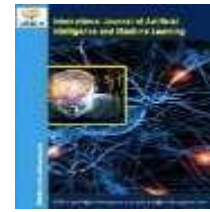




International Journal of Artificial Intelligence and Machine Learning

Publisher's Home Page: <https://www.svedbergopen.com/>



Research Paper

Open Access

AXIS-Bench: A Pilot Benchmark For Interactivity and Accessibility Conformance in Front-End Code Generation

Harsha Vardhan Reddy Nooka¹, Sailesh Kotha², Rajesh Thondamanati³

¹ORCID ID - <https://orcid.org/0009-0000-1680-6605>, Email: hnooka93@gmail.com

²ORCID ID - <https://orcid.org/0009-0008-4727-7839>, Email: saileshkotha@gmail.com

³ORCID ID - <https://orcid.org/0009-0009-7417-420X>, Email: rajeshsvr54@gmail.com

Abstract

Large language models are increasingly used to translate visual designs and natural-language specifications into front-end code, yet existing design-to-code benchmarks score generated pages primarily on visual and structural fidelity to a reference screenshot, and existing accessibility-focused resources evaluate static prompt-code pairs rather than the behavioral, keyboard-operable interactions that most often break accessibility in practice. We present AXIS-Bench, a pilot benchmark that targets the intersection of interactivity and accessibility. Its first release comprises twelve interactive UI component specifications, including disclosure widgets, modal dialogs, custom selection controls, tabbed navigation, sortable data tables, a carousel, a live-validated form, and a keyboard-operable drag-and-drop list, each paired with an explicit WCAG 2.2 and WAI-ARIA Authoring Practices Guide (APG) interface contract. We evaluate three generation conditions: a naive non-semantic baseline, a single-pass ("zero-shot") large language model, and the same model prompted with the relevant APG pattern text ("guideline-grounded"). Evaluation uses two independent, fully automated harnesses: the axe-core accessibility engine, and a purpose-built Playwright test suite that locates every element exclusively through its accessibility-tree role and accessible name, so that a component which is not exposed correctly to assistive technology cannot pass its interactivity test by accident. On this pilot suite, the naive baseline passes 0 of 12 interactivity and ARIA-conformance tasks, the zero-shot condition passes 6 of 12 (McNemar exact $p = 0.031$ against naive), and the guideline-grounded condition passes all 12 ($p = 0.031$ against zero-shot; $p < 0.001$ against naive). We additionally find that axe-core's static rule set, run on isolated components, is dominated by page-scaffold checks that do not distinguish the two LLM conditions, motivating the paired functional harness. We release the task specifications, all 36 generated solutions, and both evaluation harnesses to support replication and extension.

Keywords: front-end code generation; design-to-code; UI-to-code; web accessibility; WCAG; WAI-ARIA; large language models; benchmark; interactivity testing.

1. Introduction

Translating a visual design or a natural-language interface specification into working front-end code, commonly called design-to-code or UI-to-code generation, has become one of the more visible applications of large language models (LLMs) in software engineering. A lineage of benchmarks and datasets has tracked this capability: pix2code first framed the problem as sequence generation from a GUI screenshot [1]; WebSight scaled the idea to 823,000 synthetic screenshot-HTML pairs for training vision-language models [2]; Design2Code curated 484 real webpages to benchmark multimodal code generation against reference screenshots [3]; WebCode2M and Web2Code extended training and evaluation to millions of real, Common-Crawl-sourced instances [5, 6]; and Sketch2Code pushed the input modality further, from finished screenshots to hand-drawn wireframes [4].

Across this lineage, evaluation is dominated by perceptual and structural similarity to a reference: CLIP similarity, block-match ratios, DOM-tree comparisons, and, in several cases, pairwise human preference between the rendered original and the rendered generation. These are reasonable proxies for "does the page look and read like the design," and Design2Code in particular reports that human evaluators preferred GPT-4V-generated pages over the original in 64% of pairwise comparisons [3], a striking result about visual quality. None of these metrics, however, say anything about whether the generated markup can be operated from a

keyboard, whether a screen reader announces its state changes, or whether its interactive widgets expose the roles and names that assistive technology depends on. A page can score well on every visual-similarity metric in the Design2Code suite while being entirely unusable to a screen-reader or keyboard-only user.

A parallel line of work has begun to ask the accessibility question directly. WebA11yBench released 5,000 prompt-code pairs curated from a single model's outputs to support fine-tuning toward WCAG-compliant HTML, CSS, and JavaScript [8]. Suh et al. compared human-written code against LLM regenerations using the IBM Equal Access Checker and QualWeb, and found that naive LLM output already reduced simple violations such as missing alt text and low contrast relative to a human baseline, but that both humans and models struggled with complex ARIA state management, and that closing that harder gap required an explicit feedback loop rather than a single generation pass [9]. That finding is the starting point for this paper: the accessibility failures that survive naive LLM generation are concentrated precisely in the interactive, stateful widgets whose correct behavior depends on keyboard event handling, focus management, and live-region announcements, not in the static, single-pass checks that tools like axe-core, AChecker, or QualWeb can perform without executing the page.

A third line of work evaluates interactivity functionally rather than visually. FrontendBench executes generated front-end code inside a Puppeteer/Jest sandbox against 148 prompt-test pairs spanning demos, tools, mini-games, and data visualizations, reporting 90.5% agreement between its automatic scoring and expert human judgment [7]. This is the right evaluation paradigm for interactivity, treating the browser as an executable oracle rather than an image to be compared, but FrontendBench's test suite targets general functional correctness and does not specifically encode WCAG success criteria or WAI-ARIA Authoring Practices Guide (APG) patterns as pass/fail conditions.

AXIS-Bench (Accessibility x Interactivity Suite Benchmark) is designed to sit at the intersection these three literatures leave open: it specifies interactive widgets against named WCAG 2.2 success criteria and APG patterns, it evaluates them by executing the generated code against automated behavioral test oracles rather than by visual similarity or LLM-as-judge scoring, and its test locators are resolved through the accessibility tree itself, so a widget that is not exposed correctly to assistive technology cannot pass its functional interactivity test by accident. This last design choice is what distinguishes AXIS-Bench methodologically from a generic functional-testing harness: the same assertion that checks "does pressing Escape close the dialog" also requires the dialog to have been locatable by role and accessible name in the first place.

