



ScreenAudit: Detecting Screen Reader Accessibility Errors in Mobile Apps Using Large Language Models

Mingyuan Zhong
Computer Science & Engineering
University of Washington
Seattle, Washington, USA
myzhong@cs.washington.edu

Ruolin Chen
The Information School
University of Washington
Seattle, Washington, USA
ruolin@uw.edu

Xia Chen
Carnegie Mellon University
Pittsburgh, Pennsylvania, USA
stooloveu@gmail.com

James Fogarty
Computer Science & Engineering
University of Washington
Seattle, Washington, USA
jfogarty@cs.washington.edu

Jacob O. Wobbrock
The Information School
University of Washington
Seattle, Washington, USA
wobbrock@uw.edu

Abstract

Many mobile apps are inaccessible, thereby excluding people from their potential benefits. Existing rule-based accessibility checkers aim to mitigate these failures by identifying errors early during development but are constrained in the types of errors they can detect. We present ScreenAudit, an LLM-powered system designed to traverse mobile app screens, extract metadata and transcripts, and identify screen reader accessibility errors overlooked by existing checkers. We recruited six accessibility experts including one screen reader user to evaluate ScreenAudit's reports across 14 unique app screens. Our findings indicate that ScreenAudit achieves an average coverage of 69.2%, compared to only 31.3% with a widely-used accessibility checker. Expert feedback indicated that ScreenAudit delivered higher-quality feedback and addressed more aspects of screen reader accessibility compared to existing checkers, and that ScreenAudit would benefit app developers in real-world settings.

CCS Concepts

• **Human-centered computing** → **Accessibility systems and tools.**

Keywords

Mobile accessibility, large language models, accessibility audit.

ACM Reference Format:

Mingyuan Zhong, Ruolin Chen, Xia Chen, James Fogarty, and Jacob O. Wobbrock. 2025. ScreenAudit: Detecting Screen Reader Accessibility Errors in Mobile Apps Using Large Language Models. In *CHI Conference on Human Factors in Computing Systems (CHI '25)*, April 26–May 01, 2025, Yokohama, Japan. ACM, New York, NY, USA, 19 pages. <https://doi.org/10.1145/3706598.3713797>



This work is licensed under a Creative Commons Attribution 4.0 International License. *CHI '25, Yokohama, Japan*
© 2025 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-1394-1/25/04
<https://doi.org/10.1145/3706598.3713797>

1 Introduction

Mobile applications frequently fail to meet accessibility standards [23, 52, 53, 72], rendering app contents and functionality inaccessible to people with disabilities. Despite efforts to promote developer awareness [8, 30, 64] and availability of developer toolkits [7, 16, 28], a recent large-scale longitudinal study [23] found no significant improvements in mobile app accessibility. Meanwhile, an industry survey [2] found that 72% of developers are *not* certain that their company's digital services are accessible.

Contributing to the lack of accessibility improvement and developer confidence may be the limited accessibility error coverage in current automated checkers (e.g., Google's Accessibility Scanner) [10, 28, 41]. For instance, Carvalho et al. found that blind and partially sighted users encountered 40 distinct types of accessibility problems across eight mobile apps and websites [10], yet automated checkers detected only four of these problems [41]. Our own analysis of student developer usage of standard Android development tools revealed that automated checkers were perceived as ineffective and inconsistent and were challenging to use and to interpret.

The importance of accessibility-first design has long been recognized for creating inclusive user interfaces [51, 59]. This approach aligns with guidelines suggesting that prioritizing accessibility from the initial stages of development is crucial [8, 29, 48, 57]. Although automated checkers are widely adopted in the industry [1] and are valuable for identifying issues early and cost-effectively, their current limitations call for improvements in coverage and quality. Recent advancements in large language models (LLMs) [45, 63] demonstrate unprecedented UI understanding capabilities [35, 65, 73] and have been adopted to automatically operate mobile apps, surfacing interface and accessibility issues for software quality assurance [39, 62].

Inspired by these results, we present *ScreenAudit*, an exploratory accessibility checker for Android apps targeted at identifying screen reader accessibility errors and surfacing them to app developers. ScreenAudit aims to enhance the quality, coverage, and interpretability of current automated accessibility checkers by incorporating large language models (LLMs). In contrast to existing accessibility checkers, ScreenAudit automates TalkBack

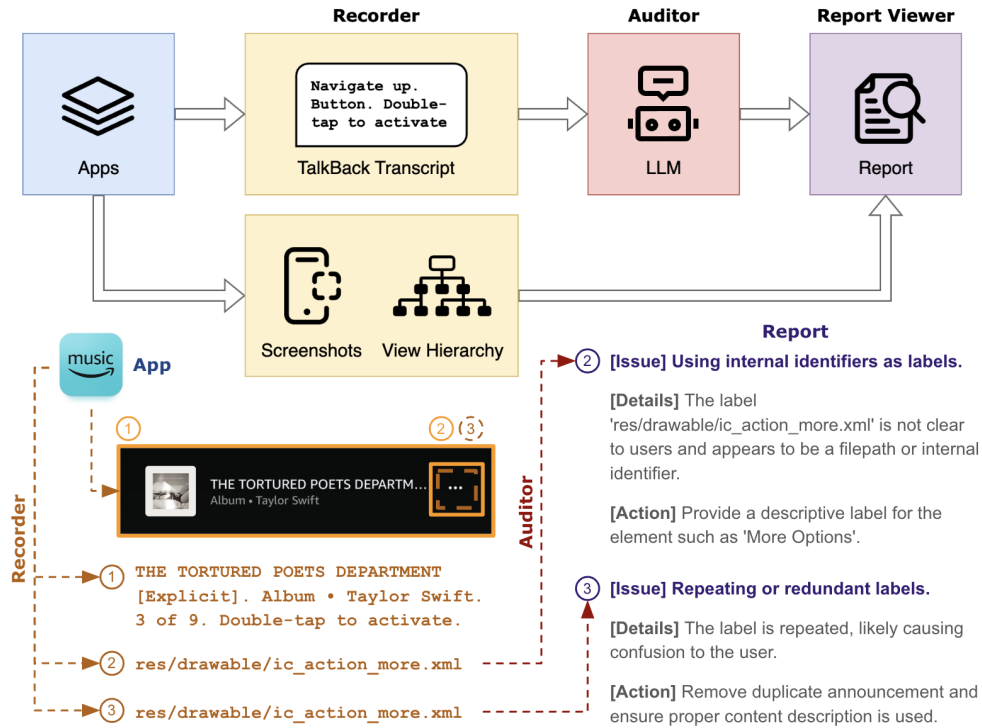


Figure 1: ScreenAudit captures accessibility metadata from an app screen, including TalkBack transcripts. An LLM is used to evaluate potential screen reader accessibility errors, which are presented in a report. In the example from Amazon Music, ScreenAudit identifies the errors of using internal identifiers and redundant labels and provides actionable advice.

(Android’s default screen reader) to traverse and collect speech output from app screens, and then utilizes OpenAI’s GPT-4o [45] to interpret these outputs and identify common accessibility errors (see Figure 1). We prompt the LLM with general instructions about TalkBack output and accessibility failures, without limiting to particular guidelines. Shaped by developer feedback, our reports include screen reader outputs, identified issues and their explanations, and suggested actions for repair.

We evaluated ScreenAudit reports through an expert study. Six accessibility experts (one blind screen reader user and five sighted experts) analyzed a total of 14 mobile app screens independently without prior exposure to our reports. Findings show a strong correlation between expert assessments and the reports generated by ScreenAudit. Of 163 errors identified by experts, ScreenAudit uncovered 69.2%, with a precision of 71.3%. Notably, ScreenAudit identified at least four additional issues initially overlooked by the experts but confirmed upon review. Although limitations exist, experts found ScreenAudit to be effective and easy to use. We constructed an accessibility error dataset based on expert audits and evaluated performance of different prompting strategies. A prompt that provided general accessibility guidance and encouraged LLM’s contextual lookup yielded the best result.

Our goal in creating ScreenAudit is not to replace user testing during development but to complement it. Early and frequent testing is essential. ScreenAudit aims to provide immediate,

actionable feedback for developers, allowing user testing sessions to focus on more complex, nuanced issues that are best understood through user interaction.

In summary, our main contributions are:

- (1) The ScreenAudit system, an accessibility audit tool for developers that captures screen reader announcements and automatically generates accessibility reports with explanations using an LLM.
- (2) A study with accessibility experts that demonstrates the coverage and usefulness of ScreenAudit.
- (3) An evaluation of different LLM prompts and their effects on the detection of accessibility errors.

2 Related Work

Our research builds on related work in automated accessibility evaluation tools and their limitations and related work in automated assessments using large language models (LLMs).

2.1 Automated Accessibility Evaluation Tools

Automated evaluation tools have been extensively used to identify accessibility issues in software [1]. Compared to manual inspection, they offer advantages in efficiency and scalability. Automated accessibility evaluation tools can be broadly categorized into two types: code-level linters and runtime analyzers.

Code-level linters, such as for Android Studio [32], React Native [43], GitHub [3], and Deque’s Axe Accessibility Linter [18], provide basic accessibility checking directly in common developer tools and platforms. Linters almost always employ rule-based static analyses to check for potential accessibility errors and highlight problems in code during app development. However, they can only check a limited set of guidelines [19, 32, 43, 55] and are unable to examine dynamically generated contents or components during runtime.

Runtime accessibility checkers complement linters in the developer workflow by detecting problems in running applications. One type of accessibility checker analyzes accessibility errors on individual app screens or pages, such as Google’s Accessibility Scanner [28], Apple’s Accessibility Inspector [7], Deque’s Axe [16], and WAVE [66]. They inspect and analyze every element in the UI hierarchy of each screen, offering additional insights beyond static analysis [58]. However, these checkers still require manual effort in navigating to and interacting with a specific screen under test.

To address this, another line of work combines the analysis of individual screens with programmatic exploration of apps (i.e., crawling). MATE [22] randomly activates interactive elements in an app to visit different app states and check for accessibility errors heuristically. To simulate the experience of assistive technology users, Groundhog [54] invokes assistive services during automated exploration to test whether UI elements can be focused and activated. Similarly, BAGEL [14] automatically performs keyboard navigation on a webpage to detect accessibility issues in navigation. Some of these automated exploration techniques have been adopted in large-scale assessments of mobile app accessibility [13, 23, 72].

Recent research has started to study and mitigate the problems encountered during app crawling. For example, apps often have designs that hinder programmatic access to certain screens, such as those behind a login screen, an ad, or pop-up windows [23]. To visit potentially blocked screens, juGULAR [4] uses a statistical classifier to detect such states and replays recorded event sequences to bypass them. Another problem arises when presenting accessibility issues to developers: issues identified across an app can be redundant and overwhelming, as screens are often re-visited and UI components re-used. To de-duplicate and summarize issues, Swearngin et al. [61] developed a screen grouping model that supports the reporting of unique accessibility issues in apps.

Still, limitations exist in all current automated accessibility evaluation tools, as a survey found existing tools covered only 8 of 64 accessibility guidelines from BBC and WCAG 2.0 [58]. Deque’s own assessment claimed that their tool covered 16 of the 50 Success Criteria under WCAG 2.1 Level AA [17]. In an evaluation of blind and partially sighted people’s usage of mobile websites and apps, users encountered 40 distinct types of accessibility problems [10], yet current automated checkers detected only four of these problems [41]. Many frequent problems blind and low vision (BLV) users encounter, such as inappropriate feedback, unclear or confusing functionality, inconsistent navigation, are not covered by existing checkers [10].

The scope of our exploratory work here focuses on individual app screens, consistent with tools widely adopted in the industry [7, 16, 28]. We perform automated crawls within the

scope of an app screen, similar to prior work [14, 54], to capture the perspective of assistive technology users. We expand on existing runtime accessibility evaluation tools by extending the types of accessibility errors that they can detect.

