AI Coding Agents and Web Accessibility: Keyboard and Screen-Reader Access to Public-Benefit Forms

Shirshak Jung Thapa

Florida Atlantic University

November 2026. DOI: https://doi.org/10.5281/zenodo.24429402. *License: CC BY 4.0.*

Abstract

A web form can pass an accessibility scan and still be impossible to finish by keyboard. We asked whether code changes by AI coding agents break keyboard completion of public-benefit application forms, and whether the projects' own checks and an accessibility scanner would notice. On three open-source applications, a scripted keyboard driver completed form tasks before and after each change, with a pointer driver as control. None of 41 merged agent pull requests that touch form code broke keyboard completion (0 of 41; 95% interval 0 to 9%), and none of 91 that touch interface or form code did (0 to 4%). One pull request introduced radio buttons with a non-standard keyboard pattern. None of 48 changes commissioned from two AI models broke completion (0 to 7%). We planted barriers by hand in one application; scripted runs confirmed ten keyboard barriers and four losses of information exposed to assistive technology. The project's linter, security scan and unit and controller tests reported none of the 14, and the scanner reported a new problem for one, a removed label, and for none of the keyboard barriers. Recorded NVDA speech was unchanged for eight of nine sampled pull requests and lacked something at the planted step for ten of the 14. Checked by hand, all ten keyboard barriers stopped the plain keyboard, while NVDA's browse mode got past all six tried. In these applications the AI changes kept keyboard access, but breaks like those we planted passed every project check we ran.

Keywords: web accessibility; keyboard accessibility; screen readers; AI coding agents; large language models; pull requests; regression testing; public-benefit applications; automated accessibility testing; empirical software engineering

1. Introduction

Picture someone applying for help with food or housing costs on a public website. They do not use a mouse. They move through the form with the Tab key, type their answers, press Space to tick a box and Enter to continue. Half way through, they reach a Continue button that a recent code change turned into a plain box that reacts only to a mouse click. Tab passes over it. Enter does nothing. The form is not broken for anyone with a mouse, so nobody else notices, and an automated accessibility check run on the page may report nothing new either. For this person the application stops there.

People who rely on the keyboard, or on devices and software that act through it, include many who are blind and use a screen reader and many who cannot use a mouse precisely. For them, whether a form can be operated by keyboard largely decides whether it can be completed at all. Public-benefit applications raise the stakes: the online form is often the main way to apply, and a form that cannot be finished means a benefit that is not applied for.

These forms change often, and more and more of the changes are written by AI coding agents. An agent is a tool that reads a project's code, edits its files and opens a pull request, which is a proposed change that the project's maintainers review and merge. Such changes are reviewed by people and checked by the project's automated tests. Neither is built to notice a keyboard barrier. A reviewer looks at the changed lines of code and does not usually open the form and tab through it. An automated test checks only what its author wrote it to check, so unless someone wrote a test for keyboard use, none fails.

Automated accessibility scanners exist for this purpose, and some projects run one. A scanner inspects the page as it stands: it checks, for example, that every field has a name and that text has enough contrast. Whether a control responds to a key press, or whether focus can leave a field, is behaviour, and a scan of a page does not press keys.

So two questions are open for these applications. Do the changes that AI agents actually make, and the changes they make when asked for ordinary form work, break keyboard completion of the form? And if a change did break it, would anything the project already runs notice? Earlier work has measured the accessibility of web pages generated from scratch by language models, and has tested keyboard operation and whole tasks on existing pages (Section 7). We found no study that runs a keyboard-only completion of a whole form task before and after each real AI change to an existing web application and reports what the project's own checks and a scanner caught.

The unit of this study is therefore a whole task, not a single element: open the form, enter the answers, meet a validation error, correct it, review the answers and submit, by keyboard alone. We measured three things at each step. First, whether the keyboard driver, a script that sends only key presses, can complete the task, with a pointer driver that clicks as the control. Second, what the page exposes to assistive technology, such as each field's name and whether it is marked required. Third, as a description only, what the NVDA screen reader says.

We ran this on three open-source public-benefit applications, each built and run on one machine: the MyFriendBen benefits screener, a screener that helps people understand work requirements for food and health-coverage benefits, and the Bloom affordable-housing application. The study has three parts. In Part 1 we measured every merged pull request by an AI coding agent that touches the interface or form code of these applications. In Part 2 two AI models carried out six ordinary form change requests on one application, in two wordings and twice each. In Part 3 we planted keyboard barriers and losses of exposed information by hand in that application, one at a time, and asked whether its linter, its security scan, its tests and the scanner would report them.

The AI changes did not break keyboard completion. None of the 41 merged agent pull requests that touch form code broke it (95% interval 0 to 9%). Neither did any of the 91 that touch interface or form code (0 to 4%), or any of the 48 commissioned changes (0 to 7%). One pull request did change how keyboard users answer three Yes/No questions, in a way that still works but departs from the standard pattern. An earlier, stricter version of our driver counted it as a break; we report it under both.

The planted barriers tell the other half. Ten keyboard barriers and four exposure losses were confirmed by the scripted runs. The project's linter, its security scan and its unit and controller tests reported none of them. The scanner reported a new problem for one, a field whose label no longer names it, and for none of the ten keyboard barriers.

What the NVDA screen reader said was recorded in separate runs and is reported in Section 4.5 as a description. For eight of nine pull requests without a flag it said the same before and after. For ten of the 14 planted cases it no longer said something at the planted step.

The author then checked the planted cases by hand, with the keyboard and NVDA (Section 4.6). All ten keyboard barriers stopped the plain keyboard. NVDA's browse mode got past the six barriers it was tried on, because in that mode NVDA activates a control itself.

Taken together: in these applications the AI changes we measured left keyboard completion intact, while barriers of the kind we planted passed every check of the project that we ran, and almost always the scanner too. Section 2 explains keyboard operation, what scanners check and the applications. Section 3 describes the design, Section 4 the results and Section 5 what they mean. Section 6 sets out the limits of the study, Section 7 the related work and Section 8 concludes.

2. Background

2.1 Operating a form by keyboard

On a web page, one element at a time has keyboard focus: it receives the keys that are pressed. Tab moves focus to the next element that can take it, and Shift+Tab to the previous one; the sequence is called the tab order. Enter activates a focused button or link, Space ticks a checkbox, and the arrow keys move within a group such as a list of options. A control works by keyboard when it can be reached in the tab order and does what it should when the keys for it are pressed.

Radio buttons have a standard way of working by keyboard. Tab moves focus into the group, and each press of an arrow key moves focus to the next option and selects it at the same time [W3C APG radio group]. A form can work differently and still be usable. For example, an arrow key may change which option is selected while the focus stays on the first option. We call this a pattern deviation. It does not stop a keyboard user, but it can confuse someone who expects the standard behaviour.

Many things break keyboard operation while leaving the page fine for a mouse user. A clickable box built from a plain page element has no place in the tab order. A field taken out of the tab order cannot be reached. A script that swallows Enter or Space leaves a control that looks normal and does nothing. A field that pulls focus back to itself is a focus trap: the keyboard user cannot leave it. The Web Content Accessibility Guidelines state the general requirements, that all functionality be operable through a keyboard interface and that focus can be moved away from any component it reaches [W3C WCAG 2.2, criteria 2.1.1 and 2.1.2]. This study does not test conformance to these guidelines or to any law; it measures whether one scripted task can be completed.

2.2 What the page exposes to assistive technology

Screen readers and similar tools do not read the page's code directly. The browser builds an accessibility tree, which is a second description of the page made for such tools. In it each element has a role (button, text field, checkbox), an accessible name (usually the text of its label) and states such as required, invalid, expanded or checked. A field can also have a description linked to it, often its help text. What the tree holds is what assistive technology can tell its user. A change can keep a form working by keyboard and still remove

something from the tree. For example, it can unlink a label, so that the field is named only by its placeholder text. Or it can drop the link between a field and its help text. We call such a change an exposure loss.

