Research Square

Preprints are preliminary reports that have not undergone peer review.
They should not be considered conclusive, used to inform clinical practice,
or referenced by the media as validated information.

Vibe-Coded Interfaces Repeat Their Mistakes: An Accessibility Audit of Deployed AI-Generated Web Applications

Tibor Guzsvinecz

guzsvinecz.tibor@zek.uni-pannon.hu

University of Pannonia

Krisztián Németh

University of Pannonia

Judit Szűcs

University of Pannonia

Short Report

Keywords: accessibility debt, AI-generated code, large language models, European Accessibility Act, vibe coding, web accessibility

Posted Date: November 24th, 2026

DOI: https://doi.org/10.21203/rs.3.rs-11284210/v1

License: This work is licensed under a Creative Commons Attribution 4.0 International License.
Read Full License

Additional Declarations: No competing interests reported.

Abstract

Purpose. Vibe coding, building software through conversation with AI code generation tools without inspecting the generated code, has opened web development to people who could not previously program. Existing studies evaluate the accessibility of code that models produce for controlled prompts. This study asks how accessible the applications that vibe coders actually deploy are, and whether their accessibility defects differ from those of hand-built applications. Methods. We collected publicly deployed web applications from GitHub: projects generated with the Lovable platform (created March 2025 to August 2026) and a control group of React projects created and last modified before the release of ChatGPT. We scanned the home page of each application with axe-core against WCAG 2.2 A/AA rules and analyzed 373 Lovable and 395 control applications. Results. Detectable violations were common in both groups (76.1% of Lovable and 81.5% of control applications). Adjusted for page size and hosting, Lovable applications had fewer violation instances (IRR 0.56, 95% CI 0.42–0.75). Their defect profile differed: missing language, alternative text and form labels were rare, while unnamed buttons were twice as common (29.8% compared to 14.9%), mostly icon-only buttons using the same icon library. Defect profiles were also more homogeneous across Lovable applications, and while violations declined over the observation period, unnamed buttons did not. Conclusion. Vibe-coded applications did not show more automatically detectable violations, but they repeat the same defects. We interpret this through the concept of accessibility debt and argue that platform-level defaults are the most efficient point of intervention.

1. Introduction

Andrej Karpathy first used the term “vibe coding” in early 2025 (Karpathy, 2025). It now describes the process of creating software using natural language interaction with artificial intelligence (AI) code generating tools, with little to no code examination (Fawzy et al., 2026a). The phrase gained popularity more quickly than the practice could be examined. It covered a full range of tools in less than a year, from browser-based platforms that transform a paragraph of text into a deployed web application to agentic coding assistants. In June 2026, Lovable, one such platform, revealed that it was being utilized to launch one million new projects per week (Bort, 2026).

The literature on vibe coding has so far been framed around access in one specific sense: people who could not build software now can. Practitioner accounts name this as a primary motivation (Fawzy et al., 2026a), non-developers in survey work give it as their main reason for using the tools (Fawzy et al., 2026b), and blind programmers use conversational code generation to work around development environments that were never designed for them (Chen et al., 2026). Accessibility has a second meaning, however, and the two meanings now pull in opposite directions. The interfaces that are easier than ever to create may be harder to use for the estimated 1.3 billion people, 16% of the world population, who experience significant disability (World Health Organization, 2022). Furthermore, in the 2026 WebAIM Million analysis, 95.9% of the top one million home pages had automatically detectable WCAG failures, up from 94.8% the year before, which reversed six years of small improvements (WebAIM, 2026). The

report does not attribute the change to any cause, and neither do we. It does describe the baseline into which generated interfaces are now being added at scale.

Regulation works in the opposite direction. The Web Accessibility Directive (European Parliament and Council, 2016) imposes accessibility criteria for public sector websites and applications in the EU. E-commerce is among the private services that are subject to the criteria of the European Accessibility Act, which went into effect on June 28, 2025 (European Parliament and Council, 2019). The empirical research that follows is evaluated using WCAG, which is now in version 2.2 (W3C, 2024). At the same time that software developers are getting less and less able to examine their work, accessibility is turning into a legal property of software.

The amount of empirical research on the accessibility of code generated by large language models (LLMs) has rapidly increased (Section 2), and the results consistently show that models perform better when asked and make predictable mistakes. Almost all of this work assesses code generated for researchers' selected prompts.

This paper makes two contributions. Conceptually, it argues that accessibility defects in vibe-coded interfaces are structurally invisible to their creators, because the creator's feedback loop is visual and most accessibility defects produce no visual symptom. We call the result accessibility debt, building on the term's recent use in industry reporting (Deque Systems, 2026a), and derive the expectation that model and platform defaults should make these defects more uniform across applications (Section 3). Empirically, it reports an automated audit of deployed applications generated with a widely used vibe-coding platform, compared with hand-built applications from before the release of ChatGPT (Sections 4 and 5). The study addresses four research questions (RQs):

- RQ1. How common are automatically detectable accessibility violations in deployed vibe-coded applications, compared with hand-built applications?
- RQ2. Which types of violation differ between the two groups?
- RQ3. Are the violation profiles of vibe-coded applications more homogeneous than those of hand-built applications?
- RQ4 (exploratory). Do the violations of vibe-coded applications change over the period in which they were created?

During the preparation of this manuscript, the authors used Claude Opus 5 and Claude Opus 5.5 (Anthropic) to assist with language and structural refinement, including improvements to readability, clarity, coherence, logical flow, and the organization of the manuscript. The AI tools were used solely in an editorial capacity and did not generate research data, scientific results, or substantive interpretations. All AI-assisted suggestions and revisions were critically reviewed and edited by the authors, who retain full responsibility for the final content and conclusions of the manuscript.

2. Background

2.1. Accessibility of LLM-generated web code

Early studies asked whether general-purpose chat models produce accessible HTML. In one of the first systematic evaluations, developers prompted ChatGPT to create websites that were then checked with AChecker and WAVE; 74 of the 88 resulting sites had accessibility issues (Aljedaani et al., 2024). The same study found that the model could fix a large share of the violations once they were pointed out, in its own code and in open-source code from GitHub. A single-page case study reached a similar picture in more detail, and combined automated tools with manual keyboard and screen reader testing (Ahmed et al., 2025).

