

TaskAudit: Detecting Functionality Errors in Mobile Apps via Agentic Task Execution

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

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

Davin Win Kyi
Computer Science & Engineering
University of Washington
Seattle, Washington, USA
davin123@cs.washington.edu

Chen Li
Carnegie Mellon University
Pittsburgh, Pennsylvania, USA
meglichen23@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

Accessibility checkers are tools in support of accessible app development, and their use is encouraged by accessibility best practices. However, most current checkers evaluate static or mechanically-generated contexts, failing to capture common accessibility errors impacting mobile app functionality. In this work, we define *functionality* errors as accessibility barriers that only manifest through interaction (i.e., named according to a blend of “functionality” and “accessibility”). We introduce *TaskAudit*, which comprises three components: a Task Generator that constructs interactive tasks from app screens, a Task Executor that uses agents with a screen reader proxy to perform these tasks, and an Accessibility Analyzer that detects and reports accessibility errors by examining interaction traces. Our evaluation on real-world apps shows that TaskAudit detects 48 functionality errors from 54 app screens, compared to between 4 and 20 with existing checkers. Our analysis demonstrates common error patterns that TaskAudit can detect in addition to those from prior work, including label-functionality mismatch, cluttered navigation, and inappropriate feedback.

CCS Concepts

• **Human-centered computing** → **Accessibility systems and tools**; • **Computing methodologies** → **Intelligent agents**.

Keywords

Mobile accessibility, accessibility auditing, generative agents, automated task execution, large language models.

ACM Reference Format:

Mingyuan Zhong, Xia Chen, Davin Win Kyi, Chen Li, James Fogarty, and Jacob O. Wobbrock. 2026. TaskAudit: Detecting Functionality Errors in Mobile Apps via Agentic Task Execution. In *Proceedings of the 2026 CHI Conference on Human Factors in Computing Systems (CHI '26)*, April

13–17, 2026, Barcelona, Spain. ACM, New York, NY, USA, 20 pages. <https://doi.org/10.1145/3772318.3791415>

1 Introduction

Automated accessibility checkers are commonly used by mobile app designers, developers, and quality assurance testers to identify accessibility problems. These tools are recommended by the Web Accessibility Initiative (WAI) [67] and supported by platform owners [4, 24] and various external organizations [14, 42, 71].

However, these checkers only perform static, heuristic assessments of a given screen’s accessibility in an app. As a result, automated checkers have limited error coverage, detecting only four out of 40 problem types identified in a study with blind and low vision participants [9]. Similarly, they detected between 17% and 31% of accessibility errors as identified by experts according to a recent analysis [79]. To improve coverage, crawlers have been developed to support large language models (LLMs) in identifying additional error types not supported by current systems [79]. Generative agents have also been adopted by AXNav to automate accessibility testing in quality assurance [65], but the system still uses heuristic-based error detection.

These advancements, while encouraging, overlook an important aspect of mobile app accessibility: their functionality. We define a *functionality error* as an accessibility barrier that only manifests through interaction, where the static state of the UI appears accessible, but the dynamic behavior fails to meet expectations or WCAG criteria [8] (e.g., an action producing no feedback, a button’s function not matching its label). In this work, we focus on functionality errors that manifest through screen readers. Figure 1 shows examples from five functionality error categories we identified: *Locatability*, *Actionability*, *Label*, *Feedback*, and *Navigation*. We will introduce these in detail below.

Salehnamadi et al. [53, 54] took initial steps toward addressing this gap with accessibility crawlers that detected two types of functionality issues: button locatability and actionability. Although such systems represent a promising direction for detecting and addressing functionality errors, we show that they are prone to generating false positives in real-world apps. In addition, their rule-based approach leaves out several types of functionality errors. In our opinion, effective detection of functionality errors



This work is licensed under a Creative Commons Attribution 4.0 International License. *CHI '26, Barcelona, Spain*

© 2026 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-2278-3/26/04

<https://doi.org/10.1145/3772318.3791415>

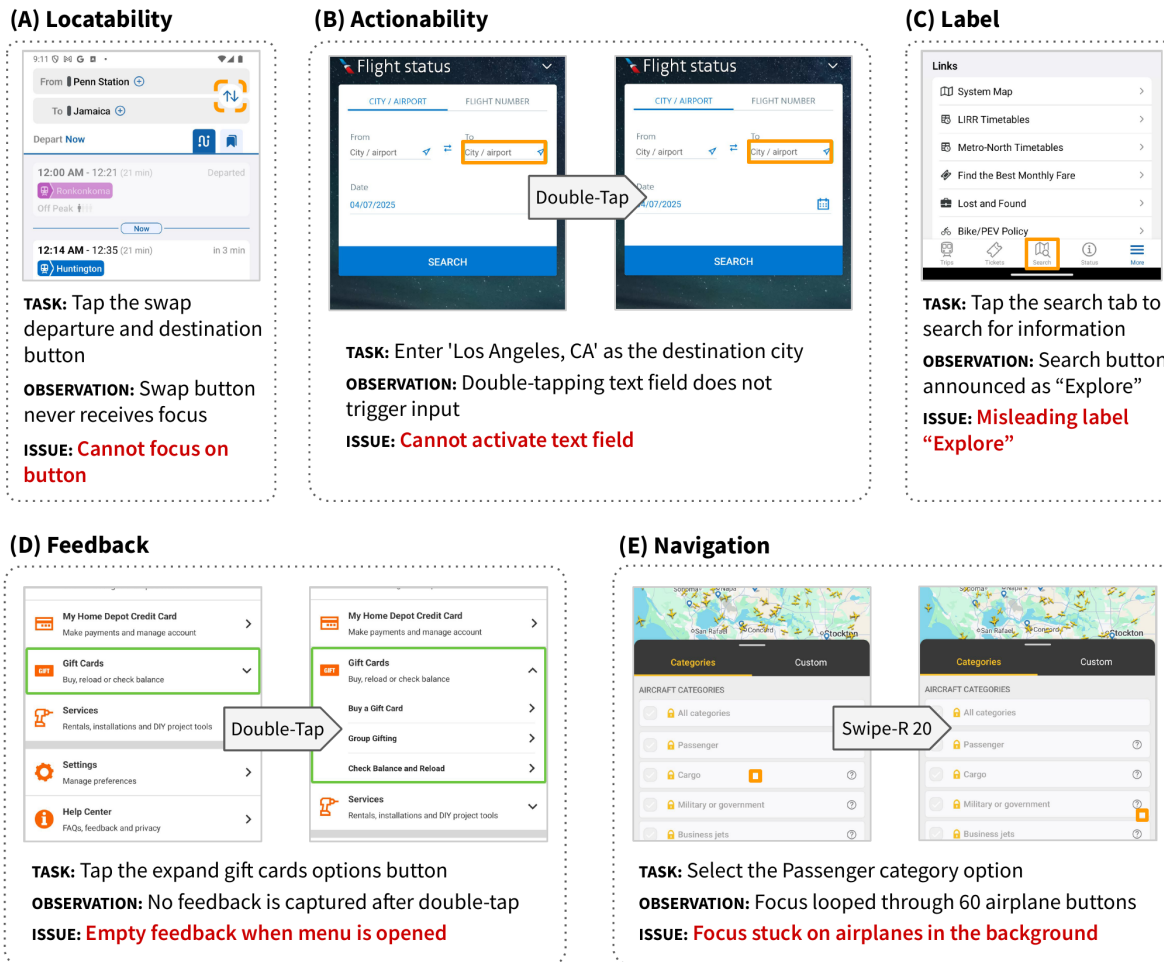


Figure 1: Exemplar errors from the five functionality error categories that TaskAudit detected: *Locatability*, *Actionability*, *Label*, *Feedback*, and *Navigation*. Each example shows the task attempted (TASK), a summary of agent observation (OBSERVATION), and its identified issue (ISSUE). Screenshots show elements with focus (solid outline) or intended element (dashed outline). “Swipe-R 20” means 20 swipe-right gestures to traverse 20 elements.

requires systems that not only adaptively explore applications via accessibility services, but also semantically interpret interface behavior.

As an exploration in expanding automated evaluation capabilities, we present *TaskAudit*, an accessibility evaluation system that focuses on detecting functionality errors through simulated interactions. *TaskAudit* comprises three components: a Task Generator that constructs interactive tasks from app screens, a Task Executor that uses agents with a screen reader proxy to perform these tasks, and an Accessibility Analyzer that detects and reports accessibility errors by examining interaction traces. Unlike Groundhog’s mechanical crawling and direct pixel-wise comparison approach [54], *TaskAudit*’s agents attempt to understand the semantic goal and evaluate the outcome of an interaction sequence. Compared to current agentic analyses such as AXNav [65], *TaskAudit* is not limited to heuristic testing (i.e., loop and missing button detection), but leverages LLMs to directly manipulate and analyze the screen

reader experience. *TaskAudit* does not use vision when operating under a screen reader to accurately simulate that experience. Table 1 outlines the technical differences between these tools.

To evaluate *TaskAudit*’s performance, we explored the following research questions:

RQ1: Can interactive tasks be identified from an app screenshot?

RQ2: Can a multi-agent system perform within-screen tasks successfully via a screen reader proxy?

RQ3: Can *TaskAudit* detect functionality errors effectively?

We conducted three experiments, each addressing one research question with existing and new datasets. Results show that the Task Generator covered 69.4% of interactive tasks in a crowd-labeled dataset (RQ1). When no accessibility errors exist, our task-executing agents performed 96.0% of within-screen tasks successfully via the screen reader proxy (RQ2). Evaluation on 54 unique screens from real-world apps shows that our strategy detected 48 out of 78

Table 1: Overview of mobile accessibility analysis techniques for interactive tasks. *TaskAudit* is the current tool compared to prior tools *Groundhog* and *AXNav*.

	Groundhog [54]	AXNav [65]	TaskAudit
Task Type	Functionality-based	Workflow-based	Functionality-based
Task Creation	Automatic, via view hierarchy	Manual	Automatic, via visual understanding
Task Execution	Mechanical crawling	Agentic execution w/ vision	Agentic execution w/o vision
Assessment	Pixel-wise comparison	Heuristic analysis	LLM analysis

functionality errors, compared to between 4 and 20 with existing checkers (RQ3). Our analysis demonstrates common error patterns that TaskAudit can detect in addition to those from prior work, including label-functionality mismatch, cluttered navigation, and inappropriate feedback.

This research makes the following contributions:

- (1) The TaskAudit system that automatically identifies interactive tasks on a mobile app screen, executes them via a screen reader proxy to test functionality, and analyzes accessibility errors by examining the execution traces.
- (2) A performance evaluation demonstrating the feasibility of multi-agent task execution via a screen reader proxy and the system’s ability to identify functionality errors.
- (3) A qualitative analysis of detected functionality errors, such as label-functionality mismatch, cluttered navigation, and inappropriate feedback.

We envision TaskAudit as being part of the development cycle: it can be run when screens or interaction flows change alongside existing accessibility checks, adding the capability to execute interactions to detect functionality errors. TaskAudit can be integrated into accessibility inspection tools and IDEs, and optionally into continuous integration (CI) pipelines as a targeted check on high-impact flows, where developers can review and triage its reports to ensure accessibility before deployment.

2 Background: Functionality Errors

We illustrate the concept of functionality errors with examples in Figure 1.

Locatability errors. Borrowing from prior work [54], one common functionality error type is Locatability error. For elements that can be invoked using touch interaction, those that cannot be focused using a screen reader have Locatability errors. Figure 1(a) shows one such example, where the swap direction button cannot be focused on.

Actionability errors. Prior work [54] also proposed another error type: Actionability error. Elements that can be focused but cannot be activated (i.e., clicked) by a screen reader have Actionability errors. As shown in Figure 1(b), the input field does not respond to activation with a double-tap.

Locatability and Actionability errors are both covered by WCAG 2.1.1 “Keyboard” [8], which requires all functionality to be operated through a keyboard-compatible interface (e.g., a keyboard, a screen reader, a switch device).