2.2 LLM-Supported Automated Assessments and UI Understanding

Large language models (LLMs) have demonstrated proficiency in conducting automated assessments that traditionally rely on human intervention. Zhang et al. [74] developed an LLM-based system for grading programming assignments that identified syntactic and semantic errors and provided feedback comparable to human instructors. Liang et al. [38] showed LLMs were also able to produce feedback on academic papers that resembled that of human peer reviewers, with over 50% of participants perceiving AI-generated feedback to be more valuable than feedback from at least some human reviewers. Closer to our current research, Duan et al. [20] demonstrated LLMs could evaluate and generate feedback for UI designs, and their performance can be significantly improved via few-shot and visual prompting techniques. Wu et al. trained UIClip [69], a vision-language model that quantifies UI design quality that can improve LLM UI code generation.

LLMs also show promising results in other UI understanding tasks, such as generating help [75], extracting macros [34], and autonomously executing tasks as agents [35, 65, 68, 73]. Such UI understanding and decision-making capabilities have been utilized in performing automated GUI and accessibility tests. GPTDroid [39] uses an LLM to explore mobile apps and automatically generate GUI tests. AXNav [62] adopts a similar approach by using LLMs to perform accessibility tests based on natural language instructions and heuristically flag accessibility issues.

These efforts demonstrate strong capability of LLMs in understanding and interacting with mobile UIs. However, such UI knowledge remains largely untapped for analyzing accessibility issues themselves. ScreenAudit aims to leverage these UI understanding and assessment capabilities to identify and explain accessibility problems in mobile apps.

3 Formative Analysis on Student Developer Perspectives

Although existing research has established the inadequacy of existing accessibility checkers, not enough is known about the perspectives of the users of these developer tools (i.e., the developers themselves). To this end, we explored the responses to an assignment’s reflection questions from student developers in an undergraduate course on Android programming, designated as an intermediate-level “core course” in our university’s computer science major. In the assignment, students were tasked with evaluating and repairing accessibility errors using the Android Accessibility Scanner [28] and TalkBack [27] for apps that they created. Full text of this assignment and the reflection questions is available in Appendix A.

The questions analyzed were assigned as part of the regular curriculum of the course. Prior to this assignment, students had received seven weeks of course materials on Android development, including two lectures on accessibility. They also had developed

six prototype Android apps covering different aspects of programming interactive applications. Our formative analysis of student feedback is based on responses stripped of all personally identifiable information (PII). We received a determination that this study was not human subjects research from our University's Institutional Review Board (IRB) because of the exclusion of all PII.

We chose to analyze student reflections because they mirrored the initial experiences and challenges that many developers have when checking and repairing accessibility failures. The structured academic setting ensured that students received uniform instructions and operated under similar conditions. Student-identified accessibility failures covered 11 of the 15 most frequently encountered problem types by blind people who use screen readers in prior work [10, 41]. Detailed procedure and analysis results are available in Appendix B.

3.1 Feedback for Current Developer Tools

Using standard thematic analysis [9], we analyzed student reflections on the problems of current developer tools. We also sought to understand the features they desired to better support accessible app development.

3.1.1 Problems. Our analysis revealed three themes in student problems with current developer tools: P1: *Limited detection scope and inconsistency in Accessibility Scanner*, where students noticed unreported problems, and identified inconsistent or false positive issues. P2: *Difficulty operating TalkBack*, where students reported encountering bugs and difficulty setting up TalkBack and navigating using gestures and hearing announcements. P3: *Lack of developer support in all developer tools*, where students had difficulty with locating relevant tools and interpreting their results on potential accessibility issues.

3.1.2 Features. Our analysis further revealed three main features students wanted in developer tools to enhance accessible app development: F1: *Integrate accessibility tool output and provide accessibility alerts during development*: Accessibility Scanner results (alerts and warnings) and TalkBack announcements should be integrated with Android Studio. F2: *Enhance explanations of issues and improve developer education*: Tools should provide examples and explanations for potential accessibility issues, along with tutorials and learning materials. F3: *Conduct code-level scanning and provide suggestions*: Tools should offer intelligent code analysis to flag potential accessibility issues and provide refactoring suggestions.

3.2 Design Implications

Our findings show that developers, especially those new to accessibility, encounter significant challenges that current development tools fail to address. We identify five design implications (D1–D5) for an accessibility checker based on the above findings and problems identified by prior work. As an exploration of incorporating LLMs to support accessibility checking, ScreenAudit focuses on the identification and explanation of errors, and implements three of the design implications (D1–D3). D4 and D5 require additional research and development in future work on developer tools.

3.2.1 D1: Expand the Coverage of Current Accessibility Checkers. Developers and prior work [10] identified the limited coverage of error types supported by existing checkers, limiting their benefits and effectiveness (P1). We design ScreenAudit to expand coverage while recognizing the benefits of rule-based checking for its efficiency, reproducibility, and usefulness for certain error types (e.g., missing labels, color contrast, target size). Therefore, ScreenAudit is intended to operate alongside existing checkers and tools, while future iterations can directly incorporate rule-based checking features [26].

3.2.2 D2: Integrate Screen Reader Output. Developers who had no prior exposure to screen readers have difficulty in setting up and operating TalkBack even after tutorials given in lectures (P2). Developers also hoped to see TalkBack's output integrated into developer tools in support of identifying or confirming problems without context-switching (F1). ScreenAudit automatically captures TalkBack outputs for the target screen and includes them in the generated accessibility report for quick reference.

3.2.3 D3: Provide Explanations and Actions for Errors Detected. Developers had difficulty interpreting the Accessibility Scanner's outputs as certain terms were considered overly technical (P3); developers desired better explanations with examples of accessible designs (F2). We designed ScreenAudit to generate concise explanations with suggested actions for each error detected. We anticipate that future iterations of our tool will provide direct links to relevant tutorials or guidelines to support developer education.

3.2.4 Additional Design Implications. We further identify two design implications: D4: *Integrate accessibility checker features directly into Android Studio or other IDEs* (P3, F1), and D5: *Provide code-level analysis and suggestions for accessibility* (P3, F3). We did not implement these features but acknowledge their promise as potential future work, whether by us or by others.

4 ScreenAudit: System Design

We created ScreenAudit based on the design implications identified in our formative analysis and in prior work. ScreenAudit is intended to function similar to current accessibility checkers: a developer opens an app screen to test on their phone or emulator and initiates a scan with ScreenAudit. After the scan is complete, the developer can view the report through a web portal on their computer.

The ScreenAudit system consists of three main components: a *Recorder* that traverses a mobile app screen and captures necessary metadata including its TalkBack output, an *Auditor* that evaluates potential screen reader accessibility errors on the screen using an LLM, and a *Report Viewer* that displays a report highlighting detected errors and provides explanations. Figure 1 illustrates ScreenAudit's system pipeline.

4.1 Recorder

ScreenAudit collects metadata of the screen being scanned with its Recorder. One important goal of the Recorder is to collect TalkBack's spoken feedback so the Auditor can analyze the same announcements a user would hear on screen readers. This also fulfills design implication D2. In contrast, traditional rule-based checkers examine a static snapshot of a screen, which ignores its

dynamic aspects (e.g., focus order, scrollable views) and cannot effectively use the context provided by neighboring elements.

4.1.1 Using Gestures to Control TalkBack and Traverse a Screen. Our initial attempt to capture TalkBack output was based on the TalkBack logs¹ and sending SWIPE_RIGHT gestures to the device. This approach captured accurate announcements but had three main drawbacks:

- (1) *The time needed to traverse a screen is long.* In our testing, even with text-to-speech (TTS) set at max speed, the average time for capturing one element was three seconds. We observe between 10 and 40 elements in most app screens before any scrolling occurs. Therefore, we typically need between 30 and 120 seconds to capture a screen’s TalkBack output, which reduces the efficiency of the entire auditing process. Three factors contribute to the overall delay: (1) The capture duration depends on content length, which can range from under one second to tens of seconds. (2) A 0.5-second delay is needed to ensure that we capture an element’s usage hint, which has a built-in delay of 0.4 seconds from TalkBack. A usage hint is necessary for conveying an element’s accessibility role, such as “Double-tap to activate.” (3) Furthermore, an extra ~0.4 seconds is needed to send a swipe gesture to the device and wait for a response from TalkBack.
- (2) *Lack of standardization of captured results.* As each developer can choose to customize their TalkBack settings, there is no guarantee when deploying ScreenAudit that a developer’s settings will be the same as ours. For example, usage hints can be hidden if turned off in TalkBack’s settings. This inconsistency will impact the audit accuracy by misleading the Auditor to believe that all usage hints are missing.
- (3) *Focus indicator is captured.* TalkBack shows a green focus indicator for the element with screen reader focus. Although this feature is valuable for users with low vision and for testing screen user support, the indicator can distract developers if captured as part of the screenshot in ScreenAudit’s report.

To address these issues, we created a custom fork of TalkBack based on its open source implementation [31] that preserves its normal speech output behaviors, but with additional capabilities: (1) Efficient speech capture and traversal using TalkBack’s internal pipeline. Specifically, whenever the text for an announcement is created, our implementation skips TTS and directly proceeds to the next focusable element. (2) Settings are stored independently of TalkBack. (3) The focus indicator can be hidden. Our improvements reduced the capture time per element to around 0.15 seconds (limited by the internal processing latency of TalkBack) while maintaining the ability to capture usage hints, which means typically a screen takes between 1.5 and 6.0 seconds to capture, regardless of spoken content length.

4.1.2 Capturing TalkBack Transcripts and Screen Snapshots. The Recorder captures TalkBack announcements by sequentially focusing on each screen reader focusable element, starting from the default element of initial focus. Similar to prior work [62]

and in keeping with the Accessibility Scanner, we limit the total number of elements traversed to the first screen or 40 elements, to prevent infinite loops or scrolls. In addition, the Recorder captures the corresponding screen snapshot with screenshot and view hierarchy data, similar to Fok et al. [23]. Each TalkBack announcement is associated with its corresponding element on screen by comparing their bounding boxes.

While traversing a screen, dynamic UI changes in content or layout may occur, such as when scrolling or loading new contents. The Recorder monitors TYPE_VIEW_SCROLLED and TYPE_WINDOW_CONTENT_CHANGED events from the Android system’s Accessibility Service to determine if a potential change has occurred and capture a new snapshot. Redundant snapshots are removed if there is no pixel-level difference.

4.2 Auditor

We focus the design and implementation of the Auditor on fulfilling design implications D1 and D3, expanding the coverage of existing accessibility checkers and providing explanations and actions for errors detected. To implement these, we use the TalkBack transcript as part of our custom prompt to request feedback from the LLM, and prompt the LLM to provide explanations and actions for each error. The Auditor sends the prompts to GPT-4o [46] (a derived model of GPT-4 [45]) for completion.

4.2.1 Initial Observation. Our initial attempt identified the strength and weakness of GPT-4o’s assessments evaluating TalkBack transcripts for screen reader accessibility without additional instructions. The main strength is its ability to infer information based on surrounding context from the transcript. For example, in a food delivery app there are two consecutive announcements: “30 min” and “Delivery time.” The LLM is able to associate them together and provide the following suggestion: “Consider grouping ‘30 min’ and ‘Delivery time’ together to avoid redundancy and to provide clear context.”

The main weakness is the model’s lack of specific knowledge of TalkBack and mobile accessibility. For example, the output “Selected, Home. Tab 1 of 4. Double-tap to activate” follows the default announcement of TalkBack in the order of *state*, *label*, *role*, *usage hint*. However, such formatting can result in false positives from the LLM, where the model moves the *state* (“Selected”) to the end, especially when the label and usage hints are more complex. In another example, the LLM claims this output “\$0 Delivery Fee on \$15+” is not clear because it did not indicate if it is plain text or an actionable item. However this assessment contradicts WCAG 4.1.2, which does not require textual labels to be indicated for conciseness.

4.2.2 Prompt Structure. To improve the accuracy of the model’s accessibility assessments, we experimented with different ways of including accessibility guidelines and principles.