Later work compared models and prompting strategies. Guriţă and Vatavu (2025) contrasted accessibility-agnostic and accessibility-oriented prompts for ChatGPT and Claude, using automated testing, expert evaluation and LLM self-reflection. Accessibility-oriented prompts improved several success criteria, while semantic structure and keyboard navigation remained problematic. A benchmark published in this journal tested four models on eleven web components and found that none reached full compliance without follow-up prompting and human intervention (Abu Doush and Kassem, 2025). A comparison of human-written and LLM-generated code found that models handled basic requirements such as color contrast and alternative text better than humans, but struggled with more complex features such as ARIA attributes; a feedback-based prompting approach narrowed the gap (Suh et al., 2025).

A third line of work treats accessibility as something to optimize directly. A11yn aligns a code model with an automated auditor as the reward signal and introduces a benchmark of real-world UI requests (Yoon et al., 2025). Post-render verification builds accessibility profiles into the generation pipeline and checks the rendered result rather than the source (Jerry et al., 2027). Related studies use LLMs to detect and repair violations in existing pages (He et al., 2025; Fathallah et al., 2025; Oyelayo et al., 2026).

Two features of this literature matter for our argument. The first is its unit of analysis: a prompt and its output, evaluated largely with automated tools. The second is who is missing. A systematic review of 33 studies on LLMs for web accessibility found that evaluation practice emphasizes technical feasibility over user-centered validation with people with disabilities (Aljedaani and Mollik, 2026). Where humans are involved, they are usually experts; Guriţă and Vatavu (2025) name the absence of users with disabilities as a limitation of their own study.

2.2. Vibe coding as a practice

Research on vibe coding is younger and comes mostly from software engineering and the psychology of programming. An early observational study of think-aloud coding sessions described vibe coding as a shift of expertise rather than its removal: the programmer still evaluates, but faster and more selectively (Sarkar and Drosos, 2025). A grey-literature review of practitioner sources identified speed as the dominant motivation and found that quality assurance is often reduced or handed back to the model (Fawzy et al., 2026a). A follow-up survey across non-developers, novices and professionals described a

perception-action gap: awareness of the risks of generated code is broadly shared, but the ability to verify and debug depends on experience, and some non-developers never check the generated code at all (Fawzy et al., 2026b). Work with novices in a hackathon setting reported fast prototyping alongside weak engineering rigor (Gama et al., 2026), and a recent state-of-the-art review covers practice, productivity and risk in general (Michels et al., 2026).

Self-assessment is a recurring problem in this literature. In a randomized trial, experienced open-source developers expected AI tools to cut their completion time by 24% and afterwards estimated a 20% reduction, while the measured effect was a 19% increase (Becker et al., 2025). The trial concerns productivity, not accessibility, but it shows how far perceived and actual effects of AI assistance can drift apart.

The risk that has received the most attention so far is security. Code completions from GitHub Copilot were vulnerable in about 40% of generated programs across high-risk scenarios (Pearce et al., 2022). In a user study, participants with an AI assistant wrote less secure code than those without one, and were more likely to believe their code was secure (Perry et al., 2023). Deng et al. (2026) extended this line to deployed vibe-coded applications, assembling a corpus of 10,517 open-source projects and analyzing a sample of publicly reachable deployments. We did not find a comparable study of the accessibility of deployed vibe-coded applications. The closest work examines accessibility in the opposite direction: Madan et al. (2025) developed heuristics for evaluating whether vibe-coding tools themselves can be used with a screen reader, and applied them to three conversational coding environments.

2.3. Accessibility in software practice

Accessibility was a weak point of software practice before generative tools arrived. Practitioners tend to treat it as a late-stage concern, and knowledge differs strongly between those with direct and indirect experience of it (Bi et al., 2022). Education does not close the gap: in a survey of US computing faculty, about 20% of respondents reported teaching accessibility topics (Shinohara et al., 2018). Users rarely fill it either. A survey of 104 users with disabilities found that accessibility problems are seldom reported in app reviews, with scepticism about impact and technical difficulty among the barriers (Santos et al., 2026).

Automated checking has well-known limits. A benchmark of evaluation tools found that they covered at most 50% of WCAG success criteria, with completeness between 14% and 38% (Vigo et al., 2013). Guideline conformance is itself an incomplete proxy: only 50.4% of the problems blind users encountered in a user study were covered by WCAG 2.0 success criteria (Power et al., 2012). Recent work on mobile apps finds the same distance between what automated interventions detect and what screen reader users experience (Huq et al., 2026). Automated remediation can also make things worse. In a study of accessibility overlays, 20 of 47 blind and low-vision survey respondents (42.55%) reported that they had abandoned or stopped visiting websites they used regularly because of an overlay (Makati et al., 2024).

2.4. The gap

The two literatures meet at a point neither has examined. Accessibility studies measure what a model produces for a prompt, assuming that someone competent will read the result. Vibe coding studies describe people who, by definition, do not read it. This paper asks what happens to accessibility when the evaluation step changes from inspecting code to looking at a rendered page, and examines the question on applications that have actually been deployed.

3. Conceptual framework: accessibility debt

The studies reviewed in Section 2.1 agree on two points: models produce accessibility violations by default, and they avoid many of them when asked. Capability is not the bottleneck. We argue that the bottleneck is the feedback loop of the person who creates the interface.

A vibe coder evaluates the output by looking at it. The loop is short and visual: describe, render, inspect, adjust. It works for properties the creator can perceive, such as layout, color and whether a button does something when clicked with a mouse. It fails for properties that produce no symptom in that loop. A missing form label, a heading hierarchy that exists only visually, a focus order that skips the main action, or a custom widget that a screen reader announces as a generic group: none of these change what the creator sees. The interface looks finished because, for the creator, it is finished.

We call the result accessibility debt: accessibility defects that are shipped because they cannot be observed in the creator's own evaluation loop, and whose cost is paid by users who are absent from that loop. The debt metaphor goes back to Cunningham (1992), who compared shipping first-time code to going into debt, acceptable as long as it is paid back promptly. The accessibility variant is already in use in industry reporting (Deque Systems, 2026a); our aim is to give it a mechanism. Three properties set it apart from ordinary technical debt.

- Invisibility: Technical debt is usually known to the team that takes it on, even when repayment is postponed. Accessibility debt in vibe-coded interfaces is not postponed; it is never registered. Survey work shows that awareness of the risks does not translate into the ability to verify (Section 2.2), and for accessibility the gap is wider, since verification needs assistive technology or knowledge of standards that non-developers have no reason to hold. Overconfidence adds to it. Users of AI assistants in a security study wrote less secure code and trusted it more (Perry et al., 2023), and an industry survey found that 88% of engineering leaders trust AI-generated code to be accessible (Deque Systems, 2026a). Automation bias, the tendency to accept the output of an automated aid without sufficient checking, is documented in both novices and experts and is not removed by training or instructions (Parasuraman and Manzey, 2010). If professionals overestimate the accessibility of generated code, there is little reason to expect better calibration from people who have never opened a screen reader.
- Externality: The creator is not the one who pays. A security flaw in a vibe-coded application can eventually hurt its owner through a breach, and research on deployed applications has begun to document such flaws (Deng et al., 2026). An accessibility defect hurts a user who silently leaves.