This paper reports the design of AXIS-Bench's first, pilot release (v0.1: 12 tasks across 7 component categories) and an initial empirical study built on it. We generate solutions to all 12 tasks under three conditions: a naive baseline with no ARIA semantics, a single-pass ("zero-shot") LLM condition, and a "guideline-grounded" condition in which the same model is additionally given the literal APG pattern text for the widget it is implementing. We evaluate all 36 resulting solutions with both an axe-core static scan and our role-based interactivity harness. The contributions of this paper are:

1. AXIS-Bench v0.1, a pilot task suite of 12 interactive component specifications spanning 7 categories, each shipped with an explicit WCAG 2.2 / APG interface contract and an executable interaction sequence (Section 3).
2. A dual automated evaluation methodology that pairs axe-core static scanning with a data-driven, accessibility-tree-based Playwright interactivity harness, along with the released harness source (Section 4).
3. A controlled comparison of three generation conditions, with all 36 generated solutions, raw results, and analysis scripts released for replication (Section 5).
4. An empirical, pilot-scale finding that grounding generation in the literal APG specification text closes a measurable interactivity and accessibility gap that naive prompting leaves open, together with the finding that a generic static scanner alone does not detect that gap (Section 6).

The remainder of the paper is organized as follows. Section 2 situates AXIS-Bench against prior design-to-code, accessibility, and interactivity benchmarks. Section 3 describes the benchmark's design principles, task taxonomy, and specification schema. Section 4 describes the dual evaluation methodology and the three generation conditions. Section 5 reports results. Section 6 discusses their implications. Section 7 reports threats to validity and limitations. Section 8 covers ethics, reproducibility, and generative-AI use disclosure. Section 9 concludes.

2. Related Work

2.1 Design-to-Code and UI-to-Code Datasets

pix2code was among the first to frame UI generation as sequence prediction from a screenshot, using a small, synthetically rendered dataset of mobile, web, and desktop mockups [1]. WebSight scaled this synthetic-data approach considerably, releasing 823,000 screenshot-HTML pairs generated from templated layouts to fine-tune vision-language models toward screenshot-to-HTML conversion [2]. Design2Code moved to real-world data, curating 484 webpages from the C4 corpus (plus an 80-example "HARD" subset) and scoring generated HTML against the original screenshot with CLIP similarity, block-match, text similarity, position similarity, and CIEDE2000 color similarity; GPT-4o was the strongest model tested, and human raters preferred GPT-4V's reconstructions over the true original page in 64% of comparisons [3]. WebCode2M scaled further still, drawing 2.56 million training instances and three held-out test splits from Common Crawl, and showed that a model trained on this real-world data outperformed models trained on synthetic corpora, while also documenting that performance degrades substantially as target code complexity increases [6]. Web2Code unified generation and comprehension into a single 1.18-million-pair instruction-tuning and evaluation resource spanning both newly rendered pages and refinements of WebSight and pix2code [5]. Sketch2Code pushed the input modality upstream from finished screenshots to 731 hand-drawn sketches paired with 484 webpage screenshot-HTML references, and found that interactive, feedback-driven generation (the model asking clarifying questions or revising after a review) improved over one-shot conversion across ten evaluated vision-language models [4].

All six resources share an evaluation philosophy: fidelity to a reference image or reference markup, measured by perceptual or structural similarity, or by human/model preference judgments. None specify, and none test, keyboard operability, focus management, or ARIA state as pass/fail criteria.

2.2 Accessibility-Aware Code Generation

WebA11yBench addresses accessibility directly, releasing 5,000 prompt-code pairs curated from a single model's (DeepSeek-V3) outputs, intended for fine-tuning toward production-ready, WCAG-compliant HTML, CSS, and JavaScript covering semantic HTML, ARIA attributes, keyboard navigation, and screen-reader compatibility [8]. As released, it is a static training/reference corpus rather than an executable benchmark: it does not report an automated pass/fail oracle over generated code.

Suh et al. conducted the most directly relevant empirical study, comparing accessibility outcomes across human-written code from ten real GitHub projects, code regenerated by GPT-4o, and code regenerated by Qwen2.5-Coder-32B-Instruct-AWQ, scored with two independent static checkers (IBM Equal Access Checker and QualWeb) against WCAG 2.1 [9]. Naive LLM regeneration already improved on the human baseline (AChecker inaccessibility rate 0.347 to 0.348 for the two models versus 0.425 for humans; QualWeb 0.105 to 0.113 versus 0.125), with the largest gains concentrated in easily automatable checks such as color contrast (nearly 49% fewer issues) and alternative-text validity (over 70% fewer issues). The authors' proposed feedback-driven method, FeedA11y, pushed the best model to an AChecker inaccessibility rate of 0.300, surpassing the human baseline on seven of ten projects, but the paper is explicit that both humans and naively prompted models continued to struggle with complex ARIA attributes. AXIS-Bench is designed to probe exactly that harder tail: rather than scoring arbitrary real-world code with a static checker, it specifies interactive widgets whose correctness is inseparable from ARIA state transitions and keyboard event handling, and tests them functionally.

2.3 Evaluating Interactivity in Generated Front-End Code

FrontendBench evaluates 148 curated prompt-test pairs across five difficulty tiers (from simple static pages to complex interactive features) by executing generated code in a Node.js/Puppeteer/Jest sandbox and scoring it against predefined test scripts, reporting 90.54% agreement with expert human evaluation and a leaderboard led by o3-mini at an 83.11% pass rate [7]. This establishes, for front-end code specifically, the same execute-and-check paradigm that HumanEval established for general program synthesis, where generated code is run against unit tests rather than compared token-for-token or judged by similarity [13]. FrontendBench's test scripts target general functional correctness (does the calculator compute the right answer, does the game respond to input) rather than accessibility semantics; none of its five task categories is built around a named WCAG success criterion or APG pattern.