Table 1 details the prompt structure we adopted in our analysis. Our prompt consists of four sections: *Introduction*, *Accessibility overview*, *Instruction*, and a *TalkBack Transcript* for the screen being evaluated. Except for the Transcript section that includes dynamic information of the transcript being analyzed, all other sections are static. We design our prompt based on our initial observations and

¹These logs can be accessed by setting TalkBack’s log level to verbose and searching for `SpeechControllerImpl`’s speaking entries.

Table 1: Prompt Structure for Analyzing TalkBack Transcripts

Section	Context	Prompt
<i>Introduction</i>	Static	You are examining the accessibility of an app based on transcripts of TalkBack. You will see a transcript of what a screen reader user will hear when they use an app. Please analyze if each interaction reflected in this transcript is accessible. ...
<i>Accessibility</i>	Static	# Accessibility Basics We want descriptions to be informative and concise. Consider if the transcript for each element conveys its content or purpose appropriately. Most important information should appear first. ...
<i>Instruction</i>	Static	# Your Task Follow these steps to evaluate the accessibility of the transcript: Step 1 - Look at the big picture. Consider each transcript entry in relation to the elements before and after it. Is this element related to nearby elements? ...
<i>Transcript</i>	Dynamic	app: "...", transcripts: [... { index: 6, transcript: "Image Search. Button. Double-tap to activate"}, { index: 7, transcript: "Appliances"}, ...]

commonly encountered accessibility problems discussed in prior work. Our full prompt can be found in Appendix C.

The Introduction section briefly describes the task. In the Accessibility overview section, we provide fundamental accessibility guidelines to TalkBack in one of two ways: (1) a *General* guideline independently written by two of the researchers and then compiled together, and (2) a *WCAG 2.1*-based list of accessibility guidelines, with items that are not relevant to transcript analysis removed. We compare these against a *Base* condition, which lacks the Accessibility overview section.

In the Instruction section, we provide chain-of-thought instructions [67] of accessibility evaluation requirements, with the aim of understanding whether the model can better check for context when explicitly instructed to do so. We include two prompts to evaluate this: (1) a *Basic* instruction that asks the model to evaluate the transcript for its accessibility, and (2) a *Contextual* instruction that has an additional step that asks the model to look at nearby elements for context.

Finally, the Transcript section is appended to the end to form a complete prompt. As these designs were made through informal testing, we evaluate all prompt designs after collecting a “ground truth” accessibility error dataset from experts (see Section 6).

4.3 Report Viewer

The Report Viewer consists of screenshots captured by the Recorder and evaluations generated by the Auditor in an expandable list view (see Figure 2). On the left, a screenshot highlights detected accessibility errors. Locations of the elements are determined based on the view hierarchy data captured by the Recorder. Hovering (or pressing the tab key) over each element will show its TalkBack transcript and detected error. On the right, a list of detected accessibility errors are shown with brief descriptions at a glance. An element can be selected on the screenshot or in the list. According to the element selected, if an error exists, the list

view expands to display its TalkBack transcript with details and recommended actions on error handling for the developers.

The list view in Figure 2 shows details on two exemplar accessibility errors detected by ScreenAudit:

- (1) The error at index 1 shows the “add” icon does not have an appropriate label and ScreenAudit suggests to add a label “Add new location”. The transcript shows that the button is entirely unlabeled by the developer, and that TalkBack attempts to mitigate the problem by auto-generating a label “+”. Therefore, the button here violates WCAG 1.1.1, which requires non-text contents to have text alternatives. The label generated by TalkBack violates WCAG 2.4.6, which requires labels to be descriptive.
- (2) The error at index 12 shows ScreenAudit identifies redundant announcement for the train element. Immediately prior to this, the element at index 11 reads “5:25 PM peak train towards Ron-kahn-kah-muh. Scheduled ...” (not shown in the screenshot). ScreenAudit identifies the train information to have existed in the previous speech, and suggests removing the label. This type of error has been identified in prior work [42] as “overly perceivable.” Specifically, this label violates WCAG 1.3.1, which requires information to be equally perceivable to all users.

5 Accessibility Error Dataset Construction and Expert Evaluation

We recruited accessibility experts with significant experience with screen readers to examine the report produced by ScreenAudit. We have two main goals in conducting this study: (1) to construct a ground-truth dataset of accessibility errors to evaluate the coverage and accuracy (recall and precision) of our system, and (2) to obtain expert subjective opinions on ScreenAudit’s reports. We received approval from our University’s Institutional Review Board (IRB) prior to the study.

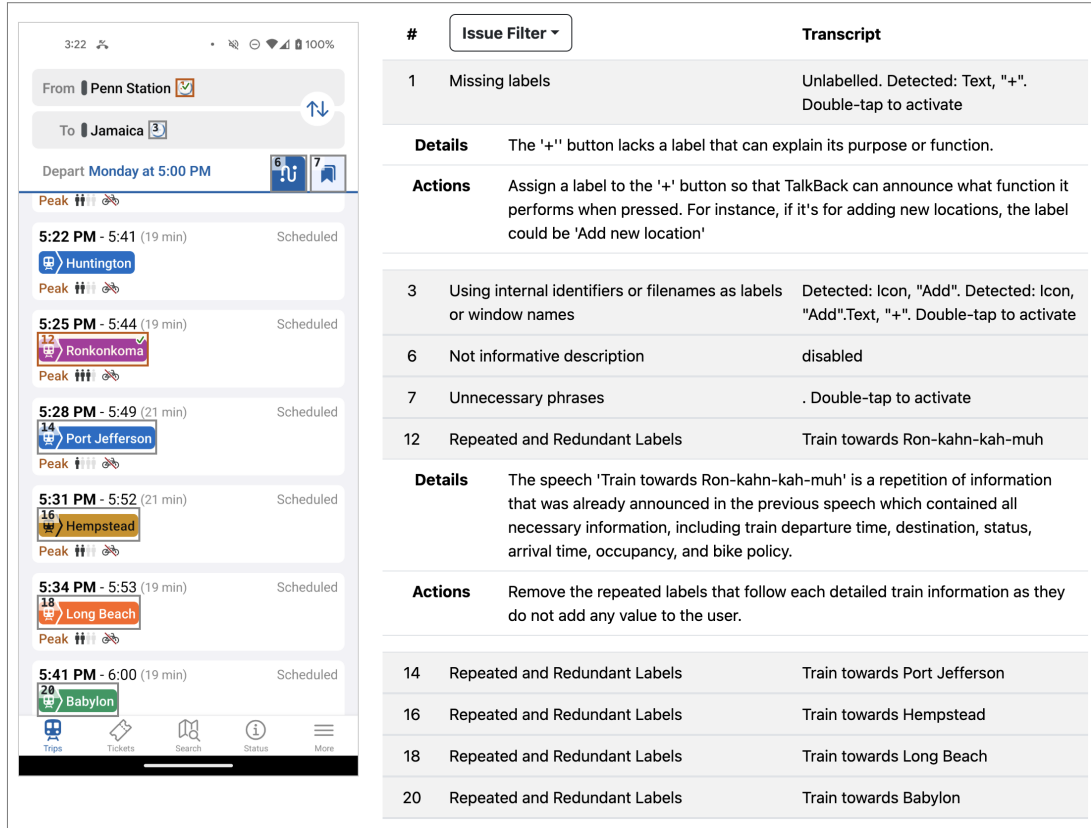


Figure 2: The Report Viewer of ScreenAudit displaying accessibility errors detected in a mobile application. The report entries for item 1 (unlabeled add button for “From” station) and item 12 (a redundant label) are expanded with their details shown. The “Issue Filter” menu (top) allows filtering of errors by type.

In this section, we first introduce how we selected the Android apps for the study, then the expert profiles and study procedure. After that, we provide details on our analysis and study results. Finally, we illustrate ScreenAudit’s success and failure cases with examples.

5.1 Android App Dataset

We prepared a dataset of 14 frequently downloaded apps for use in our evaluations. They were randomly selected from 312 apps evaluated in [23], covering nine common app categories: Education, Entertainment, Events, Food & Drinks, Housing, Music, Shopping, Sports, and Travel. During the selection process, whenever a selected app was no longer available from the Google Play store, we excluded it and selected an alternative app with a similar number of downloads. We encountered two such apps. We also excluded one app that was completely inoperable with screen readers, as our tool currently only covers elements reachable using a screen reader. We believe a combination of heuristics and LLM analysis can be used to audit such apps, taking inspirations from [14, 54], and we leave this case for future research.

5.2 Expert Profiles

We recruited six accessibility experts (E1–E6) with an average of 15.9 ($sd = 8.9$) years of experience (YoE) directly related to designing, developing, researching, or using access technology (AT) for people who are blind or low vision. All participants are currently working in accessibility-related roles and self-described as very familiar with screen readers. E1–E5 all have accessibility research or development experience at major software companies and in academic settings. E6 is blind and uses screen readers. He consults and coaches mobile app developers to help make their apps accessible. Table 2 details the profile of each expert. We provided study information to each participant as part of our invitation, including its description, duration, and compensation. All participants voluntarily chose to participate in the study.

5.3 Procedure

We conducted one evaluation session with each participant, which lasted between 60 and 90 minutes. Each participant was provided with a 50 USD gift card as compensation. Each session consisted of three parts: an introduction, a few evaluation tasks, and a semi-structured interview. Time permitting, we presented up to four randomly chosen mobile app screens from our dataset.

Table 2: Expert Participant Profiles.

Id	Main Accessibility Experience	YoE	Current Role	AT Experience
E1	Developer of an iOS app designed for people who are blind or low vision at a software company.	8	PhD Student	VoiceOver, iOS Accessibility Inspector
E2	PDF Document accessibility research at a software company and at a university.	2.5	PhD Student	NVDA, VoiceOver, AxesPDF, WAVE, PDF checkers
E3	Technology specialist, web developer at a university's accessible technology center.	24	Director	JAWS, NVDA
E4	Internal accessibility consulting and usability testing at a software company's accessibility team. Teaches programmatic access and designing for people with disabilities at a university.	24	Professor	Narrator, JAWS, company's internal accessibility testing tools
E5	Research with screen reader users, including using AR for navigation and non-visual games at a software company.	12	Researcher, Software Engineer	Screen Readers, Augmentative and Alternative Communication (AAC) devices
E6	Mobile app accessibility consultant, screen reader user (blind).	25	Program Manager	VoiceOver, TalkBack

First, we introduced the basics of TalkBack and common screen reader accessibility guidelines adapted from the Appt Guidelines for mobile apps [24] as a reminder. We also demonstrated the use of Accessibility Scanner and ScreenAudit's report on an app outside of our dataset. The introduction took between 10 and 15 minutes.

Next, participants performed independent evaluations of each mobile app screen. Participants were asked to use TalkBack to interact with the screen and identify any screen reader accessibility errors and take brief notes in a blank study worksheet. Participants were also offered a report from the Android Accessibility Scanner [28]. After the assessment, we asked each participant to explain all observed errors verbally. We then presented the participant with ScreenAudit's report for the same screen, which was not offered during their evaluation. Participants were asked to note any missing issues or inaccurate descriptions. There were no time constraints. The evaluations took between 40 and 60 minutes.

Finally, we conducted a semi-structured interview to gather additional feedback. We used open-ended questions to elicit feedback on the tool's usability, coverage, and potential impact on accessibility practices. The interview took between 10 and 15 minutes.

We conducted the sessions with E1–E5 in-person and with E6 remotely. The above procedure was modified for E6 with three changes: (1) We did not explain the guidelines and use of screen readers as E6 was blind and a screen reader user. (2) Transcripts and descriptions of the app screens were made available to E6 prior to the study. (3) During the study, the researcher first verbally described the screen then went through each transcript entry and described its corresponding UI element when necessary, and asked the participant for an evaluation. E6 was then read the ScreenAudit's analysis and asked to evaluate the report entry. This process repeated for each app.

5.4 Analysis Overview

We collected audio recordings of each study session and each participant's study worksheet. Participants analyzed between 1 and 4 screens in each session, with an average of 2.3 screens. In total, participants covered 14 unique screens, each in a different app, containing 306 UI elements. We processed the collected data in two parts: (1) accessibility error analysis of all UI elements and construction of the accessibility error dataset and (2) analysis of qualitative data gathered during interviews.

