VisioAux: a crowdsource approach for accessibility evaluation in mobile devices

Alex Frederico Ramos Barboza[1], **Marcelo Medeiros Eler**[1]

[1]School of Arts, Sciences and Humanities (EACH) – University of São Paulo (USP)
Rua Arlindo Béttio, 1000 – 03828-000 – São Paulo – SP – Brazil

{alex.barboza,marceloeler}@usp.br

Abstract. Introduction: *The rapid growth in mobile device usage has increased the need for accessible applications. Automated accessibility testing approaches were developed to support accessibility evaluation, but they have limitations such as low coverage of features and screens and the lack of prioritization of issues by real-world impact.* ***Objective:*** *This paper introduces an automated crowdsourcing-based strategy to increase coverage and provide impact-based prioritization of accessibility issues in mobile applications.* ***Method:*** *We developed VisioAux, an accessibility testing service integrated into mobile apps and deployed on end users' devices. As users navigate apps in their daily activities, VisioAux performs runtime accessibility checks on apps under test. VisioAux collects reports from multiple devices and consolidates results to estimate coverage and issue severity for a given app.* ***Results:*** *A preliminary evaluation compared VisioAux with Google Accessibility Scanner, assessed its performance impact, and verified its crowdsourcing capabilities. Results indicate comparable detection performance, negligible performance overhead, and successful aggregation of accessibility reports from diverse devices.*
Keywords *Accessibility, Testing, Monitoring, Mobile, Android, Crowdsourcing*

1. Introduction

The use of mobile devices such as tablets and smartphones has increased dramatically in recent years [Cerwall 2021]. In order to guarantee the inclusion of people with disabilities in this platform, accessibility guidelines such as WCAG 2.2 [W3C 1999], BBC [BBC nd], and ABNT NBR 17060 [Da Costa Nunes et al. 2024] have been proposed to guide developers and designers to create inclusive digital products and services. However, the adoption of accessibility guidelines in the industry is quite a complex issue as developers tend to not address accessibility as a critical factor in their professional practice [Lewthwaite et al. 2023, Patel et al. 2020, Leite et al. 2021, Bi et al. 2022]. On the other hand, companies rarely expect candidates to be familiar with accessibility guidelines [Martin et al. 2022]. Not surprisingly, studies indicate a general lack of accessibility in mobile applications across many domains, sizes, and complexities [Chen et al. 2021a, Yan e Ramachandran 2019a, Eler et al. 2018a].

Accordingly, many researchers and organizations have developed automated tools for supporting accessibility testing, which is especially useful when there is a lack of testing resources. In general, automated approaches for accessibility testing are based on static or dynamic analysis. Static analysis has the advantage of quickly analyzing many software artifacts to detect accessibility issues; however, in mobile settings, such

types of analysis are quite limited since many accessibility violations can only be detected using dynamic analysis in runtime due to device configurations and screen rendering [Silva et al. 2018a, Eler et al. 2018a].

Dynamic automated testing approaches differ in how they detect accessibility violations and the way each screen of the app under test is reached [Silva et al. 2018a, Eler et al. 2018a]: using test scripts, manual exploration, or fully automated. The limitation of test scripts is that they require specialized professionals to develop thorough test sets to achieve a high coverage of application screens, which hinders the adoption of such strategy [Kochhar et al. 2015, Silva et al. 2018a]. The limitation of manual exploration is that it is time-consuming and it is difficult to repeat across different versions or test sessions, leaving many parts of the application uncovered [Matos et al. 2023]. To overcome the limitations of scripts and manual strategies, fully automated approaches rely on algorithms that mimic user interaction to explore the application under test and run accessibility checks during the exploration. However, fully automated approaches also have limitations, such as the low coverage of features and screens, especially when the exploration depends on context-aware or sensitive information (e.g. passwords, personal data), or complex user interaction.

In addition to the specific limitations of different strategies, automated approaches have intrinsic limitations. First, they cannot detect all accessibility issues since the detection of many violations depends on specific usage contexts and human evaluation; therefore, relying only on automated approaches is not recommended [Mateus et al. 2020]. Nevertheless, they are useful to make testing activities more productive considering the accessibility violations they can detect. Second, accessibility bug reports produced by automated tools cannot specify the impact of each accessibility violation for the end user, thus making it difficult to prioritize the issues to be fixed.

In this context, this work proposes an innovative strategy to mitigate some of the limitations of fully automated approaches: automated crowdsourcing-based accessibility evaluation. In the context of software development, crowdsourcing is "the performance of specific software development tasks on behalf of an organization by a large and typically undefined group of external people with the necessary expertise" [Stol e Fitzgerald 2014]. Therefore, typically, crowdsourcing strategies for software evaluation consist of experts or end users carrying out tasks to evaluate the presence of accessibility barriers in parts of a given application. In our approach, instead of experts evaluating mobile apps on different devices and scenarios, accessibility testing is automatically carried out by a testing service we designed to be deployed on the end user's device as an accessibility service of the operating system (e.g. Android).

In practice, developers interested in enabling automated crowdsourcing testing of their mobile apps can include our testing service as a dependency of their mobile project using artifact repositories (e.g. Maven). When the mobile app is installed or updated, the testing service will also be installed on the device upon user approval. Accordingly, when the user opens an application under test, our testing service runs accessibility checks in each new screen explored by the end user, enabling continuous monitoring of accessibility issues in these applications. The crowdsourcing aspect of this work is that the accessibility bug reports produced by the testing service in each device will be sent to a server where

all bug reports received by all devices will be combined and consolidated. In addition, our approach relies on users to visit the different screens of a mobile application that will be tested, not as experts or testers with a planned scenario, but as real people who use mobile applications during their daily activities.

Such an approach helps mitigate two specific issues. First, achieving high feature and screen coverage is difficult in automated approaches. In our approach, high coverage will be achieved by the combination of many distinct features and screens explored by users of different profiles who use applications for different purposes. The second issue our approach helps to mitigate is the prioritization of accessibility bugs. Considering that bug reports come from many devices, information concerning the most frequently accessed features can be used along with the type of issues information to create specific impact measures. Furthermore, our approach can be used both during development time, with intermediate versions of the application evaluated by specific stakeholders, or during operation time, when the applications are already installed on end users' devices.

In that sense, this paper makes three key contributions: first, it introduces an innovative approach to accessing and reporting accessibility issues through the application of crowdsourcing principles. Second, it proposes a novel severity measure that enables the prioritization of accessibility issue fixes. Finally, it presents VisioAux, a framework designed to work as an accessibility testing service that can run on end users' devices along with monitored mobile applications. Three aspects of our approach were evaluated: a) violation detection capabilities; b) performance overhead introduced by VisioAux on mobile devices; and c) crowdsourcing features. Considering the limitations of our preliminary evaluation, our findings show that the violation detection capabilities of VisioAux are comparable to established tools such as Google Accessibility Scanner; VisioAux does not introduce significant overhead; and VisioAux can manage data received from different devices and produce combined accessibility reports.

The remainder of this paper is organized as follows: Section 2 shows the current state of the art in accessibility evaluation for mobile devices; Section 3 details the VisioAux methodology and its architecture; Section 4 discusses results obtained with VisioAux; and Section 5 presents concluding remarks.

2. Related Work

The mobile development industry has established resources for mobile accessibility testing, which include tools frequently used by developers and testers to identify problems in their mobile apps. For example, Lint is a static analysis tool that provides suggestions for accessibility improvements [Google 2024b]. Robolectric [Robolectric 2024] and Espresso [Google 2024a] are testing frameworks that can be extended to perform accessibility checks [Eler et al. 2018b]. Additionally, Accessibility Scanner is an accessibility testing tool created by Google. It can be enabled, and while the app under test is being executed, it suggests improvements for accessibility purposes [Aashutosh 2023].

Despite the availability of these tools, accessibility problems are still consistently found in smartphone applications [Alshayban et al. 2020, Chen et al. 2021b]. Hence, in recent years, a considerable amount of scientific effort has been done in the field of human-computer interaction and software engineering to meet those accessibility criteria [Silva et al. 2018b, Dias et al. 2021]. These scientific studies can be classified into two

groups. The first group of studies consists of automatic correction of a specific subset of accessibility issues. The second group is characterized by methods capable of identifying a broader range of accessibility issues.

2.1. Automatically repairing issues

The first group leverages advanced algorithmic techniques to automatically fix violations of a subset of requirements from WCAG in smartphones. For example, AccessFixer [Zhang et al. 2023a] uses R-GCNs [1] to automatically repair small touch areas, low padding, and low contrast. Initially, AccessibilityScanner is used to identify screens with accessibility problems, and through the technique developed, AccessFixer captures information about these identified problems and formulates strategies to correct the accessibility problems.

Analogous to AccessFixer, AGAA (Android GUI Accessibility Adapter), [Xu et al. 2023] corrects undersized texts and redundant information issues. The proposed technique is implemented using an SDK (*Software Development Kit*) made up of three modules. The first module abstracts the graphical interface elements and the relationship between these elements as a data structure. The second SDK module generates high-quality repair solutions using genetic algorithms, and the third SDK module applies the solution found to the source code.