Users with disabilities rarely report accessibility problems even through channels designed for feedback (Santos et al., 2026). No error report arrives and no crash is logged, so the signal that could close the loop is lost where it is generated.

- Replication at scale: A human developer produces idiosyncratic errors. A model produces its defaults, for everyone. Models trained on web corpora that contain accessibility flaws tend to reproduce them (Yoon et al., 2025). If a model habitually renders a custom dropdown without keyboard support, that defect is copied into every application whose creator asked for a dropdown and nothing else. Combined with platform-level volume (Deng et al., 2026), this turns individual model habits into population-level barriers.

This framing also resolves an apparent contradiction in how vibe coding is discussed. Vibe coding has opened software creation to people previously excluded from it, blind programmers included (Chen et al., 2026). The same technology that widens access to making software can narrow access to using it, and both effects come from one property: the model takes over work that the creator no longer needs to understand. That is empowering when the work is syntax. It becomes a liability when the work is making sure other people can use the result.

A last point concerns automated auditing. Most empirical studies in this area rely on tools such as axe-core, whose coverage is limited (Section 2.3). They check whether an attribute is present, not whether it means anything: in one analysis of 300 LLM-generated interfaces, 541 semantic violations such as generic button labels passed conventional checks (Calò et al., 2026). An automated check in the vibe-coding loop would reduce the debt, but it would not remove it, and it could produce a signal of compliance that is harder to question than no signal at all. The history of accessibility overlays, marketed as automated fixes and abandoned by many of the users they were meant to serve (Makati et al., 2024), shows how that can go.

The framework leads to expectations that an audit of deployed applications can check, within the limits of automated testing. If model capability is not the bottleneck, vibe-coded applications need not show more violations overall than hand-built ones, and some classes of error that models handle well may even become rarer (RQ1, RQ2). If defaults are replicated at scale, the violations that do occur should be more alike from one vibe-coded application to the next than violations in hand-built applications are (RQ3). The framework makes no prediction about which specific rules will differ; that is an empirical question.

4. Method

4.1. Sampling

We collected applications through the GitHub repository search API on 23 September 2026. There were two groups: a vibe-coded group, and a control group.

Regarding the former, Lovable, a browser-based platform that generates full web applications from natural-language prompts, creates a default README for exported projects that links to the project page on lovable.dev. We searched for repositories whose README contains the string “lovable.dev/projects” and that were created between 1 March 2025 and 31 August 2026. Searches were run month by month, retrieving up to 200 repositories per month. This fingerprint differs from the one used by Deng et al. (2026), who searched repository code for the author metadata that Lovable writes into the HTML; both identify projects that originate from the platform, but neither can show that the code was not edited afterwards.

In the case of the latter, we searched for repositories that mention React in their name, description or topics, were created between 1 January 2020 and 31 October 2022, and were last pushed before 30 November 2022, the release date of ChatGPT. Restricting the last push date excludes projects that may later have been modified with generative tools. React was chosen because Lovable generates React applications.

In both groups we kept only repositories whose homepage field pointed to a deployment on a common static hosting service (lovable.app, vercel.app, netlify.app, github.io, pages.dev), and only one repository per owner. This yielded 880 Lovable and 470 control applications. We drew a random sample of 470 applications from each group (seed 20260923).

4.2. Measurement

Each application’s home page was loaded once in headless Chromium through Playwright for Python 1.63.0 (Microsoft, 2026), with a 1280 × 800 viewport, a 30-second loading timeout and a further 3 seconds for client-side rendering. We then injected axe-core 4.13.0 (Deque Systems, 2026b) and ran all rules tagged for WCAG 2.0, 2.1 and 2.2 at levels A and AA. For each application we recorded the violated rules, the number of affected elements per rule (violation instances), the impact level reported by axe-core, and the number of elements in the rendered document as a measure of page size. Scans took place on 23 September 2026 between 18:44 and 20:15 UTC.

To characterize the most distinctive violation, unnamed buttons, we rescanned all analyzed applications with at least one button-name violation later the same evening. axe-core reports a shortened HTML excerpt for each violating element, which omits child elements, so for each flagged button we located the element through the CSS selector reported by axe-core and recorded whether it contained an SVG element, whether it contained any text, and the class names of the button and of its icon. From the class names we identified the icon library (Lucide icons carry the class lucide) and buttons whose class string contains inline-flex items-center justify-center, the opening of the base classes of the shadcn/ui Button component in its pre-v4 releases (shadcn-ui, 2025). Of the 252 Lovable buttons matched this way, 248 also carried the shadcn/ui theme token ring-offset-background, which supports the attribution. The current v4 component uses a different class order and would not be matched, so this count is a lower bound.

Applications were excluded when the page returned an HTTP error status, failed to load, or rendered as an effectively empty page (fewer than 15 elements or fewer than 20 characters of visible text).

4.3. Analysis

Analyses were run in R 4.6.1 (R Core Team, 2026). For RQ1 we compared the share of applications with at least one violation and the distributions of violated rules and violation instances between groups, with Wilcoxon rank-sum tests. Because the vibe-coded pages were considerably larger, we also fitted negative binomial regressions (MASS 7.3–65; Venables and Ripley, 2002) of violation instances and of distinct violated rules on group, log page size and hosting service (reference: vercel.app). As a sensitivity analysis, the model for violation instances was refitted on applications hosted on vercel.app only, the one service well represented in both groups. For RQ2 we compared the share of applications violating each rule with Fisher's exact tests and Holm correction across all rules observed (Holm, 1979). For RQ3 we represented each application with at least one violation as a binary vector of violated rules and computed Jaccard distances. Homogeneity was tested as the difference in multivariate dispersion between groups (Anderson, 2006), and differences in profile location with PERMANOVA controlling for hosting (Anderson, 2001), both with 999 permutations in vegan 2.7-6 (Oksanen et al., 2026). The button-level data were analyzed descriptively. For RQ4 we used the repository creation date as a proxy for when a Lovable application was generated and modeled violation instances (negative binomial) and the presence of unnamed buttons (logistic regression) on months since March 2025, controlling for log page size.

