Prompt and Circumstance: Evaluating Accessibility in LLM-Generated HTML Form Components

Yumeng Ma
Paul G. Allen School of Computer Science & Engineering | DUB Group
University of Washington
Seattle, Washington, USA
yumengma@cs.washington.edu

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

Joe Devon
GAAD Foundation
Henderson, Nevada, USA
joe@gaad.foundation

Aaron Gustafson
Microsoft
Seattle, Washington, USA
aaron.gustafson@microsoft.com

Michael Fairchild
Microsoft
Lincoln, Nebraska, USA
mfairchild365@proton.me

Jennifer Mankoff
Paul G. Allen School of Computer Science & Engineering
University of Washington
Seattle, Washington, USA
jmankoff@uw.edu

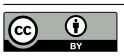
Venkatesh Potluri
School of Information
University of Michigan
Ann Arbor, Michigan, USA
potluriv@umich.edu

Abstract

Large language models (LLMs) are used to generate user interface (UI) code, yet the accessibility of this code is difficult to evaluate in a controlled way. Existing evaluations typically examine full-generated webpages or loosely specified tasks, which makes it difficult to determine where accessibility outcomes come from. We introduce PACE for "Prompt Accessibility Controlled Evaluation," a benchmarking testbed for controlled evaluation of accessibility in LLM-generated HTML form components. Using PACE, we evaluated the accessibility of 58 form components generated by five developer-centric LLMs with variations across three system prompts with increasing accessibility guidance, and three request variants containing increasing detail about each form components. We collected 10 trials per configuration for a total of $n = 26{,}100$ outputs. Our findings show that accessibility in LLM-generated code moved with *prompt* construction and with the *circumstance* around it (e.g., model, form component, and measure used to assess the output). This pattern calls for evaluation designs that can better separate sources of variation in the accessibility of LLM-generated code.

CCS Concepts

• **Human-centered computing** → **Accessibility systems and tools**; **Empirical studies in accessibility**.



This work is licensed under a Creative Commons Attribution 4.0 International License.
ASSETS '26, Vila Nova de Gaia, Portugal
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2521-0/26/10
https://doi.org/10.1145/3797867.3828975

Keywords

Accessibility, large language models, code generation, benchmarking, form components, prompt construction.

ACM Reference Format:
Yumeng Ma, Jacob O. Wobbrock, Joe Devon, Aaron Gustafson, Michael Fairchild, Jennifer Mankoff, and Venkatesh Potluri. 2026. Prompt and Circumstance: Evaluating Accessibility in LLM-Generated HTML Form Components. In *The 28th International ACM SIGACCESS Conference on Computers and Accessibility (ASSETS '26), October 25–28, 2026, Vila Nova de Gaia, Portugal*. ACM, New York, NY, USA, 32 pages. https://doi.org/10.1145/3797867.3828975

1 Introduction

Artificial intelligence (AI) coding tools powered by large language models (LLMs) can generate code, apply accessibility attributes, and build web pages from text prompts. These tools are part of front-end workflows, where developers describe what they want and receive code in return. This shift reflects a move toward conversational programming [10], where developers specify behavior through prompts instead of writing code line by line.

This shift also raises new questions about the accessibility of the user interfaces (UIs) these systems output [14]. Even in domains where LLMs are expected to excel, such as improved coding efficiency, evidence is mixed [16]. Further, when developers use LLMs to code, they may not inspect or verify the result [34]. When this practice is combined with the general lack of education about accessible code development [28, 45] and the known access problems in the training data for LLMs, there is a significant risk that accessibility will decline with LLM-generated code.

These concerns are amplified by the accessibility barriers that already persist across the web. The WebAIM Million report,[1] which documents accessibility conformance failures across the top one million home pages, has shown year after year that the vast majority of websites are inaccessible. These pages are part of the public web corpus on which LLMs are trained, so it is reasonable to expect inaccessible patterns in generated code. Worryingly, the 2026 WebAIM Million report showed a 1.1 percentage-point increase in pages with detectable accessibility failures, as defined by the Web Content Accessibility Guidelines (WCAG).[2] These problems become consequential in reusable UI components within design systems and component libraries, where an accessibility failure can propagate across many applications [31]. Interactive form components, such as inputs, selection controls, and buttons, are central to user interaction and are known to introduce recurring accessibility challenges [32, 33].