2.3 What scanners check

An automated accessibility scanner loads a page and tests the elements it finds against a set of rules. We use axe-core, a widely used open-source scanner. Its own documentation says it finds on average 57% of the issues covered by the guidelines [axe-core repository]. Its rules check, for example, that form fields have labels, that the page has one main landmark and one top-level heading, and that text contrasts enough with its background. A rule can see that a button lacks a name. It cannot see that a button ignores the Enter key, because it never presses one. It is well known that scanners miss much of what keyboard users meet [Deque coverage report; Playwright documentation]. We use the scanner here as one of the checks a project might run.

2.4 The applications

We looked for open-source web applications that meet three conditions. They serve a United States public-benefit program through a form that an applicant fills in. They run on one machine without a paid service or a login to an outside service. And at least one merged agent pull request touches the code of the form. Three applications were measured.

MyFriendBen is a benefits screener: a household answers questions about its members, income and expenses, and the screener lists the programs it may qualify for [MyFriendBen repositories]. Its front end is written in React. We ran its Colorado configuration with its own backend on the same machine.

The work-requirements tool of Code for America helps a person find out whether work requirements for food and health-coverage benefits apply to them and prepares a form they can hand in [work-requirements tool repository]. It is a Ruby on Rails application. We ran its North Carolina configuration.

Bloom is a platform for affordable-housing listings and applications [Bloom repository]. We ran its public site with its own staging seed data and completed the guest application for one seeded listing.

A fourth application, the Summer EBT portal for a children's summer food benefit [Summer EBT portal repository], met the conditions but could not be run on the study machine, because its back end needs a database server that is available only as a container. Five other candidates were left out: either none of their merged agent pull requests touches an applicant's form, or they could not be installed without paid packages.

3. Study design

We wrote the plan down on 19 November 2026, after a trial on MyFriendBen and before any main run. It fixed the questions, the applications, the measures, the definitions, the three parts and the analysis. We made four additions before looking at any before-and-after comparison: three task files per form instead of one, an extra run in a narrow window, the use of screen-reader speech as a description, and a corrected rule for radio buttons. Each is explained where it applies.

3.1 Tasks

A task file is a list of steps that fill in a form from start to finish. Examples of steps are "fill the text field named First or given name", "choose No in the radio group about gig work" and "activate the Continue button". Each step names its target only by its role and accessible name, the way assistive technology sees it. For each application we wrote three task files for the newest version of the code, which we call head.

- The **main** task goes through the whole form with valid answers and one planned validation error, which it then corrects.
- The **errors** task submits every page empty or with wrong input, so that every error message appears. For each error it checks whether assistive technology is told about it, and then corrects it.
- The **branch** task chooses answers that open the sections of the form the main task does not reach.

The main task also runs in a window 320 pixels wide, because narrow layouts change which controls are shown and in what order. Table 1 gives the length of each task file.

Table 1. The applications, their form tasks and the merged agent pull requests measured

Application (configuration run)	Form task	Task files (steps)	Agent pull requests touching interface or form code	Of these, touching form code	Testable (form)
MyFriendBen benefits screener (Colorado)	benefits screener, language page to results	main 44, errors 90, branch 98 (branch with yearly amounts 99)	83	37	82 (36)
Work-requirements tool (North Carolina)	work-requirements screener, location to "You've created your form"	main 44, errors 50, branch 47	6	4	6 (4)
Bloom housing portal (public site, seeded listing)	guest housing application, listing to confirmation	main 43, errors 45, branch 54	3	1	3 (1)

Task files are counted in steps at the newest commit. The MyFriendBen branch task with yearly amounts replaces the branch task under the corrected keyboard driver; its errors task was run under the first driver only. A fourth application, the Summer EBT portal, could not be run on the study machine.

Each task ends at the last page the application reaches without outside services: the results page of MyFriendBen, the "You've created your form" page of the work-requirements tool, and the confirmation page of Bloom. A task written for head may not fit an older version of the code, because pages and questions change over time. For an older version we adapted the task file only as far as needed for the pointer driver to complete it. Both sides of a change always ran the same adapted file, and every adapted file is kept.

3.2 The instrument

The **keyboard driver** is a script that operates Google Chrome through Playwright, a browser-automation library [Playwright]. It sends only key presses: Tab, Shift+Tab, Enter, Space, the arrow keys, Escape, Home, End, the page keys and typed characters. It never clicks, never moves the mouse and never runs a script that

changes the page, and a check of its own source code enforces this on every validation run. To decide what to press, it reads the page: which element has focus, its role, its name, its value and the visible text. To reach a target it presses Tab until the target has focus. Each step ends as done or as blocked. A blocked step gets one of four reasons: the target is not in the accessibility tree, it cannot be reached by Tab, focus is trapped, or the target cannot be operated by keyboard. A blocked step ends the task.

The **pointer driver** runs the same task with ordinary mouse clicks and typing. It is the comparison. If the pointer run fails too, the page is broken for everyone, and the keyboard result says nothing about keyboard access.

The **radio rule** was corrected before we looked at any comparison. The first version of the driver expected every radio group to follow the standard pattern. On MyFriendBen it therefore reported two radio groups as not operable, although a keyboard user can operate them. In one, the arrow keys change the selection without moving focus. In the other, each radio button is its own Tab stop and takes Space or Enter. The corrected driver tries three ways in turn: the standard pattern, the arrow keys with selection only, and Tab with Space or Enter. If the second or third way works, it records a pattern deviation. It reports a radio button as not operable only if none of the three selects it. For MyFriendBen we report results under both rules, and the corrected rule is the main one.

The **exposure record** notes what the page tells assistive technology at each step. For the target of the step it keeps the role, the name and where the name comes from, the description, and the states required, invalid, expanded, checked and disabled, before and after the step. After a step that should produce an error, it keeps how the error is exposed. After a page loads, it keeps the page title and the first heading. Comparing two runs gives, for each step, what was lost, gained or changed. The record comes from the accessibility tree that Chrome offers.

The **scanner** is axe-core. It runs inside the page during the keyboard runs in the wide window and records every rule the page violates at that moment.

Validation. Before trusting the instrument, we ran it on a test site built for the purpose. The site has one good version of a form and many variants, each with one planted change. Harmless variants change the page without hurting keyboard use, for example a scrambled tab order that still works. Broken variants each contain one keyboard barrier. Exposure variants remove information while the keyboard still works. In the last validation run before the measurements, all 65 of 65 cases behaved as expected: 58 variants plus 7 extra runs in the narrow window and with the error sweep. Every broken variant blocked the keyboard run at the planted step, with the expected reason, while the pointer run completed. The exposure comparison was as expected for all 25 of 25 judged variants. axe-core reported something new for none of the 30 planted barriers.

3.3 Definitions

These definitions were fixed in advance and applied the same way to every change.

- A **keyboard regression** is counted when the keyboard run completes before the change and is blocked after it, while the pointer run still completes, and a second keyboard run gives the same result.
- An **exposure loss** is counted when the before-and-after comparison reports the same lost item at the same step in two separate pairs of runs.

- A **scanner change** is a rule that the scanner reports after the change and not before, or before and not after.
- An **application break** is counted when every pointer run fails after the change although it completed before.
- A **pattern deviation introduced** is a radio question that needs a non-standard way of answering after the change and did not before. It is flagged for checking and is not counted as a regression.

Barriers that were already there before a change are the baseline. We describe them and do not count them against the change. A flag comes from the scripted runs. Each flagged keyboard regression, exposure loss and pattern deviation was then checked by hand by the author (Section 3.8).

3.4 Part 1: merged agent pull requests

We found the agent pull requests with the markers of the AIDev dataset [Li, Zhang and Hassan 2026]. These markers recognise pull requests opened by five coding agents: Codex, Devin, Copilot, Cursor and Claude Code. We kept the pull requests that were merged and could be matched to a commit on the main branch. We then marked each one by the files it changes: whether it touches interface code, and whether it touches the code of the form that our task fills in. The list was fixed from data gathered on 18 and 19 November 2026.

For each pull request we built the application twice, once from the code just before the change and once from the code just after. We ran each task on both builds: one pointer run (a second if the first was blocked), two keyboard runs with the scanner, and for the main task two more keyboard runs in the narrow window. MyFriendBen was measured first, with all its 83 pull requests that touch interface code. For the work-requirements tool, the set also includes two pull requests that change only the Ruby code behind the form and no interface file.