2.4 Positioning AXIS-Bench

Table 1 summarizes how AXIS-Bench's evaluation basis differs from the five most closely related resources. AXIS-Bench is, to the authors' knowledge, the first published benchmark that combines an explicit per-task WCAG/APG interface contract, execution-based (rather than similarity- or preference-based) pass/fail testing of that contract's interactive behavior, and accessibility-tree-rooted test locators that make the interactivity test and the accessibility test structurally inseparable. Its pilot scale (12 tasks) is far smaller than the million-instance training corpora surveyed above; it is not positioned as a training resource or a leaderboard at that scale, but as a small, hand-verified, extensible seed suite in the tradition of early, carefully curated benchmarks, with a scale-up roadmap discussed in Section 9.

Table 1. AXIS-Bench compared with related design-to-code, interactivity, and accessibility resources.

Resource	Scale	Evaluation basis	Accessibility semantics tested	Interactivity tested
Design2Code [3]	484 pages (+80 hard)	Visual/structural similarity to screenshot	No	No
WebCode2M [6]	2.56M train / 768 test	Visual/structural similarity to screenshot	No	No
Sketch2Code [4]	731 sketches / 484 pages	Visual similarity + VLM/human judge	No	No
FrontendBench [7]	148 tasks	Sandboxed functional execution	No	Yes (general)
WebA11yBench [8]	5,000 prompt-code pairs	Static dataset; no reported executable oracle	Yes (as training target)	No
AXIS-Bench (this work)	12 tasks (pilot) x 3 conditions	axe-core static scan + role-based functional execution	Yes	Yes (ARIA-contingent)

3. AXIS-Bench: Benchmark Design

3.1 Design Principles

Five principles guided the design of AXIS-Bench and distinguish it from the resources surveyed in Section 2.

1. Executable over perceptual. A task's pass/fail outcome must come from executing the generated code against a behavioral oracle in a real browser, never from a similarity score against a reference image or from an LLM-as-judge rating.
2. Contract before code. Every task ships an explicit interface contract naming the required ARIA role and accessible name of each interactive element before any implementation exists, mirroring how the APG itself specifies a pattern. This lets the three generation conditions be compared on the same target semantics rather than being penalized for stylistically different, equally valid DOM structures.
3. Accessibility-tree-first testing. Test locators are resolved by ARIA role and accessible name, using Playwright's role-based locator API, never by CSS selector or DOM id. A component that is not exposed correctly to assistive technology cannot pass the interactivity test by accident, because the test cannot find it.
4. Grounded difficulty. Tasks are anchored to named WCAG 2.2 success criteria and named APG patterns, not to arbitrary component names, so any failure can be traced to a specific, citable specification clause.

5. Small, inspectable, and forkable at first release. The pilot favors a small number of hand-verified tasks whose expected behavior a human reviewer can independently check, over a larger but noisier auto-mined corpus, so that early releases are trustworthy building blocks rather than a source of silently wrong test oracles.

3.2 Task Taxonomy

The pilot release organizes its 12 tasks into 7 categories chosen to span the WAI-ARIA APG patterns most commonly implicated in real front-end accessibility defects: disclosure widgets, overlays and dialogs, selection controls, navigation, data display, forms, and direct manipulation. Table 2 lists each category with its task count and the WCAG success criteria most centrally exercised.

Table 2. Task taxonomy for AXIS-Bench v0.1.

Category	Tasks	Primary WCAG 2.2 criteria exercised	APG pattern(s)
Disclosure Widgets	2	1.3.1, 2.1.1, 4.1.2	Accordion; Disclosure (Show/Hide)
Overlays & Dialogs	2	2.4.3, 2.1.2, 4.1.2, 4.1.3	Dialog (Modal); Live Region (status)
Selection Controls	2	1.3.1, 2.1.1, 4.1.2	Select-Only Combobox; Editable Combobox with List Autocomplete
Navigation	2	2.4.3, 2.4.4, 4.1.2	Tabs; aria-current usage
Data Display	2	1.3.1, 2.2.2, 4.1.2	aria-sort on table headers; Carousel
Forms	1	3.3.1, 3.3.3, 4.1.3, 1.3.1	Form notifications / live region
Direct Manipulation	1	2.1.1, 2.5.7, 4.1.3	Reorderable grid/listbox with keyboard alternative

3.3 Task Specification Schema and Interface Contracts

Every task is authored as a JSON record with the following fields: a unique id and category; a title; a natural-language prompt written as a designer's handoff brief would be; a contract sentence stating the required accessible role and name of each interactive element; a list of WCAG 2.2 success criteria; the governing APG pattern and its canonical W3C URL; and a sequence of executable steps (described in Section 4.2) that operationalizes the task's behavioral requirements as an automated test. Appendix B reproduces one complete task record in full, together with its three generated solutions, as a worked example.

The interface contract is the mechanism that keeps the three generation conditions comparable. Because every condition is told, in the prompt, which accessible name and role each control must expose (for example, a button named "Delete account" that opens a dialog named "Confirm deletion"), differences in test outcomes reflect differences in how faithfully and completely a generation strategy implements the required behavior, not differences in whether the test can find the right element at all.

3.4 The Pilot Task Set (v0.1)

The twelve tasks are: a single-expand FAQ accordion; a multi-expand settings panel; a delete-confirmation modal dialog; a save-confirmation toast notification; a custom (non-native) country select; a city autocomplete combobox; a three-tab account-settings panel; a five-page pagination control; a sortable employee data table; an auto-rotating image carousel; a sign-up form with live validation and a live-updating password-strength indicator; and a keyboard-operable, reorderable to-do list. Appendix A tabulates all twelve tasks with their full WCAG and APG mappings and the number of executable test steps per task (74 steps total, mean 6.2 per task). The task set intentionally spans a difficulty gradient, from single-state toggles (the accordion) to multi-step

composite protocols (the autocomplete combobox and the keyboard-reorderable list), because Section 5 shows this gradient is where the three generation conditions separate most clearly.

4. Automated Evaluation Methodology

Figure 5 gives an overview of the full pipeline described in this section: each task specification is realized under three generation conditions, and every resulting solution is passed through two independent, fully automated evaluators.