Similarly, ScaleFix [Alotaibi et al. 2023] automatically repairs accessibility problems related to scaling issues without distorting the graphical end-user interface. Using genetic algorithms, ScaleFix minimizes text cutting and overlap between graphic interface elements, maintains the characteristics of the elements on the screen, and minimizes the number of changes needed to the original user interface. During its execution, ScaleFix generates possible candidates for repairing the screen that contain the identified accessibility problem, adopting the one that best fits the criteria mentioned.

In addition, there is Iris [Zhang et al. 2023b], repairing errors regarding color contrast by also taking into account the context of each application. In the mentioned study, a large database was built of applications without color contrast problems, so that the color combinations of these applications can be used as a reference in the next stage. In the second stage of the technique, color pairs (foreground and background colors) are selected that take into account the context of each application. Finally, the color attributes of the problematic widgets (such as buttons or labels) are identified and replaced.

SALEM (**S**ize-based in**A**naccessibi**L**ity r**E**pair in **M**obile apps) [Alotaibi et al. 2021] is a novel approach to test size-based accesibility requirements, automatically repairing possible issues. As an input, Accessibility Scanner is used to identify size-based issues (such as small touch area in buttons). Then, in a second phase, SALEM identifies the properties of elements that must be adjusted to comply with accessibility guidelines. Then, using a genetic algorithm, the best candidate for a fix is used and a new version of the mobile application is generated, without the size-based accessibility problems.

Also, COALA (**Co**ntext-**A**ware **La**bels) [Mehralian et al. 2021] generates labels for Android icons. For screen readers - such as TalkBack in Android - to work properly,

[1] R-GCN - Relational Graph Convolutional Networks

it is essential that images and icons are accompanied by explanatory alternative texts [2]. However, the authors of COALA point out that it is important for alternative texts to be enriched by different sources of information, such as the context in which they are presented on the screens. To this end, the authors developed COALA, which aims to predict these alternative texts for icons and images. By using deep learning techniques, COALA creates a dataset of texts to be used as label suggestions.

2.2. Testing and identifying accessibility problems

Although such methods and tools are undoubtedly innovative, they all share the same characteristic , *i.e.*, they are capable of fixing rather specific accessibility problems. In contrast, there is a second major group of methods capable of detecting accessibility violations on mobile devices. Although not capable of automatically fixing violations, they are able to identify them and are particularly useful in the development and testing phases.

For example, MATE [Eler et al. 2018b] mimics user actions through scripts that interact with the screen being tested. This approach automatically generates accessibility tests when a new violation is found. MATE ends its execution whenever a parameterized time limit is reached. Thus, at the end of execution, the application can be classified as accessible, supported (it can be used by assistive technologies) or inaccessible. To explore the application, MATE combines screen state abstraction with the testing framework UI Automator. A screen state represents the application screen at a given moment in time, after actions have been performed by users, such as clicking a button that shows a new popup. Thus, whenever a new state is discovered, MATE assigns each element in the UI an equal probability of being chosen and, by interacting with an element in UI randomly selected, leads to the discovery of a new state. Every time a new state is discovered, new accessibility checks are performed.

A11Puppetry [Salehnamadi et al. 2023] uses the *RaR - Record and Replay* technique. The RaR technique allows developers to record interactions with screens during development and testing. In this way, during test execution, the manual inputs made are repeated in an automated way, eliminating the need for manual interaction by testers [Lam et al. 2017]. The recorded touch gestures are then played back by A11Puppetry, with the TalkBack screen reader activated. As the interactions are replayed, A11 Puppetry searches for unfocusable elements, actions that cannot be triggered by assistive technologies, and missing speakable texts.

Mobile applications must be operable through assistive technologies, such as the TalkBack screen reader on Android. Atari (**A**utomated detector of **T**alkbAck Inte**R**active Access**I**bility Failures) [Alotaibi et al. 2022] was created for this purpose. ATARI abstracts in a first model the possible interactions via TalkBack of a disabled user on a given screen via TalkBack. ATARI then abstracts, in a second model, the possible interactions of users who do not use screen readers. These two models are compared to identify operability flaws, *i.e.* actions that users of assistive technologies cannot perform due to the lack of accessibility in the application.

By abstracting navigation between screens into a graph-like data structure, MARS [Fok et al. 2022] looks for interface elements (such as buttons) that could trigger

[2]https://www.w3.org/TR/UNDERSTANDING-WCAG20/text-equiv-all.html

navigation to a new screen. Whenever a new screen is identified, MARS detects possible absences of alternative text in images and buttons with images.

For users with low vision, it is essential that text on screens is scalable, allowing fonts to be enlarged or reduced using text resizing features. To this end, AccessiText [Alshayban e Malek 2022] is an approach capable of identifying text resizing problems. AccessiText operates in two stages. First, the Test Runner executes automation scripts to analyze a screen twice. First as viewed by a user with regular vision, then again after resizing. Data from both runs is passed to the second stage, Result Analyzer, which compares the screens to detect (i) non-resized text, (ii) missing widgets, (iii) truncated graphics, and (iv) overlapping elements.

MAC (Mobile Accessibility Checker) by IBM [Yan e Ramachandran 2019b] allows developers to integrate a library into apps enabling automatic accessibility assessment while the app is being developed or tested. During testing, MAC monitors changes in screen states and classifies accessibility errors as violations, potential violations or warnings. In order to measure how inaccessible the GUI components of an application are, the authors of this study have also proposed the IAER (Inaccessible Element Rate) metric, which estimates the percentage of GUI elements that are inaccessible and cannot be used by users with disabilities, and the AIR metric, which calculates the percentage of actual accessibility problems in relation to the maximum number of accessibility problems that could affect an application.

Latte [Salehnamadi et al. 2021] reuses preexisting GUI tests to evaluate the accessibility of mobile applications. With assistive technologies enabled (such as TalkBack and TextSwitch enabled), Latte re-runs these tests mimicking the way users with disabilities would interact with the application. If a particular task described in the test could not be successfully completed using assistive technologies, this implies that the test detected an accessibility gap. Finally, Latte collects and analyzes the results.

Xbot [Chen et al. 2021b] is a technique that aims to maximize the number of screens explored in mobile apps. Using reverse engineering, Xbot enables the automated navigation in the mobile app by extracting navigation parameters (such as data needed to navigate to a new screen). Then, in a second stage of its execution, Xbot navigates to the screens by passing the necessary parameters. Whenever a new screen is discovered, the Xbot performs new accessibility checks using the Google Accessibility Test Framework (ATF). At this stage, the graphic interface elements that make up the newly explored screen are inspected and, if there are any accessibility problems, they are reported.

Groundhog [Salehnamadi et al. 2022] is an app crawler that assesses accessibility violations without manual effort. Groundhog is a technique for evaluating accessibility by replicating ways in which users with disabilities would interact with cell phones using assistive technologies, comparing the results obtained with interactions carried out without assistive technologies. Groundhog begins by exploring the different states of the application's screens. Whenever a new screen is found, all the possible actions that can be performed are extracted. Using the command line, these elements are triggered by a server. The result of the executions is then evaluated by a test oracle, which checks the accessibility level of the application.

2.3. Research gaps in mobile accessibility evaluation

Even techniques capable of detecting a wide range of accessibility issues still cover a small percentage of screens and features of complex applications, and they do not take into account how many end users are affected by those issues, nor how frequently those issues manifest for the users. In fact, a scrutiny of the current state of the art in accessibility assessment for mobile devices shows that the majority of available tools and techniques focus on the design, testing, and development stages of mobile applications.

To the best of our knowledge, no crowdsourcing strategy has been proposed to evaluate mobile applications accessibility [Song et al. 2018]. In the domain of Web accessibility, approaches to accessibility testing based on crowdsourcing adopt traditional strategies [Alyahya 2020]: experts carry out tasks to evaluate the presence of accessibility barriers in parts of a given web application. In our approach, however, the crowd's expertise is not required, as they act as exploring agents of mobile apps so automated tests can be executed.

For mobile applications, collecting data from the end-user base (essentially crowdsourcing insights from real users) has proven to be a reliable technique for monitoring app behavior and is widely adopted in the industry. For instance, Firebase Crashlytics monitors application crashes (when apps suddenly stop working and close by themselves), ranking them by the number of affected users [Firebase 2025]. Similarly, UXCam helps teams understand how users interact with the app, such as identifying the most commonly used features [UXCam 2025]. However, assessing how this would be feasible in the context of accessibility on mobile devices raises relevant technical and scientific questions that are worth addressing. Hence, this work presents an emerging and innovative approach to assessing accessibility problems in mobile devices by leveraging crowdsourcing concepts.

3. VisioAux: a crowdsource approach for accessibility evaluation

Considering the importance of mobile accessibility and the limitations of current automated approaches as discussed before, this paper introduces an automatic accessibility assessment strategy that leverages crowdsourcing concepts to reach two specific goals: i) obtain greater coverage of the functionalities and interfaces of the applications being evaluated; and ii) gather usage data that aids in defining metrics related to the severity or the impact of identified accessibility issues, which may be beneficial in devising prioritization strategies; iii) allow the continuous monitoring of the accessibility of mobile apps while they run on users devices. Accordingly, we devised a crowdsource approach for accessibility evaluation called *VisioAux*. A general overview of this approach and the reasoning behind some design decisions are presented as follows.

3.1. Approach overview

Most accessibility evaluation approaches and tools are based on the execution of accessibility tests during development or before releasing a mobile app. In addition, most tools are executed from the perspective of the developer, who has access to the source code of the application, or as third-party entities, which have access only to the interface of the application under evaluation. The general idea behind the *VisioAux* approach is to present a new perspective for accessibility evaluation: continuous accessibility testing of mobile applications as they are used by end users on their own devices.