One common way to monitor and improve LLM performance is benchmarks. A benchmark refers to a dataset, a set of evaluation metrics, and the processing steps used to assess specific aspects of LLM behavior [35]. Several existing benchmarks evaluate AI-generated code for accessibility (e.g., [11, 18]). However, these benchmarks provide limited perspective into accessibility outcomes. Many evaluate full-generated webpages or loosely specified tasks, where multiple factors—such as layout, styling, and prompt phrasing—interact simultaneously [3, 39]. This makes it difficult to determine which aspects of prompting or model behavior contribute to accessibility failures. Other benchmarks report high-level accessibility outcomes [22] but do not verify whether the required semantic attributes and relationships within a component are present and correctly connected.

To address this gap, we consider form component generation in a controlled experiment. We created PACE, for "Prompt Accessibility Controlled Evaluation," a benchmarking testbed for structured evaluation of accessibility in LLM-generated HTML form components. PACE supports comparisons across *model*, *system instruction*, *component specification*, and *request variant*. We quantified accessibility through node-level Axe pass rate for automated audit performance and normalized schema score for component-level quality.

This controlled setup lets us test the effect of each of these conditions on accessibility outcomes. We investigate three research questions using PACE:

- **RQ1-Effects:** How does accessibility differ across *model*, *system instruction*, and *request variant*?
- **RQ2-Interaction:** How do *system instruction* and *request variant* interact to influence accessibility outcomes?
- **RQ3-Component** How does *component specification* impact the accessibility of generated outputs?

Our study evaluates five development-centered LLMs across three system instructions with stepwise increases in accessibility guidance, 58 form components, and three request variants with progressively greater request detail, with ten trials per configuration, for a total of $n = 5 \times 3 \times 58 \times 3 \times 10 = 26{,}100$ generated outputs.

Our work makes three contributions.

- **Benchmarking Framework** We introduce PACE, a configurable and reproducible benchmarking testbed for controlled evaluation of accessibility in LLM-generated HTML form components. We have made the PACE codebase[3] openly available to support reuse and extension.
- **Experimental Design** We present a large-scale experimental design that isolates how *model*, *system instruction*, *component specification*, and *request variant* relate to accessibility outcomes. We have released the study outputs and prompts[4] to support replication and future comparison.
- **Findings** We report findings that clarify how accessibility in AI-generated code depends on these conditions. We observe pronounced differences across models, mixed effects across prompt construction, and substantial differences across form components.

Our results carry implications for how accessibility in AI-generated code should be evaluated and understood: (1) accessibility in AI-generated code should not be treated as a stable capability that is improved through *prompt* construction, because it also depends on evaluative *circumstance* (e.g., model, form component, and measure used to assess the output); and (2) evaluation design influences what can be claimed about accessible generation in the first place. PACE offers one way to study these dependencies under controlled conditions.

2 Related Work

We situate PACE within three connected lines of work. We begin by discussing the positioning and adoption of AI coding assistants to motivate why generated code requires careful evaluation in practice. We then review studies showing that accessibility remains an understudied aspect of AI-generated code and establish that prompting can impact accessibility outcomes. We close by discussing work on alignment and benchmarking to show why a controlled benchmark is needed to separate the effects of *model*, *system instruction*, *request variant*, *component specification*, and *measurement choices*.

2.1 AI Coding Assistants Create New Evaluation Challenges

Studies of real development practice show that developers use AI for drafting code [37], rewriting code [1], documentation [17], translation [29], and other software maintenance tasks [36], at times with substantial human review and revision after generation [46]. Yet evidence about ultimate benefit is still emerging. Becker et al. [7] found that experienced developers using AI tools on open-source tasks took longer to complete their work, though they expected these tools to help them work faster.

These works steer toward how AI outputs should be evaluated in practice. Recent survey work suggests that developers do not treat AI-generated code as inherently trustworthy but approach it with greater caution, review effort, and scrutiny [19]. However, evidence also notes that engagement may decline as agentic coding assistants take over more of the workflow, with developers focusing more on whether outputs appear correct than on understanding or verifying the underlying process [8]. Liang et al. [21] further

[1] https://webaim.org/projects/million/
[2] https://www.w3.org/WAI/standards-guidelines/wcag/
[3] https://github.com/momentine/pace
[4] https://doi.org/10.5281/zenodo.21303068

found that developers might give up on generated code, struggle to control tool output, and report that AI's suggestions frequently fail to meet functional or non-functional requirements. Prior work also warns that AI code generation can encourage over-reliance, create a false sense of understanding, and reproduce biased or poor coding patterns [6]. These problems can also be harmful to accessibility. When measuring web accessibility, UIs can render correctly but still fail to encode the semantic relationships upon which assistive technologies rely [5]. For example, screen readers depend on HTML structure in the DOM to locate headings, controls, and relationships between UI elements [12]. When these relationships are missing or incorrectly expressed in markup, navigation can become difficult or impossible. Evaluating accessibility in AI-generated web pages therefore requires attention to generated code semantic, not just visual appearance.