Table 3: Accessibility error categories identified by experts and their corresponding WCAG 2.1 success criteria.

Error Category	Success Criteria	Count
Missing Label	1.1.1 Non-text Content	39
Label Quality	2.4.4 Link Purpose (In Context) 2.4.6 Headings and Labels 4.1.2 Name, Role, Value	42
Structure & Grouping	1.3.1 Info and Relationships 1.3.2 Meaningful Sequence 2.4.3 Focus Order 3.2.3 Consistent Navigation	54
Heading	2.4.2 Page Titled 2.4.10 Section Headings	16
Functionality	2.1.1 Keyboard 3.2.1 On Focus 3.2.2 On Input 4.1.3 Status Messages	12
(No Error)	/	143
Total	/	306

5.5 Accessibility Error Analysis and Dataset Construction

To obtain an objective accessibility error determination of all UI elements, two researchers independently analyzed participant analyses of all screens and assigned relevant transcripts or worksheet notes to their relevant UI elements. The researchers then individually summarized the accessibility error(s) for each UI element based on these transcripts and notes, while cross-referencing WCAG 2.1 [50] and MagentaA11y’s native app accessibility guidelines [40]. Finally, the researchers combined their analysis and discussed any inconsistencies, resulting in a final list of accessibility errors for all UI elements.

Our analysis revealed 163 accessibility errors across the 14 screens evaluated. To further understand the types of errors encountered, we classified the identified errors into five categories: (1) *Missing Label*, (2) *Label Quality*, (3) *Structure & Grouping*, (4) *Heading*, (5) *Functionality*. These classifications are guided by expert comments and the WCAG 2.1 accessibility guidelines. We identified the most relevant success criteria observed within each error category and their number of occurrence as a reference, detailed in Table 3.

5.6 Illustrative Examples

We show examples to illustrate the accessibility errors identified by the experts and the outputs from ScreenAudit and Accessibility Scanner. Figure 3 shows seven exemplar UI element TalkBack transcripts, expert-identified accessibility errors, and outputs from the tools. Each example may show more than one error, because (1) a UI element may trigger multiple TalkBack announcements or (2) a single announcement may contain more than one error.

Specifically, we consider ScreenAudit’s outputs for examples ① – ④ to be consistent with expert analyses, while its outputs for examples ⑤ – ⑦ are inconsistent. Next, in Sections 5.7.3 and 5.7.4, we discuss these examples in the context of expert comments.

5.7 Qualitative Feedback from Experts

We received expert feedback during their evaluation tasks and after the tasks during the interview. While analyzing feedback, we followed guidelines for thematic analysis [9] which involved an inductive and deductive coding process. Experts in general found ScreenAudit to be (1) *effective and consistent*, (2) *easy to understand and to use*, (3) *capable of uncovering neglected errors*, but also identified its (4) *limitations*.

5.7.1 Effectiveness and Consistency. Experts found ScreenAudit consistently and effectively identified major accessibility issues within mobile applications.

It definitely did a better job than the current tool because I think the current tool would like identify some things and then randomly miss it for a very similar issue. This was a lot more consistent. (E1)

I see a ton of benefit in the fact that it points out a whole bunch of different things to think about and it makes it really easy too. (E1)

I think it picked up most of what I picked up. (E2)

But it does seem to be pulling up all the main points that I did, and I can’t think of what else it

would say. That’s great to know. But it’s interesting. (E3)

The fact that this is a tool is extremely helpful. (E6)

5.7.2 Ease of Understanding and Use. Experts found ScreenAudit’s interface and generated reports accessible and easy to comprehend, which makes it user-friendly for developers who may not be deeply familiar with accessibility.

So I think the two views of the UI and where they are in the UI, but also just being able to go through a list is, is really helpful. Also, you have an accessible view with the list, which is always good. (E1)

The issues are in a human readable format. So I think over time, you would really like there, it’s easy to comprehend. (E4)

5.7.3 Uncovering Neglected Errors. ScreenAudit identified accessibility errors that experts considered to be hard to identify or errors that experts initially missed. These cases require the model to understand interface structure or identify low-quality labels.

(1) **Structure Understanding:** In example ① of Table 2, ScreenAudit impressed E1 on its ability to understand navigation. In example ②, E2 appreciated ScreenAudit’s ability to identify the context.

① I think search results are a very special case in the sense that it’s one of the more difficult navigation things and it would be very difficult to pick that up. So, like, there’s something about navigation, which I would actually say was probably a huge problem. But it did. (E1)

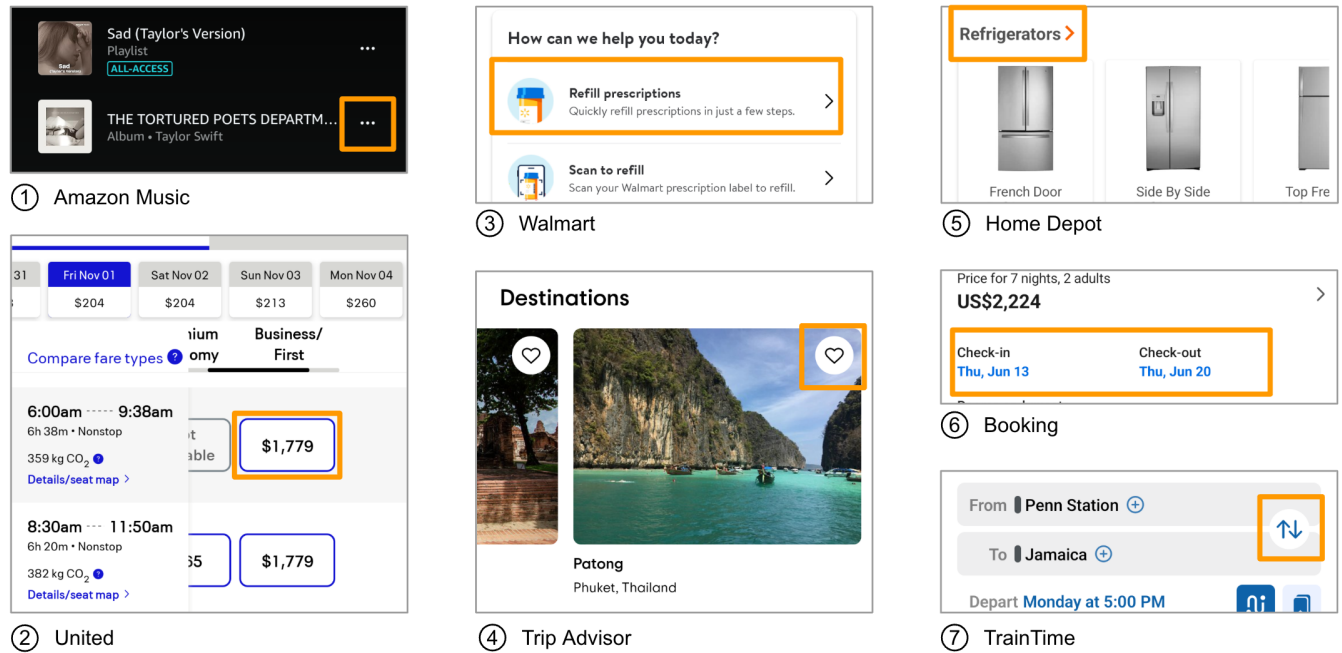
② And since this is not a part of any header, then maybe it should just simply talk about it. But since it’s a part of the column, it should identify it in the form of a table. So that it identified is really great. (E2)

(2) **Identifying Low-Quality Labels:** In example ③, ScreenAudit identified an extra announcement “2 Period”, which was initially missed by E5 during the study. In example ④, E3 initially did not recognize the repeated announcements that were detected by ScreenAudit.

④ Quite nicely. I wasn’t necessarily... Was I hearing all the “double-tap” and “double-tap and hold” stuff? Oh, I think I was just tuning out. It says they are identical. That’s true. And the corresponding location it’s associated with is what I was saying. (E3)

5.7.4 Limitations. Experts identified three main limitations in their feedback during and after the studies: (1) *inaccurate results*, (2) *not fully understanding context*, and (3) *unable to test app functionality*.

(1) **Inaccurate Results:** Sometimes, ScreenAudit identifies an error, but its explanation does not directly address the root cause. In example ⑤, ScreenAudit identified a structural issue but did not suggest add a heading. E3 noted this and commented that, while the output was incorrect, it could provoke developers to think about their design.



#	App & UI Element	TalkBack Transcript	Expert Analysis	ScreenAudit	Accessibility Scanner	Type
①	Amazon Music, "View More" Button	res/drawable/ic_action_more.xml (<i>Repeated for consecutive swipes.</i>)	(1) Improper content description for Image Button. (2) Duplicated button.	(1) Using internal identifiers as labels. (2) Repeating or redundant labels.	Multiple clickable items share this location on the screen.	LQ SG
②	United, Price Selection	\$1,779. Double-tap to activate.	Table element should announce column and row headers.	The price without context is unclear.	Multiple items have the same description.	Head
③	Walmart, "Refill Prescriptions"	...Quickly refill prescriptions in just a few steps 2 Period Opens in a browser...	Unnecessary and confusing text "2 Period" in label.	The statement '2 Period' in the speech output is confusing.	n/a	LQ
④	Trip Advisor, "Save to Trip" Button	Save to a trip. Double-tap to Save to a trip. Double-tap and hold to Save to a trip.	(1) Relationship between button and the trip to be saved unclear. (2) Repeated speech unnecessary.	(1) Button label unnecessarily repeats the action hint. (2) Multiple labels are identical with no unique context.	n/a	SG LQ
⑤	Home Depot, "Refrigerators" Heading	Refrigerators. Double-tap to activate.	(1) No heading indication. (2) Activation target unclear.	Elements ... seem to be related and should be grouped.	n/a	Head Func
⑥	Booking, "Check-in," "Check-out" Options	– Check-in – Check-out – Thu, Jun 13 – Thu, Jun 20	Grouping and ordering issue.	n/a	n/a	SG
⑦	TrainTime, "Switch" Button.	(<i>Not announced.</i>)	The "Switch From/To" button is not focusable.	n/a	n/a	Func

Figure 3: Accessibility errors identified by ScreenAudit and Accessibility Scanner. Axe [16] did not identify any of the errors listed. Abbreviations: LQ = Label Quality. SG = Structure & Grouping. Head = Heading. Func = Functionality.

- (2) **Not Fully Understanding Context:** Experts identified the main limitation of ScreenAudit to be its limited ability in understanding context surrounding elements (e.g., example ⑥), while noting that all current tools have problem in this aspect.

⑥ I think the only thing was like sort of missing things, which I think were usually because it was more context based. Which I also understand isn't really, um, isn't really something there's even like a set standard for. (E1)

- (3) **Unable to Test App Functionality:** Another limitation experts identified was that ScreenAudit was not able to directly test the functionality on the UI. In example ⑦, the switch button could not be focused by the screen reader and was ignored. E5 identified this in his testing and pointed out:

⑦ It's not catching interaction things though, and I think that's a really critical problem. None of the tools do. (E5)

6 Performance Evaluation

In this section, we evaluate the performance of ScreenAudit and provide insights into its prompt design. We conduct our analyses in five respects: (1) Comparing the performance of ScreenAudit and two current accessibility checkers. (2) Comparing the performance of ScreenAudit's five prompts. (3) Understanding whether ScreenAudit's outputs are consistent across multiple runs. (4) Examining ScreenAudit's performance by each accessibility error category. (5) Comparing the performance of currently available LLMs and understand whether these prompting techniques can be transferred to other LLMs.

6.1 Experiment Method

All our experiments were performed on the accessibility error dataset created in the expert study. Unless otherwise specified, we used OpenAI's GPT-4o, 2024-08-06 version.

For each analysis, two of the researchers independently compared a tool's output for each UI element to the corresponding ground truth label collected in the expert study. Each researcher assigned "consistent" or "inconsistent" to each of the tool's outputs, which we refer to as *Correctness*. An output was found to be *inconsistent* if it did not identify an issue, identified a different issue, or provided an incorrect explanation. After this process was done, the researchers discussed until an agreement was reached on all elements.