4.4. Ethics

Only publicly available pages were loaded, once each, without logging in, submitting forms or interacting with the application. No personal data were collected. Applications are reported in aggregate, and the repository names are not published with the results.

5. Results

5.1. Sample

Of the 470 sampled applications in each group, 395 control and 373 Lovable applications could be analyzed. The most frequent reason for exclusion in both groups was an HTTP 404 response (43 control, 79 Lovable), followed by empty pages (21 and 11). Hosting differed between groups: 352 of the 373 analyzed Lovable applications ran on vercel.app, while the control applications were spread across github.io (158), netlify.app (138) and vercel.app (96). The full list of exclusions can be observed in Table 1.

Table 1
Exclusions by group.

Reason for exclusion	Control	Lovable
HTTP 404	43	79
Empty page	21	11
HTTP 451	6	2
Other HTTP error (402, 500, 503)	4	3
Network error or timeout	1	2
Total excluded	75	97
Analyzed	395	373

5.2. Prevalence of violations (RQ1)

Detectable violations were common in both groups (Table 2). At least one violation was found in 81.5% of control applications and 76.1% of Lovable applications, and at least one violation of critical or serious impact in 81.0% and 74.8%. The median application violated one rule in both groups. The number of distinct rules violated was slightly lower in the Lovable group (mean = 1.47 compared to 1.75; Wilcoxon $p = 0.037$), while the number of violation instances did not differ (median = 3 in the control group and median = 4 in the Lovable group; $p = 0.50$; Fig. 1).

The groups differed in page size. The median Lovable home page contained 311 elements, against 111 in the control group. Table 3 reports negative binomial models of the two violation measures, adjusted for page size and hosting. Page size was the strongest predictor of violation instances: each unit increase in log page size multiplied the expected number of instances by 2.61 (95% CI [2.39, 2.86]; Model 1). With page size and hosting held constant, Lovable applications had fewer violation instances (*IRR* = 0.56, 95% CI [0.42, 0.75], $p < 0.001$; Model 1) and violated fewer distinct rules (*IRR* = 0.66, 95% CI [0.55, 0.79], $p < 0.001$; Model 2). The result held among applications hosted on vercel.app only (96 control, 352 Lovable; *IRR* = 0.50, 95% CI [0.37, 0.69]; Model 3).

Table 2
Descriptive statistics by group.

Group	Control	Lovable
n	395	373
Apps with ≥ 1 violation, %	81.5	76.1
Apps with ≥ 1 critical or serious violation, %	81.0	74.8
Distinct rules violated, median (IQR)	1 (1)	1 (1)
Violation instances, median (IQR)	3 (7.5)	4 (10)
Violation instances, mean	10.7	9.6
Serious instances, median	2	3
Page size (DOM elements), median	111	311

Table 3
Negative binomial models of accessibility violations. Values are incidence rate ratios with 95% confidence intervals. * $p < 0.05$, ** $p < 0.01$, *** $p < 0.001$. “Other” hosting comprises github.io, netlify.app and pages.dev. Model 3 is a sensitivity analysis restricted to the one hosting service well represented in both groups.

Group: Lovable (ref. control)	Model 1: Violation instances	Model 2: Distinct rules violated	Model 3: Violation instances, vercel.app only
	0.56 [0.42, 0.75]* **	**0.66 [0.55, 0.79]*** **	**0.50 [0.37, 0.69]*** **
Log page size (elements)	2.61 [2.39, 2.86]***	1.36 [1.29, 1.43]***	2.88 [2.53, 3.27]***
Hosting: lovable.app (ref. vercel.app)	0.40 [0.19, 0.84]*	0.72 [0.43, 1.20]	-
Hosting: other (ref. vercel.app)	1.07 [0.81, 1.43]	0.96 [0.80, 1.14]	-
Intercept	0.06 [0.04, 0.10]***	0.39 [0.29, 0.52]***	0.04 [0.02, 0.07]***
N	768	768	448
θ (dispersion)	0.68	31.49	0.72

5.3. Types of violation (RQ2)

The top types of violation are shown in Fig. 2, while all can be found in the Appendix in Table A1.

Color contrast was the most frequent violation in both groups (54.7% of control and 58.2% of Lovable applications) and did not differ between them. Four rules differed after Holm correction. Three concern errors that were common in hand-built applications and nearly absent from Lovable applications: a

missing document language (13.2% compared to 0.5%), images without alternative text (12.9% compared to 1.9%) and form fields without labels (11.6% compared to 2.4%). The fourth went the other way. Buttons without an accessible name occurred in 29.8% of Lovable applications and 14.9% of control applications (Holm-adjusted $p < 0.001$). Insufficient target size was also more frequent in Lovable applications (11.3% compared to 6.6%), but the difference did not survive correction.

5.4. Homogeneity of violation profiles (RQ3)

Among applications with at least one violation (322 control, 284 Lovable), Lovable applications lay closer to their group centroid in rule space (mean Jaccard distance to centroid = 0.48) than control applications (0.56). The difference in dispersion was significant (permutation test of multivariate dispersion, $F(1, 604) = 21.8$, $p = 0.001$). The groups also differed in the location of their profiles (PERMANOVA, $F(1, 602) = 11.3$, $R^2 = 0.018$, $p = 0.001$), with a small additional effect of hosting ($F(2, 602) = 2.26$, $R^2 = 0.007$, $p = 0.005$). All permutation tests used 999 permutations, so $p = 0.001$ is the smallest attainable value.

5.5. Change over time (RQ4, exploratory)

Within the Lovable group, applications created later had fewer violations. The share with at least one violation was 81.8% among the 187 applications created between March and November 2025 and 70.4% among the 186 created between December 2025 and August 2026, even though later home pages were larger (median 339.5 compared to 282 elements). Adjusted for page size, expected violation instances fell by about 2.6% per month (*IRR* = 0.974 per month, 95% CI [0.949, 0.999], $p = 0.041$). Unnamed buttons did not follow this trend. They occurred in 32.1% and 27.4% of applications in the two periods, and the monthly change was not significant (*OR* = 0.983, 95% CI [0.941, 1.027], $p = 0.45$).

5.6. What the unnamed buttons contain

The rescan reproduced the violation in 110 of the 111 Lovable and all 59 control applications (Table 4). In the Lovable group, most unnamed buttons (70.4%) contained an SVG icon and no text; in the control group, most (60.0%) contained neither an icon nor text, as is typical of buttons drawn with CSS. Lucide icons appeared in 72.8% of the unnamed Lovable buttons and in 94.5% of the affected Lovable applications, but in none of the control applications. In 44.5% of the affected Lovable applications, at least one unnamed button also carried the shadcn/ui class signature.