2.2 (Un)Stability of Accessibility in AI-Generated Code

Studies that examine accessibility of LLM-generated code consistently report weaknesses in how web applications are structured. Websites generated by ChatGPT frequently contained accessibility violations in default outputs, like missing form labels, absent alternative text, vague link descriptions, and poor contrast [3]. While LLM-generated implementations sometimes reduced basic violations, they still struggled with correct ARIA usage and with organizing UI elements in ways assistive technologies can reliably interpret[39]. Mowar et al. [23, 24] further showed that improvements from AI coding assistants depended on whether developers were already aware of accessibility practices, suggesting that default code suggestions from models do not consistently produce accessible results. These findings collectively show that accessibility is fragile in many AI systems.

In practice, developers typically start by issuing short natural-language requests to coding assistants and then decide whether the generated output is usable or requires revision [4]. However, most of the above studies evaluate generated code after prompting strategies or developer intervention have been applied. To understand the accessibility characteristics present at earlier stages of interaction, PACE makes it possible to observe accessibility-relevant structures before any manual correction or iterative prompting takes place.

Studies suggest that prompt construction can affect the accessibility of generated UIs, but not reliably enough to ensure correct markup. Guriță and Vatavu [14] found that prompts encouraging accessibility practices improved generated banking interfaces in areas such as heading structure, focus behavior, and labeling, yet multiple barriers still remained. Prior work also shows that LLM outputs are highly sensitive to prompt phrasing and format, with small wording changes producing measurable differences in generated results [9, 26, 30]. Motivated by this sensitivity, PACE treats prompt construction as two parts: *system instruction*, which provides general guidance, and *request variant*, which specifies the task. This separation lets us distinguish effects tied to instruction context from those tied to task phrasing.

2.3 Alignment and Benchmarking Address Different Accessibility Stages

Prior work addresses accessibility in AI-generated code at different points in the pipeline, but these efforts are rarely organized in a way that makes their scope easy to compare. Table 1 summarizes this landscape across intervention stage, evaluation target, and what each line of work can and cannot isolate. This perspective distinguishes work that changes model behavior during training from work that evaluates outputs after generation, as well as benchmarking efforts that enable controlled comparison across models.

3 The PACE Testbed

PACE is a testbed for configuring and running controlled experiments on accessibility in LLM-generated HTML form components. Here, HTML form components refer to generated HTML UI widgets used for form interaction, such as text fields, email fields, radio button groups, and toggle switches. A component specification is the benchmark definition for one such widget, specified by its type, label, and attributes.

PACE has four manipulated factors:

- *model* — the LLM used to generate the widget (e.g., GPT, Claude, Gemini, etc.).
- *system instruction* — a fixed high-level prompt that guides the model's overall output behavior. This acts as background context (e.g., *You are a coding assistant. Generate HTML only. Do not include CSS or JavaScript.*).
- *component specification* — the widget to be generated, defined by its {*type*}, {*label*}, and {*attribute*} (e.g., *{component} = text field; {label} = What color is an orange?; {attribute} = with hint "orange"*).
- *request variant* — a prompt template that describes the task, instantiated with values from the component specification (e.g., *Create a {component} for "{label}" {attribute}*, which becomes *Create a text field for "What color is an orange?" with hint "orange".*)

PACE generates test cases by taking all combinations of these factors. Each combination defines one condition. For each condition, PACE builds a complete prompt by combining the *system instruction* with a *request variant* filled using the component specification. This prompt is then sent to the model to generate an HTML output. Sampling parameters (e.g., temperature, top-p, maximum tokens) are fixed within a run. Each condition is repeated a user-defined number of times to capture variance in model outputs. For every run, PACE stores the generated HTML and its configuration metadata. The testbed's workflow is shown in Figure 1.

Following this workflow, we describe three parts of PACE: how accessibility is evaluated, how experiments are executed through the backend pipeline, and what users can configure through the interface.

3.1 Accessibility Evaluation

PACE supports two complementary evaluation mechanisms: node-level Axe pass rate and normalized schema score. These mechanisms evaluate different properties of generated accessibility, though we note that they do not provide a complete evaluation

Table 1: Prior work addressing accessibility in AI-generated code at different stages of the pipeline. Existing efforts differ in whether they intervene during training, evaluate outputs after code generation, or provide shared benchmarking infrastructure.