3.5 Part 2: commissioned changes

For the work-requirements tool we wrote six requests for ordinary changes, of the kinds its own merged agent pull requests show. We wrote them before any change was generated. Two limit what a field accepts: the length of a name (W1) and the characters in a case number (W2). One handles an unsupported language in the web address (W3). One sends a person back if they open the signature page too early (W4). One loads a script only on the pages that need it (W5). One gives every page a browser-tab title (W6). Each request came in two wordings. Wording A is the request alone. Wording B adds the sentence "Keep the form fully usable with a keyboard alone and with a screen reader."

Each change was made by an AI agent working in a fresh copy of the code at head. The agent could only read and edit files. We used two models, Claude Sonnet 5.5 and Claude Opus 5.5, and ran each one twice for every request and wording. That gives 6 requests, 2 wordings, 2 models and 2 repetitions, 48 changes in all. Each change was built and then measured against head with the same three tasks as in Part 1.

3.6 Part 3: planted barriers

We then broke the work-requirements tool on purpose, one change at a time, in its real page templates. We call each such change a plant. In a first pass we planted ten keyboard barriers of the kinds validated on the test site: click-only controls, controls taken out of the tab order, key handlers that swallow Enter or Space, a focus trap, and a button that stays disabled until the mouse moves. We also planted four exposure losses (a

label removed, an error not announced, a required field not marked, and a lost link to help text). A plant counts only if the scripted runs confirm it. A keyboard barrier must block the keyboard run in both runs while the pointer run completes. An exposure loss must appear in both pairs of runs. After a first pass, six further variants aimed at the same kinds were planted, marked with the suffix b. All 20 plants are reported.

3.7 The projects' own checks

For every commissioned change and every plant we ran three of the work-requirements tool's own checks, set up the way its repository sets them up. The first is its Ruby linter (standardrb), which checks code style. The second is its security scanner (brakeman). The third is its automated tests, which the project calls specs: the unit, model, request and controller specs (1,064 examples at head). The unchanged code at head passes all three. For the pull requests of Part 1 we recorded three things from each repository at each merge: which automated workflows it runs, whether any of them runs an accessibility scanner, and which test files the pull request changed.

3.8 Screen-reader speech and the hand check

The third measure records what the NVDA screen reader says at each step of a keyboard run [NVDA]. NVDA is started with its own settings folder and a silent speech synthesizer, so nothing is spoken aloud, and it writes everything it would say to a log with the time. The keyboard driver runs the task in a visible Chrome window, pausing 600 milliseconds after each key press and 1.5 seconds after each step. For each step we keep what NVDA said at three moments: when the target received focus, while the step was under way, and after its last key press. We keep only speech from inside a step's time window, and only while the test window was in front. The raw log is deleted after each run.

Before comparing two runs we remove differences that have nothing to do with the page: spacing and capital letters, the site's address, position phrases such as "2 of 3", landmark announcements, announcements about the browser window, and the order of items within a step. After a page loads we compare only its title and first heading, because how much more NVDA reads depends on the length of the page. What remains is listed for each step as lost, gained or changed, with the kind of item: a name, a role, a state, an error message or other text.

In these runs the keys go to the browser, not through NVDA. The record therefore shows what NVDA says as focus moves and the page changes during a keyboard run. On the test site, before the measurements, three runs of the good version gave identical speech after these rules. The speech comparison agreed with the exposure comparison on three of four exposure variants. In the fourth, NVDA read a placeholder as the field's name, which sounds exactly like a label. Some harmless variants also changed the speech, for example a different kind of date control. Speech is therefore reported as a description that accompanies the two measures.

This validation was repeated with the final version of the instrument at the start of the capture night, in 33 runs. Three runs of the good version again gave identical speech. Four of five exposure variants lost the expected item in the speech; the fifth was again the placeholder. Of the 20 harmless variants that were judged, 9 lost at least one spoken item, which confirms that a difference in speech alone does not show harm.

The speech captures were run in one night, on 21 November 2026, one run at a time on the unattended study machine. A small helper script kept the test window in front. The captures cover four groups: MyFriendBen

#2157, the two commissioned changes with a scanner change (on all three tasks), nine pull requests without a flag, and each of the 14 counted plants compared with the unplanted form. The unplanted form was captured once per task, and that record is the "before" side for every plant and for both commissioned changes. The nine pull requests without a flag are all nine unflagged pull requests of the work-requirements tool and Bloom.

The author, who tests accessibility professionally, then checked each flagged case and each counted plant by hand, with the keyboard and NVDA, and recorded what a user meets. This hand check was done on 21 November 2026, after the captures, in Chrome with NVDA. For the plants, all 14 were put into one copy of the form, so that the form had to be walked through only once. The scripted pointer run completed this copy, and the scripted keyboard run stopped at its first barrier. At each planted control the author first tried the plain keyboard, meaning with NVDA off or in its focus mode, where keys go straight to the page. For six of the ten keyboard barriers the author also tried NVDA's browse mode, in which NVDA acts on the control itself. Pull request #2157 was checked on the MyFriendBen build made after the change.

3.9 Analysis

We report counts and proportions with exact 95% intervals (Clopper-Pearson) [Clopper and Pearson 1934]. An interval says how high the true rate could plausibly be, given what we counted. For example, 0 breaks in 41 changes is compatible with a true rate of up to 9%. We give the numbers for each application, for the three together, and for Part 2 by model, wording and request. No other statistical test is used.

Three practical details. The work-requirements task ends before the tool's PDF step, because that step needs a package we did not install. The planned "state not reported" plant was replaced by a lost help-text link, because no control on the task paths reports such a state. And a run that failed at its first step on a page-load timeout was repeated.

4. Results

4.1 Before any change: what was already there

Each application already had some barriers and gaps at head, before any change we measured. These form the baseline. On MyFriendBen, the Referral Source dropdown has no accessible name, so the driver finds it through its visible label. Its Yes/No income questions, which one of the agent pull requests introduced, do not follow the standard radio pattern. And 19 of the error checks of the full errors task fail at head: the error is shown on the screen, but assistive technology is not told about it. On the work-requirements tool, the text box for details about what prevents work has no label linked to it, and 4 error checks fail at head in the same way (first name, last name, e-mail confirmation and signature). On Bloom, every error check of the errors task passed at head. We removed the error checks that already fail at head from the task files, so that a later change is judged only on what was there to lose.

Under the first radio rule, the driver could not complete the MyFriendBen main task at head at all: the run stopped at step 24, the first of the Yes/No questions. Under the corrected rule it completes, with a pattern deviation at five steps.

4.2 Part 1: the merged pull requests did not break keyboard completion

Table 2 gives the counts.

Table 2. Part 1: merged agent pull requests, flagged / testable

Set	Testable	Keyboard regression	Exposure loss	Scanner change	Application break	Pattern deviation introduced
MyFriendBen, form code	36	0 of 36 (0-10%)	0 of 35 (0-10%)	3 of 35 (2-23%)	0 of 36 (0-10%)	1 of 36 (0-15%)
Work-requirements tool, form code	4	0 of 4 (0-60%)	0 of 4 (0-60%)	0 of 4 (0-60%)	0 of 4 (0-60%)	0 of 4 (0-60%)
Bloom, form code	1	0 of 1 (0-98%)	not compared	not compared	0 of 1 (0-98%)	0 of 1 (0-98%)
All three, form code	41	0 of 41 (0-9%)	0 of 39 (0-9%)	3 of 39 (2-21%)	0 of 41 (0-9%)	1 of 41 (0-13%)
MyFriendBen, interface or form code	82	0 of 82 (0-4%)	0 of 81 (0-4%)	4 of 81 (1-12%)	0 of 82 (0-4%)	1 of 82 (0-7%)
Work-requirements tool, interface or form code	6	0 of 6 (0-46%)	0 of 6 (0-46%)	0 of 6 (0-46%)	0 of 6 (0-46%)	0 of 6 (0-46%)
Bloom, interface or form code	3	0 of 3 (0-71%)	0 of 2 (0-84%)	0 of 2 (0-84%)	0 of 3 (0-71%)	0 of 3 (0-71%)
All three, interface or form code	91	0 of 91 (0-4%)	0 of 89 (0-4%)	4 of 89 (1-11%)	0 of 91 (0-4%)	1 of 91 (0-6%)
MyFriendBen under the first radio rule, form code	36	1 of 36 (0-15%)	0 of 35 (0-10%)	2 of 35 (1-19%)	0 of 36 (0-10%)	0 of 36 (0-10%)
MyFriendBen under the first radio rule, interface or form code	82	1 of 82 (0-7%)	0 of 81 (0-4%)	6 of 81 (3-15%)	0 of 82 (0-4%)	0 of 82 (0-4%)

Pooled rows use the corrected keyboard driver for MyFriendBen. For two pull requests (MyFriendBen #2157 and Bloom #6333) the two sides ran different task versions, so only completion and the keyboard pattern were compared. They are left out of the exposure and scanner columns, whose denominators are therefore smaller.