To realize this vision, our approach relies on two main components: a testing service and a server. The testing service is executed along with the mobile app under evaluation and works as a continuous monitoring agent that runs accessibility checks on each screen rendered on the mobile device. In this approach, instead of using algorithms to mimic user interaction, the users themselves are the agents that explore the mobile applications as they perform their daily activities on mobile devices. In this scenario, the users are not taking part in a test session as this is not perceptible to them. Therefore, the only screens and features analyzed by the testing service are the ones spontaneously used by the end user. The results of all accessibility checks are thus sent to a server that combines and consolidates the results coming from the many devices where the testing service is running. The information collected does not contain sensitive data, as will be discussed later in this section.

The reasoning behind our strategy is that accessibility issues can only be detected if screens are somehow reachable by tools or human testers. However, mimicking user interaction to achieve suitable screen or feature coverage is limited, and manually exploring large and complex mobile applications is costly and time-consuming. Hence, our approach relies on the fact that, in real environments, distinct end users explore different features and screens of the same mobile applications. For example, one individual might utilize an application solely to read and send emails, whereas another might also employ it for scheduling meetings, and yet another might use it for making video calls. Even if a single user does not visit many screens of the application, we hypothesize that the combination of the screens visited by all users of the application will cover most of their features and screens.

In this context, we claim that we can achieve our first goal of reaching greater app coverage by combining the results of the testing executed on the features and screens accessed by many distinct users. Notice that, differently from traditional crowdsourcing testing approaches, which rely on users with disabilities or as experts to run tests on specific parts of a given application, our approach relies on the crowd only in the sense that their daily routine using mobile applications will provide the exploration of screens that will be checked against accessibility criteria by our testing service.

Considering that the server will consolidate data coming from many devices, we believe we can achieve the second goal, which is to gather information to devise prioritization strategies. For instance, we can measure which features and screens are most frequently accessed by end users. Taking into account how many end users were affected by a bug is already a common measure of severity for other types of bugs, such as crashes [Firebase 2025]. At the same time, different accessibility issues might have a different impact on end users with disabilities. For that reason, it is also important to take into account the nature of the accessibility violation itself.

At the daily reality of the software industry, where accessibility issues coexist with a plethora of other types of bugs and resources to fix are limited, it is essential to address accessibility violations based on the severity of their impact on users. Therefore, data gathered from mobile devices and accessibility checks could be combined with information regarding the features within a context, type of issue detected, and so forth, to propose a proxy measure for impact and severity. By introducing the possibility of calculating severity and impact using different metrics, this work aims to facilitate the

adoption of accessibility guidelines by highlighting which problems could be corrected first.

3.2. VisioAux architecture

In order to meet the requirements of our approach, we propose an implementation of *VisioAux* as a three-module architecture as presented in Figure 1: VisioSDK, VisioBack and VisioDashboard. VisioSDK works as a testing service, which runs accessibility checks on mobile apps while used by end users. Accessibility bug reports are thus sent to VisioBack, which works as the server that combines and consolidates bug reports from many sources. Finally, development teams use VisioDashboard to explore the accessibility bug reports retrieved from VisioBack to make informed decisions concerning bug fixing. Each component of this architecture is described as follows.

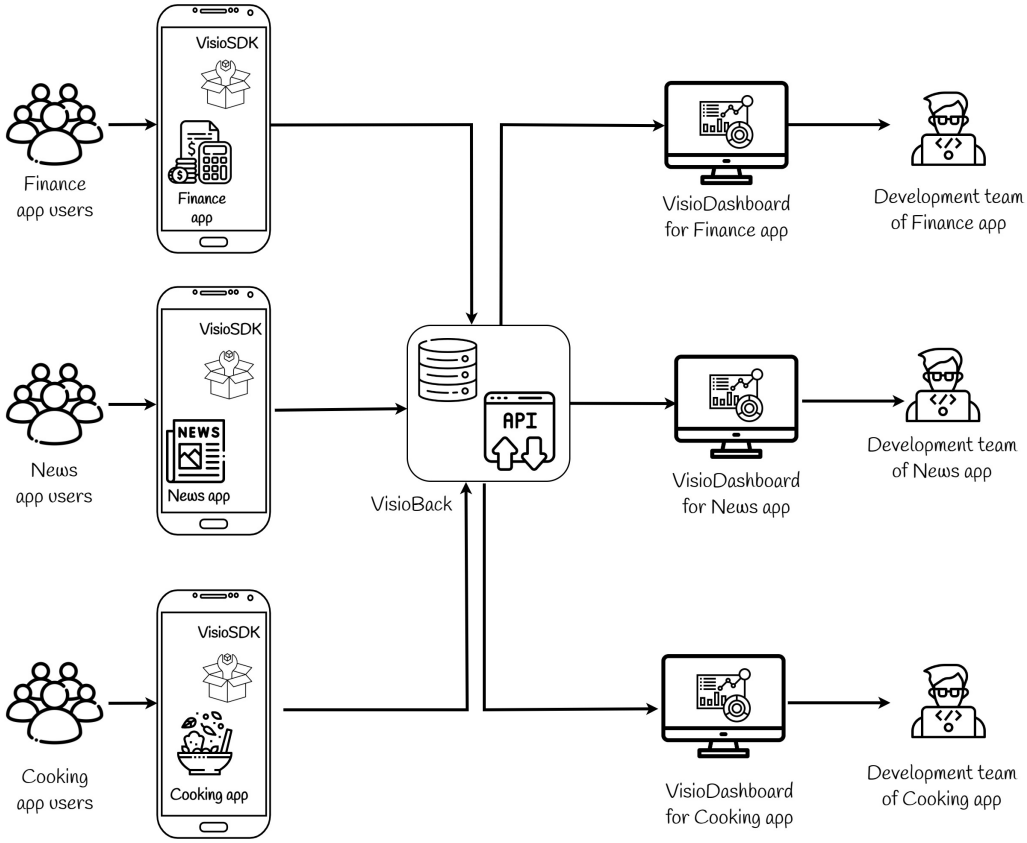

Figure 1. The architecture of *VisioAux* and its modules

3.2.1. VisioSDK

The VisioSDK component plays the role of the testing service discussed in this approach overview. [3] Developers can integrate it into the source code of their mobile application during the development phase. By declaring VisioSDK a dependency, developers enable

[3]The source code repository for VisioSDK can be found at `https://github.com/afbarboza/VisioSdk`.

their mobile applications to automatically run accessibility evaluations in the background. The integration of VisioSDK in mobile applications is straightforward.

In the context of Android development, VisioAux can be added to a project as a dependency via Gradle[4], as presented in Listing 1. Each time a new SDK is added to an Android app, a new version of the app must be released to the public. This ensures that the update reaches end users. This is a common process for mobile developers [Android Developers 2024d]. For example, adding SDKs such as firebase-analytics for tracking user behavior, retrofit2 for making network requests, or glide for loading and displaying images, requires integrating the SDK into the app's source code. When development teams integrate the VisioSDK in a similar way, possibly available in Maven repositories in the future, a new app version must be rolled out. Starting from that version, users will benefit from VisioAux's accessibility inspection features. As a result, it is not necessary to release a duplicated app, with VisioAux integrated, to the general public.

After declaring VisioSDK as a dependency, developers need to provide their own implementation of the `Application` base class and register the `VisioAuxLifecycleTracker` class as the lifecycle callback. This step is exemplified in Listing 2. This step ensures that VisioSDK will be notified when a new screen is shown to the user so it can run accessibility checks. After these two steps, the integration is complete. Notice that this approach is intended to be adopted by developers that want to enable online testing features in their own applications, and not for developers that want to test third party apps.

Listing 1. VisioSDK integration

```
/* ... Declares VisioSDK as a dependency of the app ... */
implementation("com.android.visioaux:visioaux:0.0.1")
```

Listing 2. Registering VisioAux to listen to changes in screens

```
public class AppApplication extends Application {
  public void onCreate() {
        super.onCreate();
        registerActivityLifecycleCallbacks(
                new VisioAuxLifecycleTracker()
        );
    }
}
```

VisioSDK implements an Android Service [5] called `AccessibilityService`, which listens to events called `AccessibilityEvent` (e.g. click, scroll) that might lead to some transition in the current state of the user interface [Android Developers 2024c]. Accordingly, every time an `AccessibilityEvent` is fired by the user, VisioSDK triggers accessibility checks to identify accessibility violations. Through this approach, the coverage of the relevant screens for users is done by the users themselves. As they interact with the application, they become the agents that provide information on which screens and UI states should be evaluated.

[4]Gradle is a build automation tool used to manage various software development tasks, including compiling and testing.

[5]In Android, `Services` enable the execution of long-running operations in the background.

Every triggered `AccessibilityEvent` has its corresponding `AccessibilityNodeInfo`. The `AccessibilityNodeInfo` is a data structure that represents the portion of the screen that triggers the `AccessibilityEvent`. The main role of VisioSDK is to map the AccessibilityNodeInfo to the portion of the View that may contain accessibility violations. Algorithm 1 shows an overview of the implementation of this process. First, it maps the information from the `AccessibilityNodeInfo` provided by the Android `AccessibilityService` (Line 2) to an Android View; next, it runs accessibility checks on that view (Line 3); finally, it sends a report to VisioBack including data on the bug detected and device information, such as manufacturer, Android Version, screen size and density, and so forth.