Work	Stage	Primary target	What it supports	What it does not isolate
A11yn [47]	Training-time alignment	Model optimization for accessibility	Shows that accessibility can improve through alignment objectives during training	Prompt effects, *component specification* effects, and measurement effects at generation time
Abu Doush & Kassem [2]	Post-generation evaluation	Accessibility of generated HTML components	Shows that syntactically valid outputs can still fail WCAG requirements	Whether failures come from model, prompt, component specifications, or measurement
Aljedaani et al. [3] and Guriţă & Vatavu [14]	Page- or task-level evaluation	Accessibility of generated pages, screens, or task outputs	Shows that accessibility weaknesses persist in realistic generation settings	Visible factor-level interpretation since multiple aspects of the task change at once
Microsoft's A11y LLM Eval[a]	Benchmark evaluation	Default accessibility performance under no accessibility guidance	Supports model comparison under a shared no-guidance setup using WCAG pass rates and failure counts	Component-specific structure, prompt-level effects, and measurement beyond the benchmark's selected WCAG checks
AIMAC[b]	Benchmark evaluation	Accessibility of generated web pages across design categories	Supports large-scale comparison across models under shared prompts using automated browser-based auditing	Component-level factor isolation, prompt construction effects, and relationships beyond automated testing alone
WebAccessBench [18]	Benchmark evaluation	Accessibility reliability in generated web interfaces under realistic prompting conditions	Supports comparison across models and guidance regimes using raw defect burden, clean output rate, Structure-normalized burden, and Aggregate benchmark score	*component specification*-level factor isolation, since it evaluates full interface artifacts and does not separate prompt construction and interaction demands at the level of individual components
SciArena[c] and Terminal-Bench[d]	Benchmarking infrastructure	Controlled comparison across models	Shows the value of shared infrastructure for surfacing systematic model differences	Accessibility-specific behavior and component-level accessibility structure

[a]https://microsoft.github.io/a11y-llm-eval-report/ [b]https://aimac.ai/ [c]https://allenai.org/blog/sciarena [d]https://www.tbench.ai/

of accessibility (they do not capture all barriers tied to dynamic interaction, page context, or human judgment). Node-level Axe pass rate evaluates the HTML that is present and asks whether evaluated nodes trigger automated accessibility violations under the selected Axe rules. Normalized schema score asks whether, given the component specification, the model generated the structure, semantics, and relationships expected for that widget, such as labels, descriptions, error relationships, grouping structure, required-state encoding, and correct element and attribute choices. An output can receive a strong Axe result because it avoids detectable violations on generated markup, yet still receive a lower schema score because it omits part of the expected pattern, and vice versa. We therefore report both metrics to distinguish violation-free markup from complete realization of the intended pattern.

3.1.1 Axe. Axe-core (Axe) [5] is a widely used open-source accessibility testing engine for websites and other HTML-based UIs. PACE uses Axe to audit each generated output with a fixed subset of 26 rules relevant to form components (Appendix A, Table A.1). For each output, Axe returns rule-level results categorized as violations, passes, incomplete, and inapplicable.

Because Axe reports results at the rule and node level, PACE aggregates those results at the node level by grouping all checks associated with the same DOM node. Since multiple rules may apply to one node, each node receives one outcome using the following