Table A1
Rule-level prevalence with Fisher's exact tests and Holm-adjusted *p*-values.

Rule	Control n (%)	Lovable n (%)	Difference (pp)	*p*	*p* (Holm)
color-contrast	216 (54.7)	217 (58.2)	3.5	0.344	1.000
button-name	59 (14.9)	111 (29.8)	14.8	< 0.001	< 0.001
link-name	73 (18.5)	80 (21.4)	3.0	0.321	1.000
html-has-lang	52 (13.2)	2 (0.5)	-12.6	< 0.001	< 0.001
image-alt	51 (12.9)	7 (1.9)	-11.0	< 0.001	< 0.001
label	46 (11.6)	9 (2.4)	-9.2	< 0.001	< 0.001
target-size	26 (6.6)	42 (11.3)	4.7	0.030	0.813
meta-viewport	27 (6.8)	15 (4.0)	-2.8	0.112	1.000
link-in-text-block	16 (4.1)	5 (1.3)	-2.7	0.026	0.724
list	16 (4.1)	9 (2.4)	-1.6	0.227	1.000
scrollable-region-focusable	16 (4.1)	8 (2.1)	-1.9	0.149	1.000
select-name	16 (4.1)	8 (2.1)	-1.9	0.149	1.000
document-title	10 (2.5)	0 (0.0)	-2.5	0.002	0.055
listitem	8 (2.0)	2 (0.5)	-1.5	0.109	1.000
nested-interactive	8 (2.0)	6 (1.6)	-0.4	0.790	1.000
aria-progressbar-name	3 (0.8)	7 (1.9)	1.1	0.212	1.000
svg-img-alt	7 (1.8)	4 (1.1)	-0.7	0.548	1.000
aria-input-field-name	6 (1.5)	2 (0.5)	-1.0	0.288	1.000
aria-valid-attr-value	5 (1.3)	1 (0.3)	-1.0	0.218	1.000
frame-title	5 (1.3)	4 (1.1)	-0.2	1.000	1.000
aria-hidden-focus	2 (0.5)	4 (1.1)	0.6	0.439	1.000
aria-allowed-attr	4 (1.0)	1 (0.3)	-0.7	0.374	1.000
aria-roles	4 (1.0)	0 (0.0)	-1.0	0.125	1.000
role-img-alt	4 (1.0)	0 (0.0)	-1.0	0.125	1.000

Rule	Control n (%)	Lovable n (%)	Difference (pp)	*p*	*p* (Holm)
aria-command-name	3 (0.8)	0 (0.0)	-0.8	0.250	1.000
aria-prohibited-attr	2 (0.5)	1 (0.3)	-0.2	1.000	1.000
aria-required-children	2 (0.5)	1 (0.3)	-0.2	1.000	1.000
html-lang-valid	0 (0.0)	1 (0.3)	0.3	0.486	1.000
aria-required-attr	1 (0.3)	0 (0.0)	-0.3	1.000	1.000
aria-required-parent	1 (0.3)	0 (0.0)	-0.3	1.000	1.000
autocomplete-valid	1 (0.3)	0 (0.0)	-0.3	1.000	1.000
definition-list	1 (0.3)	0 (0.0)	-0.3	1.000	1.000
object-alt	1 (0.3)	0 (0.0)	-0.3	1.000	1.000

Table 4
Content of unnamed buttons by group: icon-only, empty and text-containing buttons, icon library and component signature, at button and application level.

Buttons	Control	Lovable
Unnamed buttons	417	635
Icon only (SVG, no text), *n* (%)	167 (40.0)	447 (70.4)
No icon and no text, *n* (%)	250 (60.0)	155 (24.4)
Text not exposed as name, *n* (%)	0 (0.0)	33 (5.2)
Lucide icon, *n* (%)	0 (0.0)	462 (72.8)
shadcn/ui class signature, *n* (%)	0 (0.0)	252 (39.7)
Applications		
Applications with unnamed buttons	59	110
≥ 1 icon-only button, *n* (%)	44 (74.6)	95 (86.4)
≥ 1 Lucide icon, *n* (%)	0 (0.0)	104 (94.5)
≥ 1 shadcn/ui signature, *n* (%)	0 (0.0)	49 (44.5)

6. Discussion

6.1. Not worse, but different

The audit does not support the simple claim that vibe coding makes the web less accessible. Roughly three in four deployed Lovable applications had detectable violations, a share close to that of the pre-ChatGPT control group. Both figures are below the 95.9% reported for popular home pages by WebAIM (2026), but that analysis used a different tool on a different population of sites, so the numbers are not directly comparable. Once page size is taken into account, Lovable applications had fewer violation instances. This is consistent with the experimental literature, in which models handle basic requirements such as alternative text and color contrast reasonably well (Suh et al., 2025), and with the capability argument of Section 3: models are able to produce accessible markup.

Part of the improvement is probably not the model's doing. A missing document language is a property of the page template, and a platform that starts every project from the same template fixes it once for all of them. The present data cannot separate the contribution of the template from that of the model, and we do not attempt to.

6.2. Replication at scale

The finding that matters most for the framework is the combination of RQ2 and RQ3. Vibe-coded applications failed in more uniform ways, and one failure stood out. Unnamed buttons were twice as common in Lovable applications, and the button-level rescan shows what they are: mostly icon-only buttons, most of them with a Lucide icon (72.8%), often inside a button component with the shadcn/ui class signature. An icon-only button is also the clearest illustration of the invisibility property. It looks complete and it works when clicked, so nothing in a visual feedback loop signals a problem; a screen reader, however, announces it only as "button". The defect is invisible to the creator and repeated across applications, which is exactly the pattern the concept of accessibility debt describes.

Homogeneity has a practical upside. A defect that is spread across many applications because of one default can be removed by changing that default. For a platform, requiring an accessible name whenever an icon is the only content of a button, whether through the component library, the code model or a check before deployment, is a single change that the data suggest would affect a large share of deployed applications. Hand-built applications, with their more varied failures, offer no comparable lever. The exploratory trend analysis adds a further detail. Violations overall declined across the eighteen months covered by the sample, as one would expect with improving models and templates, while the share of applications with unnamed buttons did not change significantly. A defect that survives general improvement is the signature of a default rather than of chance.