Label errors. We identify a Label error as a semantic mismatch between the element’s label and its actual functionality. Figure 1(c)

shows one such example, where the “search” tab is incorrectly labeled as “explore” and thus can mislead a screen reader user who is looking for the search feature. Label errors are covered under WCAG 4.1.2 “Name, Role, Value” and can be relevant to WCAG 1.1.1 “Non-text Content” depending on implementation [8]. While some Label errors may be detectable with static accessibility checking (e.g., a missing label), many potential issues cannot be definitively determined without examining functionality (e.g., a pictogram that may be interpreted in different ways), and therefore we included this category for completeness.

Feedback errors. This type of error relates to feedback quality after an action. It occurs when actionable items, such as input fields, do not trigger an announcement after activation, or trigger feedback that is uninformative. Figure 1(d) shows one such example, where activating a dropdown menu resulted in visual changes, but did not generate any audible feedback. This can lead to confusion for screen reader users regarding the current state of the app. These are covered by WCAG 3.2.2 “On Input” and WCAG 4.1.3 “Status Messages” [8]. Compared to Label errors that concern announcement quality *before* interactions, Feedback errors relate to quality *after* interactions.

Navigation errors. Navigation errors occur when the structure of interactive elements impedes efficient movement through the interface using a screen reader. For example, in Figure 1(e), a large number of focusable airplane elements clutter the page, making it almost impossible to reach the main menu. This type of error can lead to frustration and inefficiency during sequential navigation. Navigation errors are addressed by WCAG 1.3.1 “Info and Relationships,” which requires content structure be programmatically determinable, and WCAG 2.4.1 “Bypass Blocks,” which requires mechanisms to skip repetitive content [8].

3 Related Work

Our research builds on related work on (1) manual accessibility inspection practices, (2) automatic accessibility evaluation techniques, and (3) LLM-driven task execution and its applications in accessibility evaluation.

3.1 Accessibility Inspection Practices

The Web Content Accessibility Guidelines (WCAG) is an international standard for web content accessibility [8]. Although developed for the web, many of the WCAG requirements apply to mobile apps, and specific guidelines have been created for developers of mobile platforms [5, 25, 41] and for the accessibility inspection of mobile apps [22, 39, 73].

In addition to guidelines, user testing is critical to ensure app accessibility. In an assessment of WCAG 2.0, Rømen and Svanæs found that these guidelines only covered 32% of accessibility problems experienced by users with disabilities [50]. In another study comparing the more recent WCAG 2.1, Mateus et al. found that users encountered 36 types of accessibility problems, while the guidelines only covered 22 (61%) [40].

User testing is not always feasible during accessibility assessments due to practical limitations. Evaluation protocols, such as WCAG-EM [19] and EN 301 549 [20], provide frameworks for systematically assessing websites and apps' accessibility. However, their instructions were found to be nonspecific, especially when evaluating mobile apps [58]. In the same study, automated tools were also found to be limited in scope and availability. Evaluators therefore relied on manual examination, developing their own methodologies and using tools only for specific tasks such as checking color contrast. Similarly, Pereira and Duarte reported that practitioners found automated checkers to be limited and relied on manual testing for accurate assessments [57].

3.2 Automated Accessibility Evaluation

To support accessible app development and evaluation, a variety of automated accessibility evaluation tools have been created. While these tools have been found to be under-utilized in current accessibility inspection practices [57, 58], they have the potential to greatly improve the efficiency and coverage of accessibility testing.

Linters, such as those for Android Studio [26], React Native [43], GitHub [1], and Deque's Axe Accessibility Linter [15] provide code-level accessibility checking for a limited set of issues during development. To evaluate accessibility in a more realistic setting and assess dynamically generated components, runtime accessibility checkers analyze a static snapshot of a webpage or an app screen to check for predefined errors. Examples include Google's Accessibility Scanner [24], Apple's Accessibility Inspector [4], Deque's Axe [14], WAVE [71], and Google Chrome's Lighthouse [28].

To automate accessibility checking, crawlers programmatically invoke UI controls and expand the coverage of app states and functionalities. MATE randomly activates elements and applies heuristic checking similar to runtime checkers [18]. To detect functionality errors, Groundhog [54] and BAGEL [11] implement crawlers to simulate the activation of UI controls or navigation within a screen. These automated checkers can overwhelm developers with redundant recommendations. Swearngin et al. [64] addressed this by introducing a screen grouping model that deduplicates and summarizes unique accessibility issues in apps.

Recent large-scale mobile accessibility assessments adopt crawling techniques to capture app snapshots for analyses [10, 21, 75]. Beyond automated crawlers, Ross et al. adopt an epidemiological lens on mobile accessibility [52]. In particular, they examine image-based button labeling practices across thousands of Android apps [51] and measure the prevalence and distribution of accessibility problems in mobile ecosystems. These studies characterize how accessibility errors emerge and persist at scale, and they motivate techniques that provide deeper analyses within individual apps beyond static analysis.

Another line of work expands accessibility evaluation coverage with AI. López-Gil et al. [37] explored automating certain WCAG success criteria using large language models (LLMs) that previously required manual checking. HindDroid [36] generates hints for inputs with an implication of supporting accessible app development and testing. ScreenAudit [79] crawls a screen for screen reader transcripts and generates an accessibility report based on an LLM analysis. Recent work has also sought to provide code-level support, using LLMs, to detect and correct web accessibility errors including missing alt-text, improper headings, inadequate color contrast, and non-descriptive link descriptions [13, 29, 46].

Currently, automated tools remain limited in detecting functionality errors [40, 57, 58]. Our work combines crawler-based and AI-driven techniques to improve coverage of these errors.

3.3 LLM-Driven Task Execution

The core technology that enabled TaskAudit is automated task execution by LLM-driven agents, which are built upon recent advances in UI understanding and multi-agent systems. Here, we introduce current UI understanding techniques, automated task execution, and their applications in accessibility evaluation.

3.3.1 UI Understanding. To acquire semantic understandings of a given UI, machine learning models have been developed to detect common UI components [38, 77], predict whether they are interactive [56, 63], and generate captions for each component [35, 38]. On the screen level, models such as Screen Parsing [74] and Graph4GUI [31] aim to recover structural information between UI components. Screen2Vec [34] enables robust screen comparisons and generalized task flow learning. Screen2Words [68] generates textual screen summaries using a multi-modal approach. Recent advances in vision language models have led to OmniParser [38] and ScreenAI [7], which combine many of the above capabilities to support screen annotation and question answering.

Much of the UI understanding work laid the foundation for automated task execution. However, in our system, we rely solely on screen reader outputs for task execution, without using screenshots. Instead, we leverage OmniParser [38] to support the generation of interactive tasks to be evaluated by TaskAudit.

3.3.2 Automated Task Execution. Recent studies have utilized learning-based agents that leverage semantic UI understandings to automate task execution. Agents rely on large language models (LLMs) to interpret interfaces and execute user instructions, such as navigating web and mobile interfaces [30, 62, 72, 80]. Other systems further integrate natural language prompts from users to guide task completion [66], or use pixel-based recognition techniques to infer UI semantics necessary for task execution [61]. Multi-agent implementations distribute responsibilities across individual agent modules dedicated to planning, decision-making, and reflection or monitoring: Mobile-Agent-V2 [69] separates planning, action, and reflection tasks among dedicated agents. Prompt2Task [30] decomposes the task execution pipeline into specialized agents handling prompt analysis, information retrieval, task parsing, grounding to UI actions, and task assessment. VisionTasker [62] integrates UI understanding, task planning, and execution within modular sub-components of a single agent. Common themes across

these approaches include the distribution of responsibilities such as semantic interpretation, interaction planning, and task execution. In TaskAudit, we adopt a similar multi-agent design but adapt it to operate via a screen reader, without “seeing” the screen.

3.3.3 Application in Accessibility Evaluation. Traditional accessibility testing tools such as GroundHog [54] and FeedLack [32] offer thorough identification of certain accessibility errors by mechanically crawling app states or analyzing source code to detect missing user interface feedback. Without semantic understanding, they lack coverage of nuanced functionality issues. AXNav [65] employs multiple agents (planner, action, evaluation) that execute pre-defined tasks to assess UI accessibility. However, AXNav operates the app using conventional controls and does not directly evaluate the user experience provided by accessibility services. ScreenAudit [79] uses an LLM to analyze TalkBack transcripts with screen metadata, but primarily relies on mechanical crawling without directly testing functionality.

Building upon multi-agent task execution approaches [65, 69] and LLM-driven accessibility checkers [36, 37, 79], we leverage LLM-driven task execution agents to directly evaluate functionality through a screen reader proxy. Unlike existing tools that mechanically crawl interfaces or examine metadata [32, 54, 79], our approach targets semantic-dependent interactions, surfacing errors tied to the actual screen reader user experience.

4 The TaskAudit System

Informed by our understanding of functionality errors, we created TaskAudit to explore detecting such errors in mobile apps through automated task execution.

As shown in Figure 2, the TaskAudit system consists of three components: a *Task Generator* that constructs interactive tasks from a mobile app screen, a *Task Executor* that performs each task through a screen reader proxy using custom task-executing agents, and an *Accessibility Analyzer* that examines collected task execution traces to determine and report any errors. We adopted OpenAI’s GPT-4o [44] for TaskAudit, but expect the system design to be adaptable to other language models.

4.1 Task Generator

The Task Generator constructs potential tasks from a screenshot captured on the screen under evaluation. As no consistent guidelines currently exist for mobile accessibility evaluation [59], we reference the Website Accessibility Conformance Evaluation Methodology (WCAG-EM) 1.0 document published by the W3C [19]. In particular, Requirement 4.b on accessibility auditing is the most relevant and involves checking “functionality, entering data, notifications, and other interaction.” Therefore, we designed the Task Generator to generate a *task specification* for each interactive element appearing in the screenshot. We chose a screenshot-based approach that allows the system to identify interactive elements that are rendered visually but not properly exposed in the accessibility tree, as we will demonstrate through Groundhog’s results in Section 5.3 and illustrated in Figure 6. This also allows the Task Generator to function where direct access to an app’s view hierarchy is not available, reflecting many real-world quality assurance

workflows (e.g., an app may hide or obfuscate the view hierarchy or use a framework that does not properly expose metadata).

To ensure reliable and consistent task generation, the Task Generator adopts a three-step process, with the first two steps in line with recent work in UI understanding [7, 38]. Figure 3 illustrates the process with an example from an app that we examined.

- (1) *Identifying interactive element candidates* from a screenshot using Paddle OCR [33] for text extraction and OmniParser’s icon detection model [38].
- (2) *Captioning.* Candidates are individually cropped from the screenshot with surrounding context and sent to GPT-4o for captioning. Specifically, we prompt the model to output a descriptive name and element category (i.e., *information*, *action*, *input*, and *navigation*) for each element. These categories are derived from the WCAG-EM 1.0 [19], and Apple [6] and Google’s [27] classifications of UI components. Figure 3 includes the descriptive names overlaid over their respective elements.
- (3) *Task specification generation.* The Task Generator filters out informational elements from the candidate list. The resulting list of interactive elements and an annotated screenshot (i.e., each element is indexed and highlighted in its bounding box) are sent to GPT-4o to generate specifications of interactive tasks. For each task, the model is prompted to generate a description with necessary details (e.g., time, location, name), identify a corresponding element and any prerequisites, and propose a task success criterion for the Task Executor and the Accessibility Analyzer to determine if a task is successful.

4.2 Task Executor

Given a task description from the Task Generator, the Task Executor attempts to perform that task through a *screen reader proxy*, using a multi-agent execution workflow. The goal is to simulate how a person using screen reader would interact with the app, and collect screen reader output for accessibility evaluation.

4.2.1 Screen Reader Proxy. A screen reader serves as a *proxy* between an app’s original interface and the manifest interface that a blind person perceives [78], converting a graphical user interface to a mostly linear interface announced by voice. In our system, the screen reader proxy links the task-executing agents and the *environment*, i.e., the app interface manifested through the screen reader. We adopt a modified version of Android’s TalkBack [79] that allows direct capture of transcripts from its internal pipeline before they are announced.