Of the 91 testable pull requests that touch interface or form code, none broke keyboard completion of any task: 0 of 91 (95% interval 0 to 4%). Of the 41 that touch form code, none did: 0 of 41 (0 to 9%). No comparison showed information removed from the accessibility tree (0 of the 89 compared for it), and none broke the application for the pointer driver either. One MyFriendBen pull request was not testable, because the first page of the version before it shows nothing with the backend we ran.

One pull request changed how three questions are answered by keyboard. MyFriendBen #2157 introduced Yes/No radio buttons for three income questions. Under the corrected driver, both keyboard runs after the change complete the task, in the wide and in the narrow window. At the five steps that answer No to one of the three questions, they record that the arrow keys change the selection but focus does not follow. That is a pattern deviation introduced by the change, 1 of 41 among the form pull requests. Under the first,

stricter radio rule, the same runs stop at step 24, the first of these answers, while the pointer run completes. That meets the definition of a keyboard regression: 1 of 36 MyFriendBen form pull requests under that rule. Both readings describe the same behaviour. A keyboard user who presses the arrow keys and then Tab can finish the task. One who expects the standard pattern may not. The author's hand check found the same behaviour (Section 4.6).

The scanner changes look like noise. All four scanner changes in the pooled set are MyFriendBen flags in two rules, on the first page of a task. One rule requires a main landmark and the other a top-level heading. The same version of the code gives different answers for these rules from one run to the next. Under the corrected driver, two keyboard runs of the same version and task disagree about the main-landmark rule on the first page in 77 of 297 pairs. This points to a scan taken before the first page had finished loading. We therefore do not read these flags as changes made by the pull requests.

The two smaller applications gave the same result. The six work-requirements pull requests and the three Bloom pull requests produced no flag of any kind. Bloom #6333 changed the vouchers question behind a feature flag, so we ran the version after the change with the flag switched on. The question is then a group of checkboxes instead of Yes/No buttons. Because the two sides ran different task versions, only completion was compared there, as for MyFriendBen #2157.

What the projects themselves run. None of the three repositories runs an accessibility scanner in any automated workflow at any of the merge commits. The MyFriendBen front end does not switch on the accessibility rules of its code linter (`eslint-plugin-jsx-a11y`) at any of its 83 merge commits. Bloom's linter settings at head include the recommended set of those rules. The work-requirements tool's linter checks Ruby code only. Of the 83 MyFriendBen pull requests, 52 added or changed a test file.

4.3 Part 2: commissioned changes did not break it either

All 48 commissioned changes built, and all were measured. Their size varied from 11 to 277 changed lines, with a median of 33. Table 3 gives the results.

Table 3. Part 2: commissioned changes to the work-requirements tool

Group	Changes measured	Keyboard regression	Exposure loss	Scanner change	Application break	Linter reports	Security scan reports	Specs fail
All changes	48	0 of 48 (0-7%)	0 of 48 (0-7%)	2 of 48 (1-14%)	0 of 48 (0-7%)	3	0	12
Sonnet 5.5	24	0 of 24 (0-14%)	0 of 24 (0-14%)	2 of 24 (1-27%)	0 of 24 (0-14%)	0	0	6
Opus 5.5	24	0 of 24 (0-14%)	0 of 24 (0-14%)	0 of 24 (0-14%)	0 of 24 (0-14%)	3	0	6
Wording A	24	0 of 24 (0-14%)	0 of 24 (0-14%)	0 of 24 (0-14%)	0 of 24 (0-14%)	2	0	7
Wording B	24	0 of 24 (0-14%)	0 of 24 (0-14%)	2 of 24 (1-27%)	0 of 24 (0-14%)	1	0	5
Request W1	8	0 of 8 (0-37%)	0 of 8 (0-37%)	0 of 8 (0-37%)	0 of 8 (0-37%)	3	0	0
Request W2	8	0 of 8 (0-37%)	0 of 8 (0-37%)	0 of 8 (0-37%)	0 of 8 (0-37%)	0	0	0
Request W3	8	0 of 8 (0-37%)	0 of 8 (0-37%)	0 of 8 (0-37%)	0 of 8 (0-37%)	0	0	6
Request W4	8	0 of 8 (0-37%)	0 of 8 (0-37%)	0 of 8 (0-37%)	0 of 8 (0-37%)	0	0	6
Request W5	8	0 of 8 (0-37%)	0 of 8 (0-37%)	0 of 8 (0-37%)	0 of 8 (0-37%)	0	0	0
Request W6	8	0 of 8 (0-37%)	0 of 8 (0-37%)	2 of 8 (3-65%)	0 of 8 (0-37%)	0	0	0

Wording A is the request alone; wording B adds "Keep the form fully usable with a keyboard alone and with a screen reader." No change introduced a pattern deviation. Linter = standardrb, security scan = brakeman, specs = the repository's non-feature RSpec examples.

None of the 48 broke keyboard completion, none lost exposed information, and none broke the application: 0 of 48 (95% interval 0 to 7%). The same holds for each model and for each wording separately: 0 of 24 (0 to 14%). The two scanner changes are both Sonnet changes to the page-title request in wording B, and both are improvements. The changes added a language attribute to the page's root element, so the scanner rule that asks for one stopped reporting. Since nothing broke under either wording, the extra sentence in wording B made no measurable difference.

The project's own checks did react, but to things other than accessibility. The linter reported a style issue in 3 of the 48 changes, all Opus changes to the name-length request. One existing spec failed in each of 12 changes. All 12 are on the two requests that change what a page does when it is reached in an unexpected way: an unsupported language (W3) and the early signature page (W4), with 6 of 8 changes on each. The security scan reported nothing.

4.4 Part 3: planted barriers went unnoticed by the project's checks

Twenty plants were made and all built. Fourteen were confirmed by the scripted runs and are counted: 10 keyboard barriers and 4 exposure losses. Table 4 lists every plant.

Table 4. Part 3: barriers planted in the work-requirements tool

Plant	Kind	What was changed	Counted	What the measure recorded	Scanner: new rules	Linter	Security scan	Specs
k01	click-only control	Continue button of the age-range page replaced by a that submits the form on click only	yes	keyboard blocked at step 10 (target-not-in-accessibility-tree); pointer completes	none	silent	silent	pass
k02	control out of the tab order	first-name field of "Tell us about yourself" gets tabindex -1	yes	keyboard blocked at step 33 (target-not-reachable-by-tab); pointer completes	none	silent	silent	pass
k03	control out of the tab order	county dropdown of the location page gets tabindex -1 whenever it is filled	yes	keyboard blocked at step 5 (target-not-reachable-by-tab); pointer completes	none	silent	silent	pass
k04	key handler swallows Space	Space is swallowed on the checkboxes of the caring-for-someone page	yes	keyboard blocked at step 13 (target-not-operable-by-keyboard); pointer completes	none	silent	silent	pass
k05	key handler swallows Enter and Space	Enter and Space are swallowed on the Continue button of the e-mail page	yes	keyboard blocked at step 40 (target-not-operable-by-keyboard); pointer completes	none	silent	silent	pass
k06	focus trap	the signature field gives focus back to itself when it loses it	no	keyboard completes	none	silent	silent	pass
k07	click-only control	Yes/No answer buttons (all yes/no questions) replaced by elements that submit on click only	yes	keyboard blocked at step 11 (target-not-in-accessibility-tree); pointer completes	none	silent	silent	pass
k08	click-only control	"Create my form" link of the milestone page replaced by a that navigates on click only	no	pointer blocked too (step 32)	none	silent	silent	pass