6.3. Implications

For platforms and model developers: The most efficient point of intervention is upstream. Accessible defaults for common components, starting with icon buttons, and alignment of models with accessibility checks (Yoon et al., 2025) address defects at the level where they are replicated. The experience with

accessibility overlays (Makati et al., 2024) is a warning against applying automated fixes after deployment instead.

For creators and educators: Because the defects that remain are invisible in the creator’s loop, they have to be made visible: an audit panel next to the preview, or a keyboard-only and screen reader check as a routine step. Accessibility teaching, already marginal in computing curricula (Shinohara et al., 2018), needs to shift from writing correct markup to verifying generated output.

For regulation: Since 28 June 2025 the European Accessibility Act has covered e-commerce services, while exempting microenterprises that provide services (European Parliament and Council, 2019, Art. 2(2)(f), Art. 4(5)). Many vibe-coded applications are likely to be built by exactly such small operators. Whether accessible defaults at platform level could reach the users that the exemption leaves out is a question for the Commission’s review of the Directive, due by 28 June 2030 (Art. 33).

Lastly, for research: The audit answers only the first of several questions raised by the framework. Studies with users of assistive technology are needed to establish what automated tools miss in generated interfaces (Calò et al., 2026; Power et al., 2012). Tracking studies could repeat the audit as platforms and models change, and interventions that make accessibility visible inside the creator's loop can be tested experimentally.

6.4. Limitations

Automated testing detects only part of the accessibility problems that users encounter (Vigo et al., 2013), so all prevalence figures are lower bounds, and conclusions apply to automatically detectable violations. Only the home page of each application was scanned. The vibe-coded group represents one platform, identified through a README fingerprint; exported projects may have been edited afterwards by hand or with other tools. The control group consists of hobby and portfolio projects from 2020–2022, which are comparable in origin but not in age, framework version or tooling, and survivorship among old deployments may have favored better-maintained projects. Hosting differed between groups; we controlled for it statistically and in a vercel.app-only sensitivity analysis, but residual confounding cannot be excluded. Lovable’s own documentation states that new applications created from 13 May 2026 use TanStack Start rather than React with Vite (Lovable, 2026), so the platform’s default stack changed within the observation window, which may contribute to the trend. Class-name signatures identify the icon library and component style reliably, but they do not show whether the code was generated by the model or copied from the platform's template. The trend analysis relies on repository creation dates, which approximate but do not equal the time of generation, and it was not planned in advance. Finally, page size was strongly associated with violation counts, and the adjusted comparison depends on treating it as a covariate rather than as part of the effect.

7. Conclusion

Vibe coding has made it possible for people without programming skills to build and deploy web applications. In an audit of 373 deployed Lovable applications and 395 hand-built applications from before ChatGPT, the vibe-coded applications did not show more automatically detectable violations. They replaced a set of familiar errors with a narrower and more uniform one, and the most frequent new error, the unnamed icon button, is precisely the kind of defect a creator cannot see by looking at the page. Accessibility debt in vibe-coded applications is therefore less a matter of volume than of repetition. That makes it both harder to notice and easier to fix, provided the fix is made where the defaults are set.

Declarations

Competing interests

The authors declare no conflicts of interest.

Declarations

of generative AI use

Funding

This research received no funding.

Ethics approval

Not applicable. The study analyzed publicly accessible web pages without logging in, submitting forms or collecting personal data.

Data availability

Data is available upon request.

Author Contribution

Tibor Guzsvinecz: Conceptualization, Methodology, Software, Formal analysis, Investigation, Data curation, Visualization, Writing – original draft, Writing – review & editing. Krisztián Németh: Validation, Writing – original draft, Writing – review & editing. Judit Szűcs: Validation, Writing – original draft, Writing – review & editing. All authors read and approved the final manuscript.

Data Availability

Data is available upon request.

References

1. Abu Doush, I., Kassem, R.: Can generative AI create accessible web code? A benchmark analysis of AI-generated HTML against accessibility standards. Univ. Access. Inf. Soc. **24**(4), 3483–3506 (2025). https://doi.org/10.1007/s10209-025-01250-2
2. Ahmed, A., Fresco, M., Forsberg, F., Grotli, H.: From code to compliance: assessing ChatGPT's utility in designing an accessible webpage. A case study. (2025). arXiv:2501.03572
3. Aljedaani, W., Habib, A., Aljohani, A., Eler, M.M., Feng, Y.: Does ChatGPT generate accessible code? Investigating accessibility challenges in LLM-generated source code. In: Proceedings of the 21st International Web for All Conference (W4A '24), Singapore, pp 165–176. (2024). https://doi.org/10.1145/3677846.3677854
4. Aljedaani, W., Mollik, R.H.: Large language models for web accessibility: a systematic literature review. In: Proceedings of the 23rd International Web for All Conference (W4A '26), Dubai, pp 160–171. (2026). https://doi.org/10.1145/3800424.3800452
5. Anderson, M.J.: A new method for non-parametric multivariate analysis of variance. Austral Ecol. **26**(1), 32–46 (2001). https://doi.org/10.1111/j.1442-9993.2001.01070.pp.x
6. Anderson, M.J.: Distance-based tests for homogeneity of multivariate dispersions. Biometrics. **62**(1), 245–253 (2006). https://doi.org/10.1111/j.1541-0420.2005.00440.x
7. Becker, J., Rush, N., Barnes, E., Rein, D.: Measuring the impact of early-2025 AI on experienced open-source developer productivity. (2025). arXiv:2507.09089
8. Bi, T., Xia, X., Lo, D., Grundy, J., Zimmermann, T., Ford, D.: Accessibility in software practice: a practitioner's perspective. ACM Trans. Softw. Eng. Methodol. **31**(4), 1–26 (2022). https://doi.org/10.1145/3503508
9. Bort, J.: Lovable says it has hit $500M in annualized revenue, with 1 million new projects a week. TechCrunch, 9 June 2026. (2026). https://techcrunch.com/2026/06/09/lovable-says-it-has-hit-500m-in-annualized-revenue-with-1-million-new-projects-a-week/
10. Calò, T., Gurita, A.-E., De Russis, L.: Measuring the semantic accessibility gap in LLM-generated web UIs. In: Extended Abstracts of the 2026 CHI Conference on Human Factors in Computing Systems (CHI EA '26), pp 1–5. (2026). https://doi.org/10.1145/3772363.3799364
11. Chen, N., Qiu, L.K., Wang, A.Z., Wang, Z., Yang, Y.: Programmers who use screen readers in the vibe coding era: adaptation, empowerment, and new accessibility landscape. In: Proceedings of the 2026 CHI Conference on Human Factors in Computing Systems (CHI '26), pp 1–20. (2026). https://doi.org/10.1145/3772318.3790726