The screen reader proxy accepts one or multiple swipe gestures (for navigating between elements), or a single double-tap gesture (for clicking on an element) in each interaction. These gestures are consistent with other screen readers, such as VoiceOver [3]. We also support standard Android gestures, including going back and text entry. The gestures are invoked using our modified TalkBack service on an Android emulator, and the screen reader proxy exports the resulting transcripts using the Android logging system, which are then forwarded to the Task Executor via a socket connection. Other data, such as screenshots and accessibility metadata, are collected separately and saved for later analysis. We did not support touch exploration in line with prior work [53, 54, 79] due to a

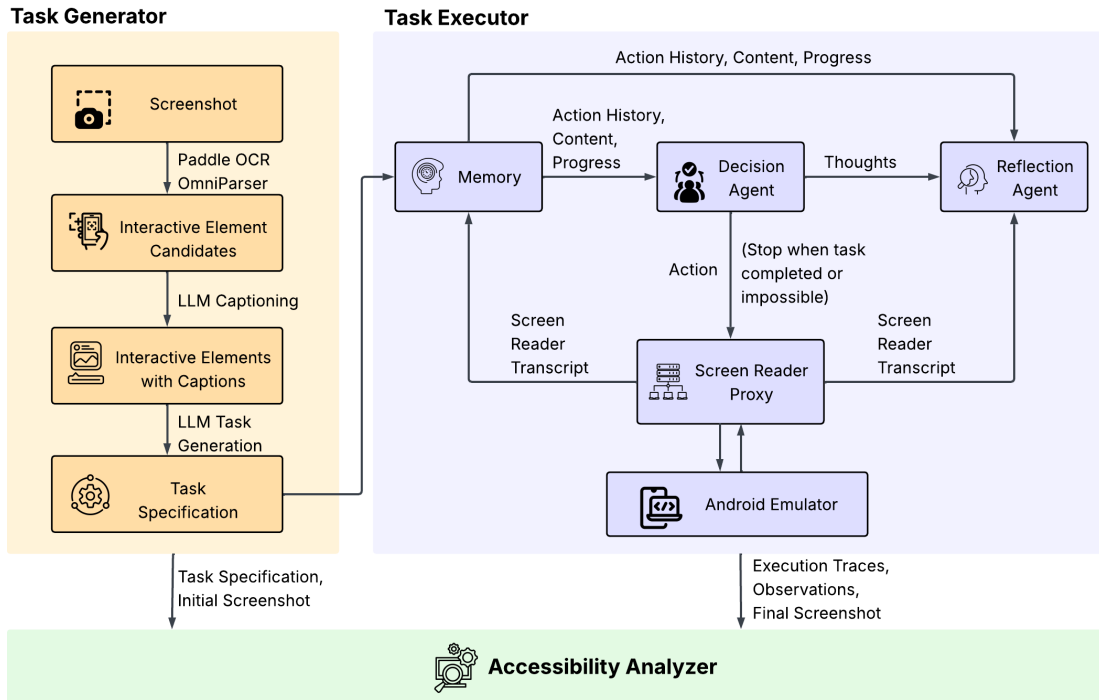


Figure 2: TaskAudit’s system architecture, comprising a Task Generator, Task Executor, and Accessibility Analyzer. The Task Generator produces task specifications from an initial screenshot. The Task Executor accepts a task and utilizes a multi-agent approach to execute it via a screen reader proxy on an Android emulator. Execution traces and related metadata are sent to the Accessibility Analyzer for evaluation.

lack of behavioral data on touch exploration from screen reader users, accessibility guidelines requiring full keyboard navigation support [8], and the exploratory nature of our work. We recognize this as an opportunity for future research.

4.2.2 Multi-Agent Execution Workflow. We implement the Task Executor as a multi-agent system supported by LLMs. Inspired by recent task-executing agents [30, 62, 69], we use distinct Decision and Reflection agents so the system can explicitly assess each action’s outcome before planning the next step, which is particularly important because the agents do not see screenshots. At each step, the Decision agent proposes an action, the proxy executes it and returns transcripts, and the Reflection agent evaluates the outcome, updating a shared Memory store. Figure 4 illustrates this loop with a task execution example. Prompts are available in Appendix A.

Memory. The Memory mechanism stores task-relevant information of the user’s goal, environment setup, and execution history of previous steps to help execute the task.

Decision Agent. The Decision agent generates actions based on the task description, current progress, transcripts, and prior reflections. It operates within a predefined *operation space* of four environment actions (swipe, double-tap, back, type) and three meta-actions (wait, mark task as complete, mark task as impossible).

Reflection Agent. The Reflection agent assesses whether each action’s outcome matches expectations given the prior state

and transcripts and, when it fails, identifies likely reasons. This reflection process follows prior work on task-executing agents [69]. Reflections are stored step-by-step to support later analysis but are not themselves treated as accessibility judgments.

4.2.3 Exploration and Navigation Details. We run the Task Executor on top of Android emulator snapshots [17] and, for each screen under analysis, capture and restore an Android Virtual Device (AVD) snapshot before executing each task to ensure consistent execution.

Because a screen typically does not produce transcripts until interacted with, the Decision agent must first explore the environment and identify relevant elements, so we explicitly prompt it to perform up to $k = 20$ “swipe right” actions (up to three attempts) until it detects an element related to the task, allowing a maximum of 60 elements to be explored.

In some cases, TalkBack’s list-handling logic makes the number of swipes needed to revisit elements inconsistent with the number of announced elements, which can confuse the agent. We mitigate this by allowing the agent to specify an object in the original transcript, and the screen reader proxy programmatically navigates until that element is matching based on string similarity.

4.3 Accessibility Analyzer

The Accessibility Analyzer processes task execution traces to interpret the outcome of each interaction and identify possible

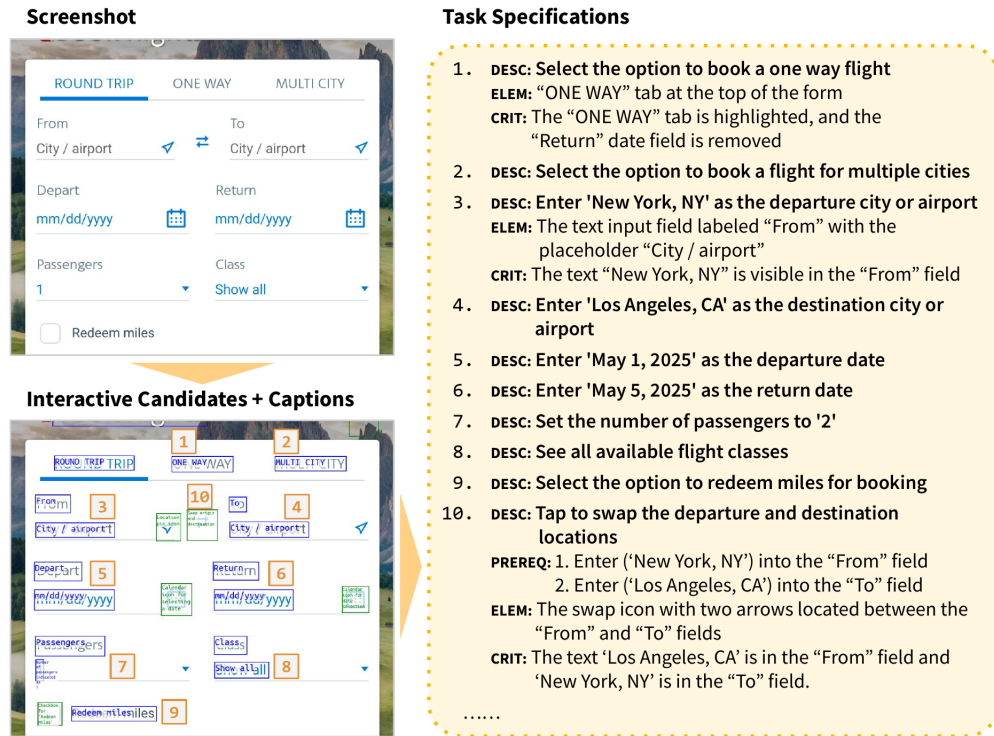


Figure 3: TaskAudit’s Task Generator parses the interactive elements from a screenshot and generates task specifications. For brevity, we only present selected examples from the specifications, including DESC for description, PREREQ for prerequisites, ELEM for element, and CRIT for success criterion.

functionality errors. The analyzer performs a step-by-step evaluation of the execution trace. Because a task may be completed before all planned actions are executed, the overall Task Specification is checked after every action. Central to the analysis is the execution trace collected during task execution. Each trace includes a list of executed steps which contains the following information corresponding to each action performed via the screen reader proxy:

- (1) Summary of execution progress before each action.
- (2) Thoughts and reasoning behind each action proposed.
- (3) Observation (screen reader transcripts) after each action.
- (4) Reflection of whether each action met expectation.

The analysis is conducted in a two-stage process for each step in the trace, which we illustrate with an example in Figure 5. Prompts are available in Appendix A.3.

First, for each action performed, we use GPT-4o to determine its outcome. The model’s goal is to check if the overall task’s success criterion has been met during any of the steps, allowing for early task completion. The model also validate the immediate action’s accessibility using a set of action-specific heuristics. These heuristics include verifying that a keyboard appears after tapping a text field, that the screen content changes after a scroll action, and that new elements are announced after activating a menu.

Second, if an action is marked as a FAILURE, its trace and the original Task Specification are further analyzed using a specialized prompt. This model uses a structured, Chain-of-Thought style

reasoning process to determine if the failure was caused by accessibility errors in the app, or that no accessibility error has occurred if the failure is attributable to other factors, such as an agent execution error or a change in the app’s state. This process ensures that every action is validated against both its immediate expected outcome and the task’s final goal, and that failures are reviewed before being flagged as accessibility errors.

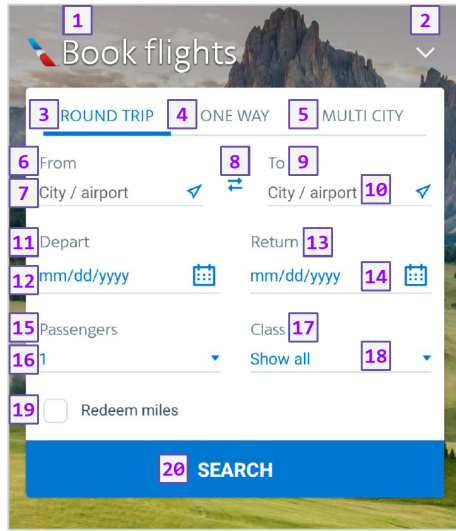
5 Performance Evaluation

TaskAudit adapts and combines many components towards implementing automated assessments of functionality errors. While some of these techniques have been published and evaluated in prior work, here we systematically evaluate them in the context of detecting functionality errors.

We conducted experiments to assess TaskAudit’s performance in three respects: (1) its ability of identifying tasks on a screen, (2) its ability of executing tasks through a screen reader proxy, when no accessibility errors are involved, and (3) its ability of detecting functionality errors. These first two assessments correspond to the Task Generator and the Task Executor respectively and the third experiment evaluates the entire TaskAudit pipeline.

To support these experiments, we adapted existing datasets and created our own datasets as needed. We report results in similar tasks from prior work where available.

Task: Enter 'Los Angeles, CA' as the destination city or airport



(a) Action: { 'action_type': 'SWIPE_RIGHT', 'repetitions': '20' }

Transcripts

- 1 Book flights. keyboard hidden. Book flights. Double-tap to activate.
- 2 Unlabelled, Button. Double-tap to activate, Labels available, use Tap with 3 fingers to view.
- 3 selected, Round trip, Tab, 1 of 3, In list, 3 items.
- 4 One way, Tab, 2 of 3. Double-tap to select.
- 5 Multi City, Tab, 3 of 3. Double-tap to select.
- 6 Opens search, Out of list.
- 7 City / airport, Edit box. Double-tap to edit text, Double-tap and hold to long press.
- 8 reverse city button, Button. Double-tap to activate.
- 9 ... 20 SEARCH, Button. Double-tap to activate.