6.2 Analyses and Results

6.2.1 Comparison of Accessibility Evaluation Tools. We compared ScreenAudit against two widely-used accessibility checkers for Android: Google's Accessibility Scanner [28] and Deque's Axe [16]. Axe was not able to run on one screen in the dataset (United's booking screen), therefore that screen was excluded for Axe.

Table 4 shows precision, recall, and F1 score for each of the tools tested. An analysis of variance based on mixed logistic regression [25, 60] indicated a statistically significant effect of *Tool* on *Correctness*, $\chi^2(2, N=2142) = 170.1, p < .001$. Post hoc pairwise comparisons, corrected with Holm's sequential

Bonferroni procedure [33], indicated that ScreenAudit vs. Accessibility Scanner ($Z = 8.08, p < .001$), ScreenAudit vs. Axe ($Z = 11.02, p < .001$), and Accessibility Scanner vs. Axe ($Z = 4.11, p < .001$) were significantly different. The Accessibility Scanner and Axe's lower recall agrees with prior research that current automated checkers only detect a small subset of accessibility errors [10, 41]. ScreenAudit is able to cover a much higher percentage of errors with a modest reduction in precision.

Table 4: Performance metrics for ScreenAudit, Accessibility Scanner, and Axe.

Tool	Precision	Recall	F1
ScreenAudit *	.713	.622	.664
Accessibility Scanner	.729	.313	.438
Axe	.812	.171	.283

* ScreenAudit's statistics are averages of the five prompts evaluated in Section 6.2.2.

6.2.2 Comparison of Prompts. We tested the following five prompts according to their structures outlined in Section 4.2.2: (1) *Base*: a baseline condition with only an introduction and task instructions. (2) *General*: *Base* prompt with added general accessibility guidelines. (3) *Contextual*: *Base* prompt with added instructions about looking for contextual cues. (4) *General_Contextual*: Combined prompt with *General* accessibility guidelines and *Contextual* instructions. (5) *WCAG_Contextual*: Combined prompt with *WCAG* accessibility guidelines and *Contextual* instructions.

Table 5 shows the evaluation results for the five prompts. An analysis of variance based on mixed logistic regression [25, 60] indicated a statistically significant effect of *Prompt* on *Correctness*, $\chi^2(4, N=1530) = 18.66, p < .001$. Post hoc pairwise comparisons, corrected with Holm's sequential Bonferroni procedure [33], indicated that *Base* vs. *Contextual* ($Z = 3.28, p = .010$) and *Base* vs. *General_Contextual* ($Z = 4.02, p < .001$) were significantly different.

Our results show that ScreenAudit's prompt with general accessibility guidelines and contextual instructions has the highest recall and F1 score. It appears that the addition of either general accessibility guidance or contextual instruction alone does not improve model performance over the *Base* prompt. However, a combination of both improves the performance. Interestingly, adding the WCAG guidelines instead of general accessibility guidance does not yield better performance compared to the *Base* prompt. One possible reason is that the WCAG guidelines are abstract and technical, making it difficult for the LLM to effectively utilize. We further discuss this finding in Section 7.2.

6.2.3 Consistency. We ran the *General_Contextual* prompt four additional times on the accessibility error dataset to evaluate ScreenAudit's output consistency. We performed an analysis of variance based on mixed logistic regression [25, 60] on the five runs and found no detectable difference in *Correctness*, $\chi^2(4, N=1530) = 3.24, p = .52$. Given the amount of data analyzed, any meaningful difference in output would likely have been

Table 5: ScreenAudit’s prompt performance on the accessibility error dataset collected in the expert study.

Prompt	Introduction	Accessibility	Instruction	Precision	Recall	F1
<i>Base</i>	Yes	/	Basic	.797	.577	.669
<i>General</i>	Yes	General	Basic	.696	.589	.638
<i>Contextual</i>	Yes	/	Contextual	.667	.650	.658
<i>General_Contextual</i>	Yes	General	Contextual	.708	.699	.704
<i>WCAG_Contextual</i>	Yes	WCAG	Contextual	.698	.595	.642

detected; therefore, we found the model output to be consistent across the evaluated runs. On average, the model achieved a precision of .723 ($SD = .038$) and a recall of .692 ($SD = .023$), with an overall F1 score of .707.

6.2.4 ScreenAudit Performance by Error Category. We further break down each prompt’s and each accessibility checker’s performance by accessibility error category in Table 6 and visualize their true positives, true negatives, and classification errors in Figure 4. As expected, Missing Label errors are relatively easy to detect by automated checkers and therefore have higher detection rates across all prompts and checkers. Nevertheless, ScreenAudit outperforms rule-based checkers in all accessibility error categories. Comparing those elements with no errors, rule-based checkers have lower false positive rates than ScreenAudit. Future research can investigate whether rule-based checkers can support LLMs, for example through Retrieval Augmented Generation (RAG), to lower false positive rates.

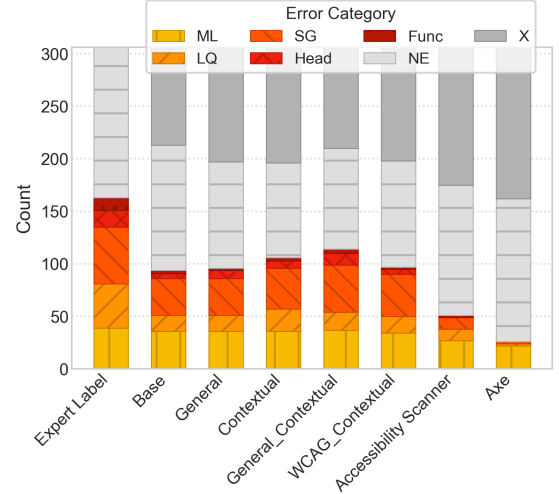
For ScreenAudit, it appears that general accessibility guidelines improved the model’s performance in detecting errors with Headings, while contextual instructions improved the model’s performance in detecting errors with Label Quality and Structure & Grouping. The lower performance of all prompts in Label Quality and Functionality is likely because ScreenAudit does not currently explore the interactivity or functionality of elements. Therefore, it becomes more difficult for ScreenAudit to detect such errors as many are context-dependent.

Table 6: Break down of prompt or tool’s accuracy (in percentages %) by accessibility error category.

Prompt/Tool	ML	LQ	SG	Head	Func	NE
<i>Base</i>	92.3	35.7	64.8	31.2	25.0	83.2
<i>General</i>	92.3	35.7	64.8	50.0	16.7	70.6
<i>Contextual</i>	92.3	50.0	72.2	43.8	25.0	62.9
<i>General_Contextual</i>	94.9	40.5	83.3	68.8	33.3	67.1
<i>WCAG_Contextual</i>	87.2	38.1	74.1	31.2	16.7	70.6
A11y Scanner	69.2	26.2	20.4	0.0	16.7	86.7
Axe	56.4	5.3	4.0	0.0	0.0	94.9

Abbreviations: • ML = Missing Label • LQ = Label Quality • SG = Structure & Grouping • Head = Heading • Func = Functionality • NE = No Error

6.2.5 Comparison of LLMs. We compared the performance of GPT-4o, the LLM that we used in the above experiments,

**Figure 4: Stacked bar plot for each prompt or tool’s performance by error category, with expert labels on the left as reference. Counts for ML, LQ, SG, Head, Func represent the number of true positives. Count for NE represents the number of true negatives. Count for X represents the number of classification errors.**

against other state-of-the-art LLMs: OpenAI o1 [47], Anthropic’s Claude 3.5 Sonnet [5], Google’s Gemini 1.5 Pro [15], and Meta’s Llama 3.1 405B Instruct [21]. Table 7 shows each model’s recall, precision, and F1 score when using the *Base* and the *General_Contextual* prompts on the accessibility error dataset. OpenAI o1 and Gemini both achieved better performance with the *General_Contextual* prompt over the *Base* prompt. Claude’s performance change was negligible, while Llama 3.1’s performance decreased. These variations among LLMs may result from the differences in model steerability [12], or be influenced by our experiment process, which focused on GPT-4o.

7 Discussion

Our results show that ScreenAudit achieves significant coverage of screen reader accessibility errors in mobile apps, with a recall of 69.2%. Our fundamental insight is to simulate *how* a user interacts with apps using a screen reader and directly analyze that experience. We explored the effectiveness of leveraging LLMs for performing these analyses and providing explanations for the detected accessibility errors, revealing promising research directions for future accessibility evaluation tools. In this section,

Table 7: Comparison of LLM performance on the accessibility error dataset with the Base and General_Contextual (GC) prompts. Δ = change.

Model	Precision			Recall			F1		
	Base	GC	Δ	Base	GC	Δ	Base	GC	Δ
GPT-4o (2024-08-06)	.797	.723	-.074	.577	.692	+.115	.669	.707	+.038
OpenAI o1 (2024-09-12)	.606	.675	+.069	.577	.679	+.102	.591	.677	+.086
Claude 3.5 Sonnet (2024-10-22)	.714	.724	+.010	.337	.337	.000	.458	.460	+.002
Gemini 1.5 Pro-002	.582	.617	+.035	.607	.693	+.086	.595	.653	+.058
Llama 3.1 405B Instruct	.686	.644	-.042	.429	.380	-.049	.528	.478	-.050

we discuss how ScreenAudit complements and has the potential of integrating rule-based checkers, how some of our technique relates to heuristic evaluation, and the potential of simulating user testing with LLMs.

7.1 ScreenAudit Complements Existing Accessibility Checkers in Creating Accessible Apps

Our goal in developing ScreenAudit is not to replace current rule-based accessibility checkers with an LLM-based checker. Instead, we recognize the speed, cost-effectiveness, and consistency of these rule-based checkers. We envision integrating the techniques of ScreenAudit into open-source accessibility checkers to combine the strengths of rule-based checkers and those of ScreenAudit. We also hope to explore utilizing the results from rule-based checkers [6, 16, 26] and crawler-based checkers [14, 54, 66] as part of LLM input to provide additional context or to validate the correctness of LLM evaluations and iteratively build a final assessment.

Although we primarily target screen reader accessibility errors, the same accessibility representations support a range of access technologies, such as switch access, keyboard navigation, and voice control. Given that developers have the most comprehensive understanding of their app’s UI layouts and visual design intent, they are best positioned to provide and ensure the accuracy of accessibility metadata. By directly supporting developers, we aim to promote the creation of more accessible apps.

7.2 LLMs and Heuristic Evaluation

In our experiments, we found that providing general accessibility guidance to the LLM yielded better results than providing it with WCAG guidelines. Although this appears counter-intuitive, there has been research that concluded WCAG guidelines only covered a subset of all accessibility issues encountered by users [11, 49]. We also draw parallel to Nielsen’s research on usability heuristics [44], where these heuristics have been shown to effectively guide usability evaluation and approximate the results from user studies. We differ from usability heuristics by tailoring our heuristics to accessibility issues and using LLMs instead of experts or specialists. Nevertheless, our results suggest that adopting a heuristic evaluation approach with LLMs may be beneficial compared to exhaustively listing all possible error types.

7.3 Towards Simulated User Testing with LLMs

Research has studied LLM-supported user behavior simulations for web-marketing campaigns [36] and smartwatch interfaces [71]. However, while LLMs can hypothesize how an interface may respond to users with different abilities, they are unable to test the various ways of interacting with an interface, such as with screen reader, eye tracking, or switch access. Without such information, LLMs can be prone to hallucinations [71]. ScreenAudit grounds LLM analysis of accessibility by directly interacting with an app using a screen reader. Although unlikely to eliminate *all* hallucinations, the approach of interacting with access technologies and analyzing their feedback can be adapted to simulate diverse abilities from users. Contextual information from such interactions should reduce the likelihood of undesired response from the LLMs, enabling fast and cost-effective feedback for designers and developers.

8 Limitations and Future Work

Although ScreenAudit demonstrates promising capabilities in detecting screen reader accessibility errors, we also identify its limitations and suggest directions for future research.

