance in fixing the issue other than sometimes a single high-level non-contextualized fix suggestion. FIXALLY takes as input an issue detected by one of these tools, its description, information about the impacted UI element in the form of a screenshot (i.e., Accessibility Inspector [10]), and the app source code, and both localizes the UI element and provides multiple relevant fix suggestions.

Some work combines run-time accessibility scanners with app crawlers to detect and report accessibility issues [12], [34], [35]. While these tools can surface more issues to developers, the amount of issues reported by these tools can be overwhelming [36] especially when even localizing and fixing one issue is already challenging. These tools are also not connected to the underlying source code, which can often be a reason for developers to ignore the reported issues from these tools.

Another area of work has developed single-purpose, mostly machine-learning based, techniques to detect specific accessibility issues such as color issues [13], touch target size [16], missing labels [14], and text scaling [15]. These solutions have predominately focused on Android apps, which have very different specification than iOS apps and are not implemented in SwiftUI. Zhang et. al [6] detect and repair UI elements for the iOS VoiceOver screen reader using an approach that could be platform agnostic. However, there remains a huge gap between these solutions and the original source code where the developer must make the fix. Even if the localization issue were solved by these methods, there would still be a need to generate alternate fix solutions. As we learned in our formative interviews, fixing one issue may introduce other issues, so there is likely no one-size-fits-all fix for each issue category. Developers need to consider many other requirements to determine the correct fix (e.g., input from design teams, impact on other areas of the app). In contrast to prior solutions, FIXALLY both localizes the code for the impacted UI element and suggests multiple candidate fixes. Future versions of FIXALLY can also incorporate some of these

techniques into its pipeline for issue detection, and then rely on the capabilities of its LLM to suggest candidate solutions.

B. LLM-based program repair and fault localization

Recent progress in LLMs have advanced their application in automating program repair tasks. A systematic survey on LLMs for automated program repair [37] reviewed 127 studies, covering various aspects of this problem, but none specifically address GUI or accessibility issues.

The only study focused on GUI issues is ACCESS [38], which examines LLM capabilities in correcting web accessibility violations by exploring different prompt engineering techniques to fix issues reported on specific HTML tags. This approach is limited to textual data from web pages and does not address mobile app issues, where it is necessary to identify the location of the problematic GUI element in the code.

Recently, researchers have expanded beyond prompt engineering to enhance LLM-based bug repair performance for GitHub issues. They have employed various techniques, including fine-tuning on specific datasets [39]–[43], or more advanced strategies such as retrieval-augmented generation to guide search space [44], agents interacting with other tools [45], [46], or multi-agent systems operating in different steps [47]–[49].

In these works [48], [49], researchers drew inspiration from human roles in real-world scenarios, such as Manager and Developer, to design LLM agents with specific actions. Tao et al. [48] enforce collaboration between agents to localize a file and implement a patch. Chen et al. [49] proposed four collaborative plans for agents to address issues, with the Manager agent selecting a plan from pre-defined options. These studies demonstrate how modeling software engineering processes with agents can enhance issue resolution capabilities. However, these approaches only address functional issues reported on GitHub.

In contrast to prior related work, our work targets accessibility issues in mobile apps using a novel multi-agent LLM architecture. It designs agents inspired by the steps developers take to solve these problems while also considering their unique characteristics, such as the diversity of accessibility issues, various fix strategies, and the need to analyze GUI images with reported issues. We developed FIXALLY, which analyzes issues in mobile apps, employs a plan-localize-fix process, and produces plausible fixes. Additionally, FIXALLY incorporates a feedback loop that uses natural language error messages to enable the LLM to reflect [50] on and address its own failures.

IX. CONCLUSION

Fixing accessibility issues in mobile apps is a challenging task for developers. While automated scanners can identify these issues, they often fall short in guiding developers to the exact location in the code and suggesting appropriate

fixes. Towards addressing these needs, we proposed a plan-localize-fix technique, operationalized through a multi-agent LLM architecture called FIXALLY. Evaluations on FIXALLY demonstrate its capabilities in suggesting plausible fix suggestions and highlight how the tool assists the developer in the decision-making process for selecting and applying accessibility-related fixes. Our results suggest that applying LLMs to fix accessibility issues in source code is an encouraging direction.