Algorithm 1 VisioSDK algorithm for checking the accessibility of a screen

```
1: procedure EXPLOREACCESSIBILITYNODE(n: AccessibilityNodeInfo)
2:     view ← GETVIEWDETAILS(n)
3:     accessibilityIssues ← CHECKVIEW(view)
4:     REPORTACCESSIBILITYISSUES(accessibilityIssues)
```

It is also important to consider that screens can change over time without any user interaction. For example, when a popup appears in the app as a result of a network response. To handle such cases and to improve screen coverage, VisioSDK schedules a new accessibility evaluation every 3 seconds. This time interval can be personalized based on developer's preferences.

The accessibility checks of VisioSDK are run by the VisioATF, a modified version of the Google Accessibility Test Framework (ATF) for accessibility evaluation, which are used by tools such as Espresso Accessibility [Android Developers 2024a] and Accessibility Scanner [Giebel 2020]. The Google ATF is an open-source library that performs accessibility checks on Views [Google 2024]. VisioATF modified the Google ATF to avoid throwing exceptions when accessibility violations are found, so the application is not closed, and to include more information on the issues found to allow for the classification of violations based on impact and severity[6]. Table 1 shows the list of checks conducted by the ATF and reported by VisioSdk while the application is in use.

In order to classify the impact that different issues could represent to the user, VisioATF adopted the well-established concept of conformance levels from WCAG [(WAI) 2008]. WCAG defines the three conformance levels A, AA, and AAA. Therefore, each acceptance criterion of the WCAG is assigned a conformance level. Level A is the most basic level of conformance. As dictated by WCAG, to be classified as AA for an application or screen, all the A and AA acceptance criteria must be satisfied. Also, to be classified as AAA for an application or screen, all the A, AA, and AAA acceptance criteria must be satisfied. For example, if an app succeeds in meeting the minimum contrast for texts (level AA) for all its labels but fails at least once to provide one alternative text to non-text content (level A), then the screen is classified as A. As a result of this definition, basic violations of WCAG should be given higher priority to be fixed. Table 2 shows examples of the checks carried out by the VisioATF and also the conformance level assigned to each of them (A, AA or AAA).

[6]The source code repository for VisioATF can be found at https://github.com/afbarboza/VisioATF.

Table 1. List of checks carried out by ATF adopted by VisioATF

Check	Definition
ClassNameCheck	Check if the role of the component is correctly determined.
ClickableSpanCheck	Check to ensure that ClickableSpan is not being used in a TextView
DuplicateClickableBoundsCheck	Checks for touch area overlap
DuplicateSpeakableTextCheck	Verifies if two elements share identical audio descriptions.
EditableContentDescCheck	Check to ensure that an editable text is not labeled by a contentDescription
ImageContrastCheck	Checks the minimum required contrast for images.
LinkPurposeUnclearCheck	Checks if links have unclear purposes.
RedundantDescriptionCheck	Checks for inadequate or redundant audio descriptions
SpeakableTextPresentCheck	Checks for the presence of alternative text where necessary.
TextContrastCheck	Verifies the minimum contrast ratio in text
TextSizeCheck	Checks for text scaling issues.
TouchTargetSizeCheck	Checks the minimum touch target size.
TraversalOrderCheck	Checks issues related to incorrect navigation by assistive technologies.
UnexposedTextCheck	Checks for text not visible to assistive technologies.

Table 2. Examples of checks executed by VisioATF and WCAG conformance level

Check name	Conformance level	Violated WCAG criterion
ImageConstrastCheck	AA	1.4.11
TouchTargetSizeCheck	AAA	2.5.5
UnexposedTextCheck	A	1.1.1
SpeakableTextPresentCheck	A	2.5.3

3.2.2. VisioAux and privacy concerns

Whenever a mobile app requires access to restricted data or functionality (e.g., location, list of contacts, or the camera) it requests the appropriate permission from the Android operating system, which must then be granted by the user [Android Developers 2024f]. This process helps ensure user privacy. Since VisioSDK functions as an accessibility service, the user must explicitly grant accessibility permission for it to perform its tasks [Android Developers 2024b]. This behaves similarly to other accessibility services, such as Accessibility Scanner or TalkBack. Figure 2 illustrates how users can enable or disable VisioAux: users must activate VisioAux Service and allow it to access internal information. The displayed message is the default message set by the Android system

and cannot be changed.

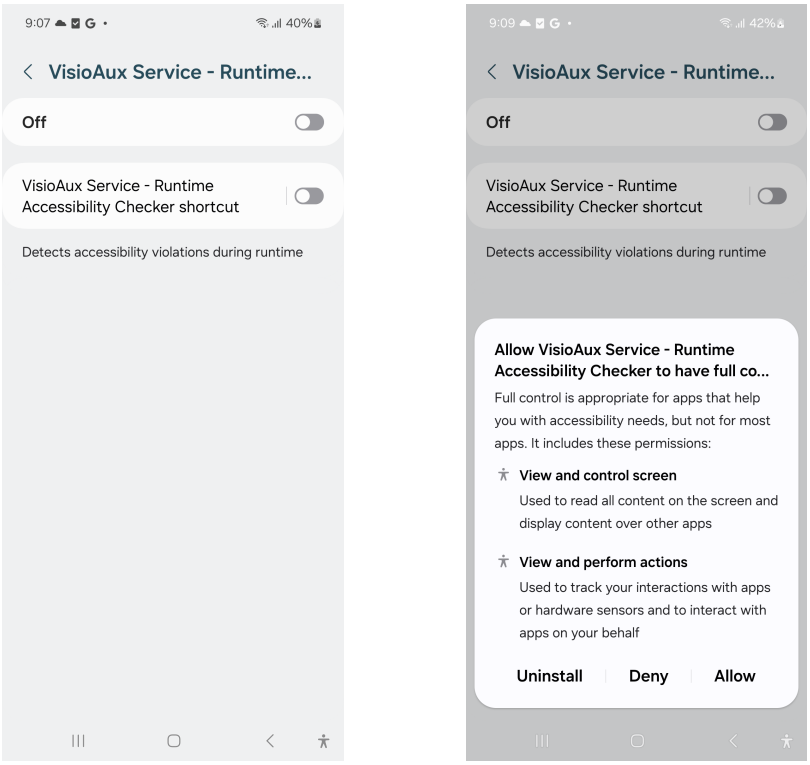

Figure 2. Android settings screen where users can enable or disable VisioAux.

It is also worth mentioning that VisioSDK operates on a per-app basis. If the developers of the Finance app shown in Figure 1 want to integrate VisioAux analysis, VisioSDK will not inspect the accessibility of the News app, for instance. The analysis of the News app is possible if, and only if, its developers integrate VisioSDK into the app's source code. This behavior is enforced by the Android kernel, which isolates processes and prevents one app from spying on another [Tanenbaum 2009].

3.2.3. VisioBack

VisioBack is a backend application implemented in Java using the Spring Boot framework [7]. It exposes a REST API to VisioSdk, enabling VisioSdk to communicate to a remote backend - VisioBack - about errors encountered while the app was being used. VisioBack [8] is capable of receiving these multiple reports and persisting them in a relational database. In this way, it becomes possible not only to report errors, but also to measure which accessibility errors have been most recurrent, or how many times a particular accessibility error has manifested itself to the end user. How VisioBack is capable of distinguishing these multiple errors is explained in further detail in the next paragraphs.

[7]https://spring.io/why-spring
[8]The source code repository for VisioBack can be found at https://github.com/afbarboza/VisioBack.

Whenever VisioBack receives a new accessibility error report, it queries the relational database to check whether that error has been reported before or whether it is a new accessibility error. To compare whether a reported error is equivalent to an error already stored in the database for a particular app, VisioBack applies the algorithm illustrated in Listing 2. As shown in Listing 2, VisioAux considers two errors to be equivalent if they are violating the same WCAG criteria (e.g., low contrast below the recommended), can be found on the same screen, and also on the same widget.

Algorithm 2 Algorithm to check for equivalent accessibility errors

Input: a duple of accessibility errors reported by VisioSdk, E_1 e E_2
Output: **True** if, and only if, $E_1 == E_2$. **False**, otherwise.

```
if E1.type ≠ E2.type then
    return false
else if E1.screen ≠ E2.screen then
    return false
else if E1.widget ≠ E2.widget then
    return false
else
    return true
```

For example, consider E_1 as a new small touch area problem on the home screen reported by VisioSdk. Also consider E_2 as a small touch area problem on the home screen, with E_2 previously recorded in the database. If E_1 occurs on the forward button, while E_2 occurs on the back button, then, according to the algorithm in Listing 2 these errors are not equivalent. In this scenario, VisioBack creates a new record in the database.

Now, if E_1 and E_2 are found to be equivalent, VisioBack has a different approach. In this case, VisioBack increments the occurrence counter for the given accessibility issue, instead of creating a new database record. In addition, as VisioBack keeps track of the uniquely identified devices, it becomes possible to count how many times a particular user has faced a particular accessibility issue.

Next, for the objectives of this research to be effectively achieved, it is essential that VisioAux is able to classify the violations found in a given application by the severity they represent to end users - especially those with disabilities. How this classification is done and how VisioAux presents a consolidated report to the development teams is the responsibility of VisioAux's third and final module, the VisioDashboard.