8.1 UI Element Coverage

One of the main constraints of ScreenAudit is that it utilizes TalkBack’s core pipeline to traverse screen reader focusable elements on an app screen. This dependency limits ScreenAudit’s ability to explore and detect elements that are actionable but unfocusable to screen readers. We expect future developments to additionally utilize structured UI information, such as the view hierarchy from the Accessibility API, or extracted UI structures from screenshots [70], so that all elements perceivable by sighted users will be examined.

8.2 Acquiring Additional Context

As experts pointed out in their comments, context understanding is an important yet unsolved problem of all accessibility checkers. Although our adoption of LLMs provides a first step towards addressing this challenge, additional research is still needed. Existing approaches, such as Groundhog [54] and BAGEL [14], can be adopted to specifically address the limitation in assessing UI element reachability and interactivity. From an interaction trace, LLM-supported automated checkers can utilize information in the screens immediately before the current screen to acquire

context. More broadly, when exploring UI elements on screen, an agent-driven approach can be adopted [62, 73] to decide whether to further examine an element and how to perform corresponding accessibility actions.

8.3 Approximating Real-World Usage

Although linear navigation is common for screen reader users, people also adopt different ways of interacting with their devices. As ScreenAudit does not currently support other forms of navigation, accessibility errors are only detected when they present in a linear way. Touch exploration, for example, does not follow such pattern. Although crawler-based accessibility evaluation [22, 23] can cover many app screens, they are less efficient and may miss important features in an app, as they typically explore randomly. When accessibility evaluation experts audit an app, they carefully select screen samples to make sure they cover an app's "core purpose and typical user interaction pathways" [56]. An agent-driven approach can again be explored to simulate these interactions, learn from how developers and auditors navigate and select screen samples by potentially adopting a cognitive walkthrough approach [37], and expose currently undetectable accessibility errors.

8.4 Direct Code Auditing and Refactoring

Although linters exist in current development tools [3, 18, 32], they are rule-based and limited to the set of accessibility errors they are programmed to detect. In this paper, we did not address student developer requests for direct code auditing and refactoring. We intend to explore ways to integrate code-level analysis and support in future iterations of ScreenAudit, such as automatically surfacing code snippets that caused an error, suggesting fixes, and re-evaluating the modified app to confirm repairs.

9 Conclusion

ScreenAudit presents a novel approach to accessibility assessment in mobile apps through the integration of large language models. By directly invoking the screen reader and examining its transcripts, ScreenAudit identifies accessibility errors that are previously undetectable with traditional methods, such as grouping and label quality errors, improving coverage from 31% to 69%. ScreenAudit also complements existing rule-based checkers with nuanced evaluations and contextual insights. Our results uncover opportunities to enhance ScreenAudit by integrating more contextual information, improving interactive context understanding and UI element coverage, and supporting diverse user interactions. Ultimately, this work sets the stage for aligning automated tools more closely with access technology user needs for inclusive mobile experiences, while opening avenues for integrating LLM-driven heuristic evaluations and simulated user testing to address a wider range of accessibility challenges and user abilities.

Acknowledgments

We thank Nigini Oliveira for his generous support in the student developer study, Douglas Goist for his feedback on system design,

and the NSITE team for participant resources. We thank our participants and anonymous reviewers for their feedback.

This work was supported in part by Google and by the University of Washington Center for Research and Education on Accessible Technology and Experiences (CREATE). Any opinions, findings, conclusions or recommendations expressed in our work are those of the authors and do not necessarily reflect those of any sponsor.

References

- [1] Level Access. 2022. 2022 State of Digital Accessibility. <https://www.levelaccess.com/resources/state-of-digital-accessibility-report-2022/>
- [2] Level Access. 2024. Fifth Annual State of Digital Accessibility Report: 2023–2024. <https://www.levelaccess.com/resources/fifth-annual-state-of-digital-accessibility-report-2023-2024/>
- [3] AccessLint. 2024. Automated and continuous web accessibility testing. <https://accesslint.com/>
- [4] Domenico Amalfitano, Vincenzo Riccio, Nicola Amatucci, Vincenzo De Simone, and Anna Rita Fasolino. 2019. Combining Automated GUI Exploration of Android apps with Capture and Replay through Machine Learning. *Information and Software Technology* 105 (2019), 95–116. doi:10.1016/j.infsof.2018.08.007
- [5] Anthropic. 2024. Claude 3.5 Sonnet. <https://www.anthropic.com/claude/sonnet>
- [6] Apple. 2023. Use VoiceOver Recognition on your iPhone or iPad. <https://support.apple.com/en-us/111799>
- [7] Apple. 2024. Accessibility Inspector. <https://developer.apple.com/documentation/accessibility/accessibility-inspector>
- [8] Apple. 2024. Human Interface Guidelines - Accessibility. <https://developer.apple.com/design/human-interface-guidelines/accessibility>
- [9] Virginia Braun and Victoria Clarke. 2006. Using thematic analysis in psychology. *Qualitative research in psychology* 3, 2 (2006), 77–101.
- [10] Michael Crystian Nepomuceno Carvalho, Felipe Silva Dias, Aline Grazielle Silva Reis, and André Pimenta Freire. 2018. Accessibility and usability problems encountered on websites and applications in mobile devices by blind and normal-vision users. In *Proceedings of the 33rd Annual ACM Symposium on Applied Computing* (Pau, France) (SAC '18). Association for Computing Machinery, New York, NY, USA, 2022–2029. doi:10.1145/3167132.3167349
- [11] Andreia R. Casare, Celmar G. da Silva, Paulo S. Martins, and Regina L. O. Moraes. 2016. Usability heuristics and accessibility guidelines: a comparison of heuristic evaluation and WCAG. In *Proceedings of the 31st Annual ACM Symposium on Applied Computing* (Pisa, Italy) (SAC '16). Association for Computing Machinery, New York, NY, USA, 213–215. doi:10.1145/2851613.2851913
- [12] Trenton Chang, Jenna Wiens, Tobias Schnabel, and Adith Swaminathan. 2024. Measuring Steerability in Large Language Models. In *Neurips Safe Generative AI Workshop 2024*. 19 pages. <https://openreview.net/forum?id=y2J5dAqCjW>
- [13] Sen Chen, Chunyang Chen, Lingling Fan, Mingming Fan, Xian Zhan, and Yang Liu. 2022. Accessible or Not? An Empirical Investigation of Android App Accessibility. *IEEE Transactions on Software Engineering* 48, 10 (2022), 3954–3968. doi:10.1109/TSE.2021.3108162
- [14] Paul T. Chiou, Ali S. Alotaibi, and William G.J. Halfond. 2023. BAGEL: An Approach to Automatically Detect Navigation-Based Web Accessibility Barriers for Keyboard Users. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems* (Hamburg, Germany) (CHI '23). Association for Computing Machinery, New York, NY, USA, Article 45, 17 pages. doi:10.1145/3544548.3580749
- [15] Google DeepMind. 2024. Gemini Pro. <https://deepmind.google/technologies/gemini/pro/>
- [16] Deque. 2021. axe: Accessibility Testing Tools and Software. <https://www.deque.com/axe/>
- [17] Deque. 2024. The Automated Accessibility Coverage Report. <https://accessibility.deque.com/hubfs/Accessibility-Coverage-Report.pdf>
- [18] Deque. 2024. axe DevTools Accessibility Linter. <https://www.deque.com/axe/devtools/linter/>
- [19] Deque. 2024. axe DevTools Linter Accessibility Rules. <https://docs.deque.com/linter/4.0.0/en/axe-linter-rules>
- [20] Peitong Duan, Chin-Yi Cheng, Gang Li, Bjoern Hartmann, and Yang Li. 2024. UICrit: Enhancing Automated Design Evaluation with a UI Critique Dataset. In *Proceedings of the 37th Annual ACM Symposium on User Interface Software and Technology* (Pittsburgh, PA, USA) (UIST '24). Association for Computing Machinery, New York, NY, USA, Article 46, 17 pages. doi:10.1145/3654777.3676381
- [21] Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. 2024. The Llama 3 Herd of Models. arXiv:2407.21783 [cs.AI] <https://arxiv.org/abs/2407.21783>
- [22] Marcelo Medeiros Eler, Jose Miguel Rojas, Yan Ge, and Gordon Fraser. 2018. Automated Accessibility Testing of Mobile Apps. In *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*. IEEE Computer