Reflection: The operation thought was to explore the screen by swiping right 20 times to gather information about the available elements and identify the input field where we can enter the destination. The screen reader transcript after the operation provides detailed information about the elements on the screen, including various buttons and edit boxes. ...

(b) Action: { 'action_type': 'SWIPE_LEFT', 'repetitions': '10' }

Transcripts: 19 18 ... 11 10 City / airport, Edit box. Double-tap to edit text, Double-tap and hold to long press.

Reflection: ... The screen reader transcript after the operation shows that the focus is now on "City / airport, Edit box" at index 10. This indicates that the operation action successfully navigated to the desired input field. ...

(c) Action: { 'action_type': 'DOUBLE_TAP' }

Transcripts: 10 <no_feedback>

Reflection: ... the screen reader transcript after the operation indicates '<no_feedback>', and the keyboard status remains unchanged, as it has not been activated. This suggests that the double-tap gesture did not result in the expected outcome ...

(d) Action: { 'action_type': 'SWIPE_RIGHT', 'repetitions': '20', 'description': 'Swipe right 20 times to explore the screen and gather more information about the elements present, aiming to find the correct input field or any feedback' }



Figure 4: TaskAudit's Task Executor executes a task which ultimately leads to a functionality error. In each step, the Decision agent generates an action, transcripts are collected through the screen reader proxy, and the Reflection agent assesses the outcome. Throughout the process, the accessibility focus changes. The last focused element in each step is highlighted in the transcripts. This example includes an input control that does not respond to activation (Step c), which led the Decision agent to attempt other pathways (Step d). However, this proved to be impossible and the execution ultimately terminated after a loop was detected (not shown for brevity).

5.1 Task Identification

5.1.1 Method. We evaluate TaskAudit's task identification performance by comparing the interactive elements identified by the Task Generator against human annotated datasets from prior work [35, 56]. Specifically, we ran the Task Generator over a human-annotated screenshot dataset containing UI element tappability and captions. An element was considered detected if over 80% of the bounding box extracted by the Task Generator overlaps with a tappable element from the dataset. Two authors independently

compared system-generated with crowdsourced captions, flagging captions as inconsistent if they were off-topic or had missing information. The authors discussed any disagreements until they reached a consensus.

We did not compare with GroundHog [54] or other accessibility checkers in this experiment, because they require analyses of running apps which we cannot perform on our annotated dataset. However, we provide a detailed comparison of existing techniques in Section 5.3.

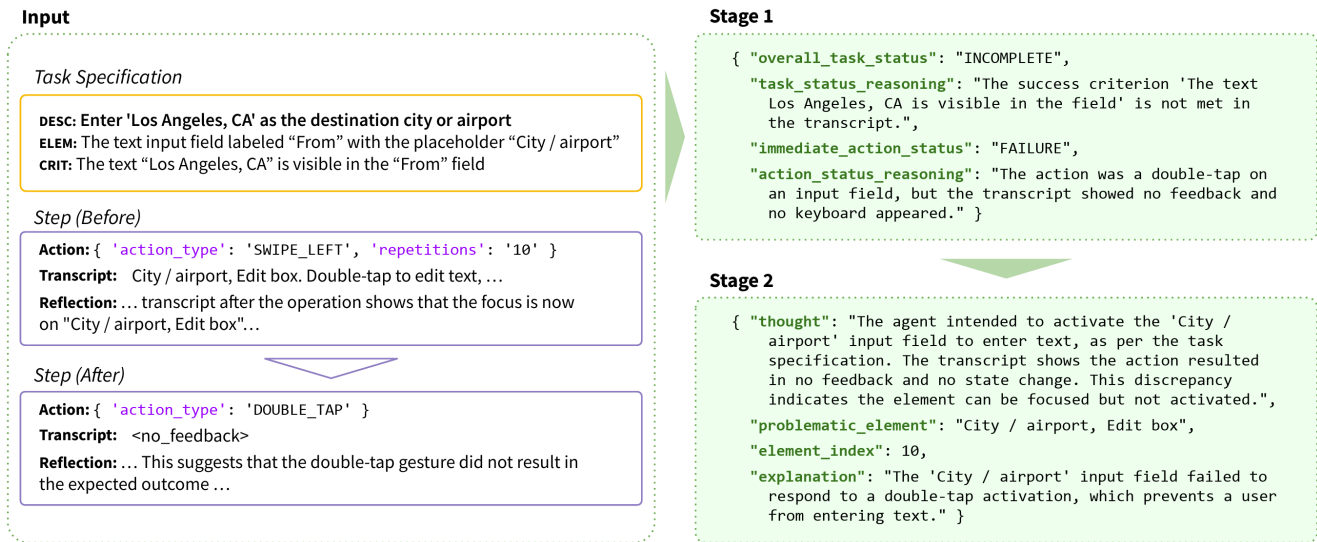


Figure 5: TaskAudit’s Accessibility Analyzer processes one step in the execution trace from Figure 4 [(b) to (c)]. It uses a two-stage prompting process to identify the root cause and reports an error.

5.1.2 *Datasets.* We used three datasets in this experiment:

- (1) RICO [12], a large-scale dataset of mobile app screenshots.
- (2) A crowdsourced *UI tappability* dataset [56], where each UI element is labeled “tappable” or “not tappable.”
- (3) A crowdsourced *widget captioning* dataset [35], where each UI element has a caption about its functionality.

The UI tappability and widget captioning datasets are random subsets of the RICO dataset. The intersection of the two datasets contained 2,041 UI elements from 1,086 screens. Each element in the widget captioning dataset had two or three human-generated captions. Two of the authors independently reviewed and manually removed 815 elements whose captions appear to describe different functionalities (e.g., one is “filter results” and the other is “menu bar”). These inconsistencies are likely due to screenshot inconsistencies or crowdworker errors from the original dataset. The final dataset contained 1,226 unique elements from 813 screens.

5.1.3 *Results.* Our Task Generator detected 911 of the 1,226 tappable UI elements (74.3%). Among these, 69.5% of all buttons (643 of 925), 73.0% of selection inputs (46 of 63), and 93.3% of text inputs (222 of 238) were detected. We did not calculate precision, as no available dataset has complete labels for all tappable elements. False positives will have a limited impact on the final assessment, as the Task Executor identifies such elements as non-actionable during its run. As a reference, Zhang et al. [77] developed a UI element detection model that achieved a mean Average Precision (mAP) of 71.3% on a proprietary dataset. For generated captions, 851 (93.4%) were consistent with the crowdsourced captions. The overall actionable element identification rate was 69.4% (851 of 1,226).

5.2 Task Execution via Screen Reader Proxy

5.2.1 *Method.* We measured the task execution success rate via screen reader proxy by generating and executing tasks on apps known to be accessible. Our test set included apps with no known missing-label errors from Fok et al. [21] and all Google-developed apps from the Android in the Wild dataset [49], excluding Chrome. From this set, we manually examined 26 apps to identify 36 screens with no apparent functionality errors.

We ran our Task Generator on these screens to produce and execute 299 unique tasks via the screen reader proxy. A task was considered successful if the agent correctly focused on a non-actionable element, activated an actionable element, or entered data into an input field. Since each task only involved interacting with or altering the state of one element, we expected the task success rate to be high.

5.2.2 *Results.* Our agents successfully executed 287 of the 299 tasks, achieving a 96.0% success rate. Most failures (7 tasks, 2.3%) were caused by the LLM generating incomplete task descriptions, such as specifying a search action without providing a query text. The remaining 5 failures (1.7%) were due to content refreshing on the screen between task generation and execution, a delay our system mitigates but cannot entirely eliminate. The high success rate on these accessible screens demonstrates the robustness of our agentic task execution.

5.3 Functionality Error Detection

5.3.1 *Method.* We evaluated whether TaskAudit is capable of detecting and reporting functionality errors and compared its performance against three state-of-the-art accessibility checkers: the Android Accessibility Scanner [24], ScreenAudit [79], and Groundhog [54]. We evaluated them on 54 unique screens from 14 apps. Three of the authors independently examined all generated

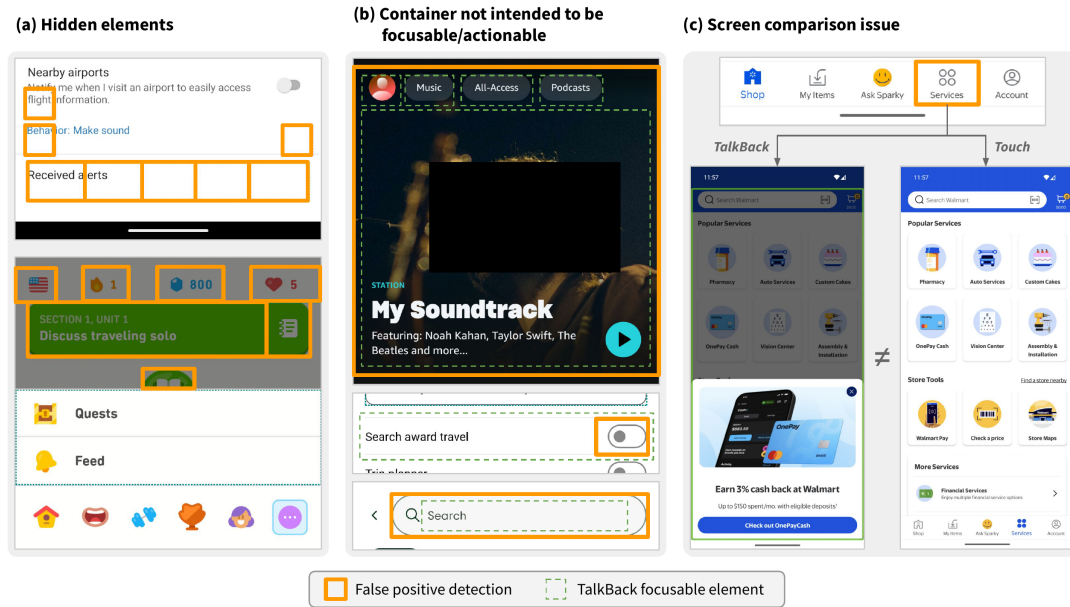


Figure 6: Examples of Groundhog’s false positive detections highlighted in orange. (a) Elements covered by an overlay are incorrectly flagged. (b) Containers are incorrectly flagged when alternative elements provide full functionality and are focusable. Making the containers focusable would instead cause duplicates. (c) An Actionability error being incorrectly flagged because of a pop-up ad captured during screen comparison.

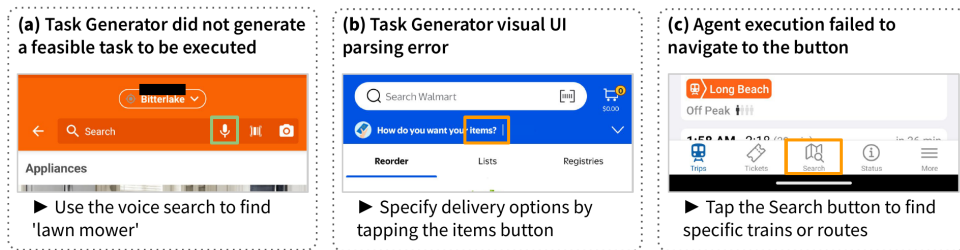


Figure 7: Examples of TaskAudit’s false positive detections with corresponding task descriptions below. (a) Task Generator created an impossible task as the Executor has no speech capability. (b) Task Generator made a visual parsing error. (c) The Executor failed to navigate to an accessible button.

Table 2: Performance comparison of accessibility checkers in detecting functionality errors.