Figure 5. AXIS-Bench pipeline: task specification, three generation conditions, and dual automated evaluation

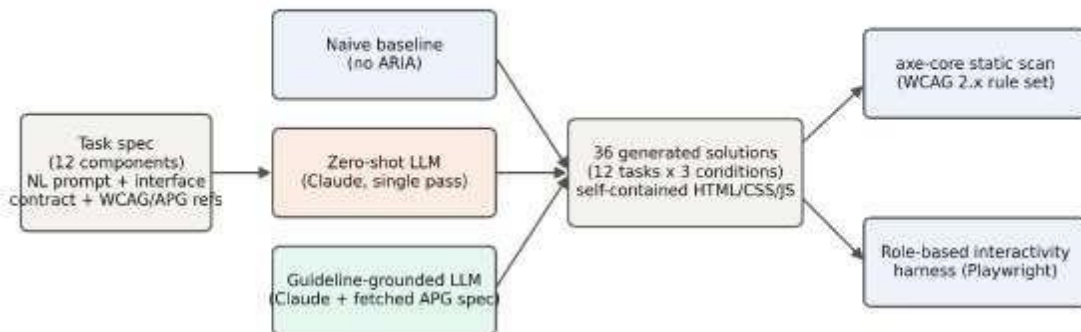


Figure 5. AXIS-Bench pipeline: task specification, three generation conditions, and dual automated evaluation.

4.1 Static Accessibility Scanning (axe-core)

Each of the 36 generated solutions (12 tasks x 3 conditions) is loaded in a headless Chromium instance under Playwright, and scanned with Deque Systems' axe-core accessibility engine [12], restricted to the wcag2a, wcag2aa, wcag21aa, wcag22aa, and best-practice rule tags. For each solution we record every violation's rule id, impact level (critical, serious, moderate, or minor), the WCAG tags it maps to, and the number of affected DOM nodes.

4.2 Role-Based Interactivity Testing

Each task's JSON record includes a sequence of 4 to 10 executable steps, interpreted by a purpose-built Playwright test runner (also released as part of this work). Steps are one of two kinds. Action steps drive the page: click a named role, press a key on the currently focused element, focus a named role and then press a key, fill or type into a field, or wait. Check steps make an assertion: that a named role is visible or hidden; that a role appears a specific number of times; that a named element has a given ARIA attribute value (with the input/textarea value property read directly, rather than the static HTML attribute, since the latter does not track programmatic updates); that the currently focused element is, or is contained within, a named role; or that some element carrying aria-live, role="status", role="alert", or role="log" has announced text containing a given substring within a bounded polling window. Every step in a task's sequence must succeed for that task to count as a pass; partial credit (steps passed out of steps attempted) is also recorded and reported as a secondary metric. Across the 12 pilot tasks, the harness executes 74 total steps (mean 6.2 per task, range 3 to 10).

4.3 Locating Elements Through the Accessibility Tree

Every locator in the interactivity harness is resolved by ARIA role and accessible name (Playwright's getByRole), never by CSS selector or DOM id. This is a deliberate methodological choice, not an implementation convenience: it means the interactivity test and the accessibility test are not independent measurements of the same artifact but are structurally coupled. A component that is visually and behaviorally functional, for example a dropdown that opens and closes correctly on click, but that fails to expose the correct role or accessible name, will fail the interactivity test at the very first step, because the test literally cannot find the element through the

same lens a screen reader would use. Section 6 discusses what this coupling reveals that a purely visual or purely code-structural evaluation would miss.

4.4 Code-Level Metrics

As descriptive, non-evaluative context, we additionally compute four static metrics per solution file by regular-expression analysis of the source: non-blank lines of code; count of explicit `aria-*` attributes; count of explicit `role=""` attributes; and count of semantic HTML tags (`button`, `nav`, `table`, `th`, `td`, `label`, `fieldset`, `legend`, `dialog`, `main`, `header`, `footer`, `section`, `article`) versus generic `div` elements. These are reported as descriptive statistics correlating with, not substituting for, the functional and static-scan results.

4.5 Generation Conditions (Systems Under Test)

Three conditions were generated for every task, all authored in a single session by the same practitioner-and-AI-assisted workflow described in Section 8, using Claude (Anthropic) as the generating model:

1. Naive. A representative accessibility-unaware implementation: functional on click and, where relevant, on native-element defaults, but with no ARIA roles, states, or properties, and no custom keyboard handling beyond what the browser provides for free on the HTML elements used.
2. Zero-shot LLM. A single-pass generation from the task's natural-language prompt and interface contract alone, using the model's general training-time knowledge of accessible web development, with no additional reference material supplied in the prompt.
3. Guideline-grounded LLM. The same model, generating from the same prompt and contract, but with the literal, freshly retrieved W3C APG pattern text for the relevant widget (the exact keyboard-interaction table and required roles/states/properties, as quoted in Sections 3 to 4 of the APG pages cited in Appendix A) included in context.

We emphasize a scope limitation stated in full in Section 7: because all three conditions were produced by the same model family in one controlled setting, this study is an ablation of prompting and grounding strategy, isolating the effect of supplying the literal specification text, and not a cross-model leaderboard. No claim is made, or supported by this data, about how other model families would perform under either prompting condition.

5. Experimental Results

5.1 Interactivity and ARIA-Pattern Conformance

Figure 1 shows the task-level pass rate under all three conditions. The naive baseline passes 0 of 12 tasks (0%). The zero-shot LLM condition passes 6 of 12 tasks (50%). The guideline-grounded condition passes all 12 of 12 tasks (100%). At the finer-grained step level (partial credit across all 74 executable steps), naive passes 21 of 74 steps (28.4%), zero-shot passes 66 of 74 (89.2%), and guided passes 74 of 74 (100%); that is, even where the zero-shot condition fails a task outright, it typically satisfies most of that task's steps and fails on a specific, identifiable behavior, discussed per-task below.