3.2.4. VisioDashboard

VisioDashboard is a web page implemented using Thymeleaf [9]. Whenever the development team wants to check the accessibility issues found in their app as reported by many devices running VisioSDK, they can resort to VisioDashboard for this purpose [10]. Figure 3 illustrates how VisioDashboard presents some of the accessibility analysis for

[9] https://www.thymeleaf.org/

[10] VisioDashboard shares its source code repository with VisioBack, available at https://github.com/afbarboza/VisioBack.

developers. Each row represents an accessibility issue found by VisioATF and reported by VisioSDK.

The first column, severity, shows the severity value σ calculated for that accessibility problem. The second column shows the category of the accessibility problem (e.g., small touch area). The third column is a message to developers about how the problem can be addressed. The message clearly identifies the widget with the reported accessibility issue and provides straightforward guidance on how to fix it in the next app release. The table is sorted by the severity σ in descending order. In future implementations, VisioDashboard will also implement different visualizations according to the development team requirements for accessibility reports and also provide translation support for other languages, such as Brazilian Portuguese.

Severity	Type	Message
15.0	CONTENT_LABELING	View add_button: This item may not have a label readable by screen readers.
15.0	CONTENT_LABELING	View subtract_button: This item may not have a label readable by screen readers.
7.5	LOW_CONTRAST	View countTV: The item's text contrast ratio is 2.46. This ratio is based on a text color of #999999 and background color of #EEEEEE. Consider increasing this item's text contrast ratio to 3.00 or greater.
5.0	TOUCH_TARGET_SIZE	View subtract_button: This item's size is 10dp x 10dp. Consider making this touch target 48dp wide and 48dp high or larger.
5.0	TOUCH_TARGET_SIZE	View add_button: This item's size is 10dp x 10dp. Consider making this touch target 48dp wide and 48dp high or larger.
1.33	TOUCH_TARGET_SIZE	View material_switch: This item's size is 47dp x 27dp. Consider making this touch target 48dp wide and 48dp high or larger.
1.0	TOUCH_TARGET_SIZE	View material_switch: This item's size is 46dp x 27dp. Consider making this touch target 48dp wide and 48dp high or larger.

Figure 3. Issues reported by VisioDashboard

The severity metric (σ) presented in the first column of Figure 3 is calculated by the Equation (1), where N is the number of users affected by that particular issue. Value if I is calculated by Equation (2), where ι_i is the amount of times the i-th user found the issue.

$$\sigma = \frac{N \times I}{\alpha} \tag{1}$$

$$I = \sum_{i=1}^{N} \iota_i \tag{2}$$

The value of α is associated with the WCAG conformance level, and it is mathematically defined as follows.

(3)

The reasoning behind severity estimation. Classifying accessibility problems may not be as trivial as it appears to be. To discuss this aspect, consider a hypothetical food recipe application for mobile phones. In a certain version, the development team chooses to integrate VisioSdk's accessibility analysis. This application contains multiple screens and, unfortunately, two distinct accessibility errors that have not yet been corrected. The first error, E_1, concerns the resizing of the text. Users with low vision, willing to scale the font size to be able to see the description of a recipe in larger font size, are faced with text that cannot be resized. This error is a violation of the success criterion 1.4.4 of WCAG 2.2, which has a compliance level of AA. At the same time, on another screen that is less frequently accessed, there is a picture and nutritional information about an ingredient. The ingredient picture, which contains important information about allergies, does not have an alternative text that describes all the information presented in the image. This error, hereinafter called E_2, is a violation of WCAG 2.2 criterion 1.1.1 and has a compliance level of A.

The number of users N who have encountered problem E_1 while trying (unsuccessfully) to enlarge the font is the first variable taken into account by VisioAux. The premise of this work is that the more users who are affected by E_1, the higher the priority should be for correcting it. In this example, consider that $N_{E1} = 10$ and $N_{E2} = 5$. If only N was considered, since $N_{E1} > N_{E2}$, E_1 should be fixed first.

However, the number of affected users may not be the only variable of interest. The number of times that the same user encounters E_1 is also relevant. In this example, consider that 10 users encountered the error E_1, with each user encountering the problem 1 time. This measure, here called incidence and denoted by I, would be $I_{E1} = 10$, since $10 \times 1 = 10$. In contrast, for error E_2, each of the five users ($N_{E2} = 5$) encountered the absence of alternative text three times, totaling 15 incidents of E_2, since $I_{E2} = 5 \times 3 = 15$. Incidence I is then defined as the sum of times all users encounter an error.

If only the number of users N is considered, then E_1 might have a higher severity. If only the incidence I is considered, then E_2 would have a higher severity. At this point, it becomes clear that both variables should be taken as indicators of the severity of an accessibility error. This is precisely what VisioAux does by taking the severity as being proportional to the number N of users affected and the incidence I of an error.

In the example mentioned above, if only N and I were considered in Equation (1), the severity of error E_1 (σ_{E1}) would be 100, since N = 10 and I = 10 (10×10). The severity of error E_2 (σ_{E2}) would be 75, since N = 5 and $I = 15$ (5×15). Since $\sigma_{E1} > \sigma_{E2}$, the error E_1 would naturally represent a higher severity. Since the conformance level should matter, a third variable was introduced as α in Equation (1), which represents the conformance level of an accessibility problem. In the provided example, α_{E1} is equal to 2, since the error E_1 violates a principle with conformance level AA. On the other hand, α_{E2} is equal to 1, since the error E_2 violates a WCAG principle with conformance level A. Following Equation (1), the severity of error E_1 (σ_{E1}) is 50 and the severity of error E_2 (σ_{E2}) is 75, which means E_2 would have priority to be fixed.

VisioDash extension. Considering that different development teams have different perspectives on bug severity, especially when contextual information regarding features

are taken into account, future implementations of VisioDash will allow developers to personalize the estimation of the severity of each accessibility violation found using information gathered by VisioSDK within mobile devices and any other information added by the development team.

4. Preliminary evaluation and emerging results

At its current stage, three studies were conducted to evaluate the feasibility and the applicability of VisioAux in a real environment. The first study was conducted to understand how effective VisioSDK is in detecting accessibility issues in mobile apps. The second study was conducted to understand the impact of VisioSDK on mobile app performance. The third study was conducted to investigate the capabilities of VisioAux in gathering accessibility reports originating from various devices. The three studies are described as follows.

4.1. Study 1 - Accessibility issues detection capabilities

The first concern regarding an accessibility testing tool is whether it is capable of detecting accessibility issues. Even though VisioAux was built upon the Google Accessibility Testing Framework, which has been thoroughly tested by many well-known approaches and tools, we proposed the following research question to perform a preliminary evaluation of VisioAux:

> **RQ:** To what extent is VisioAux capable of detecting and reporting accessibility violations in mobile applications?

We executed the following steps to answer this research question. First, we randomly selected ten applications of different categories from the F-Droid repository, with a varying number of downloads to enable the generality of our results to various domains. Table 3 shows the list of apps selected in this step. Each app was modified to include the VisioAux dependency and the required changes were implemented (cf. Listings 1 and 2). Next, we defined three different tasks for each application, which correspond to possible user interactions. Considering the need to reproduce such tasks many times, we created test scripts to automate their execution in mobile settings [11].

Following, we established a baseline of comparison to evaluate the capability of VisioAux to identify accessibility issues. Accordingly, we decided to compare the results obtained with VisioAux with results obtained with Google Accessibility Scanner, which is frequently used in experiments related to automated accessibility evaluation for mobile applications [Zhang et al. 2023a, Alotaibi et al. 2023, Salehnamadi et al. 2023, Alotaibi et al. 2022, Salehnamadi et al. 2022, Chen et al. 2021b, Alotaibi et al. 2021, Salehnamadi et al. 2021]. Finally, we executed the three tasks of the ten apps (30 executions) on two specific settings: i) the original app with accessibility issues being detected by Accessibility Scanner; and ii) the app version with the VisioSDK integrated with accessibility violations being detected by VisioSDK and VisioATF. All measures were taken to make sure the intended version was running on the mobile device, such as uninstalling the previous version and rebooting the device.

[11] The source code repository for VisioExperiments, which contains the test scripts, is available at https://github.com/afbarboza/VisioExperiment.

Table 3. Selected apps for VisioAux experiments.

App Name	Category	Number of Downloads
Catima	Finance	5000
Decisions	Sports & Health	5000
Simple Draw	Graphics	100000
PaperWallet	Finance	10000
PxerStudio	Graphics	100
Aegis	Security	100000
AVNC	Connectivity	1000
Hauk	Navigation	500
Shaarlier	Internet	1000
Fosdem	Events	10000

Table 4 shows the comparison between the results obtained with Accessibility Scanner and VisioAux. In total, 332 accessibility violations were reported combining all results, but only 249 issues were found by both tools. Some violations were found ony by Accessibility Scanner (38) and some only by VisioAux (45). For some apps, VisioAux performed better. For example, for the Catima application, a total of 22 violations were reported: 17 were detected by both VisioAux and Accessibility Scanner, but VisioAux reported 5 problems that were not identified by Accessibility Scanner and no issues were detected only by Accessibility Scanner. On the other hand, considering the Decisions app, a total of 12 accessibility problems were reported: 6 issues were found by both VisioAux and Accessibility Scanner; one accessibility problem was found only by VisioAux; and five accessibility problems were found only with Accessibility Scanner.