Tool	Precision	Recall	F1
Accessibility Scanner	- *	0.052	-
ScreenAudit	- *	0.130	-
Groundhog	0.142	0.256	0.183
TaskAudit	0.676	0.615	0.644

* We did not calculate precision for Accessibility Scanner and ScreenAudit because (1) they are not designed for detecting functionality errors and that (2) they reported many other aspects of accessibility issues not covered by our dataset. Separating these from functionality errors will inherently be subjective and we defer to related work [79] for a more detailed comparison on these tools.

reports against the observations recorded in our functionality dataset, recording their outcomes.

5.3.2 Dataset. We created a functionality error dataset of 78 error instances, building on top of a recently published accessibility error dataset that included 14 apps with one unique screen per app [79]. Their dataset included 12 instances related to “functionality,” covering elements’ keyboard access, focus, input, and feedback. To cover a more diverse set of screens and functionalities, we included one additional screen under each navigational tab from each app. We included these additional screens because (1) top-level tabs are usually on critical paths of app functionality, and (2) they cover a diverse set of interactive components, such as buttons, text inputs, drop-downs, and selection elements. In total, our dataset contained 54 unique screens. As these apps were randomly selected according

Table 3: Breakdown of causes for incorrect outcomes for Groundhog and TaskAudit.

Tool	Outcome	System Component of Error Origin	Cause	Count (%)	
Groundhog	False	Locatability	--> Hidden element	27 (22.9%)	
		Detection	--> Container not intended to be focusable	19 (16.1%)	
	Positive	Actionability	--> Destination screen comparison issue	50 (42.4%)	
		Detection	--> Container not intended to be actionable	22 (18.6%)	
	<i>Total False Positives</i>				<i>118, FPR: N/A</i>
	False Negative	<i>Locatability and Actionability Detection only</i>			<i>17, FNR: 45.9%</i>
<i>Total False Negatives under all categories</i>			<i>58, FNR: 74.4%</i>		
TaskAudit	False Positive	Agent Execution	--> Inaccurate navigation	4 (17.4%)	
			--> Verbose or confusing transcripts	2 (8.7%)	
			--> Screen reader proxy unreliable	1 (4.3%)	
	False Positive	Task Generator	--> Visual UI parsing error	12 (52.2%)	
			--> Incorrect contextual understanding	4 (17.4%)	
			<i>Total False Positives</i>		
	False Negative	Accessibility Analyzer	--> Did not identify error	6 (20%)	
			--> Used alternative, accessible interaction	3 (10%)	
Task Generator		--> Did not generate task	21 (70%)		
<i>Total False Negatives</i>				<i>30, FNR: 38.5%</i>	

to prior work [79], we expect them to be representative of typical mobile apps.

Three of the authors independently interacted with informational and interactive elements on all screens using both direct touch and Android TalkBack. Following WCAG-EM [19], the authors recorded observed behaviors and any accessibility issues. The authors conducted the interactions on an app outside of the selected apps to align their inspection methodologies before starting the annotation process. No automated tools were used, except for TalkBack to simulate screen reader interactions. We included interactions that received at least two annotations in the dataset. In total, we examined 475 unique interactions and identified 78 interactions containing accessibility errors. We classified these errors into five general categories: *Locatability* (25), *Actionability* (12), *Label* (9), *Feedback* (11), and *Navigation* (21).

We treat each manually identified functionality error as a ground-truth case and compare tool reports against this set. In our analysis, we computed performance metrics, such as precision and recall, with respect to app-level functionality errors only. For TaskAudit, interaction traces that the Accessibility Analyzer classified as agent or infrastructure failures were not reported as accessibility errors and therefore not considered as positive detections.

5.3.3 Results. Table 2 shows performance metrics for the accessibility checkers evaluated on the functionality error dataset. As expected, Accessibility Scanner and ScreenAudit only detected a very limited set of functionality errors (4 and 10, respectively)

because they did not capture or analyze the interactions with screen readers. We did not calculate their precision, in part because they reported other aspects of accessibility issues not covered by our dataset, and in part because they were not designed to analyze functionality errors in the first place.

Groundhog. It detected 20 functionality errors (54.1%) correctly out of a total of 37 in the Locatability and Actionability categories, which corresponds to an overall recall of 25.6%. Unfortunately, in these real-world apps, Groundhog demonstrated low precision (14.2%). Table 3 shows the reasons why many false positives were observed. For Locatability errors, 46 of the 63 identified elements (73.0%) were false positives, mainly due to inconsistent or faulty view hierarchy information supplied by the app. For Actionability errors, 72 of the 75 identified elements (96.0%) were false positives. The main reason was the challenge when comparing the resulting screens using touch and a screen reader (e.g., different dynamic contents or ads, inconsistent banners or pop-ups). We illustrate some of these examples in Figure 6.

As Groundhog did not report all elements analyzed, (and thus we do not know the true negatives), and because it was not designed to detect all functionality errors, we did not attempt to calculate its false positive rate (FPR). However, given Groundhog’s significantly higher number of false positives (118) than that of TaskAudit (23) when operated on the same dataset, we expect its FPR to also be higher.

TaskAudit. TaskAudit generated and executed 292 tasks. Out of the 78 functionality errors, it correctly identified 48 (61.5%), with

a precision of 66.2%. TaskAudit correctly marked 212 (72.6%) of all tasks as passing with no accessibility issues. Table 3 details the causes for TaskAudit’s false positives and false negatives. TaskAudit generated 23 false positive detections ($FPR = 9.8\%$). Of these, seven were due to agent execution issues. One common problem was navigating the interface inaccurately due to LLM misunderstanding. Task generation failures accounted for another 16 instances. The majority (12) stemmed from errors in visual UI understanding, including OCR errors in two cases. Another four problems were caused by incorrect contextual understandings of non-clickable elements, such as advertisements within apps. We illustrate some of these examples in Figure 7.

TaskAudit missed 30 functionality errors in the dataset. These omissions were caused by the Task Generator failing to generate a relevant task (21 cases), the Accessibility Analyzer not detecting an error (6 cases, including the 2 redundant controls), and the agent using an alternative, accessible interaction sequence which bypassed the error (3 cases).

When comparing the results, Groundhog achieves higher recall than TaskAudit within Locatability, but it does so by reporting substantially more false positives, resulting in low precision. In contrast, TaskAudit reports fewer total errors, but with much higher precision, and it additionally detects categories that Groundhog cannot represent.

Cost. Groundhog spent an average of 766 seconds ($sd = 872$ s) analyzing each screen. TaskAudit spent an average of 1129 seconds ($sd = 904$ s) per screen, consuming 280 k input tokens and 16 k output tokens. At the August 2025 GPT-4o pricing, the cost amounts to about US\$ 0.61 per screen.

6 Qualitative Evaluation

To further understand the *types* of functionality errors detected, we performed qualitative analysis on the experiment results. Our analysis followed guidelines set forth in the WCAG 2.2 [8] and the WCAG-EM 1.0 [19]. We also cross-referenced the expert labels for functionality errors [79] to ensure consistency.

As an overview, we summarized the functionality error coverage for accessibility checkers in Table 4, where we also included AXNav [65] for reference. The coverage indications do not mean the checkers can detect *all* errors in a category, but at least *some* errors.

Figure 8 shows a breakdown of functionality errors detected by Groundhog and TaskAudit. The two tools demonstrate complementary capabilities within Locatability and Actionability (the only categories Groundhog supports). In these areas, Groundhog detected several errors missed by TaskAudit (7 Locatability, 2 Actionability). TaskAudit additionally covers Label, Feedback, and Navigation errors, which fall outside of Groundhog’s scope. For missed detections in these categories, most occurred in Navigation (12 cases), primarily because the Task Generator did not produce a relevant task (6 cases), with 3 additional cases missed by the Accessibility Analyzer and 3 cases where the agent performed an alternative accessible sequence. For Feedback errors, 1 case was missed due to the Task Generator and 3 because the Accessibility Analyzer failed to detect the issue. For Label errors, all 2 cases were due to the Task Generator not generating the relevant tasks.

Table 4: Detection coverage of different accessibility checkers for each functionality error category.

	Locatability	Actionability	Label	Feedback	Navigation
Accessibility Scanner [24]	–	–	○	–	–
ScreenAudit [79]	–	–	○	–	○
Groundhog [54]	○	○	–	–	–
AXNav* [65]	○	○	–	–	○
TaskAudit (ours)	○	○	○	○	○

○ indicates that the tool detected at least some errors in the category, or claims to support detection.

– indicates the tool did not detect errors of this type in our evaluation and does not claim to support their detection.

* We did not evaluate AXNav as it is not publicly available. Coverage is inferred from its system description in [65].

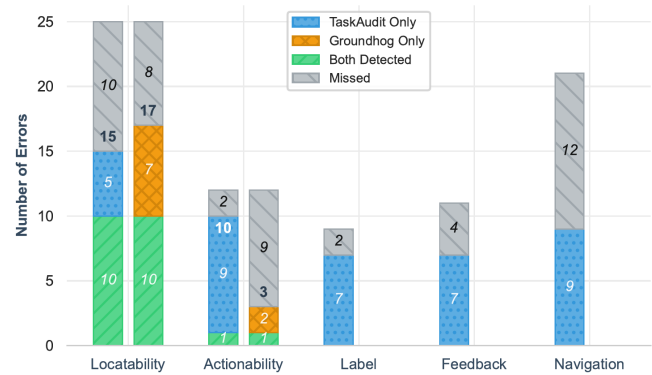


Figure 8: Errors detected by Groundhog and TaskAudit by category. Bold numbers indicate total errors detected. Groundhog does not support the Label, Feedback, and Navigation categories, and therefore these are not included in the chart.

Figure 1 and Figure 9 show eight exemplar functionality errors detected by TaskAudit, which we classify into our five error categories. Two examples represent Locatability errors: a swap-direction button that is not focusable at all (A), and an interface where the screen reader’s focus becomes trapped (F). Example (B) is an Actionability error, where an input field can be focused but not activated. For Label errors, we identified a direct *label-functionality mismatch* where a "search" tab was mislabeled as "explore" (C), as well as a play button with a missing label that made it undiscoverable (G). Example (D) highlights an issue with *inappropriate feedback*: a drop-down menu that changed visually but gave no auditory confirmation. Finally, we classify two examples as *cluttered navigation*: an instance caused by an excessive number of focusable elements (E), and a case where activating a button leads to a pop-up where the subsequent elements do not have proper focus order (H).

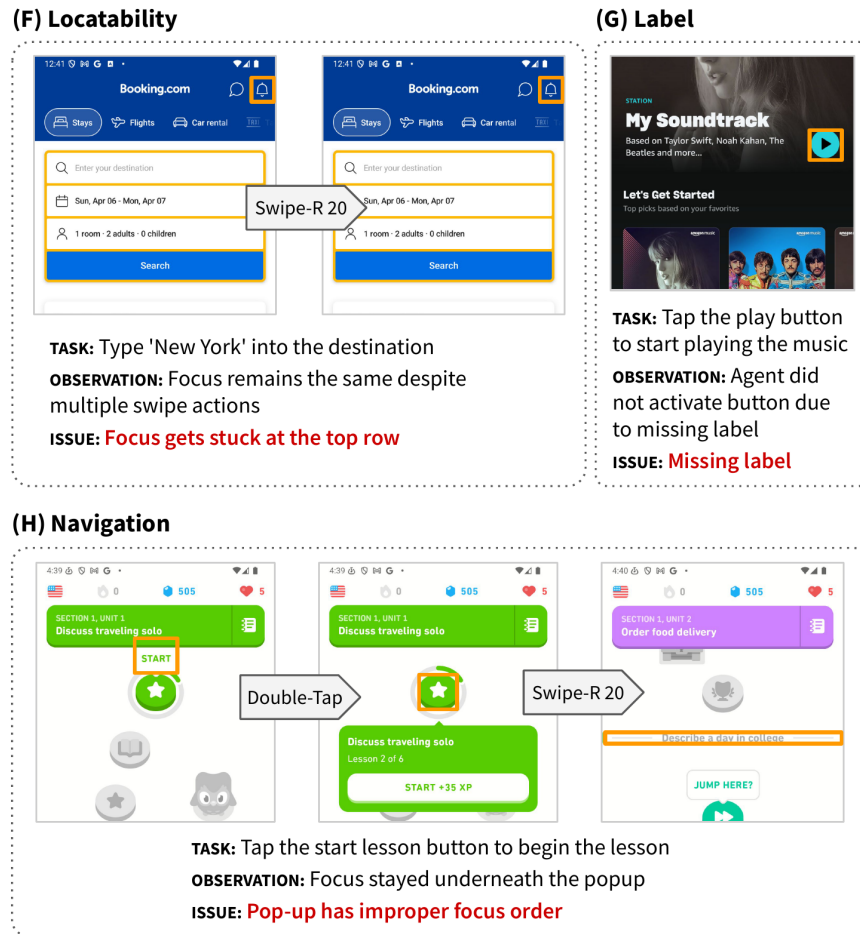


Figure 9: Additional functionality errors TaskAudit detected, with their task descriptions (TASK), agent observations (OBSERVATION), identified issues (ISSUE), and elements of focus (solid outline). (Continued from Figure 1.)