Plant	Kind	What was changed	Counted	What the measure recorded	Scanner: new rules	Linter	Security scan	Specs
k09	key handler swallows keys	all key presses on the state dropdown of the location page are swallowed except Tab	yes	keyboard blocked at step 4 (target-not-operable-by-keyboard); pointer completes	none	silent	silent	pass
k10	control disabled until mouse use	Continue button of "Tell us about yourself" stays disabled until a mouse move	yes	keyboard blocked at step 37 (target-not-operable-by-keyboard); pointer completes	none	silent	silent	pass
e01	label removed	the label of the first-name field loses its link to the field (for attribute removed)	no	both drivers blocked (step 33)	label, label-title-only	silent	silent	pass
e02	error not announced	on the age-range page the error message is hidden from assistive technology and unlinked from the group	no	no loss recorded	none	silent	silent	pass
e03	required not marked	the last-name field loses its required attribute and aria-required	no	no loss recorded	none	silent	silent	pass
e04	description lost (stands in for "state not reported")	the birthdate group loses the link to its help text and error message (aria-describedby removed)	yes	step 35: description removed (was "For example: January 19 2000")	none	silent	silent	pass
k06b	focus trap	Tab and Shift+Tab are swallowed in the signature field (focus cannot leave by keyboard)	yes	keyboard blocked at step 44 (focus-trapped); pointer completes	none	silent	silent	pass
k08b	click-only control	"Create my form" link replaced by a with exactly the same visible text that navigates on click only	yes	keyboard blocked at step 32 (target-not-in-accessibility-tree); pointer completes	none	silent	silent	pass

Plant	Kind	What was changed	Counted	What the measure recorded	Scanner: new rules	Linter	Security scan	Specs
e01b	label removed	the first-name label loses its link to the field and the field gets a placeholder with the same words (name comes from the placeholder)	yes	step 33: name now comes only from the placeholder (was from label)	label-title-only	silent	silent	pass
e02b	description lost (stands in for "error not announced")	the first-name field loses the link to its help text (aria-describedby removed); no error of this application is exposed at head, so an announced error cannot be lost	yes	step 33: description removed (was "Legally, as it appears on your ID.")	none	silent	silent	pass
e03b	required not marked	the "(required)" marker is removed from the label text of the last-name field	no	no loss recorded	none	silent	silent	pass
e04b	description lost	the phone-number field loses the link to its help text (aria-describedby removed)	yes	step 36: description removed (was "A caseworker may use this number if they need to contact you directly about your case.")	none	silent	silent	pass

Planted 20, counted 14 (10 keyboard barriers, 4 exposure losses). Plants with the suffix b were added after a first pass. Steps are those of the main task (44 steps), except e02, which was run on the errors task.

The ten confirmed keyboard barriers stopped the keyboard run at the planted step in both runs, wide and narrow, while the pointer run completed. They are:

- a Continue button rebuilt as a plain box that submits only on a click (k01);
- the Yes/No answer buttons of every question rebuilt the same way (k07);
- a link rebuilt as a plain element with the same visible text (k08b);
- a field and a dropdown taken out of the tab order (k02, k03);
- key handlers that swallow Space on checkboxes, Enter and Space on a button, or every key but Tab on a dropdown (k04, k05, k09);
- a button that stays disabled until the mouse moves (k10);
- a signature field that swallows Tab and Shift+Tab, so that focus cannot leave it (k06b).

The four confirmed exposure losses are of two kinds. Three removed the link between a field and its help text (e02b, e04, e04b). One made the first-name field take its name from a placeholder instead of its label (e01b).

The project's own checks reported none of the 14. The linter and the security scan were silent on every plant, and all 1,064 spec examples passed on every plant.

The scanner reported a new rule for 1 of the 14 and for none of the 10 keyboard barriers. The one was e01b: the scanner has a rule against fields that are named only by a title or a placeholder, and that rule reported it. Over all 20 attempts it fired for the two label plants only (e01 and e01b). To the scanner, a Continue button that cannot be reached by Tab, a checkbox that ignores Space and a signature field that traps focus all looked the same as before.

Six plants were not counted, and they show what each measure can and cannot see.

- In k06 the signature field pulled focus back to itself whenever it lost it. The keyboard driver still completed the task, so for it this was not a trap.
- In k08 the pointer driver could not find the rebuilt link either. The plant therefore broke the page for both drivers and says nothing about the keyboard. Its replacement is k08b, which keeps the visible text.
- In e01 the field lost its name entirely, so neither driver could find it by name. Our definition records that as an application break. It shows that the scripts could not locate the field; it does not show that a person could not fill it in.
- Plants e02 and e03 removed an error announcement and a required marking. The application did not expose either of them at head in the first place, so there was nothing to lose.
- In e03b the "(required)" text was removed from a label. The comparison reported a changed name, not a lost state, and the definition does not count that.

Each counted plant was then checked by hand (Section 4.6).

4.5 What the screen reader said

Every capture run finished. What follows compares the steps whose capture was valid.

Eight of the nine pull requests without a flag sounded the same. For eight of the nine sampled pull requests, the six of the work-requirements tool and Bloom #6134 and #6312, NVDA said the same before and after at every step: nothing lost, nothing gained, nothing changed. Bloom #6333 differs at one step, the vouchers question, and the difference is the one the change intends when its feature flag is on. Before, NVDA said "No, radio button, checked". After, it said "None of the above, check box, not checked" and, once the box was ticked, "checked".

The page-title changes were heard as page titles. For the two commissioned changes with a scanner change, both made for the page-title request, nothing was lost or gained on any of the three tasks. What changed is the title NVDA reads when a page opens. Before, every page was announced as "getbenefitshelp". After, NVDA reads the page's heading followed by "| getbenefitshelp". This happens at 20 steps of the main task, 24 of the errors task and 19 of the branch task. The two changes gave the same speech as each other.

After #2157, NVDA announces the new questions as radio buttons, but not the answer the arrow keys choose. The two sides of MyFriendBen #2157 ran different task versions, because the change replaces a table of income sources by three Yes/No questions. The first 20 steps are the same actions on both sides and gave identical speech. The later steps differ and cannot be compared one to one. In the run after the change, when Tab reaches one of the new questions, NVDA reads the question as a group and then "Yes, radio button, not

checked, 1 of 2". At each of the five steps that answer No, the driver then pressed the Down arrow twice: the first press selects Yes and the second moves the selection to No. NVDA said "checked" and then "not checked". It did not say the word "No" at either press. Where the next Tab press landed on the chosen answer, NVDA read "No, radio button, checked, 2 of 2". This is how the pattern deviation of Section 4.2 sounds: the user hears that the answer changed, but is not told which answer is now chosen until focus moves. One step on each side, a toggle button on the health-insurance page, could not be captured, because focus had left the page for the browser's toolbar.

For ten of the 14 counted plants, NVDA no longer said something at the planted step. Table 5 lists what was lost. For seven of the ten keyboard barriers, NVDA no longer says something at the blocked step that it says on the unplanted form. In four of them the name and role of the control are never spoken, because the control can no longer be reached (k01, k06b, k07, k08b). In three, the result of the key press is never spoken: "checked" after Space on the checkbox (k04), the chosen state "North Carolina" (k09), and the title of the next page after Continue (k05). For the other three (k02, k03, k10) the capture of the blocked step was not valid, because focus had left the page for the browser's toolbar. The speech at the block itself was therefore not recorded, and the steps before it sounded the same.

For the three plants that unlink a help text (e02b, e04, e04b), NVDA no longer read that text when the field received focus, and nothing else changed. For e01b, the label replaced by a placeholder with the same words, the speech was identical, as on the test site: NVDA reads the placeholder as the field's name. The exposure record and the scanner both report this change. The speech capture cannot hear it.