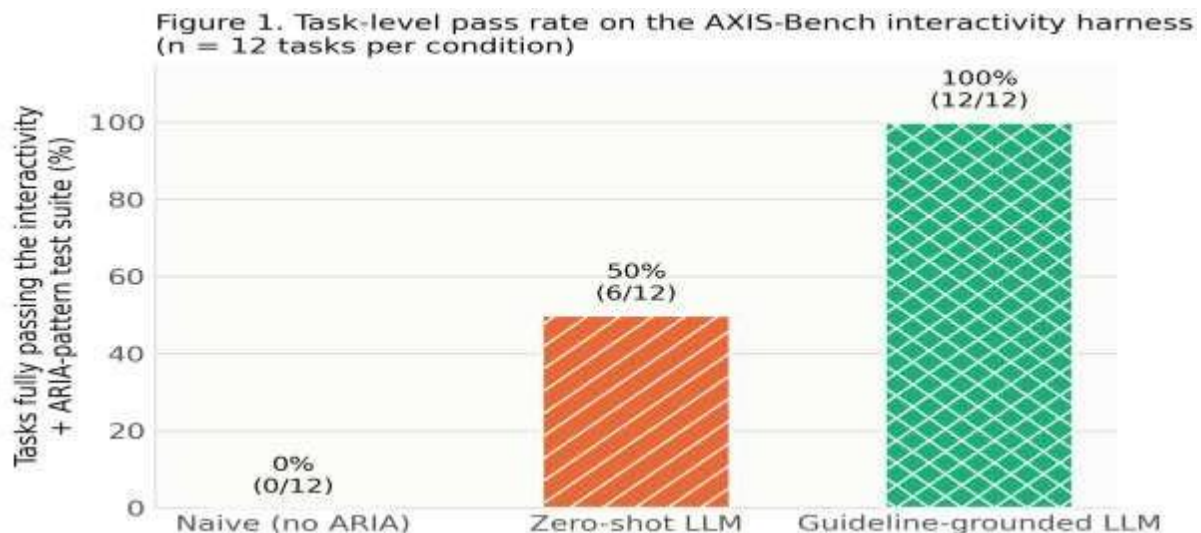


Figure 1. Task-level pass rate on the AXIS-Bench interactivity harness (n = 12 tasks per condition).

Table 3 reports the specific first failure recorded for every task the zero-shot condition did not pass outright; the guideline-grounded condition resolved every one of these by following the literal APG requirement quoted in its prompt.

Task	Zero-shot outcome	First failing behavior	Governing requirement
Delete-confirmation modal	Fail (9/10 steps)	Focus was not returned to the invoking button after the dialog closed	APG Dialog (Modal): focus returns to the invoking element on close
City autocomplete combobox	Fail (6/7 steps)	Arrow keys and Enter did not select a highlighted suggestion; Enter kept the raw typed text	APG Editable Combobox: arrow keys move a highlight, Enter accepts it
Pagination control	Fail (4/5 steps)	aria-current was updated on direct page-link clicks but not when advancing via the Next button	Consistent aria-current="page" on the current page link
Image carousel	Fail (4/5 steps)	Auto-rotation stopped only via the explicit pause button, not when a control received keyboard focus	APG Carousel: rotation stops when any element in the carousel receives focus
Sign-up form with live validation	Fail (7/9 steps)	The password-strength label updated visually but was not exposed in a live region	WCAG 4.1.3 Status Messages
Keyboard-reorderable to-do list	Fail (2/4 steps)	Only mouse drag-and-drop reordering was implemented; no keyboard alternative was provided	WCAG 2.5.7 Dragging Movements: a single-pointer alternative is required

Table 3. Zero-shot condition failures, all resolved by the guideline-grounded condition.

5.2 Category-Level Analysis

Figure 2 disaggregates the task-level results by component category. The two categories built from single, well-known state toggles (Disclosure Widgets) were solved by both LLM conditions; every other category shows the zero-shot condition solving exactly one of its two tasks (or, for the single-task Forms and Direct Manipulation categories, failing outright) while the guideline-grounded condition solves all of them. This pattern is consistent with Suh et al.'s finding that LLM accessibility gains concentrate in simply-checked criteria while complex, composite ARIA state management remains difficult under naive prompting [9]; AXIS-Bench's task selection was designed to probe exactly that composite-state tail, and the category-level results bear that design intent out.

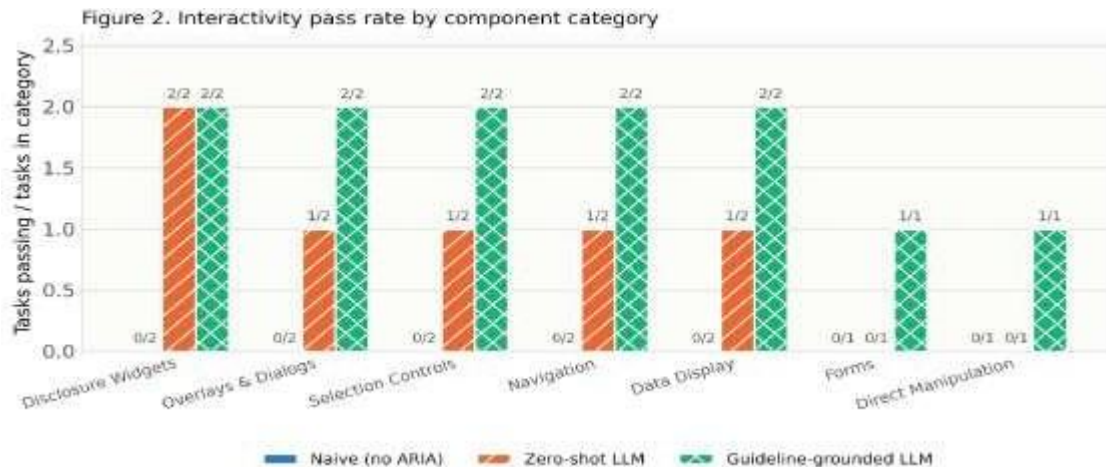


Figure 2. Interactivity pass rate by component category.