7 Discussion

TaskAudit presents a novel approach of evaluating mobile app accessibility enabled by task-executing agents. In this section, we discuss TaskAudit’s implications, limitations, and opportunities for future work.

7.1 Implications on Automated Accessibility Testing

TaskAudit demonstrates the feasibility of implementing automated testing of functionality errors for a variety of error types. To our knowledge, TaskAudit is the first automated tool capable of detecting several types of errors previously missed by accessibility checkers: label-functionality mismatch, inappropriate feedback, and cluttered navigation.

Our key insight is the direct simulation of screen reader user experience using multi-agent task execution to uncover errors. We expand on prior work such as Groundhog [54] by integrating agentic execution, reflection, and semantic understanding. We show that our multi-agent design achieves high reliability in

executing tasks via the screen reader proxy. This agentic method is particularly effective at detecting error types requiring contextual understanding, such as inappropriate feedback, label-functionality mismatches, and navigation issues, as these depend on assessing an action’s outcome against its intended purpose. We also show that LLM-supported accessibility auditing [79] can be transferred to examine functionality errors from automated interaction traces and detect multiple types of errors.

7.1.1 Complementing Existing Methods. Our results also highlight the complementary strengths of different automated methods. While TaskAudit’s semantic analysis allows it to detect entire categories of errors missed by other tools, Groundhog’s structured crawling identified several Locatability and Actionability errors that our agent-based approach did not (Figure 8). This suggests that a future hybrid approach, combining agentic semantic analysis with exhaustive mechanical crawling, could yield more comprehensive accessibility audits. For example, accessibility metadata inspection can be combined with Task Generator outputs to filter out invalid controls (e.g. those that are occluded or hidden) and flag potential

Locatability and Actionability errors, which can then be quickly evaluated with a method like Groundhog. This would allow TaskAudit to focus on addressing elements with no apparent Locatability and Actionability errors, which would reduce the amount of agentic execution needed, improving efficiency and reducing cost.

On a larger scale, ecosystem-level analyses [10, 21, 52, 75] identify common accessibility failures and quantify their prevalence across large app populations. TaskAudit complements these methods by providing task-based functionality analysis on a smaller set of critical flows. A practical pipeline might first apply large-scale screening to locate high-risk apps and screens, then deploy TaskAudit as a targeted check on those subsets to evaluate functionality errors.

7.1.2 Extensibility. We designed TaskAudit’s architecture to be modular. The Task Generator, Task Executor, and Accessibility Analyzer communicate through structured representations. For example, different LLMs or prompting strategies can be used for task generation and alternative agent frameworks can be used for execution. The screen reader proxy can also be replaced with other assistive technology proxies, such as voice-based interfaces or switch access systems.

At the same time, TaskAudit is different from many recent computer-use [45, 70] and mobile agents [69, 76] that operate primarily over screenshots. Our agents do not observe the raw pixels, but act on TalkBack outputs, and must reason about the UI from the perspective of a screen reader user. This interface makes the system less dependent on a model’s vision capabilities, but more dependent on its ability to follow instructions and maintain context over an interaction trace. We see an opportunity to fine-tune smaller language models that learn to operate screen readers directly, potentially via reinforcement learning on TalkBack transcripts and action traces. This could enable future agents to execute accessibility tasks more reliably and at lower cost.

7.1.3 Adapting to Other Interaction Modalities. The effectiveness of TaskAudit in detecting Feedback and Navigation errors demonstrates the value of analyzing interaction sequences from a non-visual, transcript-based perspective. This has implications beyond screen readers. Similar agentic methods could be adapted to test other non-traditional interaction modalities, such as voice-based interfaces or switch access systems, where the correctness of an interaction depends on the sequence and quality of non-visual feedback. For example, voice-based interfaces can be evaluated by simulating voice commands, navigating through voice-based menus, and analyzing spoken feedback transcripts. Switch access can be tested by simulating sequential inputs and analyzing the navigation order, focusability and actionability of elements.

7.1.4 Integrating into Development Workflows. In practice, accessibility is typically checked in various stages during software development. Teams run automated rule-based tools inside their IDEs and continuous integration (CI) pipelines [14, 15], complement them with runtime checkers like the Accessibility Scanner [24], while also rely on manual assistive technology walkthroughs using tools such as TalkBack [23, 47].

We envision TaskAudit fitting into this workflow by integrating with existing accessibility inspection tools and IDEs. It can also be

automatically triggered when screens or interaction flows change, alongside existing accessibility checks. Similar to how code quality gates [55] and security testing [48] can block deployment when CI pipelines fail key criteria, TaskAudit can be included into regression test suites or CI pipelines for high-impact flows. For example, organizational policy can require a feature to launch only when its functionality audit report has been reviewed and reproducible functionality errors are resolved.

Although task-based audits introduce additional up-front costs (i.e., primarily in LLM calls), catching functionality errors earlier can reduce expensive last-minute fixes. By surfacing problems before user testing, TaskAudit can reduce engineering and QA effort spent resolving accessibility issues that require redesigns.

Practically, running task-based audits requires infrastructure to deploy apps, manage emulators, and integration into existing test environments, which can increase both compute and engineering overhead. We note that similar infrastructure is already common in modern development workflows for end-to-end testing [2, 16, 60], and prior work in accessibility evaluation will require similar engineering efforts in deployment [65, 79].

7.2 Limitations

TaskAudit currently cannot perform touch exploration, limiting its ability to detect certain errors that depend on spatial interaction patterns common among screen reader users. This constraint means errors that appear exclusively when users tap directly on elements might remain undiscovered. The system’s effectiveness is also bounded by the capabilities of its components. Task generation relies on visual UI understanding; related failures, such as OCR errors or misinterpreting elements, occurred in our evaluation and can limit interaction coverage. Furthermore, the quality of the generated task prompts remains a crucial LLM-dependent limitation. Poor or overly simplistic task generation can lead to limited coverage of real-world usage scenarios, which contributed to missed errors in our evaluation. Of the 30 false negatives in TaskAudit, 21 can be attributed to the Task Generator. A potential way to increase the diversity of generated tasks is to run the generator multiple times and select the “best” tasks (e.g., according to their coverage and detail levels).

A limitation we observed during task execution is that our reflection agent can sometimes be less effective because it relies solely on the next screen’s initial transcript to assess task progress. As this accurately reflects a screen reader user’s experience, it can make the agent miss visually available cues that indicate task failure or success, resulting in unnecessary explorations. Task execution agents also encountered issues, sometimes struggling with inconsistent interface behavior, string matching limitations, overly verbose screen reader transcripts, or proxy reliability, resulting in false positives or overlooked errors.

Additionally, the system currently cannot detect errors involving functionality redundancy, such as when multiple controls perform the same action. While we adopted one effective agent setup, we have not systematically compared alternative language models, agent architectures, or combinations thereof. Different groupings of models and agents may yield varying efficiencies or accuracies that we do not currently capture.

Finally, TaskAudit currently incurs computational costs due to multiple LLM calls for task generation, execution, and analysis. The time spent per screen is also substantially longer than Groundhog. We anticipate that combining TaskAudit with rule-based analysis (see Section 7.1.1) can help reduce the number of screens requiring agentic evaluation, thereby lowering overall costs. Model optimization, such as hosting smaller, fine-tuned models locally (see Section 7.1.2) can also help improve efficiency and reduce costs.

7.3 Future Work

While we proposed the concept of “functionality error,” we did not attempt to construct a complete taxonomy of these errors. Instead, we relied on prior work and qualitative observations during our experiments to identify five error categories. TaskAudit provides a useful technical framework that can potentially speed up the identification of new error instances. This will enable future research to explore whether additional error types exist, their impact on users, and practical solutions to addressing them.

We previously discussed optimization opportunity in execution, where fine-tuning smaller language models directly on TalkBack interaction can potentially lead to reliability and cost improvements. In parallel, incorporating screenshot analysis can further enhance reflection-agent decisions. In cases where it is visually obvious that a control cannot be focused or activated, augmenting the agent with vision models when assessing interaction outcomes could terminate obviously impossible tasks and improve testing efficiency [7, 38, 68].

Another potential direction is to automatically suggest actionable solutions after detecting accessibility errors. Future research may leverage code-analysis methods to highlight relevant code segments causing these issues, along with tailored recommendations for developers. Such integration could ease remediation and improve mobile app accessibility.

8 Conclusion

In this paper, we introduced the concept of “functionality errors” that are often overlooked in accessibility evaluation and quality assurance due to their complexity and lack of tool support. TaskAudit expands current automated accessibility error checking by focusing on interactive tasks and automating the evaluation of these tasks using a multi-agent approach. Our results show the potential of agent-driven accessibility evaluation and simulated user testing in identifying accessibility errors. By enabling earlier and more comprehensive detection of functionality issues, TaskAudit can support app developers and QA professionals in creating more inclusive mobile experiences. Future work can explore the integration of rule-based checkers and programmatic crawlers to improve error detection rate. Opportunities also exist for conducting code-level analysis and generating remedies to remove the errors.

Acknowledgments

This work was supported in part by a Google Research Award and by an AWS Agentic AI Amazon Research Award. Any opinions, findings, conclusions or recommendations expressed in our work are those of the authors and do not necessarily reflect those of any supporter.