Table 4. Comparison between the accessibility issues detected by Accessibility Scanner (A.S) and VisioAux (V.A.)

App	Both	Only A.S.	Only V.A.	Total
Catima	16	0	8	24
Decisions	7	8	0	15
Simple Draw	43	1	6	50
PaperWallet	56	13	1	70
PxerStudio	31	0	2	33
Aegis	18	2	2	22
AVNC	10	6	1	17
Hauk	37	4	1	42
Shaarlier	25	4	0	29
Fosdem	6	0	24	30
Total	**249**	**38**	**45**	**332**

All accessibility issues detected by VisioSDK while the apps were being executed were reported to VisioBack. In addition, VisioDashboard successfully computed the severity score σ for each issue, correctly determining whether an issue was new or had already occurred multiple times. Also, it sorted all the issues in descending order based on the defined severity metric σ. Our findings show the most recurring problems found

in this experiment are mostly related to text resizing, color contrast (in images and text), and alternative text. Interestingly, similar results can be observed in previous studies [Yan e Ramachandran 2019b, Chen et al. 2021b].

4.2. Study 2 - Performance

In operating systems for mobile devices, the thread model defines a main thread, which is often called the UI Thread, as responsible for handling user interactions, such as button clicks or scroll operations. It is also the thread responsible for rendering the graphical interface elements [Android Developers 2024g, Apple Inc. 2024]. If the UI thread is taking too much time executing long-running operations, such as waiting for the response of network requests, the user experience could easily be degraded by a non-responsive app [Android Developers 2024e]. Considering that VisioSdk detects changes in the screen state and, together with VisioATF, identifies accessibility issues in the graphical interface, it is only natural to question whether VisioAux could compromise user experience, making the application unresponsive or even unusable. Thus, it led to the following research question:

> **RQ:** Can VisioAux compromise the user experience by affecting the performance of monitored mobile applications?

To objectively answer this question, we designed an experiment to compare the performance of the ten mobile applications with and without the VisioSDK integrated. First, we wrote test scripts to automate the execution of the 30 scenarios mentioned in the first study (three scenarios for each app). Next, we ran the test scripts to execute each scenario considering both the mobile application with and without the VisioSDK integrated. For each of the 30 tasks executed, the execution time was recorded as $\delta = t - t_0$, where t represents the final time instant of the task, and t_0 represents the initial time instant of the task. Therefore, δ represents the amount of time each task took to execute. To mitigate the effects of external variables (e.g. caching), each of the 30 tasks was repeated three times and we calculated the average of all executions for each task.

Since we are not interested in the task execution time itself, but rather in evaluating whether VisioAux could degrade performance, our measure of interest is given by the difference between the execution times of the tasks with and without VisioAux. Assuming that VisioAux would take more time, we define this difference as $\Delta = \delta_{VA} - \delta$. Table 5 shows the average execution time for each test scenario for both versions of the app (Original or with VisioSDK). Notice that no Δ (overhead) value exceeds 2.5 seconds for the whole test scenario, which means that no overhead was observed beyond even half the time required for an Android application to become unresponsive at a given specific task, i.e., 5 seconds [Android Developers 2024e]. In some cases, the app version with VisioSDK integrated ran faster than the original version. This does not mean that VisioSDK makes the app more efficient; rather, it shows that other factors such as networking and Android services may have a more significant impact than the overhead introduced by the accessibility checks.

4.3. Study 3 - Crowdsourcing Capabilities of VisioAux

The first premise of VisioAux is to enable a single device to report accessibility issues across multiple screens. The second is to allow the aggregation of reports from different

Table 5. Average execution time (in ms) for each task of each app, considering both the original app and the version with the VisioSDK integrated.

App-Task	Original	VisioSDK	Δ	App-Task	Original	VisioSDK	Δ
1-1	17949	16858	-1091	6-1	13577	13952	375
1-2	31891	29487	-2404	6-2	7523	7577	54
1-3	17455	15659	-1796	6-3	7461	7560	99
2-1	24984	24716	-268	7-1	9636	9591	-45
2-2	32344	31976	-368	7-2	10299	10220	-79
2-3	16296	16195	-101	7-3	7666	7595	-71
3-1	36101	36272	171	8-1	3975	3926	-49
3-2	23890	23974	84	8-2	13618	13659	41
3-3	13711	13570	-141	8-3	15394	15631	237
4-1	6754	6752	-2	9-1	12275	12133	-142
4-2	13412	13576	164	9-2	19732	19728	-4
4-3	20763	20702	-61	9-3	33255	33068	-187
5-1	9892	9819	-73	10-1	14043	14116	73
5-2	16769	17070	301	10-2	11499	11220	-279
5-3	7526	7476	-50	10-3	11345	11587	242

devices, facilitating the classification of issues for development teams and achieving higher coverage given the distinct features and screens visited by multiple users. Based on these premises, we pose the following research question:

> **RQ:** To what extent is VisioAux capable of covering different screens and aggregating accessibility reports from multiple devices?

To assess VisioAux's potential in this regard, we selected Fosdem[12], an event organization app with over ten thousand downloads in the Google Play Store. We defined three main usage flows representing three types of user interactions as illustrated by Figure 4: a) *Flow 1* simulates a user exploring the details of a talk within a specific topic (three screens); b) *Flow 2* represents a user searching for information about a talk (one screen); c) *Flow 3* involves a user navigating through the app's settings screen (one screen). In total, six distinct devices were used: three for Flow 1, two for Flow 2, and one for Flow 3, with no device reused across flows. Each flow was manually explored by the authors of this study. To adequately address the research question, VisioSDK generates a unique identifier for each device based on UUID and timestamp, without collecting user or device data, thus ensuring anonymity. Additionally, VisioSDK also logs the screen name whenever an accessibility issue is detected.

Accordingly, we queried the VisioBack relational database to obtain the number of detected devices, the number of identified screens, and the number of devices per screen.According to this preliminary evaluation, VisioBack successfully identified six different smartphones accessing the application; detected that a total of six distinct screens were explored by combining the results of the distinct devices; and determined how many devices explored each screen (i.e., three smartphones accessed the first flow). As a

[12]https://play.google.com/store/apps/details?id=be.digitalia.fosdem

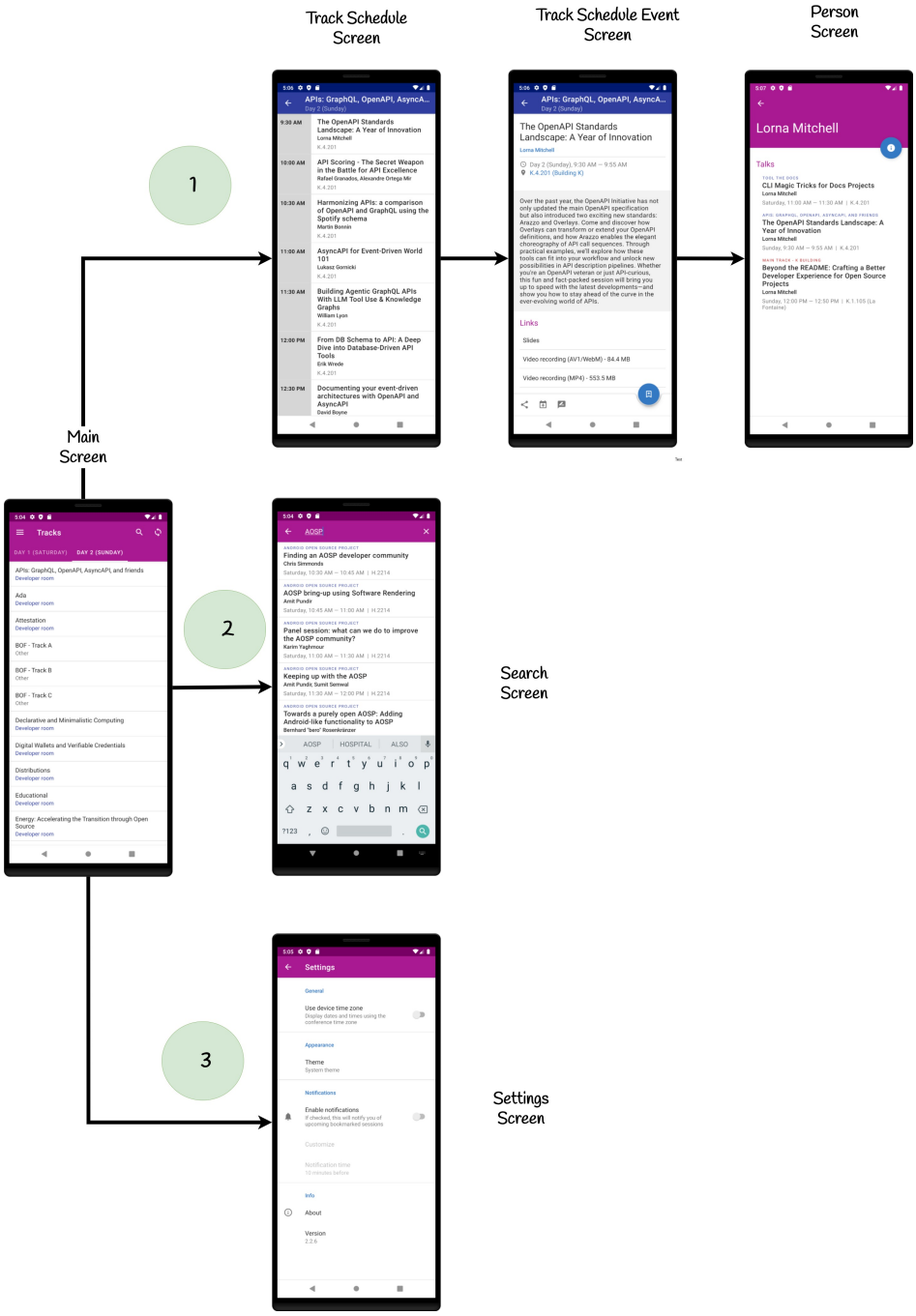

Figure 4. The three flows defined for the app Fosdem

preliminary evaluation, our findings suggest that VisioAux is mature enough to undergo future and further evaluation with real users and real testing scenarios.