REFERENCES

- [1] Android, “Build more accessible apps,” <https://developer.android.com/guide/topics/ui/accessibility>, Google, 2022, last Accessed: May 6, 2022.
- [2] Apple, “Accessibility on ios,” <https://developer.apple.com/accessibility/ios/>, Apple, 2022, last Accessed: May 6, 2021.
- [3] W3C, “Wcag 2 overview,” <https://www.w3.org/WAI/standards-guidelines/wcag/>, W3C, 2024, last Accessed: July 18, 2024.
- [4] A. Alshayban, I. Ahmed, and S. Malek, “Accessibility issues in android apps: state of affairs, sentiments, and ways forward,” in *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE, 2020, pp. 1323–1334.
- [5] C. Vendome, D. Solano, S. Liñán, and M. Linares-Vásquez, “Can everyone use my app? an empirical study on accessibility in android apps,” in *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2019, pp. 41–52.
- [6] X. Zhang, L. de Greef, A. Swearngin, S. White, K. Murray, L. Yu, Q. Shan, J. Nichols, J. Wu, C. Fleizach *et al.*, “Screen recognition: Creating accessibility metadata for mobile applications from pixels,” in *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*, 2021, pp. 1–15.
- [7] A. S. Ross, X. Zhang, J. Fogarty, and J. O. Wobbrock, “Examining image-based button labeling for accessibility in android apps through large-scale analysis,” in *Proceedings of the 20th International ACM SIGACCESS Conference on Computers and Accessibility*, 2018, pp. 119–130.
- [8] T. Bi, X. Xia, D. Lo, J. Grundy, T. Zimmermann, and D. Ford, “Accessibility in software practice: A practitioner’s perspective,” *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 31, no. 4, pp. 1–26, 2022.
- [9] Google, “Get started with accessibility scanner - android accessibility help,” 2024. [Online]. Available: <https://support.google.com/accessibility/android/answer/6376570>
- [10] A. Inc., “Accessibility Programming Guide for OS X: Testing for Accessibility on OS X,” March 2022. [Online]. Available: <https://developer.apple.com/library/archive/documentation/Accessibility/Conceptual/AccessibilityMacOSX/OSXAXTestingApps.html>
- [11] F. Mehralian, N. Salehnamadi, S. F. Huq, and S. Malek, “Too much accessibility is harmful! automated detection and analysis of overly accessible elements in mobile apps,” in *2022 37th IEEE/ACM International Conference on Automated Software Engineering*, IEEE. Rochester, Michigan, USA: ACM New York, NY, USA, 2022.
- [12] N. Salehnamadi, F. Mehralian, and S. Malek, “Groundhog: An automated accessibility crawler for mobile apps,” in *2022 37th IEEE/ACM International Conference on Automated Software Engineering*, IEEE. Rochester, Michigan, USA: ACM New York, NY, USA, 2022.
- [13] Y. Zhang, S. Chen, L. Fan, C. Chen, and X. Li, “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*, 2023, pp. 1255–1267.
- [14] F. Mehralian, N. Salehnamadi, and S. Malek, “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*, 2021, pp. 107–118.
- [15] A. S. Alotaibi, P. T. Chiou, F. M. Tawsif, and W. G. Halfond, “ScaleFix: An Automated Repair of UI Scaling Accessibility Issues in Android Applications,” in *Proceedings of the 39th IEEE International Conference on Software Maintenance and Evolution (ICSME)*, October 2023.
- [16] A. S. Alotaibi, P. T. Chiou, and W. G. Halfond, “Automated repair of size-based inaccessibility issues in mobile applications,” in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2021, pp. 730–742.
- [17] Apple Developer, “Swiftui - build user interfaces for any apple device,” <https://developer.apple.com/xcode/swiftui/>, 2024, accessed: 2024-07-20.
- [18] I. D. Foundation, “Affinity diagrams,” Dec 2017. [Online]. Available: <https://www.interaction-design.org/literature/topics/affinity-diagrams>
- [19] A. Inc., “Xctest,” Aug 2023. [Online]. Available: https://developer.apple.com/documentation/xctest/user_interface_tests
- [20] testableapple, “xcmonkey: Stress testing tool for ios apps,” 2024, accessed: 2024-07-20. [Online]. Available: <https://github.com/testableapple/xmonkey>
- [21] T. Su, G. Meng, Y. Chen, K. Wu, W. Yang, Y. Yao, G. Pu, Y. Liu, and Z. Su, “Guided, stochastic model-based gui testing of android apps,” in *Proceedings of the 2017 11th joint meeting on foundations of software engineering*, 2017, pp. 245–256.
- [22] A. Inc., “accessibilityidentifier,” July 2024. [Online]. Available: <https://developer.apple.com/documentation/uikit/uiaccessibilityidentification/1623132-accessibilityidentifier>
- [23] OpenAI, “Gpt-4,” <https://openai.com/index/gpt-4/>, 2024, accessed: 2024-07-20.
- [24] vsouza, “Awesome ios,” <https://github.com/vsouza/awesome-ios>, Accessed: 2024.
- [25] dkhaming, “Open source ios apps,” <https://github.com/dkhaming/open-source-ios-apps>, Accessed: 2024.
- [26] Apple Inc., “Performing accessibility audits for your app,” Apple Developer Documentation, July 2024, accessed: 2024-08-01. [Online]. Available: <https://developer.apple.com/documentation/accessibility/performing-accessibility-audits-for-your-app>
- [27] Orange Open Source, “Issue #703: Check accessibility issues on ios17,” GitHub, 2023, accessed: 2024-08-01. [Online]. Available: <https://github.com/Orange-OpenSource/ods-ios/issues/703>
- [28] Tree-sitter, “Tree-sitter,” <https://tree-sitter.github.io/tree-sitter/>, 2024, accessed: 2024-07-24. [Online]. Available: <https://tree-sitter.github.io/tree-sitter/>
- [29] Google, “Android lint,” 2022. [Online]. Available: <https://support.google.com/accessibility/android/answer/6376570>
- [30] N. Salehnamadi, A. Alshayban, J.-W. Lin, I. Ahmed, S. Branham, and S. Malek, “Latte: Use-case and assistive-service driven automated accessibility testing framework for android,” in *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*, 2021, pp. 1–11.
- [31] G. O. S. Framework), “Earl grey: ios ui automation test framework,” 2022. [Online]. Available: <https://github.com/google/EarlGrey>
- [32] Google, “Espresso,” 2021. [Online]. Available: <https://developer.android.com/training/testing/espresso>
- [33] M. Padure and C. Pribeanu, “Comparing six free accessibility evaluation tools,” *Informatica Economica*, vol. 24, no. 1, pp. 15–25, 2020.
- [34] M. M. Eler, J. M. Rojas, Y. Ge, and G. Fraser, “Automated accessibility testing of mobile apps,” in *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2018, pp. 116–126.
- [35] A. Swearngin, J. Wu, X. Zhang, E. Gomez, J. Coughenour, R. Stukenborg, B. Garg, G. Hughes, A. Hilliard, J. P. Bigham *et al.*, “Towards automated accessibility report generation for mobile apps,” *ACM Transactions on Computer-Human Interaction*.
- [36] S. F. Huq, A. Alshayban, Z. He, and S. Malek, “# a11ydev: Understanding contemporary software accessibility practices from twitter conversations,” in *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*, 2023, pp. 1–18.