Table 5. What NVDA no longer said for each counted plant, compared with the unplanted form

Plant	Keyboard run with the plant	No longer spoken at the planted step
k01	blocked at step 10	on focus: "continue", "button"; after the step: "getbenefitshelp"
k02	blocked at step 33	not compared: the capture of step 33 was not valid
k03	blocked at step 5	not compared: the capture of step 5 was not valid
k04	blocked at step 13	after the step: "checked"
k05	blocked at step 40	after the step: "getbenefitshelp"
k07	blocked at step 11	on focus: "no", "button"; after the step: "getbenefitshelp"
k09	blocked at step 4	after the step: "north carolina"
k10	blocked at step 37	not compared: the capture of step 37 was not valid
k06b	blocked at step 44	on focus: "sign & create my form", "button"; after the step: "getbenefitshelp"
k08b	blocked at step 32	on focus: "create my form", "link"
e04	completed	on focus: "for example: january 19 2000"
e01b	completed	nothing; the speech was identical
e02b	completed	on focus: "legally, as it appears on your id."
e04b	completed	on focus: "a caseworker may use this number if they need to contact you directly about your case."

Speech is shown after the normalisation of Section 3.8, in lower case. At the newest commit every page of the work-requirements tool has the browser title "getbenefitshelp", which NVDA reads when a page opens; where it is listed after the step, the next page did not open. The unplanted form was captured once and is the before side of every row.

4.6 The hand check

All ten keyboard barriers stopped the plain keyboard. With NVDA off or in focus mode, the author could not operate any of the ten planted controls by keyboard, as the scripted runs had recorded. Tab skipped the control or never reached it (k01, k02, k03, k07, k08b, k10), the keys did nothing (k04, k05, k09), or focus could not leave the field (k06b). Table 6 gives what was found at each.

Table 6. The hand check of the 14 counted plants, in the order they are met in the form

Plant	With the plain keyboard (NVDA off or in focus mode)	With NVDA
k09	blocked: NVDA off: the arrow keys and typing did nothing on the state dropdown; the state had to be chosen with the mouse	browse mode not tried
k03	blocked: Tab from the state dropdown went to Continue and skipped the county dropdown	browse mode not tried
k01	blocked: Tab never landed on Continue	browse mode gets past it: NVDA read it as "clickable"; Enter opened the next page
k07	blocked: Tab never landed on Yes or No	browse mode gets past it: NVDA read "clickable Yes, clickable No" in one go; Enter moved to the next page
k04	blocked: NVDA in focus mode: Space did not tick the checkbox	browse mode gets past it: Space ticked the checkbox and NVDA said so
k08b	blocked: Tab never landed on "Create my form"	browse mode gets past it: NVDA read it as "clickable", not as a link; Enter opened the next page
k02	blocked: Tab went to the middle name and the last name and skipped the first name	browse mode not tried
k10	blocked: with the mouse untouched, Tab did not land on Continue	browse mode not tried
k05	blocked: NVDA off: Enter and Space on Continue did nothing	browse mode gets past it: NVDA read "Continue, button"; Enter opened the next page
k06b	blocked: Tab and Shift+Tab did not move focus out of the signature field	browse mode gets past it: after Escape, the Down arrow reached "Sign & create my form" and Enter opened the last page
e04	not a keyboard barrier	lost: on focus NVDA did not read "For example: January 19 2000"
e04b	not a keyboard barrier	lost: on focus NVDA did not read the sentence about a caseworker
e02b	not a keyboard barrier	lost: on focus NVDA did not read "Legally, as it appears on your ID."
e01b	not a keyboard barrier	no loss: with a name typed in, NVDA still read "First or given name (required), edit"; the name is still announced

Checked by the author on 21 November 2026 with NVDA 2026.3.1 in Google Chrome, on one copy of the form that carries all 14 plants.

NVDA's browse mode got past all six barriers it was tried on. In browse mode NVDA does not pass the key on to the page. It activates the control itself. The three click-only controls (k01, k07, k08b) were read as "clickable", and Enter on them moved to the next page. Space ticked the checkbox whose key handler swallows Space (k04). Enter worked on the button whose handler swallows Enter (k05). And from the trapped signature field the author could leave with Escape and reach the last button (k06b). The other four barriers (k02, k03, k09, k10) were not tried in browse mode. Getting past a barrier is not the same as hearing the right thing: the rebuilt link was announced as "clickable", not as a link, and the Yes and No answers were read together in one phrase.

The three unlinked help texts were not read, and the placeholder case sounded the same. On focus, NVDA did not read the help text of the first-name field, the birthdate group or the phone field (e02b, e04, e04b). For e01b, NVDA still announced "First or given name (required)" with a name typed into the field, so the author heard no loss, as in the recordings.

Pull request #2157 behaves as the scripted runs recorded. With NVDA off and focus on Yes, the first press of the Down arrow selected Yes and the second selected No, while the focus ring stayed on Yes. Tab then moved focus to No. In NVDA's browse mode the Down arrow read the next line without changing the answer. Space on No selected it, and NVDA said "checked".

5. What it means

In these applications, the AI changes kept keyboard access. Across 91 merged agent pull requests and 48 commissioned changes, not one blocked a keyboard user who could complete the task before. The intervals say how strong that result is. If AI changes to form code broke keyboard use more often than about 9% of the time, we would very probably have seen at least one break among the 41 form pull requests. For the commissioned changes the figure is about 7%. The one difference we found is the Yes/No questions of #2157. A keyboard user can work around it, and a strict reading of the standard pattern counts it as broken, so we report it both ways.

The projects' checks were not built to notice. In the application where we planted barriers, the linter, the security scan and the unit and controller specs said nothing about any of the ten keyboard barriers that stop the task. That is not a fault of those tools. A Ruby linter checks Ruby style, a security scanner looks for security flaws, and the specs check what the server returns. But it means this project could merge a change that stops a keyboard user from finishing the form, and none of these checks would object. Someone would have to try the form with the keyboard.

A scanner would not have closed the gap. None of the three projects runs one. Adding axe-core would have caught one of the fourteen confirmed plants, a label problem, and none of the keyboard barriers. Where the scanner did report changes in the pull requests, the reports varied from run to run on two rules about page structure. A scan can tell whether the parts of a page have names. It cannot tell whether pressing a key does anything.

The speech record shows what a user would stop hearing. In the captures, NVDA no longer read the help text of the three fields whose link we removed. No check of the project and no scanner rule reported that loss.

At seven blocked steps NVDA never spoke the control, or the result of the key press, that it speaks on the unplanted form. Two things it cannot do alone. It cannot tell a placeholder from a label, because NVDA reads both the same, and on the test site it changed for 9 of 20 harmless variants. So recorded speech works best next to the other measures, as evidence for a person who reviews a change.

A barrier for the keyboard is not always a barrier for a screen reader. In the hand check, NVDA's browse mode got past all six planted keyboard barriers it was tried on, because it activates a control itself instead of sending a key to the page. Those barriers still stop a person who uses the keyboard without a screen reader, and they stop the Tab key for everyone. So the keyboard run of this study speaks for keyboard-only use. For people who use a screen reader, the losses that matter may be ones a keyboard run does not see: a link announced only as "clickable", or a help text that is no longer read.

What would notice is a keyboard run of the task. The instrument used here is a script that walks through the form by keyboard and compares the result with a pointer run. It caught every barrier counted in Part 3, and nothing else the project runs caught any of them. If a project kept such a check, "the form can still be completed by keyboard" would become something its build verifies on every change. The check needs task files that follow the form, and they are updated as the form changes.

Why the AI changes broke nothing here. The barriers we planted come from three sources: replacing a built-in control with a home-made one, scripts that intercept keys, and taking a control out of the tab order. The six commissioned requests asked for none of that. They asked for limits on input, a language fallback, a redirect, a script loaded on fewer pages and page titles. Changes that replace form controls are where the planted barriers suggest the risk lies, and they are the natural next thing to measure.