5. Discussion

Our crowdsourcing approach to accessibility software testing demonstrates significant promise, as evidenced by three preliminary experimental findings. First, when compared against an established tool like Accessibility Scanner, VisioAux detected a similar set of accessibility issues and sometimes identified violations that were not even detected by

Accessibility Scanner. With future implementation of actionable tests for more testable accessibility criteria defined by accessibility guidelines such as WCAG, we believe our approach has the potential to be adopted in real environments.

Second, the performance overhead measurements revealed that VisioSDK operates efficiently alongside the application under evaluation, imposing minimal or even no resource demands. This technical efficiency is critical for real-world adoption, as developers often resist solutions that degrade user experience or require extensive instrumentation. Our findings show that accessibility checks run smoothly, in a way that applications under evaluation will not have their performance compromised, disrupting user experience.

Third, VisioAux, specifically the VisioSDK component, was capable of collecting accessibility issues across different screens. Another component, VisioBack, was responsible for gathering and centralizing various accessibility reports from multiple devices into a single database, enabling a comprehensive analysis of the app's accessibility for different user profiles. Based on the results obtained, VisioAux addresses the identified gaps in the state of the art by showing promising results in enabling the monitoring of mobile applications through crowdsourcing.

We believe that one of the reasons that no or little overhead is introduced during the process is the fact that the accessibility checks are run within the mobile application, accessing information as internal views. Such an approach is quite different from when external tools need to access information from mobile apps under testing using only Android Services and sometimes having to take screenshots and run image processing algorithms to calculate contrast ratio, widget size, and so forth. We also leverage Android's asynchronous programming capabilities, particularly Kotlin Coroutines, to ensure that the UI thread remains responsive. [Google 2024].

Compared to related work, our approach differs from existing automated methods in many ways. Firstly, it doesn't use test scripts or algorithms; instead, users navigate the app, and accessibility checks are conducted by the testing service deployed at users' device. Secondly, our method achieves better feature and screen coverage by leveraging multiple users' explorations rather than scripts or a few experts. Lastly, it tracks screen visits and violations to prioritize fixing accessibility bugs. It is important to emphasize that our approach was designed to complement existing approaches, including manual evaluation, and adds new testing alternatives such as monitoring app accessibility during runtime.

Scope and limitations. We understand that our findings are limited because they are based on preliminary studies. Nevertheless, they sufficiently justify additional development and assessment of our methodology. This would lead to more robust conclusions regarding its applicability, ultimately paving the way for more intricate experiments in real-world environments with both end users and development teams.

Other mobile platforms, such as iOS, were not considered in this research. Extending VisioAux to mobile or web counterparts would require additional highly skilled personnel with expertise in those platforms. However, by establishing the scientific foundations of VisioAux, we believe that developing analogous approaches becomes

primarily an engineering effort.

It is also worth mentioning that the WCAG encompasses a broad spectrum of accessibility success criteria. While some of these criteria can be evaluated algorithmically, a considerable number require human judgment for proper assessment [Alonso et al. 2010]. This work does not aim to address those criteria that can only be evaluated through human testing, such as the presence of adequate captions or subtitles in media, which are essential for deaf users.

6. Conclusion and future directions

In this paper, we proposed an innovative way to leverage crowdsourcing concepts to perform accessibility testing of mobile applications. Differently from traditional crowdsource testing approaches, in which the crowd's expertise in software testing is used to thoroughly assess a software application, VisioAux does not require any expertise from participants. Rather, they act as mobile app explorers who will expose different interface configurations and screens to the VisioSDK, which in turn will run accessibility checks.

By proposing crowdsourcing as a way to evaluate and monitor accessibility issues, we aim to improve the current landscape in digital accessibility evaluation for mobile devices. This approach isn't about replacing existing pre-release testing methods, such as the necessary manual testing by users with disabilities and specialists, but rather serving as a complement to them, offering development teams additional insights. Our objective is to elevate accessibility to the same level of importance as other software quality attributes, such as security and performance, through the implementation of similar monitoring strategies. Moreover, our work underscores how crowdsourcing can democratize accessibility testing, making it more scalable and inclusive without compromising on technical rigor or user experience.

Our approach differs from traditional accessibility monitoring in two main aspects. In traditional accessibility monitoring, an automated tool periodically runs accessibility checks as a third-party entity on the monitored application to identify accessibility issues. The limitation of this approach is that it is based on automated tools, which in general provide limited coverage, as discussed before. Our approach relies on the end user as the agent that explores the application as they use it in daily activities. In addition, our approach is based on a testing service that runs along with the mobile application being monitored, which means it has access to the internal structure of the app, without the inherent limitations of mobile tools that access the evaluated apps as third-party entities.

Findings from our preliminary studies motivate further development and investigation as the capabilities of detecting accessibility issues for our tool are comparable to well-established tools such as Accessibility Scanner, and no significant overhead is introduced in the process when it comes to the regular usage of the mobile app being continuously assessed. Considering this scenario, many future works are necessary to consolidate our results and deliver a more robust framework that can be adopted in a real-world environment.

The first study we intend to conduct as future work is a large-scale experiment in which a mobile app running with VisioSDK integrated is deployed on multiple devices and used by real end users for a certain period of time. The results obtained concerning

accessibility issues detected and feature/screen coverage will be compared to results obtained by the state-of-the-art approaches that automatically evaluate the accessibility of mobile applications.

The second study concerns the assessment of the relevance of the metrics collected from the devices and applications under evaluation for determining the severity and the impact of the accessibility violations detected. Such investigation will be conducted with accessibility development teams that have large experience in reporting, prioritizing, and fixing accessibility bugs. Therefore, this evaluation will be useful to evaluate the proposed metric for severity and impact and the flexibility of VisioDashboard in allowing development teams to personalize severity metrics calculations.

With this approach, we envision a scenario in which accessibility testing services will be natively embedded in future versions of operating systems such as Android to allow for checking the accessibility of all mobile apps, reporting back to developers a list of accessibility violations, thus allowing development teams to continuously improve mobile accessibility by addressing accessibility bug reports received according to established prioritization metrics and strategies they define.

Ethical concerns

This study did not involve experiments with human subjects. All testing scenarios in our first two experiments were defined and automated by the authors of this paper. The third study was conducted by the authors using their own devices to simulate actions of three distinct users. Considering future endeavors, we emphasize that VisioAux does not collect any personal information from the user; only detected accessibility issues are sent to VisioBack. Additionally, VisioSDK operates only with the user's explicit consent, a requirement enforced by the Android operating system through permission settings.

Acknowledgments

ChatGPT and DeepL were used to assist with the translation of some text from Portuguese to English, in addition to the grammatical revision of the English texts written by the authors of this paper.

References

Aashutosh, K. (2023). *Mobile Accessibility Rituals: WCAG 2.1 Made Easy for Mobile Apps*. Amazon Italia.

Alonso, F., Fuertes, J. L., González, A. L., e Martínez, L. (2010). On the testability of wcag 2.0 for beginners. In *Proceedings of the 2010 International Cross Disciplinary Conference on Web Accessibility (W4A)*, W4A '10, New York, NY, USA. Association for Computing Machinery.

Alotaibi, A. S., Chiou, P. T., e Halfond, W. G. (2021). Automated repair of size-based inaccessibility issues in mobile applications. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 730–742. IEEE.

Alotaibi, A. S., Chiou, P. T., e Halfond, W. G. (2022). Automated detection of talkback interactive accessibility failures in android applications. In *2022 IEEE Conference on Software Testing, Verification and Validation (ICST)*, pages 232–243. IEEE.

Alotaibi, A. S., Chiou, P. T., Tawsif, F. M., e Halfond, W. G. (2023). Scalefix: An automated repair of ui scaling accessibility issues in android applications. In *2023 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 147–159. IEEE.

Alshayban, A., Ahmed, I., e Malek, S. (2020). Accessibility issues in android apps: state of affairs, sentiments, and ways forward. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pages 1323–1334.

Alshayban, A. e Malek, S. (2022). Accessitext: automated detection of text accessibility issues in android apps. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 984–995.

Alyahya, S. (2020). Crowdsourced software testing: A systematic literature review. *Information and Software Technology*, 127:106363.

Android Developers (2024a). Accessibility checks with espresso. `https://developer.android.com/training/testing/espresso/accessibility-checking`. Accessed: 2025-04-09.

Android Developers (2024b). Accessibility service - manifest declaration. `https://developer.android.com/guide/topics/ui/accessibility/service#manifest`. Accessed: 2025-08-04.