5.3 Static Accessibility Scan (axe-core) Results

Figure 3 reports mean axe-core violation instances per solution. Under the full wcag2a/wcag2aa/wcag21aa/wcag22aa/best-practice rule set, the naive condition averages 4.25 violation instances per solution, the zero-shot condition 3.08, and the guideline-grounded condition 3.17; a two-instance critical-impact gap (a missing programmatic label on two naive-condition form inputs, axe-core rule "label") is the only impact-graded difference axe-core detects between conditions. When we exclude three rules whose violations are artifacts of testing isolated single-component pages rather than full documents ("region", "landmark-one-main", and "page-has-heading-one", which fired at nearly the same rate across all three conditions: 63, 36, and 24 total instances respectively, essentially unrelated to which condition generated the component), the mean component-scoped violation count is 0.25 for naive and exactly 0.00 for both zero-shot and guided. In other words, axe-core's static, non-interactive rule set cannot distinguish the zero-shot from the guideline-grounded condition at all; the entire 50-percentage-point interactivity gap documented in Section 5.1 is invisible to it. We return to this finding in Section 6.2.

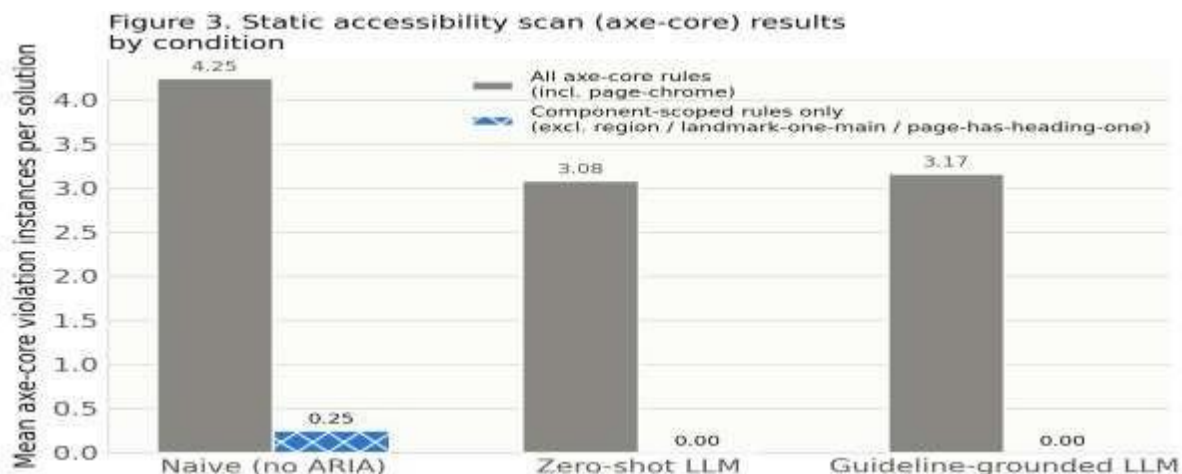


Figure 3. Static accessibility scan (axe-core) results by condition.

5.4 Code Metrics

Figure 4 reports the four static code metrics. Mean ARIA attribute count per solution rises from 0.0 (naive, by construction) to 4.9 (zero-shot) to 6.7 (guided); mean explicit role attribute count rises from 0.0 to 2.9 to 3.6; mean semantic HTML tag count rises from 2.3 to 3.8 to 3.9; and mean lines of code rise from 27.2 (naive) to 45.7 (zero-shot) to 55.5 (guided). The guideline-grounded condition's additional length over zero-shot is concentrated in exactly the code that Table 3 identifies as missing from zero-shot solutions: focus-management

bookkeeping, keyboard event handlers for the specific arrow-key and Home/End behaviors the APG specifies, and live-region status elements.



Figure 4. Static code metrics by condition.

5.5 Statistical Robustness Checks

Because outcomes are paired at the task level (the same 12 tasks under each condition), we report McNemar's exact test on the paired task-level pass/fail outcomes. Zero-shot versus naive: 6 discordant pairs, all favoring zero-shot, exact two-sided $p = 0.031$. Guided versus zero-shot: 6 discordant pairs, all favoring guided, exact two-sided $p = 0.031$. Guided versus naive: 12 discordant pairs, all favoring guided, exact two-sided $p < 0.001$. We report these for transparency and note explicitly, per Section 7, that with $n = 12$ paired tasks these results should be read as suggestive evidence appropriate to a pilot study, not as confirmatory evidence generalizing beyond this task set and this model.

6. Discussion

6.1 What Guideline Grounding Fixed

The six concrete gaps in Table 3 share a structure: each is a behavior that the APG specifies precisely, in a keyboard-interaction table or a required-properties list, but that is easy to omit under a single, unaided generation pass, because the surface-level widget already looks and mostly behaves correctly without it. Returning focus after a dialog closes, stopping carousel rotation on focus rather than only on an explicit pause click, keeping aria-current synchronized across two different code paths that both change the current page, and providing a non-drag alternative to a drag interaction are all easy to miss precisely because their absence does not break the widget's primary, most-obviously-tested interaction (open the dialog, advance the carousel, change the page, reorder the list by dragging). Supplying the literal specification text as additional context, at generation time, resolved every one of these gaps in this pilot. That is a narrow but concrete and actionable finding: for teams generating interactive components with LLM assistance, retrieving and including the relevant APG pattern page for the specific widget being built is a cheap intervention with a measurable effect on this task set.

6.2 What a Generic Static Scanner Missed

Section 5.3's finding, that axe-core's component-scoped violation rate was identically zero for both the zero-shot and guideline-grounded conditions despite a 50-percentage-point gap in functional interactivity conformance, is not a criticism of axe-core, which is a mature, widely deployed, and effective tool for what it is designed to do: static analysis of DOM structure and computed styles without executing user interaction. Every one of the six gaps in Table 3 requires simulating a keyboard interaction (a Tab sequence, an Escape key, a focus event) and observing a resulting DOM mutation or focus change, which is outside the scope of what a static analyzer can check without running the page. The practical implication is that a clean axe-core report, or

a clean report from any comparable static scanner, is necessary but not sufficient evidence that an LLM-generated interactive component is accessible; teams should pair static scanning with functional, role-based interaction tests for any component whose accessibility depends on state that changes at runtime, which in our task taxonomy is most of them.

6.3 Implications for Practice

Two practical recommendations follow directly from Sections 6.1 and 6.2 and are limited in scope to what this pilot study actually shows: first, when using an LLM to generate an interactive widget that has a documented APG pattern, including that pattern's keyboard-interaction and required-properties text in the prompt is a low-cost step that resolved every gap observed in this pilot; second, an axe-core (or similar) scan passing cleanly should not be read as evidence that a generated interactive component is fully accessible, and should be paired with functional tests that exercise the component's documented keyboard behavior.