12. Cunningham, W.: The WyCash portfolio management system. OOPSLA '92 Experience Report. ACM SIGPLAN OOPS Messenger. **4**(2), 29–30 (1992). https://doi.org/10.1145/157710.157715
13. Deng, J., Fan, Z., Meng, R.: Understanding the (in)security of vibe-coded applications. (2026). arXiv:2606.23130
14. Deque Systems: From debt to dividend: how engineering leaders can advance accessibility in the AI era. Survey report and press release, 23 July 2026. (2026a). https://www.deque.com/blog/new-report-shows-volume-of-ai-generated-code-is-significantly-increasing-accessibility-debt-risk/
15. Deque Systems: axe-core, version 4.13.0. (2026b). https://github.com/dequelabs/axe-core
16. European Parliament and Council: Directive (EU) 2016/2102 of 26 October 2016 on the accessibility of the websites and mobile applications of public sector bodies. OJ L 327, 2.12.2016, pp 1–15 (2016)
17. European Parliament and Council: Directive (EU) 2019/882 of 17 April 2019 on the accessibility requirements for products and services. OJ L 151, 7.6.2019, pp 70–145 (2019)
18. Fathallah, N., Hernández, D., Staab, S.: AccessGuru: leveraging LLMs to detect and correct web accessibility violations in HTML code. In: Proceedings of the 27th International ACM SIGACCESS Conference on Computers and Accessibility (ASSETS '25), Denver, pp 1–22. (2025). https://doi.org/10.1145/3663547.3746360
19. Fawzy, A., Tahir, A., Blincoe, K.: Vibe coding in practice: motivations, challenges, and a future outlook – a grey literature review. In: Proceedings of the IEEE/ACM 48th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP '26), pp 212–223. (2026a). https://doi.org/10.1145/3786583.3786866
20. Fawzy, A., Tahir, A., Blincoe, K.: From prompting to verification: how experience shapes vibe coding practices. (2026b). arXiv:2605.24521
21. Gama, K., Calegario, F., Jackson, V., Nolte, A., Morais, L.A., Garcia, V.: Can you feel the vibes? an exploration of novice programmer engagement with vibe coding. In: Proceedings of the IEEE/ACM 48th International Conference on Software Engineering (ICSE '26, SEET track), pp 485–496. (2026). https://doi.org/10.1145/3786580.3786992
22. Guriţă, A.-E., Vatavu, R.-D.: When LLM-generated code perpetuates user interface accessibility barriers, how can we break the cycle? In: Proceedings of the 22nd International Web for All Conference (W4A '25), Sydney, pp 124–134. (2025). https://doi.org/10.1145/3744257.3744266
23. He, Z., Huq, S.F., Malek, S.: Enhancing web accessibility: automated detection of issues with generative AI. Proc. ACM Softw. Eng. **2**(FSE), 2264–2287 (2025). https://doi.org/10.1145/3729371
24. Holm, S.: A simple sequentially rejective multiple test procedure. Scand. J. Stat. **6**(2), 65–70 (1979). https://www.jstor.org/stable/4615733
25. Huq, S.F., He, Z., He, Y., Malek, S.: Bridging the gap between automated intervention and actual user experience: a mixed-methods study on mobile accessibility issues for screen reader users. In: Proceedings of the 2026 CHI Conference on Human Factors in Computing Systems (CHI '26), pp 1–27. (2026). https://doi.org/10.1145/3772318.3791293

26. Jerry, B., Moreno, L., Martínez, P.: Accessible GenAI UI generation with post-render verification. In: Miesenberger, K., Petrie, H., Peňáz, P. (eds.) Computers Helping People with Special Needs. ICCHP 2026. LNCS 16866, pp. 394–401. Springer, Cham (2027). https://doi.org/10.1007/978-3-032-31285-3_47
27. Karpathy, A.: There's a new kind of coding I call vibe coding. Post. X (2025). https://x.com/karpathy/status/1886192184808149383
28. Lovable: FAQ. Lovable documentation. (2026). https://docs.lovable.dev/introduction/faq
29. Madan, S., Surabiyil Bindu, S., Potluri, V.: Accessibility heuristics for vibe coding interfaces. In: Proceedings of the 27th International ACM SIGACCESS Conference on Computers and Accessibility (ASSETS '25), pp 1–5. (2025). https://doi.org/10.1145/3663547.3759729
30. Makati, T., Tigwell, G.W., Shinohara, K.: The promise and pitfalls of web accessibility overlays for blind and low vision users. In: Proceedings of the 26th International ACM SIGACCESS Conference on Computers and Accessibility (ASSETS '24), St. John's, pp 1–12. (2024). https://doi.org/10.1145/3663548.3675650
31. Michels, D.L., Abu Ghazaleh, M., Lazzari, F., Kassem, N., Klein, J.: Vibe coding: practice, performance, productivity, and risk. A state-of-the-art review. (2026). arXiv:2608.20446
32. Microsoft: Playwright for Python, version 1.63.0. (2026). https://github.com/microsoft/playwright-python
33. Oksanen, J., Simpson, G., Blanchet, F., Kindt, R., Legendre, P., Minchin, P., O'Hara, R., Solymos, P., Stevens, M., Szoecs, E., Wagner, H., Bedward, M., Bolker, B., Borcard, D., Carvalho, G., De Caceres, M., Durand, S., Evangelista, H., Hannigan, G., Hill, M., Lahti, L., Martino, C., Ouellette, M., Ribeiro Cunha, E., Smith, T., Stier, A., Ter Braak, C., Weedon, J.: vegan: community ecology package. R package version 2.7-6. (2026). https://doi.org/10.32614/CRAN.package.vegan
34. Oyelayo, O., Abushaqra, G., Asadi, P., Dey, D., Costa, D.E.: LLM based web accessibility repair: an empirical study of detection, remediation, and cost. (2026). arXiv:2605.27716
35. Parasuraman, R., Manzey, D.H.: Complacency and bias in human use of automation: an attentional integration. Hum. Factors. **52**(3), 381–410 (2010). https://doi.org/10.1177/0018720810376055
36. Pearce, H., Ahmad, B., Tan, B., Dolan-Gavitt, B., Karri, R.: Asleep at the keyboard? Assessing the security of GitHub Copilot's code contributions. In: 2022 IEEE Symposium on Security and Privacy (SP), pp 754–768. (2022). https://doi.org/10.1109/SP46214.2022.9833571
37. Perry, N., Srivastava, M., Kumar, D., Boneh, D.: Do users write more insecure code with AI assistants? In: Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security (CCS '23), pp 2785–2799. (2023). https://doi.org/10.1145/3576915.3623157
38. Power, C., Freire, A.P., Petrie, H., Swallow, D.: Guidelines are only half of the story: accessibility problems encountered by blind users on the web. In: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '12), pp 433–442. (2012). https://doi.org/10.1145/2207676.2207736