References

- [1] AccessLint. 2024. Automated and continuous web accessibility testing. <https://accesslint.com/>
- [2] Appium. 2025. Appium. <https://appium.io/docs/en/latest/>
- [3] Apple. 2021. Accessibility - Vision. <https://www.apple.com/accessibility/vision/>
- [4] Apple. 2024. Accessibility Inspector. <https://developer.apple.com/documentation/accessibility/accessibility-inspector>
- [5] Apple. 2024. Human Interface Guidelines - Accessibility. <https://developer.apple.com/design/human-interface-guidelines/accessibility>
- [6] Apple. 2025. Human Interface Guidelines - Components. <https://developer.apple.com/design/human-interface-guidelines/components>
- [7] Gilles Baechler, Srinivas Sunkara, Maria Wang, Fedir Zubach, Hassan Mansoor, Vincent Etter, Victor Cărbune, Jason Lin, Jindong Chen, and Abhanshu Sharma. 2024. ScreenAI: a vision-language model for UI and infographics understanding. In *Proceedings of the Thirty-Third International Joint Conference on Artificial Intelligence (Jeju, Korea) (IJCAI '24)*. Article 339, 11 pages. doi:10.24963/ijcai.2024/339
- [8] Alastair Campbell, Chuck Adams, Rachael Bradley Montgomery, Michael Cooper, and Andrew Kirkpatrick. 2024. Web Content Accessibility Guidelines (WCAG) 2.2. *W3C* (2024). <https://www.w3.org/TR/WCAG22/>
- [9] 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
- [10] 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
- [11] 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
- [12] Biplab Deka, Zifeng Huang, Chad Franzen, Joshua Hibschan, Daniel Afergan, Yang Li, Jeffrey Nichols, and Ranjitha Kumar. 2017. Rico: A Mobile App Dataset for Building Data-Driven Design Applications. In *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology (Québec City, QC, Canada) (UIST '17)*. Association for Computing Machinery, New York, NY, USA, 845–854. <https://doi.org/10.1145/3126594.3126651>
- [13] Giovanni Delnevo, Manuel Andruccioli, and Silvia Mirri. 2024. On the Interaction with Large Language Models for Web Accessibility: Implications and Challenges. In *2024 IEEE 21st Consumer Communications & Networking Conference (CCNC)*. 1–6. doi:10.1109/CCNC51664.2024.10454680
- [14] Deque. 2021. axe: Accessibility Testing Tools and Software. <https://www.deque.com/axe/>
- [15] Deque. 2024. axe DevTools Accessibility Linter. <https://www.deque.com/axe/devtools/linter/>
- [16] Android Developers. 2021. UI/Application Exerciser Monkey. <https://developer.android.com/studio/test/monkey>
- [17] Android Developers. 2025. Snapshots. <https://developer.android.com/studio/run/emulator-snapshots>
- [18] 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
- [19] Shadi Abou-Zahra Eric Velleman. 2014. Website Accessibility Conformance Evaluation Methodology (WCAG-EM) 1.0. *W3C* (2014). <https://www.w3.org/TR/WCAG-EM/>
- [20] ETSI. 2021. Accessibility requirements suitable for public procurement of ICT products and services in Europe. https://www.etsi.org/deliver/etsi_en/301500_301599/301549/03.02.01_60/en_301549v030201p.pdf
- [21] 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
- [22] Appt Foundation. 2024. Beginners' Guide to Accessibility Testing. <https://appt.org/en/guidelines/beginnersguide-accessibility-testing>
- [23] Google. 2024. Get started on Android with Talkback. <https://support.google.com/accessibility/android/answer/6283677>
- [24] Google. 2024. Get started with Accessibility Scanner. <https://support.google.com/accessibility/android/answer/6376570>
- [25] Google. 2024. Make apps more accessible. <https://developer.android.com/guide/topics/ui/accessibility/apps>

- [26] Google. 2024. Test your app's accessibility. <https://developer.android.com/guide/topics/ui/accessibility/testing>
- [27] Google. 2025. Components – Material Design 3. <https://m3.material.io/components>
- [28] Google Developers. n.d. Lighthouse Overview. <https://developer.chrome.com/docs/lighthouse/overview>. Accessed on April 5, 2025.
- [29] Calista Huang, Alyssa Ma, Suchir Vyasamudri, Eugenie Puype, Sayem Kamal, Juan Belza Garcia, Salar Cheema, and Michael Lutz. 2024. ACCESS: Prompt Engineering for Automated Web Accessibility Violation Corrections. arXiv:2401.16450 [cs.HC] <https://arxiv.org/abs/2401.16450>
- [30] Tian Huang, Chun Yu, Weinan Shi, Zijian Peng, David Yang, Weiqi Sun, and Yuanchun Shi. 2025. Prompt2Task: Automating UI Tasks on Smartphones from Textual Prompts. *ACM Transactions on Computer-Human Interaction* (Feb. 2025). doi:10.1145/3716132
- [31] Yue Jiang, Changkong Zhou, Vikas Garg, and Antti Oulasvirta. 2024. Graph4GUI: Graph Neural Networks for Representing Graphical User Interfaces. 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 988, 18 pages. doi:10.1145/3613904.3642822
- [32] Amy J. Ko and Xing Zhang. 2011. FeedLack Detects Missing Feedback in Web Applications. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, Vancouver, BC, Canada, 2177–2185. doi:10.1145/1978942.1979337
- [33] Chenxia Li, Weiwei Liu, Ruoyu Guo, Xiaoting Yin, Kaitao Jiang, Yongkun Du, Yuning Du, Lingfeng Zhu, Baohua Lai, Xiaoguang Hu, Dianhai Yu, and Yanjun Ma. 2022. PP-OCRv3: More Attempts for the Improvement of Ultra Lightweight OCR System. arXiv:2206.03001 [cs.CV] <https://arxiv.org/abs/2206.03001>
- [34] Toby Jia-Jun Li, Lindsay Popowski, Tom Mitchell, and Brad A. Myers. 2021. Screen2Vec: Semantic Embedding of GUI Screens and GUI Components. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems* (Yokohama, Japan) (CHI '21). Association for Computing Machinery, New York, NY, USA, Article 578, 15 pages. doi:10.1145/3411764.3445049
- [35] Yang Li, Gang Li, Luheng He, Jingjie Zheng, Hong Li, and Zhiwei Guan. 2020. Widget Captioning: Generating Natural Language Description for Mobile User Interface Elements. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Association for Computational Linguistics, Online, 5495–5510. doi:10.18653/v1/2020.emnlp-main.443
- [36] Zhe Liu, Chunyang Chen, Junjie Wang, Mengzhuo Chen, Boyu Wu, Yuekai Huang, Jun Hu, and Qing Wang. 2024. Unblind Text Inputs: Predicting Hint-text of Text Input in Mobile Apps via LLM. 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 51, 20 pages. doi:10.1145/3613904.3642939
- [37] Juan-Miguel López-Gil and Juanan Pereira. 2025. Turning manual web accessibility success criteria into automatic: an LLM-based approach. *Universal Access in the Information Society* 24 (2025), 837–852. doi:10.1007/s10209-024-01108-z Long paper. Accepted: 15 February 2024; Published online: 16 March 2024. © The Author(s) 2024.
- [38] Yadong Lu, Jianwei Yang, Yelong Shen, and Ahmed Awadallah. 2024. OmniParser for Pure Vision Based GUI Agent. arXiv:2408.00203 [cs.CV] <https://arxiv.org/abs/2408.00203>
- [39] MagentaA11y. 2024. Native app accessibility checklist. <https://www.magentaa11y.com/native/>
- [40] 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
- [41] Microsoft. 2021. Accessibility overview. <https://docs.microsoft.com/en-us/windows/uwp/design/accessibility/accessibility-overview>
- [42] Microsoft. 2025. Computer Aided Accessibility Testing (CAAT). <https://www.caat.report/en>
- [43] Nearform. 2024. Eslint-plugin-react-native-a11y. <https://github.com/FormidableLabs/eslint-plugin-react-native-a11y>
- [44] OpenAI. 2024. GPT-4o. <https://openai.com/index/hello-gpt-4o/>
- [45] OpenAI. 2025. Computer-Using Agent. <https://openai.com/index/computer-using-agent/>
- [46] Achraf Othman, Amira Dhouib, and Aljazi Nasser Al Jabor. 2023. Fostering websites accessibility: A case study on the use of the Large Language Models ChatGPT for automatic remediation. In *Proceedings of the 16th International Conference on Pervasive Technologies Related to Assistive Environments* (Corfu, Greece) (PETRA '23). Association for Computing Machinery, New York, NY, USA, 707–713. doi:10.1145/3594806.3596542
- [47] Fernanda Pellegrini, Marcelo Anjos, Fabiana Florentin, Bruno Ribeiro, Walter Correia, and Jonsyberg Quintino. 2020. How to Prioritize Accessibility in Agile Projects. In *Advances in Usability and User Experience*, Tareq Ahram and Christiane Falcão (Eds.). Springer International Publishing, Cham, 271–280.
- [48] B. Potter and G. McGraw. 2004. Software security testing. *IEEE Security & Privacy* 2, 5 (2004), 81–85. doi:10.1109/MSP.2004.84
- [49] Christopher Rawles, Alice Li, Daniel Rodriguez, Oriana Riva, and Timothy Lillcrap. 2023. AndroidInTheWild: A Large-Scale Dataset for Android Device Control. In *Advances in Neural Information Processing Systems*, A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine (Eds.), Vol. 36. Curran Associates, Inc., 59708–59728.
- [50] Dagfinn RØmen and Dag Svanæs. 2012. Validating WCAG versions 1.0 and 2.0 through usability testing with disabled users. *Univers. Access Inf. Soc.* 11, 4 (nov 2012), 375–385. doi:10.1007/s10209-011-0259-3
- [51] 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
- [52] 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
- [53] Navid Salehnamadi, Abdulaziz Alshayban, Jun-Wei Lin, Iftekhar Ahmed, Stacy Branham, and Sam Malek. 2021. Latte: Use-Case and Assistive-Service Driven Automated Accessibility Testing Framework for Android. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems* (Yokohama, Japan) (CHI '21). Association for Computing Machinery, New York, NY, USA, Article 274, 11 pages.
- [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] Gerald Schermann, Jürgen Cito, Philipp Leitner, and Harald C. Gall. 2016. Towards quality gates in continuous delivery and deployment. In *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*. 1–4. doi:10.1109/ICPC.2016.7503737
- [56] Eldon Schoop, Xin Zhou, Gang Li, Zhouong Chen, Bjoern Hartmann, and Yang Li. 2022. Predicting and Explaining Mobile UI Tappability with Vision Modeling and Saliency Analysis. 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 36, 21 pages. doi:10.1145/3491102.3517497
- [57] Leticia Seixas Pereira and Carlos Duarte. 2025. Evaluating and monitoring digital accessibility: practitioners' perspectives on challenges and opportunities. *Universal Access in the Information Society* (2025). doi:10.1007/s10209-025-01210-w Long paper. Accepted: 26 February 2025. © The Author(s) 2025.
- [58] 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
- [59] 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
- [60] Amazon Web Services. 2025. AWS Device Farm. <https://aws.amazon.com/device-farm/>
- [61] Peter Shaw, Mandar Joshi, James Cohan, Jonathan Berant, Panupong Pasupat, Hexiang Hu, Urvashi Khandelwal, Kenton Lee, and Kristina Toutanova. 2023. From pixels to UI actions: learning to follow instructions via graphical user interfaces. In *Proceedings of the 37th International Conference on Neural Information Processing Systems* (New Orleans, LA, USA) (NIPS '23). Curran Associates Inc., Red Hook, NY, USA, Article 1490, 17 pages.
- [62] Yungpeng Song, Yiheng Bian, Yongtao Tang, Guiyu Ma, and Zhongmin Cai. 2024. VisionTasker: Mobile Task Automation Using Vision Based UI Understanding and LLM Task Planning. 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 49, 17 pages. doi:10.1145/3654777.3676386
- [63] Amanda Swearngin and Yang Li. 2019. Modeling Mobile Interface Tappability Using Crowdsourcing and Deep Learning. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems* (Glasgow, Scotland UK) (CHI '19). Association for Computing Machinery, New York, NY, USA, 1–11. doi:10.1145/3290605.3300305
- [64] 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
- [65] Maryam Taeb, Amanda Swearngin, Eldon Schoop, Ruijia Cheng, Yue Jiang, and Jeffrey Nichols. 2024. AXNav: Replaying 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
- [66] Minh Duc Vu, Han Wang, Jieshan Chen, Zhuang Li, Shengdong Zhao, Zhenchang King, and Chunyang Chen. 2024. GPTVoiceTasker: Advancing Multi-step Mobile Task Efficiency Through Dynamic Interface Exploration and Learning. 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 48, 17 pages. doi:10.1145/3654777.3676356
- [67] W3C Web Accessibility Initiative (WAI). 2025. Selecting Web Accessibility Evaluation Tools. <https://www.w3.org/WAI/test-evaluate/tools/selecting/>
- [68] Bryan Wang, Gang Li, Xin Zhou, Zhouong Chen, Tovi Grossman, and Yang Li. 2021. Screen2Words: Automatic Mobile UI Summarization with Multimodal Learning. 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, 498–510. doi:10.1145/3472749.3474765
- [69] Junyang Wang, Haiyang Xu, Haitao Jia, Xi Zhang, Ming Yan, Weizhou Shen, Ji Zhang, Fei Huang, and Jitao Sang. 2024. Mobile-Agent-v2: Mobile Device Operation Assistant with Effective Navigation via Multi-Agent Collaboration. In *Advances in Neural Information Processing Systems*, A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang (Eds.), Vol. 37. Curran Associates, Inc., 2686–2710. https://proceedings.neurips.cc/paper_files/paper/2024/file/0520537ba799d375b8ff5523295c337a-Paper-Conference.pdf
- [70] Xinyuan Wang, Bowen Wang, Dunjie Lu, Junlin Yang, Tianbao Xie, Junli Wang, Jiaqi Deng, Xiaole Guo, Yiheng Xu, Chen Henry Wu, et al. 2025. OpenCUA: Open foundations for computer-use agents. *arXiv preprint arXiv:2508.09123* (2025).
- [71] WebAIM. 2024. WAVE Web Accessibility Evaluation Tools. <https://wave.webaim.org/>
- [72] 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
- [73] Gian Wild. 2023. Mobile Site and Native App Accessibility Testing Methodology. In *Human Factors in Accessibility and Assistive Technology*, Matteo Zalloi (Ed.). AHFE Open Access, Vol. 87. AHFE International, USA. doi:10.54941/ahfe1003648 AHFE International Conference.
- [74] 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
- [75] 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
- [76] 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>
- [77] Xiaoyi Zhang, Lilian de Greef, Amanda Swearngin, Samuel White, Kyle Murray, Lisa Yu, Qi Shan, Jeffrey Nichols, Jason Wu, Chris Fleizach, Aaron Everitt, and Jeffrey P Bigham. 2021. Screen Recognition: Creating Accessibility Metadata for Mobile Applications from Pixels. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems* (Yokohama, Japan) (CHI '21). Association for Computing Machinery, New York, NY, USA, Article 275, 15 pages. doi:10.1145/3411764.3445186
- [78] Xiaoyi Zhang, Anne Spencer Ross, Anat Caspi, James Fogarty, and Jacob O. Wobbrock. 2017. Interaction Proxies for Runtime Repair and Enhancement of Mobile Application Accessibility. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems* (Denver, Colorado, USA) (CHI '17). Association for Computing Machinery, New York, NY, USA, 6024–6037. doi:10.1145/3025453.3025846
- [79] 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 *Proceedings of the 2025 CHI Conference on Human Factors in Computing Systems* (Yokohama, Japan) (CHI '25). Association for Computing Machinery, New York, NY, USA, 19 pages. doi:10.1145/3706598.3713797
- [80] Shuyan Zhou, Frank F. Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xianyi Cheng, Tianyue Ou, Yonatan Bisk, Daniel Fried, Uri Alon, and Graham Neubig. 2024. WebArena: A Realistic Web Environment for Building Autonomous Agents. arXiv:2307.13854 [cs.AI] <https://arxiv.org/abs/2307.13854>