- Society, Los Alamitos, CA, USA, 116–126. doi:10.1109/ICST.2018.00021
- [23] Raymond Fok, Mingyuan Zhong, Anne Spencer Ross, James Fogarty, and Jacob O. Wobbrock. 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* (New Orleans, LA, USA) (CHI '22). Association for Computing Machinery, New York, NY, USA, Article 461, 16 pages. doi:10.1145/3491102.3502143
- [24] Appt Foundation. 2024. Beginners' Guide to Accessibility Testing. <https://appt.org/en/guidelines/beginnersguide-accessibility-testing>
- [25] A. R. GILMOUR, R. D. ANDERSON, and A. L. RAE. 1985. The analysis of binomial data by a generalized linear mixed model. *Biometrika* 72, 3 (12 1985), 593–599. doi:10.1093/biomet/72.3.593 arXiv:https://academic.oup.com/biomet/article-pdf/72/3/593/704911/72-3-593.pdf
- [26] Google. 2024. Accessibility Test Framework for Android. <https://github.com/google/Accessibility-Test-Framework-for-Android>
- [27] Google. 2024. Get started on Android with Talkback. <https://support.google.com/accessibility/android/answer/6283677?hl=en>
- [28] Google. 2024. Get started with Accessibility Scanner. <https://support.google.com/accessibility/android/answer/6376570>
- [29] Google. 2024. Make apps more accessible. <https://developer.android.com/guide/topics/ui/accessibility/apps>
- [30] Google. 2024. Material Design - Accessibility. <https://material.io/design/usability/accessibility.html>
- [31] Google. 2024. TalkBack source code. <https://github.com/google/talkback>
- [32] Google. 2024. Test your app's accessibility. <https://developer.android.com/guide/topics/ui/accessibility/testing>
- [33] Sture Holm. 1979. A Simple Sequentially Rejective Multiple Test Procedure. *Scandinavian Journal of Statistics* 6, 2 (1979), 65–70.
- [34] Forrest Huang, Gang Li, Tao Li, and Yang Li. 2024. Automatic Macro Mining from Interaction Traces at Scale. In *Proceedings of the 2024 CHI Conference on Human Factors in Computing Systems* (Honolulu, HI, USA) (CHI '24). Association for Computing Machinery, New York, NY, USA, Article 1038, 16 pages. doi:10.1145/3613904.3642074
- [35] Tian Huang, Chun Yu, Weinan Shi, Zijian Peng, David Yang, Weiqi Sun, and Yuanchun Shi. 2024. PromptRPA: Generating Robotic Process Automation on Smartphones from Textual Prompts. arXiv:2404.02475 [cs.HC] <https://arxiv.org/abs/2404.02475>
- [36] Akira Kasuga and Ryo Yonetani. 2024. CXSimulator: A User Behavior Simulation using LLM Embeddings for Web-Marketing Campaign Assessment. In *Proceedings of the 33rd ACM International Conference on Information and Knowledge Management* (Boise, ID, USA) (CIKM '24). Association for Computing Machinery, New York, NY, USA, 3817–3821. doi:10.1145/3627673.3679894
- [37] Clayton Lewis, Peter G. Polson, Cathleen Wharton, and John Rieman. 1990. Testing a walkthrough methodology for theory-based design of walk-up-and-use interfaces. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (Seattle, Washington, USA) (CHI '90). Association for Computing Machinery, New York, NY, USA, 235–242. doi:10.1145/97243.97279
- [38] Weixin Liang, Yuhui Zhang, Hancheng Cao, Binglu Wang, Daisy Yi Ding, Xinyu Yang, Kailas Vodrahalli, Siyu He, Daniel Scott Smith, Yian Yin, Daniel A. McFarland, and James Zou. 2024. Can Large Language Models Provide Useful Feedback on Research Papers? A Large-Scale Empirical Analysis. *NEJM AI* 1, 8 (2024), A10a2400196. doi:10.1056/A10a2400196 arXiv:https://ai.nejm.org/doi/pdf/10.1056/A10a2400196
- [39] Zhe Liu, Chunyang Chen, Junjie Wang, Mengzhuo Chen, Boyu Wu, Xing Che, Dandan Wang, and Qing Wang. 2023. Chatting with GPT-3 for Zero-Shot Human-Like Mobile Automated GUI Testing. arXiv:2305.09434 [cs.SE] <https://arxiv.org/abs/2305.09434>
- [40] MagentaA11y. 2024. Native app accessibility checklist. <https://www.magentaa11y.com/native/>
- [41] Delvani Antônio Mateus, Carlos Alberto Silva, Marcelo Medeiros Eler, and André Pimenta Freire. 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* (Diamantina, Brazil) (IHC '20). Association for Computing Machinery, New York, NY, USA, Article 4, 10 pages. doi:10.1145/3424953.3426633
- [42] Forough Mehralian, Navid Salehnamadi, Syed Fatiul Huq, and Sam Malek. 2023. Too Much Accessibility is Harmful! Automated Detection and Analysis of Overly Accessible Elements in Mobile Apps. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering* (Rochester, MI, USA) (ASE '22). Association for Computing Machinery, New York, NY, USA, Article 103, 13 pages. doi:10.1145/3551349.3560424
- [43] Nearform. 2024. Eslint-plugin-react-native-a11y. <https://github.com/FormidableLabs/eslint-plugin-react-native-a11y>
- [44] Jakob Nielsen and Rolf Molich. 1990. Heuristic evaluation of user interfaces. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (Seattle, Washington, USA) (CHI '90). Association for Computing Machinery, New York, NY, USA, 249–256. doi:10.1145/97243.97281
- [45] OpenAI. 2023. GPT-4 Technical Report. arXiv:2303.08774 <http://arxiv.org/abs/2303.08774>
- [46] OpenAI. 2024. GPT-4o. <https://openai.com/index/hello-gpt-4o/>
- [47] OpenAI. 2024. Introducing OpenAI o1. <https://openai.com/o1/>
- [48] Fernanda Pellegrini, Marcelo Anjos, Fabiana Florentin, Bruno Ribeiro, Walter Correia, and Jonysberg Quintino. 2020. How to Prioritize Accessibility in Agile Projects. In *Advances in Usability and User Experience*, Tareq Ahrum and Christianne Falcão (Eds.). Springer International Publishing, Cham, 271–280.
- [49] Christopher Power, André Freire, Helen Petrie, and David Swallow. 2012. Guidelines are only half of the story: accessibility problems encountered by blind users on the web. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (Austin, Texas, USA) (CHI '12). Association for Computing Machinery, New York, NY, USA, 433–442. doi:10.1145/2207676.2207736
- [50] W3C World Wide Web Consortium Recommendation. 2021. Web Content Accessibility Guidelines 2.1. <https://www.w3.org/TR/WCAG21/>
- [51] Brian J. Rosmaita. 2006. Accessibility first! a new approach to web design. In *Proceedings of the 37th SIGCSE Technical Symposium on Computer Science Education* (Houston, Texas, USA) (SIGCSE '06). Association for Computing Machinery, New York, NY, USA, 270–274. doi:10.1145/1121341.1121426
- [52] Anne Spencer Ross, Xiaoyi Zhang, James Fogarty, and Jacob O. Wobbrock. 2018. 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* (Galway, Ireland) (ASSETS '18). Association for Computing Machinery, New York, NY, USA, 119–130. doi:10.1145/3234695.3236364
- [53] Anne Spencer Ross, Xiaoyi Zhang, James Fogarty, and Jacob O. Wobbrock. 2020. An Epidemiology-inspired Large-scale Analysis of Android App Accessibility. *ACM Trans. Access. Comput.* 13, 1, Article 4 (April 2020), 36 pages. doi:10.1145/3348797
- [54] Navid Salehnamadi, Forough Mehralian, and Sam Malek. 2023. Groundhog: An Automated Accessibility Crawler for Mobile Apps. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering* (Rochester, MI, USA) (ASE '22). Association for Computing Machinery, New York, NY, USA, Article 50, 12 pages. doi:10.1145/3551349.3556905
- [55] Camilo Santacruz Abadiano. 2020. Flutter accessibility suggestion tool.
- [56] Leticia Seixas Pereira, Maria Matos, and Carlos Duarte. 2024. Exploring Mobile Device Accessibility: Challenges, Insights, and Recommendations for Evaluation Methodologies. In *Proceedings of the 2024 CHI Conference on Human Factors in Computing Systems* (Honolulu, HI, USA) (CHI '24). Association for Computing Machinery, New York, NY, USA, Article 964, 17 pages. doi:10.1145/3613904.3642526
- [57] Junko Shirogane, Takayuki Kato, Yui Hashimoto, Kenji Tachibana, Hajime Iwata, and Yoshiaki Fukazawa. 2011. Method to Improve Accessibility of Rich Internet Applications. In *Information Quality in E-Health*, Andreas Holzinger and Klaus-Martin Simonc (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 349–365.
- [58] Camila Silva, Marcelo Medeiros Eler, and Gordon Fraser. 2018. 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* (Thessaloniki, Greece) (DSAI '18). Association for Computing Machinery, New York, NY, USA, 286–293. doi:10.1145/3218585.3218673
- [59] C Stephanidis, D Akoumianakis, M Sfyarakis, and A Paramythi. 1998. Universal Accessibility in HCI: Process-oriented Design Guidelines and Tool Requirements. In *Proceedings of the 4th ERCIM Workshop on User Interfaces for All*. Stockholm, Sweden, 15 pages.
- [60] Robert Stratelli, Nan Laird, and James H. Ware. 1984. Random-Effects Models for Serial Observations with Binary Response. *Biometrics* 40, 4 (1984), 961–971. <http://www.jstor.org/stable/2531147>
- [61] Amanda Swearngin, Jason Wu, Xiaoyi Zhang, Esteban Gomez, Jen Coughenour, Rachel Stukenborg, Bhavya Garg, Greg Hughes, Adriana Hilliard, Jeffrey P. Bigham, and Jeffrey Nichols. 2024. Towards Automated Accessibility Report Generation for Mobile Apps. *ACM Trans. Comput.-Hum. Interact.* 31, 4, Article 54 (Sept. 2024), 44 pages. doi:10.1145/3674967
- [62] Maryam Taeb, Amanda Swearngin, Eldon Schoop, Ruijia Cheng, Yue Jiang, and Jeffrey Nichols. 2024. AXNav: Replying Accessibility Tests from Natural Language. In *Proceedings of the CHI Conference on Human Factors in Computing Systems* (Honolulu, HI, USA) (CHI '24). Association for Computing Machinery, New York, NY, USA, Article 962, 16 pages. doi:10.1145/3613904.3642777
- [63] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention Is All You Need. In *Advances in Neural Information Processing Systems* (Long Beach, California, USA), I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (Eds.), Vol. 30. Curran Associates, Inc., Red Hook, NY, USA, 11 pages.
- [64] W3C. 2021. Mobile Accessibility: How WCAG 2.0 and Other W3C/WAI Guidelines Apply to Mobile. <https://www.w3.org/TR/mobile-accessibility-mapping/>
- [65] Junyang Wang, Haiyang Xu, Jiabo Ye, Ming Yan, Weizhou Shen, Ji Zhang, Fei Huang, and Jitao Sang. 2024. Mobile-Agent: Autonomous Multi-Modal Mobile

- Device Agent with Visual Perception. arXiv:2401.16158 [cs.CL] <https://arxiv.org/abs/2401.16158>
- [66] WebAIM. 2024. WAVE Web Accessibility Evaluation Tools. <https://wave.webaim.org/>
- [67] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. 2023. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. arXiv:2201.11903 [cs.CL] <https://arxiv.org/abs/2201.11903>
- [68] Hao Wen, Yuanchun Li, Guohong Liu, Shanhui Zhao, Tao Yu, Toby Jia-Jun Li, Shiqi Jiang, Yunhao Liu, Yaqin Zhang, and Yunxin Liu. 2024. AutoDroid: LLM-powered Task Automation in Android. In *Proceedings of the 30th Annual International Conference on Mobile Computing and Networking* (Washington D.C., DC, USA) (ACM MobiCom '24). Association for Computing Machinery, New York, NY, USA, 543–557. doi:10.1145/3636534.3649379
- [69] Jason Wu, Yi-Hao Peng, Xin Yue Amanda Li, Amanda Swearngin, Jeffrey P Bigham, and Jeffrey Nichols. 2024. UIClip: A Data-driven Model for Assessing User Interface Design. In *Proceedings of the 37th Annual ACM Symposium on User Interface Software and Technology* (Pittsburgh, PA, USA) (UIST '24). Association for Computing Machinery, New York, NY, USA, Article 45, 16 pages. doi:10.1145/3654777.3676408
- [70] Jason Wu, Xiaoyi Zhang, Jeff Nichols, and Jeffrey P Bigham. 2021. Screen Parsing: Towards Reverse Engineering of UI Models from Screenshots. In *The 34th Annual ACM Symposium on User Interface Software and Technology* (Virtual Event, USA) (UIST '21). Association for Computing Machinery, New York, NY, USA, 470–483. doi:10.1145/3472749.3474763
- [71] Wei Xiang, Hanfei Zhu, Suqi Lou, Xinli Chen, Zhenghua Pan, Yuping Jin, Shi Chen, and Lingyun Sun. 2024. SimUser: Generating Usability Feedback by Simulating Various Users Interacting with Mobile Applications. In *Proceedings of the 2024 CHI Conference on Human Factors in Computing Systems* (Honolulu, HI, USA) (CHI '24). Association for Computing Machinery, New York, NY, USA, Article 9, 17 pages. doi:10.1145/3613904.3642481
- [72] Shunguo Yan and P. G. Ramachandran. 2019. The Current Status of Accessibility in Mobile Apps. *ACM Trans. Access. Comput.* 12, 1, Article 3 (feb 2019), 31 pages. doi:10.1145/3300176
- [73] Chi Zhang, Zhao Yang, Jiaxuan Liu, Yucheng Han, Xin Chen, Zebiao Huang, Bin Fu, and Gang Yu. 2023. AppAgent: Multimodal Agents as Smartphone Users. arXiv:2312.13771 [cs.CV] <https://arxiv.org/abs/2312.13771>
- [74] Jialu Zhang, José Pablo Cambronero, Sumit Gulwani, Vu Le, Ruzica Piskac, Gustavo Soares, and Gust Verbruggen. 2024. PyDex: Repairing Bugs in Introductory Python Assignments using LLMs. *Proc. ACM Program. Lang.* 8, OOPSLA1, Article 133 (April 2024), 25 pages. doi:10.1145/3649850
- [75] Mingyuan Zhong, Gang Li, Peggy Chi, and Yang Li. 2021. HelpViz: Automatic Generation of Contextual Visual Mobile Tutorials from Text-Based Instructions. In *The 34th Annual ACM Symposium on User Interface Software and Technology* (Virtual Event, USA) (UIST '21). Association for Computing Machinery, New York, NY, USA, 1144–1153. doi:10.1145/3472749.3474812

A Assignment on Accessibility and Questions Asked in the Assignment

A.1 Assignment Description

We provide the full text of the assignment requirement on Accessibility below. Students had one week to work on the turn in this assignment.

This week's assignment aims to practice the techniques we discussed regarding Accessibility!

Step 1: Choose one of the 3 recent assignments [*assignment numbers*] from this course and examine the accessibility for the app you created.

1.1. Use the Google Accessibility Scanner and notes and screenshots of potential issues.

1.2. Try to access a feature of your App using TalkBack and take notes about difficulties and possible improvements.

We recommend using a physical device. If you only have access to the emulator, you need to download the Android Accessibility Suite (which requires an emulator with Google Play Store installed).