39. R Core Team: R: a language and environment for statistical computing. R Foundation for Statistical Computing, Vienna. (2026). https://doi.org/10.32614/R.manuals
40. Santos, P.S.H.D., Oliveira, A.D.A., Eler, M.M.: Why are accessibility app reviews rare? Exploring motivations, barriers and opportunities among users with disabilities. In: Ardito, C., et al. (eds.) Human-Computer Interaction: INTERACT 2025. LNCS 16108, pp. 176–199. Springer, Cham (2026). https://doi.org/10.1007/978-3-032-04999-5_11
41. Sarkar, A., Drosos, I.: Vibe coding: programming through conversation with artificial intelligence. In: Proceedings of the 36th Annual Conference of the Psychology of Programming Interest Group (PPIG 2025). arXiv:2506.23253 (2025)
42. shadcn-ui: [feat]: Button behavior. GitHub issue #6329, shadcn-ui/ui repository. (2025). https://github.com/shadcn-ui/ui/issues/6329
43. Shinohara, K., Kawas, S., Ko, A.J., Ladner, R.E.: Who teaches accessibility? A survey of U.S. computing faculty. In: Proceedings of the 49th ACM Technical Symposium on Computer Science Education (SIGCSE '18), pp 197–202. (2018). https://doi.org/10.1145/3159450.3159484
44. Suh, H., Tafreshipour, M., Malek, S., Ahmed, I.: Human or LLM? A comparative study on accessible code generation capability. (2025). arXiv:2503.15885
45. Venables, W.N., Ripley, B.D.: Modern applied statistics with S, 4th edn. Springer, New York. ISBN 0-387-95457-0 (2002)
46. Vigo, M., Brown, J.A., Conway, V.L.: Benchmarking web accessibility evaluation tools: measuring the harm of sole reliance on automated tests. In: Proceedings of the 10th International Cross-Disciplinary Conference on Web Accessibility (W4A '13), pp 1–10. (2013). https://doi.org/10.1145/2461121.2461124
47. W3C: Web Content Accessibility Guidelines (WCAG) 2.2. W3C Recommendation, 12 December 2024. Campbell A, Adams C, Bradley Montgomery R, Cooper M, Kirkpatrick A (eds). (2024). https://www.w3.org/TR/WCAG22/
48. WebAIM: The WebAIM Million: the 2026 report on the accessibility of the top 1,000,000 home pages. (2026). https://webaim.org/projects/million/
49. World Health Organization: Global report on health equity for persons with disabilities. World Health Organization, Geneva. ISBN 978-92-4-006360-0 (2022)
50. Yoon, J., Cho, J., Kim, J., Chung, J., Jeon, J., Yu, Y.: A11yn: aligning LLMs for accessible web UI code generation. (2025). arXiv:2510.13914

Figures

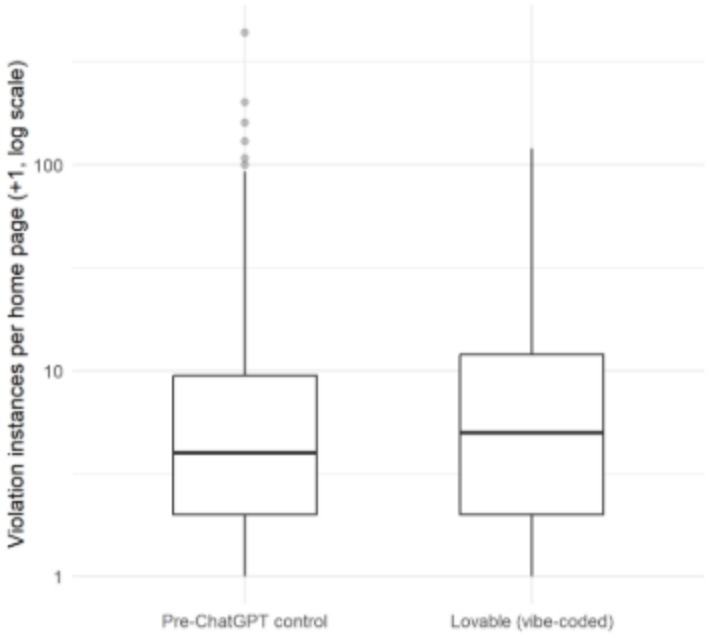

Figure 1

Violation instances per home page by group (log scale, +1 offset).

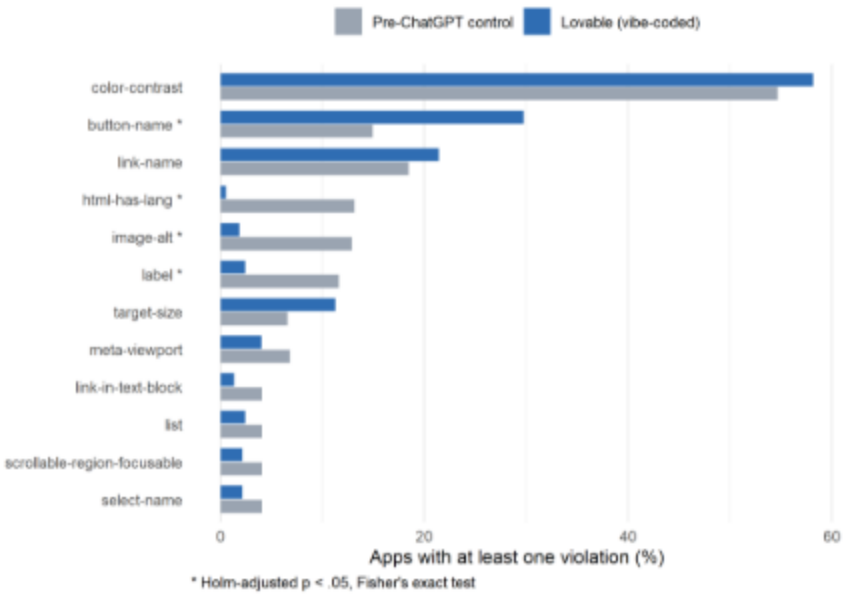

Figure 2

Share of applications violating each of the 12 most frequent axe-core rules, by group.