6. Limits of the study

6.1 What the measures show

- **What a pass means.** A completed keyboard run means that a script which already knows which control to use can finish the task by keyboard. Whether focus is visible, whether the order makes sense and whether a person would find the way are separate questions. The study does not claim conformance to any standard or law.
- **The tasks cover part of each form.** The three tasks follow one path through the form, one sweep of its errors and one alternative branch. Some branches are outside them, for example the work-requirements tool's branches for unemployment, disability, migrant work and caring for a child, and Bloom's second and later household members. A result of "no change" applies to the pages the tasks reach.
- **Older versions ran adapted tasks.** A task file adapted for an older version of the code checks less than the head file in the places where it was adapted. Among them, the oldest MyFriendBen commits covered only the first four pages, and some others ended before the results page. On MyFriendBen the backend was the head version for every commit.
- **Two comparisons are limited to completion.** For MyFriendBen #2157 and Bloom #6333 the form itself changed, so the two sides needed different task versions. For these two, completion was compared and the exposure and scanner results were not, so Table 2 leaves them out of those two columns (89 and 39 comparisons instead of 91 and 41). For Bloom #6333 the feature flag was switched on in the test database.

- **The radio rule.** The main results use the corrected rule, which was adopted before we looked at any comparison. Under the first rule the counts differ by the one pull request discussed in Section 4.2.
- **The exposure record is Chrome's.** Names, states and descriptions are the ones Chrome exposes. Another browser may expose them differently.
- **The speech captures.** In the captures the keys went to the browser, not through NVDA, so they follow Tab focus. How a screen-reader user gets through the form in browse mode is covered by the hand check. The pull requests without a flag that were captured are the nine of the work-requirements tool and Bloom. At three planted barriers the blocked step itself was not captured. The unplanted form was captured once per task, which the validation supports: repeated runs gave identical speech.
- **The hand check.** The author did the hand check on one copy of the form that carried all 14 plants together and on one build of MyFriendBen, with NVDA in Chrome. Four of the ten keyboard barriers were tried with the plain keyboard only.

6.2 What could affect the results

- **Differences from run to run.** For two scanner rules, the scan of the first page gives different answers on different runs of the same code. All scanner flags of Part 1 are of this kind. A few runs failed at their first step because the page loaded too slowly, and were repeated.
- **Which project checks were run.** We ran the work-requirements tool's linter, security scan and unit and controller specs. Its feature specs, which drive a browser, and Bloom's end-to-end tests were not part of the study. For the pull requests of Part 1 we recorded what each repository is set up to run and which test files each pull request changed.

6.3 How far the results reach

- **Three applications.** Part 1 covers three applications. Most of its pull requests come from MyFriendBen; the work-requirements tool contributes six and Bloom three. Parts 2 and 3 were run on the work-requirements tool, in its North Carolina configuration.
- **Agents and models.** The merged pull requests come from the agents the projects used: Claude Code on MyFriendBen and the work-requirements tool, Copilot on Bloom. Part 2 used two models from one provider, with six requests and tools to read and edit files. Other agents, models and kinds of request, such as redesigns that replace form controls, may behave differently.
- **One browser, one screen reader, one language.** All runs used Chrome on Windows, in English, and the screen reader was NVDA. Voice control, switch access and the other state configurations of the same applications are outside the study.

6.4 What the numbers support

A count of zero does not prove a rate of zero. The intervals give the highest rates still compatible with our counts: 4% over the 91 pull requests, 9% over the 41 form pull requests and 7% over the 48 commissioned changes, and wider for each application alone. The intervals treat each change as an independent trial. Here 82 of the 91 pull requests come from one application and the commissioned changes repeat six requests, so the bounds are best read as a guide, not as exact. The planted barriers were written by the author before they were measured. They test whether the instrument and the project's checks react; they are not a sample of the

barriers that occur in practice. The counts in Tables 2 to 5 come from scripted runs, and the one flagged pull request and the 14 counted plants were then checked by hand (Section 4.6).

7. Related work

Accessibility of code generated by language models. Several studies measure the accessibility of new code produced by language models. PACE scored HTML form components generated by five models with axe-core and a schema score [Ma et al. 2026]. Microsoft's A11y LLM Eval rendered generated pages and added hand-written assertions, some of which press keys [Microsoft A11y LLM Eval]. Suh and colleagues compared sites regenerated by two models with the human originals using accessibility checkers [Suh et al. 2025], Calò and colleagues judged the semantic quality of generated interfaces beyond what axe-core sees [Calò et al. 2026], and CodeA11y studied an accessibility assistant for Copilot with developers [Mowar et al. 2025]. A recent review covers 38 studies of language models for web accessibility [Aljedaani and Mollik 2026]. All of these measure code generated from scratch; we measure changes to existing applications, by keyboard, before and after.

Repair of accessibility issues. A11YRepair [Huang et al. 2026], AccessGuru [Fathallah et al. 2025], Oyelayo and colleagues [Oyelayo et al. 2026] and Wanscher and colleagues [Wanscher et al. 2026] use language models to repair accessibility problems and judge the result with checkers; Wanscher and colleagues report that unverified edits made the checked result worse in some trials. A11y-bench gives models real accessibility bug fixes from three repositories, validates them with the projects' tests and reports that the models introduced regressions [Tschoner 2025]. Those edits are attempts to repair accessibility. The changes we study are ordinary ones, and our test is a keyboard run of a whole task.

Keyboard and task-level testing. Chiou, Alotaibi and Halfond detect and localise keyboard accessibility failures in web pages [Chiou et al. 2021] and keyboard failures in dialogs [Chiou et al. 2023], and Winn, Chiou and Halfond detect keyboard navigation issues [Winn et al. 2026]. Flow-A11y runs multi-step scenarios in a real browser and judges interaction-dependent criteria on public sites [Fliti et al. 2026]. A11yLTLNav explores generated sites by keyboard against temporal properties [Ge et al. 2026]. On Android, Latte reuses functional tests as use cases and runs them through assistive services, counting a use case that cannot be completed as the failure [Salehnamadi et al. 2021], and TaskAudit has agents carry out tasks through a screen-reader proxy [Zhong et al. 2026]. A11y-CUA measured how often computer-use agents succeed when restricted to the keyboard [Mohanbabu et al. 2026]. Our keyboard test builds on this work. Treating the completion of a whole task as the test is Latte's idea, and running keyboard operation as an automatic check comes from the Halfond group. None of these compares an application before and after a code change.

Accessibility over time and what scanners miss. Fok and colleagues followed missing-label failures in Android apps over time [Fok et al. 2022]. That automated checks cover only part of the keyboard criteria is documented by Deque [Deque coverage report], by Playwright's own guide [Playwright documentation] and by GitHub's report on its accessibility agent [GitHub 2026].

Agent pull requests. The AIDev dataset collects pull requests opened by five coding agents on GitHub [Li, Zhang and Hassan 2026], and Khan and colleagues studied agent pull requests in mobile apps, where "ui" is one task label [Khan et al. 2026]. We found no study of what agent pull requests do to accessibility.

What is new here is the combination: whole-task keyboard runs, applied before and after real AI changes to public-benefit forms, together with what the projects' own checks see.

8. Conclusion

We asked two questions. Do AI-written changes to open-source public-benefit forms break completion of the form by keyboard? And would the projects notice if one did? In three applications, 91 merged agent pull requests and 48 commissioned changes did not break keyboard completion of the tasks we ran: 0 of 41 form pull requests (95% interval 0 to 9%) and 0 of 48 commissioned changes (0 to 7%). One pull request introduced Yes/No questions that work by keyboard in a non-standard way. When we planted barriers by hand, the project's linter, security scan and unit and controller specs reported none of the 14 that were confirmed. The scanner reported one, a label problem, and none of the ten keyboard barriers. In recorded screen-reader speech, something was no longer said at the planted step for ten of the 14, while eight of nine sampled pull requests sounded the same before and after. Checked by hand, all ten keyboard barriers stopped the plain keyboard, and NVDA's browse mode got past the six it was tried on.