Step 2: Improve the accessibility of your App: refer back to all the tips [*the instructor*] presented in class, and submit a new version OF YOUR CODE with improvements in at least a couple of categories of accessibility mistakes (find them both at the above codelab or the lecture slides.)

By YOUR CODE, we mean you should start this assignment by cloning a previous repository with your own code.

Submission: You will need to make two separate submissions: (TIP: Make sure you familiarize yourself with the rubric in the reflection part before moving forward with the coding part!)

1. Code: Access your Repo at [*link redacted*]. Submit your code.
2. Reflection: Follow the questions in [*assignment link redacted; see questions below*].

A.2 Reflection Questions

Here we provide the full text for the reflection questions asked in the assignment.

I. Issues discovered through the Google Accessibility Scanner

[*Excluded from our study*] Provide screenshots of the Scanner's results on all of your app's screens, merged together as a single image or PDF.

[Q1] How helpful was the Accessibility Scanner? Were there unexpected accessibility issues that you discovered by running the scanner? Any ideas on how the Scanner could be improved?

II. Accessibility issues discovered through TalkBack

[Q2] Recall the most common types of accessibility errors from [the previous lecture on Accessibility]. Apart from the issues discovered by the Accessibility Scanner, what issues have you discovered using TalkBack?

[Q3] Reflecting on your experience using TalkBack to interact with your app, what would you do to improve the user experience?

III. Towards better app accessibility

[Q4] Reflect on the limitations of the Accessibility Scanner and your experience using TalkBack to test accessibility. We do not want accessibility to be an afterthought. What have Android Studio and Compose done to promote/enforce accessibility in developing apps?

[Q5] To promote better app accessibility, what else the development tools could do? List a few aspects that would have helped you make your app more accessible in the first place.

B Qualitative Analysis of Student Developer Responses

B.1 Procedure

We analyzed responses that students provided to reflection questions in an assignment on Accessibility, available in the

previous Appendix section. This assignment was designed and administered as part of the regular course curriculum. Specifically, students were instructed to use the Android Accessibility Scanner to check for accessibility issues first, then use TalkBack to confirm them and look for other accessibility problems. Students were asked to reflect on their experience about analyzing and repairing accessibility issues using the Accessibility Scanner and TalkBack and discuss what else is needed in current developer tools. These reflections consisted of five questions.

Prior to this assignment, students received seven weeks of instruction on Android development with Jetpack Compose² and created six small-scale apps focusing on different aspects of interaction as assignments. The lectures included two lectures on Android Accessibility's common issues and best practices. Students were asked to select one of three apps from previous assignments: (1) a Gallery/Image Showcase app, (2) a Dessert Ordering app, (3) a Voice-to-Image Generation app.

This review was conducted as an independent analysis of student response data stripped of all personally identifiable information. We received formal determination that this project was not human subjects research from the University's IRB. All data were from undergraduate students, however no other demographic information was available. In total, 41 anonymized submissions to the assignment were collected. We excluded submissions where the student indicated that they did not complete all assignment requirements or submitted blank responses, resulting in 36 unique submissions in our analysis.

B.2 Analysis

We analyze the following research questions. RQ0 is analyzed to ensure the consistency and validity of student response data when compared to previous research results. Results for RQ1 and RQ2 are included in Section 3.

RQ0: What are the types of accessibility failures that the Accessibility Scanner *cannot* identify, but can be identified with TalkBack?

RQ1: What are the problems students had while using current developer tools (including Android Studio, Jetpack Compose, Accessibility Scanner, and TalkBack) to check for accessibility failures?

RQ2: What are the features students wanted to see in the developer tools for improving accessibility?

We used standard thematic analysis processes [9] to analyze student reflections that corresponded to each of the three research questions.

B.3 RQ0: Accessibility failures identified by TalkBack only

We developed eight initial codes which revealed three themes: (1) *improper labels and content descriptions*, (2) *inappropriate feedback for functionalities*, (3) *unclear structure or navigation order*, (4) *unexpected TalkBack behavior*. These issues align with those commonly experienced by blind users but undetectable by automated tools in prior work [41]. Specifically, students covered 11 of the 15 frequently encountered problems by blind users.

B.3.1 Improper labels and content descriptions. Our first theme shows that accessibility failures in labels and content descriptions are often ignored by the Accessibility Scanner, identifiable only through screen reader testing. For example, P5 noted missing labels for buttons, P15 noted improperly labeled buttons, P17 noted improper image alt texts, and P21 noted redundant text in icons:

For the two buttons "Buy" and "Don't Buy", the announcements were not helpful, as they just said, "Double tap to activate." It doesn't provide any information on what clicking that element will do. However, in my app, understanding what each button does is very crucial. (P5)

The forward and back buttons use a less than, greater than text to signify scrolling through the carousel. However, "less than" and "greater than" buttons, when announced, might not make sense to the user. (P15)

images were not being described with proper alt text, instead, the screenreader can only say "xxx image". (P17)

I had provided description for icons that ended up being repetitive with the subsequent text. (P21)

These accessibility failures roughly align with the following frequently encountered problems in [41]: (1) *Lack of identification (Buttons)*, (2) *Button functionality is unclear or confusing (Buttons)*, (3) *No textual alternative (Image)*.

B.3.2 Inappropriate feedback for functionalities. Our second theme shows that actions that are unannounced or unclear and inconsistent are only discovered after using TalkBack. For example, P2 found the functionality of the app confusing, P10 found the default label to be inadequate, P31 identified interactions not exposed to screen readers, and P35 noted unannounced states:

The description of the app and images, as well as the buttons are very insufficient, I can't barely understand what this app does. (P2)

I discovered the following issues: 1. share - "Double tap to activate" default click label. (P10)

I have 2 invisible UI interactions: double-clicking the image to like and long pressing to zoom in. These are two interactions that I can't do with TalkBack. Also, their results are invisible to TalkBack or users who would use it. (P31)

The app would not announce the new player upon switching screens, but instead would wait until the user would touch the text which displayed their name. This would be a huge accessibility issue. (P35)

These accessibility failures roughly align with the following frequently encountered problems in [41]: (1) *Inappropriate feedback (Controls, forms and functionality)*, (2) *Unclear or confusing functionality (Controls, forms and functionality)*, (3) *Default presentation of control or form element is not adequate (Controls, forms and functionality)*, (4) *Users cannot make sense of content (Content - meaning)*

²Jetpack Compose: <https://developer.android.com/compose>

B.3.3 Unclear structure or navigation order. Our third theme shows that inconsistent or confusing navigational orders are only discovered after using screen readers. P3 noted a lack of navigational feature in their app, P5 described how overly long interactions are frustrating, and P11 noted inappropriate navigation sequence.

Issues I have discovered using TalkBack was that without the shortcut, I didn't know how to leave the page with a swipe up function since there wasn't a button to click and also the swiping up and down didn't work with the TalkBack on. (P3)

The next issue was that in the bottom part of the app where all the statistics are listed (number of desserts sold, dessert price, etc), there were too many focusable elements, which resulted in too much user interaction. It does not make sense for the user to have to interact once to get a description of the statistic (ex: Number of desserts sold) and then interact again to get the actual value (ex: 10). It would be frustrating and tedious to get through all the statistics for each dessert that is shown, especially since knowing the statistics is helpful for the user to make their decision of whether or not to buy the dessert. (P5)

The navigation skipped the next dessert icon, transitioning from "Next Dessert" to "Items Left". (P11)

These accessibility failures roughly align with the following frequently encountered problems in [41]: (1) *Sequence of interaction is unclear or confusing (Controls, forms and functionality)*, (2) *Expected functionality not present (Controls, forms and functionality)*, (3) *Functionality does not work as expected (Controls, forms and functionality)*.

B.3.4 Unexpected TalkBack behavior. Our fourth theme shows that unexpected (buggy) TalkBack behaviors were encountered. This aligns with the frequently encountered problem “*System problems with assistive technology (System characteristic)*”.

Our analysis covered 11 of the 15 frequently encountered problems by blind users. The following four problems are not covered: (1) *Users inferred that there was functionality where there wasn't (Controls, forms and functionality)*, (2) *Inconsistent Content organization (Content - meaning)*, (3) *Meaning in content is lost (Content - meaning)*, (4) *No textual alternative (Audio, video and multimedia)*.

C Prompt for ScreenAudit

We provide the prompt *General_Contextual* used in ScreenAudit.

You are examining the accessibility of an app based on transcripts of TalkBack. You will see a transcript of what a screen reader user will hear when they use an app. Please analyze if each interaction reflected in this transcript is accessible.

The transcript represents a sequence of TalkBack interactions. Each transcript entry is feedback for one single action.

Basics of Accessibility

We want descriptions to be informative and concise. Consider if the spoken feedback for each element conveys its content or purpose appropriately. Most important information should appear first.

Each description must be unique. That way, when screen reader users encounter a repeated element description, they correctly recognize that the focus is on an element that already had focus earlier. In particular, each item within a view group such as RecyclerView must have a different description. Each description must reflect the content that's unique to a given item, such as the name of a city in a list of locations.

If the app displays several UI elements that form a natural group, such as details of a song or attributes of a message, these elements should be arranged within a container. This way, accessibility services can present the inner elements' content descriptions, one after the other, in a single announcement. This consolidation of related elements helps users of assistive technology discover the information on the screen more efficiently. Because accessibility services announce the inner elements' descriptions in a single utterance, it's important to keep each description as short as possible while still conveying the element's meaning.

If an element in the UI exists only for visual spacing or visual appearance purposes, it should not be announced.

If both type and usage hint are not present, the element is static text. Users will not confuse it with actionable item. This is expected behavior and not an accessibility issue.

Sometimes a screen contains many elements, and it makes sense to put certain actionable elements at the beginning of TalkBack focus order to make them more discoverable, such as "compose new email" button on a page of inbox messages, or a "close" or "return" button.

How to interpret the transcript

- For each transcript entry, think what the user is doing and what the user is experiencing. For example, if the user is swiping right, the user is moving to the next element. If the user is swiping left, the user is moving to the previous element. If the user is double-tapping, the user is activating the element.

- Consider the quality of labels, whether buttons are labeled effectively, whether you could infer what functionalities the interactive elements provide, and overall user experience.

- Consider the transcript without punctuations, as they are not read aloud. Do not ignore text after the punctuations.

- Consider each entry in the context of the entries before and after it, as the functionalities and meanings can be associated.

- Try your best to infer the relationship between entries, and understand the intent of the design. If the inferred intent differs from the presentation in the transcript or that it can be improved, then this may indicate accessibility issue. However, only indicate accessibility issues when they are confusing and impede understanding. Otherwise, indicate as suggestions. When making suggestions, avoid adding text if it can be understood through context.

- Cosmetic issues, such as capitalization, should not be reported.

Your task

Follow these steps to evaluate the accessibility of the transcript:

Step 1 - Look at the big picture. Consider each transcript entry in relation to the elements before and after it. Is this element related to nearby elements? If so, do you see accessibility issues that are related to the presentation order and grouping of elements? Should adjacent elements be grouped together? Many accessibility issues can be resolved if items are properly grouped. Enclose all your output for this step within triple quotes ("").

Step 2 - Based on all guidelines above, consider each transcript entry for its accessibility. What is the intent of each transcript entry? Is the intended meaning conveyed through the transcript? Build your analysis on top of Step 1. If an entry has multiple issues, separate them clearly. If an item is uninformative, consider if it needs to be associated with items. Include possible causes when appropriate. Enclose all your output for this step within triple quotes ("").

Step 3 - Convert your previous analyses into the JSON format specified. Pay attention to index number; be concise; do not remove information. If a same issue is repeated across multiple items, include the issue for EACH occurrence.

```
...
{
  "audit": [
    {
      "index": int,
      "transcript": string, # Copy the full
      transcript here
      "issue": string,      # Describe the issue
      here if you have feedback, otherwise leave
      blank
      "explanation": string, # Explain the issue
      in detail here
      "suggestion": string, # Suggested solution
      here if mentioned, otherwise leave blank
    },
    ...
  ]
}...
```