- [37] Q. Zhang, C. Fang, Y. Xie, Y. Ma, W. Sun, and Y. Y. Z. Chen, "A systematic literature review on large language models for automated program repair," *arXiv preprint arXiv:2405.01466*, 2024.
- [38] C. Huang, A. Ma, S. Vyasamudri, E. Puype, S. Kamal, J. B. Garcia, S. Cheema, and M. Lutz, "Access: Prompt engineering for automated web accessibility violation corrections," *arXiv preprint arXiv:2401.16450*, 2024.
- [39] W. Wang, Y. Wang, S. Joty, and S. C. Hoi, "Rap-gen: Retrieval-augmented patch generation with codet5 for automatic program repair," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2023, pp. 146–158.
- [40] B. Berabi, J. He, V. Raychev, and M. Vechev, "Tfix: Learning to fix coding errors with a text-to-text transformer," in *International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, vol. PMLR, 2021, pp. 780–791.
- [41] M. Fu, C. Tantithamthavorn, T. Le, V. Nguyen, and P. Dinh, "Vulrepair: A t5-based automated software vulnerability repair," in *Proceedings of the ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE. ACM, 2022, pp. 935–947.
- [42] E. Mashhadi and H. Hemmati, "Applying codebert for automated program repair of java simple bugs," in *Proceedings Companion of the 18th IEEE/ACM International Conference on Mining Software Repositories (MSR'21)*, 2021, pp. 505–509.
- [43] M. Lajkó, V. Csuvik, and L. Vidács, "Towards javascript program repair with generative pre-trained transformer," in *2022 IEEE/ACM International Workshop on Automated Program Repair*. IEEE, 2022, pp. 61–68.
- [44] C. E. Jimenez, J. Yang, A. Wettig, S. Yao, K. Pei, O. Press, and K. Narasimhan, "Swe-bench: Can language models resolve real-world github issues?" *arXiv preprint arXiv:2310.06770*, 2023.
- [45] J. Yang, C. E. Jimenez, A. Wettig, K. Lieret, S. Yao, K. Narasimhan, and O. Press, "Swe-agent: Agent-computer interfaces enable automated software engineering," *arXiv preprint arXiv:2405.15793*, 2024.
- [46] I. Bouzenia, P. Devanbu, and M. Pradel, "Repairagent: An autonomous, llm-based agent for program repair," *arXiv preprint arXiv:2403.17134*, 2024.
- [47] Y. Zhang, H. Ruan, Z. Fan, and A. Roychoudhury, "Autocoderover: Autonomous program improvement," *arXiv preprint arXiv:2404.05427*, 2024.
- [48] W. Tao, Y. Zhou, W. Zhang, and Y. Cheng, "Magis: Llm-based multi-agent framework for github issue resolution," *arXiv preprint arXiv:2403.17927*, 2024.
- [49] D. Chen, S. Lin, M. Zeng, D. Zan, J.-G. Wang, A. Cheshkov, J. Sun, H. Yu, G. Dong, A. Aliev *et al.*, "Coder: Issue resolving with multi-agent and task graphs," *arXiv preprint arXiv:2406.01304*, 2024.
- [50] N. Shinn, F. Cassano, A. Gopinath, K. Narasimhan, and S. Yao, "Reflexion: Language agents with verbal reinforcement learning," *Advances in Neural Information Processing Systems*, vol. 36, 2024.