A Prompts Used in TaskAudit

A.1 Decision

The following is the prompt used in the Decision Agent.

You are a mobile agent that performs tasks using a screen reader. By performing these tasks, we aim to identify potential accessibility issues in the app.

Background

- You are on an Android phone.
- You are using a screen reader to interact with the phone.

- You are provided with transcripts of the screen reader's output for your last action.

- If you are given no transcript, try to swipe {direction} 20 times to explore the screen. If you still cannot find the it, try swiping {direction} 20 time for at most {count} times to explore the screen before you conclude the task is impossible.

- If a transcript item is <wrap>, it means the swiping gesture has reached the end of the screen and wrapped around to the other end.

- The user's instruction/task is: <|begin_task|> {task} <|end_task|>

Screen Reader Transcript Information

In the screen reader transcript, the transcripts are in the following format:

```
...
{{
  "index": <index of the element>,
  "transcript": "<transcript of the element>"
}},
...
```

The screen reader transcripts for the last action are:

```
<|begin_transcript|>
```

```
{transcript_dict}
```

```
<|end_transcript|>
```

You are currently focused on the LAST element of the transcript list. Which is:

```
<|begin_focused_element_transcript|>
```

```
{last_element_transcript}.
```

```
<|end_focused_element_transcript|>
```

Notice:

- The transcript list is a result of your previous action.

- The length of the list depends on your action(s) and the last element of the list may not be the last element of the whole screen.

Keyboard Status

We extract the keyboard status of the current screenshot about whether the keyboard is activated.

The keyboard status is: {keyboard_status_str}

{additional_info_str}

```

### Task ###
# Propose a *single* action to be performed on the
current state.
## You should not output a list of actions to be
performed consecutively. Instead, you must output
possible immediate actions that can be performed on
the current state.
## If you don't know what to do, propose something
that would help explore the environment and gather
more information to solve the problem.
## If no talkback transcript provided nor any
context, you can propose something like
{"action_type": "{action_direction}", "repetitions":
"20"} to loop through the elements and collect info
from the talkbacks.
## Please judge if the user's instruction has
already been completed based on ### Progress ###. If
so, pick 'STATUS_TASK_COMPLETE' as 'action_type'.
## Think step by step and provide every detail.

### Output Format ###
# Step 1: Put your thought process within
`<|begin_think|>` and `<|end_think|>`.
# Step 2: Describe what you propose to do next
enclosed with `<|begin_action|>` and `<|end_action|>`.
Also please generate a brief natural language
description for each action based on your thought.
Represent all actions you propose in a list of dict
literal:
...
[{"action": <action_json>, "description":
<description_of_action>}]
...
For the <action_json>, there are only the following
types of actions to choose from:
1. {"action_type": "SWIPE_RIGHT", "repetitions":
"<count>", "stop_at": "<element_transcript>",
"stop_at_occurrence": "<occurrence>"}
2. {"action_type": "SWIPE_LEFT", "repetitions":
"<count>", "stop_at": "<element_transcript>",
"stop_at_occurrence": "<occurrence>"}
3. {"action_type": "DOUBLE_TAP"}
4. {"action_type": "PRESS_BACK"}
5. {"action_type": "TYPE", "typed_text":
"<text>"}
6. {"action_type": "WAIT"}
7. {"action_type": "STATUS_TASK_COMPLETE"}
8. {"action_type": "TASK_IMPOSSIBLE"}

### Hints: ###
- If you want to type something, please check if
the keyboard is visible (the input box is focused). If
not, you need to TOUCH the input box first.
{typing_additional_str}

```

- Use SWIPE_LEFT and SWIPE_RIGHT to explore elements. They will move current focus one by one in the direction specified. SWIPE_LEFT means moving to the previous element, SWIPE_RIGHT means moving to the next element. Use "repetition" to indicate the number of elements to move.
 - When you are exploring, set "stop_at" to "".
 - When you would like to focus on a specific element, set "stop_at" to the full transcript of that element. There may be multiple items of the same transcript; use "stop_at_occurrence" to indicate which occurrences to stop at (e.g. 1 means stop at first time we see it; 2 means skip the first one). Also set "repetition" to the number needed to find the element.
- DOUBLE_TAP means activating the focused element. You must first check if the focused element in <focused_element_transcript> is desired, then activate it to interact with it.
 - Choose STATUS_TASK_COMPLETE when you think the task has been completed according to the progress made and the final transcript.
 - Choose TASK_IMPOSSIBLE when you think the task is impossible to complete.
 - Always use double quotes for JSON keys and string values.

A.2 Reflection

The following is the prompt used in the Reflection Agent.

You are an Android mobile agent that assists a user to accomplish tasks using a screen reader.

```

### Progress before the current operation ###
{progress_str}

```

```

The keyboard status is: {keyboard_status_str_1}

```

```

### Transcript after the current operation ###

```

The screen reader transcripts after the last action are:

```

<|begin_transcript|>
{transcript}
<|end_transcript|>

```

```

The keyboard status is: {keyboard_status_str_2}

```

```

### Current operation ###

```

```

The user's instruction is: {task}.

```

In the process of completing the requirements of instruction, this operation is performed on the phone.

Below are the details of this operation:

```

# Operation thought: {thought}
# Operation action: {action}

```

```

### Response requirements ###

```

Now you need to output the following content based on the screenshots before and after the current operation:

Whether the result of the "Operation action" meets your expectation of "Operation thought"?

A: The result of the "Operation action" meets my expectation of "Operation thought".
 B: The "Operation action" results in a wrong page and I need to return to the previous page.
 C: The "Operation action" produces no changes.

Hint

You should not judge based on if this single operation can complete the user's instruction. Instead you should judge based on if this operation made any progress towards the goal.

For example, if the user's task is to search something and the operation is to open the browser:

You should not expect this browser to show the search page directly. As long as the browser is opened, you should judge this operation to be A.

You should also encourage exploration and gathering more information to solve the problem.

If the screen reader transcript is empty, it means the screen reader has not output any information. You should not judge this operation to be C.

Output format

Your output should be strictly a dict literal with the following format:

...

<|begin_think|>

Your thought for this.

<|end_think|>

<|begin_answer|>

Answer of the above question, A or B or C.

<|end_answer|>

...

A.3 Accessibility Analyzer

The following is the prompt used in Stage 1 of the Accessibility Analyzer.

Your task is to carefully analyze the provided action trace from a mobile device screen reader to determine its outcome. Pay close attention to the details.

Your analysis must consider two independent aspects:

1. Overall Task Status: Check if the 'Success Criterion' from the overall Task Specification has been met in the 'after' transcript. A task can be completed at any step.

2. Immediate Action Status: Check if the specific action just taken was successful on its own, based on the provided action-specific heuristics.

Action-Specific Heuristics:

- For a 'double-tap' on a button/menu, new elements or a screen change are expected.
- For a 'double-tap' on a text input, a keyboard or 'editing mode' is expected.
- For a 'swipe' or 'scroll', the transcript must change.

- For 'type' actions, the typed text should be announced.

Respond ONLY with a JSON object in the following format.

```
{
  "overall_task_status": "COMPLETE or INCOMPLETE",
  "task_status_reasoning": "A concise, one-sentence explanation for the overall task status, based on the success criterion.",
  "immediate_action_status": "SUCCESS or FAILURE",
  "action_status_reasoning": "A concise, one-sentence explanation for the immediate action's status, based on the heuristics."
}
```

Task Specification:

- DESC: {description}
- ELEM: {element}
- CRIT: {criterion}

Action Taken: {action_taken}

Screen Reader Transcript BEFORE Action:

```
<|begin_transcript|>
{transcript_before}
<|end_transcript|>
```

Screen Reader Transcript AFTER Action:

```
<|begin_transcript|>
{transcript_after}
<|end_transcript|>
```

The following is the prompt used in Stage 2 of the Accessibility Analyzer.

An action performed by an automated agent has failed.

Your task is to perform a detailed root-cause analysis to determine if this failure was caused by a specific accessibility error. Please be meticulous in your analysis and use the provided transcripts as evidence.

Respond ONLY with a JSON object in the following format. The 'thought' field is for your step-by-step reasoning.

```
{
  "thought": "A step-by-step analysis of the action trace. First, state the intended action. Second, describe the actual outcome from the transcripts. Third, pinpoint the discrepancy. Finally, reason whether this discrepancy constitutes an accessibility error.",
  "problematic_element": "The name of the UI element from the transcript that caused the issue. If not applicable, use 'N/A'.",
  "element_index": int,
}
```

```
"explanation": "A clear, evidence-based explanation  
for your conclusion. If an accessibility error  
occurred, explain how it fits the chosen category. If  
no accessibility error occurred, explain the likely  
non-accessibility cause (e.g., agent mistake,  
unexpected app state change)."  
}
```

Task Specification:

- DESC: {description}
- ELEM: {element}
- CRIT: {criterion}

```
Failed Action: {action_taken}  
Failure Reason from Stage 1: {stage_1_reasoning}
```

Screen Reader Transcript BEFORE Action:

```
<|begin_transcript|>  
{transcript_before}  
<|end_transcript|>
```

Screen Reader Transcript AFTER Action:

```
<|begin_transcript|>  
{transcript_after}  
<|end_transcript|>
```