[5]https://github.com/dequelabs/axe-core

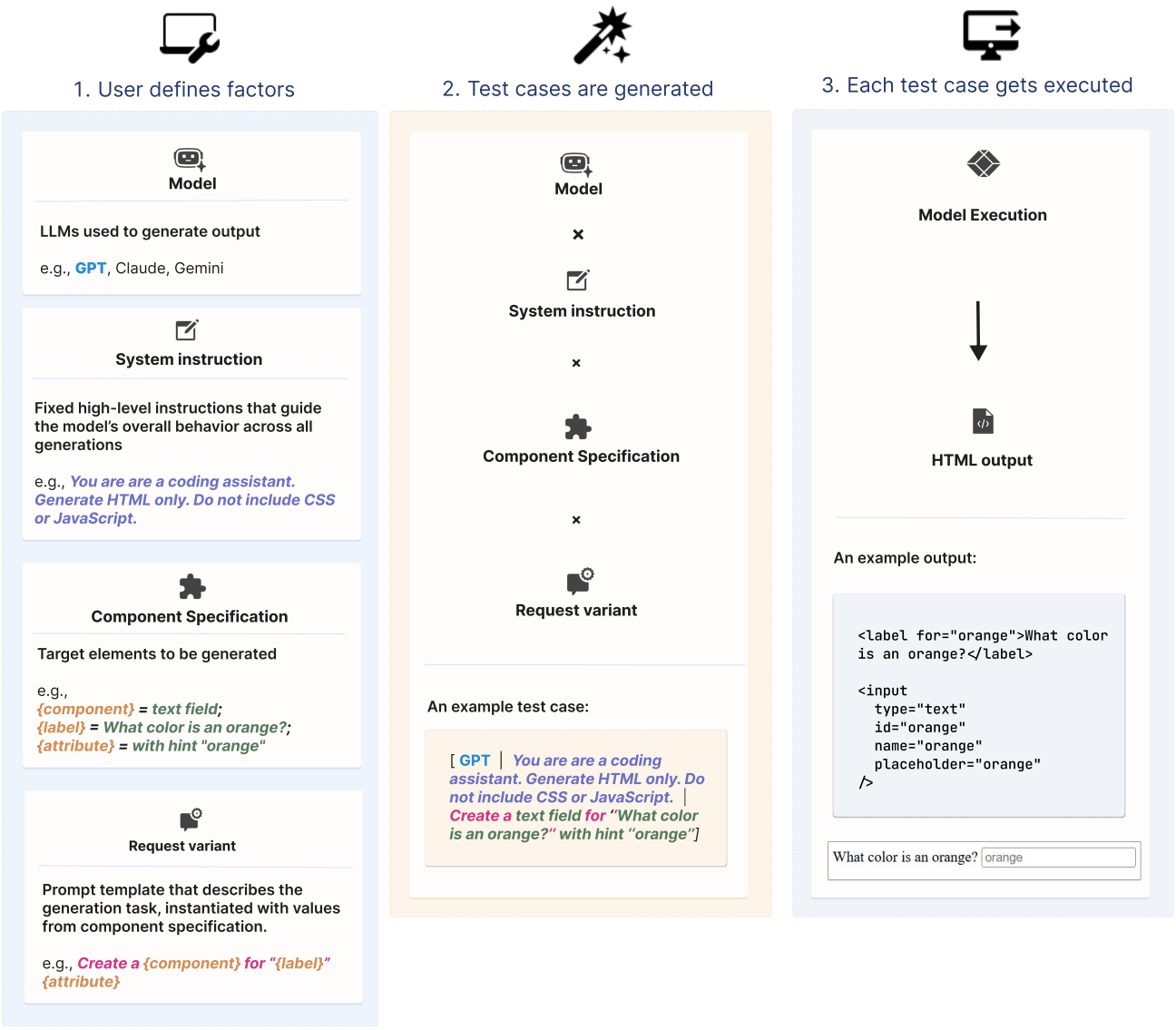


Figure 1: The PACE workflow. The user first creates an experiment by defining models, system instructions, component specifications, and request variants. The testbed then constructs test cases by enumerating combinations of these factors. Each test case is executed to produce an HTML output.

precedence: a node is marked as a violation if any associated rule reports a violation; otherwise, it is marked as incomplete if at least one associated rule is incomplete; otherwise, it is marked as a pass if all associated rules pass.

Let P denote the number of nodes that pass all applicable checks, V the number of nodes with one or more violations, and I the number of nodes with incomplete results. Nodes for which no selected Axe rules apply are excluded from evaluation counts. The node-level Axe pass rate is defined as:

$$\text{PassRate} = \frac{P}{P + V + I} \tag{1}$$

Equation 1 ranges from [0,1], with higher values indicating better accessibility performance. A value of 1.0 means that all evaluated nodes passed all applicable Axe checks. PACE computes this value for each generated widget and uses it as a dependent variable in subsequent analyses when Axe evaluation is enabled.

3.1.2 Schema. PACE computes a normalized schema score for each generated widget by evaluating the output against a predefined set of deterministic, widget -level checks derived from WCAG-relevant requirements for common form components and tied to each component specification in our benchmark. Each check is defined in advance and applied consistently to every output. Table B.1 in Appendix B lists all schema checks, their decision rules, and

their WCAG mappings. This design follows rule-based validation in software testing [43, 44], where expected structural conditions are specified in advance and then checked consistently across outputs. The schema was also cross-checked with expert industry partner-authors using outputs from our form components in the benchmark. This review helped us resolve how checks should apply across generated forms.