7. Threats to Validity and Limitations

We report the following limitations explicitly and ask readers not to generalize beyond what the data supports.

1. Single model family. Both LLM conditions, and the naive baseline, were authored using one model (Claude Sonnet 5) in one session. The results characterize this model's behavior under two prompting strategies; they are not evidence, and are not presented as evidence, about LLMs in general or about how other model families (for example GPT, Gemini, or open-weight models) would perform under either condition. Cross-model evaluation is identified as the highest-priority next step in Section 9.
2. Pilot scale. With 12 tasks, individual task idiosyncrasies can dominate aggregate statistics, and the McNemar exact tests in Section 5.5, while correctly computed, should be read as suggestive at this scale rather than confirmatory.
3. Authored, not sampled, naive baseline. The naive condition was authored by the study team to be representative of accessibility-unaware development, following common real-world patterns (click handlers on non-semantic elements, no ARIA), rather than sampled from a corpus of real production code; real naive code could be better or worse than what is modeled here.
4. Coverage of the interactivity harness. Each task's automated sequence tests the specific behaviors enumerated in that task's contract, not exhaustive coverage of everything the governing APG pattern specifies. A solution could in principle satisfy every scripted step while still having accessibility defects the harness does not probe, and, less likely given the contract-first design, could in principle fail a scripted step for a reason unrelated to accessibility.
5. Code metrics are descriptive, not evaluative. Lines of code and attribute counts are static, regular-expression-based proxies for effort and technique. They are reported as descriptive context correlating with the functional results, not as evaluation criteria in their own right, and they cannot verify correctness, only presence.
6. axe-core and the interactivity harness measure different, only partly overlapping things. Section 6.2's finding that they diverge sharply on this task set means the two scores in this paper should not be combined into, or read as interchangeable with, a single unified accessibility score.
7. Single-team authorship without independent replication. All manual judgment calls, including task selection, prompt wording, and the naive baseline's implementation choices, were made by the author team without independent inter-rater review at this pilot stage. Independent replication and adversarial review of the task set are planned before any larger release (Section 9).

8. Ethics, Reproducibility, and Generative-AI Use Disclosure

8.1 Reproducibility

All twelve task specifications (including their WCAG and APG mappings and executable step sequences), all 36 generated solution files, the axe-core and interactivity evaluation harness source code, and the raw JSON and CSV result files underlying every table and figure in this paper are released as supplementary artifacts alongside this manuscript. [Authors: insert the permanent artifact repository URL here prior to submission.]

8.2 Human Subjects and Data

This study involved no human participants, no crowdsourcing, and no personal or proprietary data. All task prompts, contracts, and reference solutions are original material authored by the study team for this work.

8.3 Generative AI Use Disclosure

Consistent with emerging publisher and funder guidance on disclosing the use of generative AI tools in research (for example, COPE's and ICMJE's position that such tools cannot be listed as authors but their use should be disclosed), the authors disclose that Claude (Anthropic) was used as a drafting and engineering assistant throughout this work: to help design the benchmark taxonomy and task specifications, to author the naive, zero-shot, and guideline-grounded reference solutions described in Section 4.5, to implement the evaluation harnesses, to run the reported experiments, and to draft this manuscript. The named authors directed this work, reviewed all generated material and reported results, and take full responsibility for the content, correctness, and conclusions of this paper.

8.4 Conflicts of Interest and Author Contributions

The authors' primary employers had no role in commissioning, funding, or reviewing this work, and no employer confidential information or non-public employer systems were used in its preparation. [Authors: please review and confirm the following CRediT-style contribution statement reflects each author's actual role before submission.] Conceptualization and benchmark design: H.V.R.N. Methodology and evaluation harness engineering: H.V.R.N., S.K. Accessibility and conformance review: R.T. Writing, original draft: H.V.R.N. Writing, review and editing: all authors.

9. Conclusion and Future Work

We introduced AXIS-Bench, a pilot benchmark that evaluates LLM-generated front-end code at the intersection of interactivity and accessibility, using executable, accessibility-tree-rooted test oracles rather than the visual-similarity or static-dataset paradigms of prior design-to-code and accessibility resources. On its first 12-task release, a naive baseline passed none of the interactivity and ARIA-conformance tasks, a single-pass LLM condition passed half, and the same model, given the literal APG specification text for each widget, passed all twelve, a gap invisible to a generic static accessibility scanner run on the same solutions. We release the full task suite, all generated solutions, and both evaluation harnesses.

Four directions follow directly from the limitations in Section 7. First, and highest priority, extending the comparison to independently sourced, third-party model families rather than a single model's behavior under two prompting strategies. Second, scaling the task suite well beyond twelve tasks, with independent, adversarial review of each new task's contract and expected behavior before it is trusted as ground truth, following the small-and-verified-first philosophy argued for in Section 3.1. Third, validating that conformance to the automated harness in this paper actually predicts usability for real assistive-technology users, through a human evaluation with screen-reader and keyboard-only participants. Fourth, extending the taxonomy beyond English-language, LTR components to internationalized and right-to-left interfaces, where several APG patterns (notably arrow-key directionality) have different correct behavior.