Android Developers (2024c). Accessibilityservice. `https://developer.android.com/reference/android/accessibilityservice/AccessibilityService`. Accessed: 2025-04-14.

Android Developers (2024d). Add build dependencies. `https://developer.android.com/build/dependencies`. Accessed: 2025-08-05.

Android Developers (2024e). App not responding (ANR). `https://developer.android.com/topic/performance/vitals/anr`. Accessed: 2025-05-10.

Android Developers (2024f). Permissions overview. `https://developer.android.com/guide/topics/permissions/overview`. Accessed: 2025-08-04.

Android Developers (2024g). Processes and Threads. `https://developer.android.com/guide/components/processes-and-threads`. Accessed: 2025-05-10.

Apple Inc. (2024). UIView | Threading Considerations. `https://developer.apple.com/documentation/uikit/uiview#Threading-considerations`. Accessed: 2025-05-10.

BBC (n.d.). Accessibility - mobile accessibility guidelines. `https://www.bbc.co.uk/accessibility/forproducts/guides/mobile/`. Accessed: 2025-04-05.

Bi, T., Xia, X., Lo, D., Grundy, J., Zimmermann, T., e Ford, D. (2022). Accessibility in software practice: A practitioner's perspective. *ACM Transactions on Software Engineering and Methodology*.

Cerwall, P. (2021). Ericsson mobility report. https://www.ericsson.com/4a03c2/assets/local/reports-papers/mobility-report/documents/2021/june-2021-ericsson-mobility-report.pdf. Accessed: 2025-08-19.

Chen, S., Chen, C., Fan, L., Fan, M., Zhan, X., e Liu, Y. (2021a). Accessible or not an empirical investigation of android app accessibility. *IEEE Transactions on Software Engineering*.

Chen, S., Chen, C., Fan, L., Fan, M., Zhan, X., e Liu, Y. (2021b). Accessible or not? an empirical investigation of android app accessibility. *IEEE Transactions on Software Engineering*, 48(10):3954–3968.

Da Costa Nunes, E. H., Castro, K. V. G. d., Anjos, E. L. d. F. d., Silva, P. V. d. S., e Monteiro, I. T. (2024). Color contrast compliance: Investigating constrast requirements in brazilian websites. In *Proceedings of the XXIII Brazilian Symposium on Human Factors in Computing Systems*, IHC '24, New York, NY, USA. Association for Computing Machinery.

Dias, J., Carvalho, D., Paredes, H., Martins, P., Rocha, T., e Barroso, J. (2021). Automated evaluation tools for web and mobile accessibility: A systematic literature review. In *International Conference on Innovations in Bio-Inspired Computing and Applications*, pages 447–456. Springer.

Eler, M. M., Rojas, J. M., Ge, Y., e Fraser, G. (2018a). Automated accessibility testing of mobile apps. In *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*, pages 116–126. IEEE.

Eler, M. M., Rojas, J. M., Ge, Y., e Fraser, G. (2018b). Automated accessibility testing of mobile apps. In *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*, pages 116–126. IEEE.

Firebase (2025). Firebase Crashlytics Documentation. https://firebase.google.com/docs/crashlytics. Accessed: 2025-04-11.

Fok, R., Zhong, M., Ross, A. S., Fogarty, J., e Wobbrock, J. O. (2022). A large-scale longitudinal analysis of missing label accessibility failures in android apps. In *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems*, pages 1–16.

Giebel, J. (2020). Accessibility evaluation tools for android mobile applications. *Tagungsband*, page 42.

Google (2024). Accessibility test framework for android. https://github.com/google/Accessibility-Test-Framework-for-Android. Accessed: 2025-04-14.

Google (2024a). Espresso. https://developer.android.com/training/testing/espresso. Accessed: 2025-04-11.

Google (2024b). Improve your code with lint checks. https://developer.android.com/studio/write/lint. Accessed: 02-jun-2024.

Google (2024). Kotlin coroutines on android. https://developer.android.com/kotlin/coroutines. Accessed: 2025-08-03.

Kochhar, P. S., Thung, F., Nagappan, N., Zimmermann, T., e Lo, D. (2015). Understanding the test automation culture of app developers. In *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*, pages 1–10. IEEE.

Lam, W., Wu, Z., Li, D., Wang, W., Zheng, H., Luo, H., Yan, P., Deng, Y., e Xie, T. (2017). Record and replay for android: Are we there yet in industrial cases? In *Proceedings of the 2017 11th joint meeting on foundations of software engineering*, pages 854–859.

Leite, M. V. R., Scatalon, L. P., Freire, A. P., e Eler, M. M. (2021). Accessibility in the mobile development industry in brazil: Awareness, knowledge, adoption, motivations and barriers. *Journal of Systems and Software*, 177:110942.

Lewthwaite, S., Horton, S., e Coverdale, A. (2023). Workplace approaches to teaching digital accessibility: establishing a common foundation of awareness and understanding. *Frontiers in Computer Science*, 5.

Martin, L., Baker, C., Shinohara, K., e Elglaly, Y. N. (2022). The landscape of accessibility skill set in the software industry positions. In *Proceedings of the 24th International ACM SIGACCESS Conference on Computers and Accessibility*, pages 1–4.

Mateus, D. A., Silva, C. A., Eler, M. M., e Freire, A. P. (2020). Accessibility of mobile applications: evaluation by users with visual impairment and by automated tools. In *Proceedings of the 19th Brazilian Symposium on Human Factors in Computing Systems*, pages 1–10.

Matos, M., Seixas Pereira, L., e Duarte, C. (2023). Evaluation of the accessibility of mobile applications: Current approaches and challenges. In *International Conference on Human-Computer Interaction*, pages 352–371. Springer.

Mehralian, F., Salehnamadi, N., e Malek, S. (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*, pages 107–118.

Patel, R., Breton, P., Baker, C. M., El-Glaly, Y. N., e Shinohara, K. (2020). Why software is not accessible: Technology professionals' perspectives and challenges. In *Extended abstracts of the 2020 CHI conference on human factors in computing systems*, pages 1–9.

Robolectric (2024). Robolectric: Unit testing framework for Android. `https://robolectric.org/`. Accessed: 2025-04-11.

Salehnamadi, N., Alshayban, A., Lin, J.-W., Ahmed, I., Branham, S., e Malek, S. (2021). 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*, pages 1–11.

Salehnamadi, N., He, Z., e Malek, S. (2023). Assistive-technology aided manual accessibility testing in mobile apps, powered by record-and-replay. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*, pages 1–20.

Salehnamadi, N., Mehralian, F., e Malek, S. (2022). Groundhog: An automated accessibility crawler for mobile apps. in 2022 37th ieee. In *ACM International Conference on Automated Software Engineering. IEEE, ACM New York, NY, USA, Rochester, Michigan, USA*.

Silva, C., Eler, M. M., e Fraser, G. (2018a). A survey on the tool support for the automatic evaluation of mobile accessibility. In *Proceedings of the 8th International Conference on Software Development and Technologies for Enhancing Accessibility and Fighting Info-exclusion*, pages 286–293.

Silva, C., Eler, M. M., e Fraser, G. (2018b). A survey on the tool support for the automatic evaluation of mobile accessibility. In *Proceedings of the 8th International Conference on Software Development and Technologies for Enhancing Accessibility and Fighting Info-exclusion*, pages 286–293.

Song, S., Bu, J., Artmeier, A., Shi, K., Wang, Y., Yu, Z., e Wang, C. (2018). Crowdsourcing-based web accessibility evaluation with golden maximum likelihood inference. *Proc. ACM Hum.-Comput. Interact.*, 2(CSCW).

Stol, K.-J. e Fitzgerald, B. (2014). Two's company, three's a crowd: a case study of crowdsourcing software development. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, page 187–198, New York, NY, USA. Association for Computing Machinery.

Tanenbaum, A. (2009). *Modern operating systems*. Pearson Education, Inc.,.

UXCam (2025). Leading mobile analytics tools. `https://uxcam.com/leading-mobile-analytics/`. Accessed: 2025-08-08.

W3C (1999). Web content accessibility guidelines 2.2. `https://www.w3.org/TR/WCAG22/`. Accessed: 2025-04-05.

(WAI), W. W. A. I. (2008). Understanding conformance requirements. `https://www.w3.org/TR/UNDERSTANDING-WCAG20/conformance.html`. Accessed: 2025-04-14.

Xu, Y., Li, Z., Liu, H., e Liu, Y. (2023). Agaa: An android gui accessibility adapter for low vision users. In *2023 IEEE 47th Annual Computers, Software, and Applications Conference (COMPSAC)*, pages 412–421. IEEE.

Yan, S. e Ramachandran, P. (2019a). The current status of accessibility in mobile apps. *ACM Transactions on Accessible Computing (TACCESS)*, 12(1):1–31.

Yan, S. e Ramachandran, P. (2019b). The current status of accessibility in mobile apps. *ACM Transactions on Accessible Computing (TACCESS)*, 12(1):1–31.

Zhang, M., Liu, H., Chen, C., Gao, G., Li, H., e Zhao, J. (2023a). Accessfixer: Enhancing gui accessibility for low vision users with r-gcn model. *IEEE Transactions on Software Engineering*.

Zhang, Y., Chen, S., Fan, L., Chen, C., e Li, X. (2023b). 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*, pages 1255–1267.