For these applications, the risk to keyboard users lies less in the AI changes we measured than in the lack of any check that would catch a break. The projects' tests click, their linters read code, and a scanner reads the page without pressing a key. A scripted keyboard run of the form, kept with the project and compared with a pointer run, would catch the barriers we planted. Two questions follow from this work. Do AI changes that replace form controls, or that redesign whole pages, behave as well? And do the barriers people meet in practice look like the ones we planted?

Declarations

Funding. No funding was received for this work.

Competing interests. The author has no relevant financial or non-financial interests to disclose.

Ethics approval. Not applicable. The study uses open-source software run locally with test data, and involves no human participants.

Use of AI. AI-based tools assisted with the programming and the writing of this paper.

Data availability

The replication package is published at https://doi.org/10.5281/zenodo.24429397 [Thapa 2026]. It contains the keyboard and pointer drivers with their validation site, the task files and their adapted versions, the build and run scripts for each application, the list of pull requests and commits measured, the commissioned changes and the planted barriers, the result records of every run behind the reported counts, the speech records of the screen-reader captures, the record of the hand check, and the scripts that regenerate every table and number in this paper.

References

- Aljedaani, W., Mollik, R. H. (2026). Large Language Models for Web Accessibility: A Systematic Literature Review. 23rd International Web for All Conference (W4A 2026). arXiv:2605.13873.

- Calò, T., Gurita, A.-E., De Russis, L. (2026). Measuring the Semantic Accessibility Gap in LLM-Generated Web UIs. Extended Abstracts of the 2026 CHI Conference on Human Factors in Computing Systems (CHI EA 2026). doi:10.1145/3772363.3799364.
- Chiou, P. T., Alotaibi, A. S., Halfond, W. G. J. (2021). Detecting and Localizing Keyboard Accessibility Failures in Web Applications. ESEC/FSE 2021.
- Chiou, P. T., Alotaibi, A. S., Halfond, W. G. J. (2023). Detecting Dialog-Related Keyboard Navigation Failures in Web Applications. 45th International Conference on Software Engineering (ICSE 2023), 1368–1380. doi:10.1109/ICSE48619.2023.00120.
- Clopper, C. J., Pearson, E. S. (1934). The Use of Confidence or Fiducial Limits Illustrated in the Case of the Binomial. Biometrika 26(4), 404–413. doi:10.1093/biomet/26.4.404.
- Deque Systems. Automated Accessibility Testing Coverage Report. https://www.deque.com/automated-accessibility-testing-coverage/
- Fathallah, N., Hernández, D., Staab, S. (2025). AccessGuru: Leveraging LLMs to Detect and Correct Web Accessibility Violations in HTML Code. arXiv:2507.19549.
- Fok, R., Zhong, M., Ross, A. S., Fogarty, J., Wobbrock, J. O. (2022). A Large-Scale Longitudinal Analysis of Missing Label Accessibility Failures in Android Apps. CHI Conference on Human Factors in Computing Systems (CHI 2022). doi:10.1145/3491102.3502143.
- Fliti, N. E., Kokorina, L., Tambon, F., Papadakis, M. (2026). Flow-A11y: Flow-Aware Accessibility Testing. arXiv:2607.03100.
- Ge, C., Peng, K., Shi, C., Greenman, B., Jiang, Y. (2026). A11yLTLNav: Automatic Detection of Accessibility Navigation Failures. arXiv:2609.17959.
- GitHub (15 May 2026). Building a general-purpose accessibility agent, and what we learned in the process. GitHub Blog. https://github.blog/ai-and-ml/github-copilot/building-a-general-purpose-accessibility-agent-and-what-we-learned-in-the-process/
- Huang, K., Zhu, L., Zhang, J., Xie, X., Chen, C. (2026). A11YRepair: Bridging Web Accessibility Barriers via Knowledge-Enhanced Divide-and-Conquer Repair. ASE 2026. arXiv:2606.21926.
- Khan, M. A., Ali, H., Rana, M., Ilyas, M. S., Bangash, A. A. (2026). On the Adoption of AI Coding Agents in Open-source Android and iOS Development. MSR 2026 Mining Challenge. arXiv:2602.12144.
- Li, H., Zhang, H., Hassan, A. E. (2026). AIDev: Studying AI Coding Agents on GitHub. arXiv:2602.09185.
- Ma, Y., Wobbrock, J. O., Devon, J., Gustafson, A., Fairchild, M., Mankoff, J., Potluri, V. (2026). Prompt and Circumstance: Evaluating Accessibility in LLM-Generated HTML Form Components. 28th International ACM SIGACCESS Conference on Computers and Accessibility (ASSETS 2026). doi:10.1145/3797867.3828975.
- Microsoft. A11y LLM Eval report. https://microsoft.github.io/a11y-llm-eval-report/
- Mohanbabu, A. G., Natalie, R., Kim, B., Guo, A., Pavel, A. (2026). A11y-CUA Dataset: Characterizing the Accessibility Gap in Computer Use Agents. CHI 2026. arXiv:2602.09310.
- Mowar, P., Peng, Y.-H., Wu, J., Steinfeld, A., Bigham, J. P. (2025). CodeA11y: Making AI Coding Assistants Useful for Accessible Web Development. 2025 CHI Conference on Human Factors in

Computing Systems (CHI 2025). doi:10.1145/3706598.3713335. arXiv:2502.10884.

- Oyelayo, O., Abushaqra, G., Asadi, P., Dey, D., Costa, D. E. (2026). LLM Based Web Accessibility Repair: An Empirical Study of Detection, Remediation, and Cost. arXiv:2605.27716.
- Playwright. Accessibility testing. https://playwright.dev/docs/accessibility-testing
- Salehnamadi, N., Alshayban, A., Lin, J.-W., Ahmed, I., Branham, S., Malek, S. (2021). Latte: Use-Case and Assistive-Service Driven Automated Accessibility Testing Framework for Android. CHI 2021.
- Suh, H., Tafreshipour, M., Malek, S., Ahmed, I. (2025). Human or LLM? A Comparative Study on Accessible Code Generation Capability. arXiv:2503.15885.
- Thapa, S. J. (2026). AI Coding Agents and Web Accessibility: Replication package [Data set]. Zenodo. https://doi.org/10.5281/zenodo.24429397
- Tschoner, M. (2025). A11y-bench: Can Large Language Models Resolve Accessibility Issues? Diploma thesis, TU Wien. https://repositum.tuwien.at/handle/20.500.12708/224566
- Wanscher, L. B., Lorensen, M. H., Shafiq, M. A., Moghaddam, M. T., Alipour, M. (2026). From Blind Edits to Verified Repair: Building Trustworthy User-Side LLM Agents for Web Accessibility. arXiv:2608.24913.
- Winn, R., Chiou, P. T., Halfond, W. G. J. (2026). Lost in Navigation: Detecting Keyboard Navigation Accessibility Issues in Web Pages. ICST 2026.
- World Wide Web Consortium (2024). Web Content Accessibility Guidelines (WCAG) 2.2. W3C Recommendation, 12 December 2024. https://www.w3.org/TR/WCAG22/
- World Wide Web Consortium. Radio Group Pattern. ARIA Authoring Practices Guide. https://www.w3.org/WAI/ARIA/apg/patterns/radio/
- Zhong, M., Chen, X., Win Kyi, D., Li, C., Fogarty, J., Wobbrock, J. O. (2026). TaskAudit: Detecting Functiona11ity Errors in Mobile Apps via Agentic Task Execution. CHI 2026. arXiv:2510.12972.

Software and applications

- axe-core, accessibility engine for automated web UI testing. https://github.com/dequelabs/axe-core
- Bloom housing platform. https://github.com/bloom-housing/bloom (head measured: commit 6f3cda99)
- MyFriendBen benefits screener. https://github.com/MyFriendBen/benefits-calculator and https://github.com/MyFriendBen/benefits-api (front-end head measured: commit 08036681)
- NVDA screen reader, version 2026.3.1. NV Access. https://www.nvaccess.org
- Playwright browser automation. https://playwright.dev
- Work-requirements self-advocacy tool, Code for America. https://github.com/codeforamerica/work-requirements-self-advocacy-tool (head measured: commit 57651811)
- Summer EBT self-service portal, Code for America (not testable in this study). https://github.com/codeforamerica/sebt-self-service-portal