References

- [1] T. Beltramelli, "pix2code: Generating Code from a Graphical User Interface Screenshot," arXiv:1705.07962, 2017.
- [2] H. Laurencon, L. Tronchon, and V. Sanh, "Unlocking the Conversion of Web Screenshots into HTML Code with the WebSight Dataset," arXiv:2403.09029, 2024.
- [3] C. Si, Y. Zhang, R. Li, Z. Yang, R. Liu, and D. Yang, "Design2Code: Benchmarking Multimodal Code Generation for Automated Front-End Engineering," in Proc. 2025 Conf. North American Chapter Assoc. Comput. Linguistics (NAACL), 2025.
- [4] R. Li, Y. Zhang, and D. Yang, "Sketch2Code: Evaluating Vision-Language Models for Interactive Web Design Prototyping," arXiv:2410.16232, 2024.
- [5] S. Yun, H. Lin, R. Thushara, M. Q. Bhat, Y. Wang, Z. Jiang, M. Deng, J. Wang, T. Tao, J. Li, H. Li, P. Nakov, T. Baldwin, Z. Liu, E. P. Xing, X. Liang, and Z. Shen, "Web2Code: A Large-Scale Webpage-to-Code Dataset and Evaluation Framework for Multimodal LLMs," in Advances in Neural Information Processing Systems 37 (NeurIPS), Datasets and Benchmarks Track, 2024.
- [6] Y. Gui, Z. Li, Y. Wan, Y. Shi, H. Zhang, Y. Su, B. Chen, D. Chen, S. Wu, X. Zhou, W. Jiang, H. Jin, and X. Zhang, "WebCode2M: A Real-World Dataset for Code Generation from Webpage Designs," in Proc. ACM Web Conf. (WWW), 2025.
- [7] H. Zhu, Y. Zhang, B. Zhao, J. Ding, S. Liu, T. Liu, D. Wang, Y. Liu, and Z. Li, "FrontendBench: A Benchmark for Evaluating LLMs on Front-End Development via Automatic Evaluation," arXiv:2506.13832, 2025.

- [8] T. Mensah-Boateng, W. Aljedaani, and J. Yuan, "WebA11yBench: A Benchmark for Web Accessibility in LLM Generated Websites," Zenodo, 2025.
- [9] H. Suh, M. Tafreshipour, S. Malek, and I. Ahmed, "Human or LLM? A Comparative Study on Accessible Code Generation Capability," arXiv:2503.15885, 2025.
- [10] World Wide Web Consortium (W3C), "Web Content Accessibility Guidelines (WCAG) 2.2," W3C Recommendation, Oct. 2023. Available: <https://www.w3.org/TR/WCAG22/>
- [11] W3C Web Accessibility Initiative, "ARIA Authoring Practices Guide (APG)." Available: <https://www.w3.org/WAI/ARIA/apg/>
- [12] Deque Systems, "axe-core: Accessibility Engine for Automated Web UI Testing." Available: <https://github.com/dequelabs/axe-core>
- [13] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. de Oliveira Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. P. Such, D. Cummings, M. Plappert, F. Chantzis, E. Barnes, A. Herbert-Voss, W. H. Guss, A. Nichol, A. Paino, N. Tezak, J. Tang, I. Babuschkin, S. Balaji, S. Jain, W. Saunders, C. Hesse, A. N. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder, B. McGrew, D. Amodei, S. McCandlish, I. Sutskever, and W. Zaremba, "Evaluating Large Language Models Trained on Code," arXiv:2107.03374, 2021.

Appendix A: Full Task List and WCAG / APG Mapping

ID	Title	Category	WCAG 2.2 SC	APG pattern	Steps
T01	Single-expand FAQ accordion	Disclosure Widgets	1.3.1, 2.1.1, 4.1.2	Accordion	8
T02	Delete-confirmation modal dialog	Overlays & Dialogs	2.4.3, 2.1.2, 4.1.2	Dialog (Modal)	10
T03	Save-confirmation toast	Overlays & Dialogs	4.1.3	Live Region (status)	3
T04	Custom country select	Selection Controls	2.1.1, 4.1.2	Select-Only Combobox	6
T05	City autocomplete combobox	Selection Controls	1.3.1, 2.1.1, 4.1.2	Editable Combobox w/ List Autocomplete	7
T06	Account-settings tabs	Navigation	2.4.3, 4.1.2	Tabs (automatic activation)	7
T07	Search-results pagination	Navigation	2.4.4, 4.1.2	aria-current usage	5
T08	Sortable employee table	Data Display	1.3.1, 4.1.2	aria-sort on column headers	5
T09	Product image carousel	Data Display	2.2.2, 4.1.2	Carousel	5
T10	Sign-up form, live validation	Forms	3.3.1, 3.3.3, 4.1.3, 1.3.1	Form notifications / live region	9

ID	Title	Category	WCAG 2.2 SC	APG pattern	Steps
T11	Keyboard-reorderable to-do list	Direct Manipulation	2.1.1, 2.5.7, 4.1.3	Reorderable grid/listbox	4
T12	Multi-expand settings panel	Disclosure Widgets	1.3.1, 2.1.1, 4.1.2	Disclosure (Show/Hide)	5

Total: 12 tasks, 74 executable interaction-test steps, mean 6.2 steps per task.

Appendix B: Worked Example Task Specification

Task T02, reproduced in full, illustrates the specification schema described in Section 3.3.

Field	Value
id	T02-modal-dialog
category	Overlays & Dialogs
title	Delete Confirmation Modal
prompt	Build a page with a button 'Delete account' that opens a modal dialog titled 'Confirm deletion' containing warning text and two buttons: 'Cancel' and 'Delete permanently'. The dialog must trap keyboard focus while open, be dismissible with Escape, and return focus to the triggering button on close.
contract	Trigger button accessible name 'Delete account'. Dialog accessible name 'Confirm deletion'. Buttons inside named 'Cancel' and 'Delete permanently'.
wcag	2.4.3 Focus Order (A); 2.1.2 No Keyboard Trap (A); 4.1.2 Name, Role, Value (A)
aria_pattern	APG Dialog (Modal) Pattern, https://www.w3.org/WAI/ARIA/apg/patterns/dialog-modal/
sequence (10 steps)	click>Delete account) -> dialog visible -> focus inside dialog -> Tab x3 -> focus still inside dialog -> Escape -> dialog hidden -> focus returned to Delete account

Table B1. Full record for task T02.

The naive solution opens a plain, unstyled-role overlay div on click, with no role, no aria-modal, no focus management, and no Escape handling; it fails at the very first check (the dialog cannot be located by role). The zero-shot solution correctly implements role="dialog", aria-modal, initial focus placement, a Tab-cycling focus trap, and Escape-to-close, but leaves focus wherever the user's last click landed rather than explicitly restoring it to the invoking button, failing only the final check. The guideline-grounded solution adds a three-line change: it records document.activeElement before opening the dialog and calls .focus() on that saved reference (falling back to the trigger button) inside its close handler, which is precisely the behavior the APG's Focus Return requirement specifies. All three solution files, in full, are included in the released artifact set.