Each applicable check produces a score $s_i \in \{0, 1, 2\}$. A score of 2 indicates that the expected accessibility condition is satisfied, 1 indicates partial satisfaction or a weaker implementation, and 0 indicates that the relevant condition is missing or incorrectly implemented. For a widget with M applicable checks, the normalized schema score is computed as:

$$S_{\text{widget}} = \frac{\sum_{i=1}^{M} s_i}{2M} \tag{2}$$

Equation 2 produces a value in [0,1], with higher values indicating that the generated widget more completely satisfies its expected accessibility checks. Table D.2 in Appendix D provides a worked example. Normalization makes scores comparable across widget s with different numbers of applicable checks. The resulting normalized schema score is used as a dependent variable in statistical analyses when Schema evaluation is enabled.

3.2 Execution Pipeline

PACE is implemented as a modular experimental pipeline that separates configuration, generation, evaluation, and storage. All experimental inputs—models, system instructions, request variants, component specifications, and scoring rules—are specified in JSON files and recorded for each run.

During execution, PACE constructs the set of combinations across selected factors, issuing code generation requests through an Azure or OpenRouter client. The provider layer lets PACE support model updates by changing model identifiers, deployment names, API versions, or API integrations but does not change the benchmark design. For OpenRouter, users can run the provided model-fetching script to refresh the available model list before configuring an experiment. For Azure, users configure the relevant deployment names, endpoint, and API version for the models available in their Azure resource. The resulting HTML fragments are stored together with their associated configuration settings.

Each fragment is then evaluated using one or both accessibility metrics. When Axe evaluation is enabled, the fragment is rendered in a headless Chromium browser, audited with Axe, and the resulting rule-level outputs are aggregated into node-level counts and an node-level Axe pass rate. When Schema evaluation is enabled, the fragment is parsed into a DOM structure and scored against the predefined widget-level checks.

All results are stored in a versioned run directory. Stored artifacts consist of generated HTML files, Schema scores, Axe evaluation results, token usage, and associated metadata. Results are exported as CSV files for downstream statistical analysis.

3.3 Interface and Configuration

PACE provides a web interface for configuring, executing, and inspecting runs. The interface is organized into three panels—*Settings*, *Run*, and *Results*—corresponding to configuration, execution, and inspection.

In *Settings*, users define models, system instructions, component specifications, and request variants (Figure 2). Accessibility scoring is configured here. For Axe, users can select the default configuration or alternative rule sets, such as the complete rule set and WCAG tag–filtered modes. For Schema, users select which checks apply to which widgets from our predefined set of checks and decision rules.

In *Run*, users configure and execute experiments through the interface (Figure 3). They select subsets of factors, set the number of repetitions, and choose whether to run Axe evaluation, Schema evaluation, or both. The interface also allows users to adjust generation sampling parameters (e.g., temperature, top-p, maximum tokens) and an optional delay between API calls. Before execution, the interface displays the total number of configurations that will be generated. During execution, it shows progress, token usage, and active runs while recording all settings for reproducibility.

In *Results*, completed runs are loaded and inspected using aggregated data from the CSV outputs. The interface provides separate views for node-level Axe pass rate and normalized schema score. Within each view, users can filter by model, system instruction, component specification, and request variant, allowing comparisons across the factorial design. The interface reports summary statistics for the selected metric and displays visualizations that show variability across factors. A preview table displays the filtered data at the row level.

The interface supports configuration and inspection of standard runs, and the underlying codebase is also configurable. Users can extend PACE by modifying API integrations, prompt templates, component definitions, and evaluation checks for further experimentation with other types of generated code or evaluation setups beyond the supported scope. We made the PACE codebase available[3].

4 Large-Scale Code Generation Study

We used PACE as an execution harness for a large-scale controlled study of LLM-generated HTML form components. We crossed five models, three system instructions (with stepwise increases in accessibility guidance), 58 form components, and three request variants (with progressively greater request detail). We collected 10 trials per configuration, yielding 2,610 configuration cells and 26,100 total HTML outputs, or 5,220 generations per model. Generation and evaluation settings were fixed for all models and runs, as shown in Table 2. We released the generated dataset from this study through Zenodo[4] to support replication and future comparison. In the following, we lay out the rationale and process for the selected models, system instructions, form components, and request variants.

4.1 Model Selection

We selected model providers using three inclusion criteria during the study window (January–March 2026): (1) they independently trained and deployed their own LLMs, (2) they offered a public

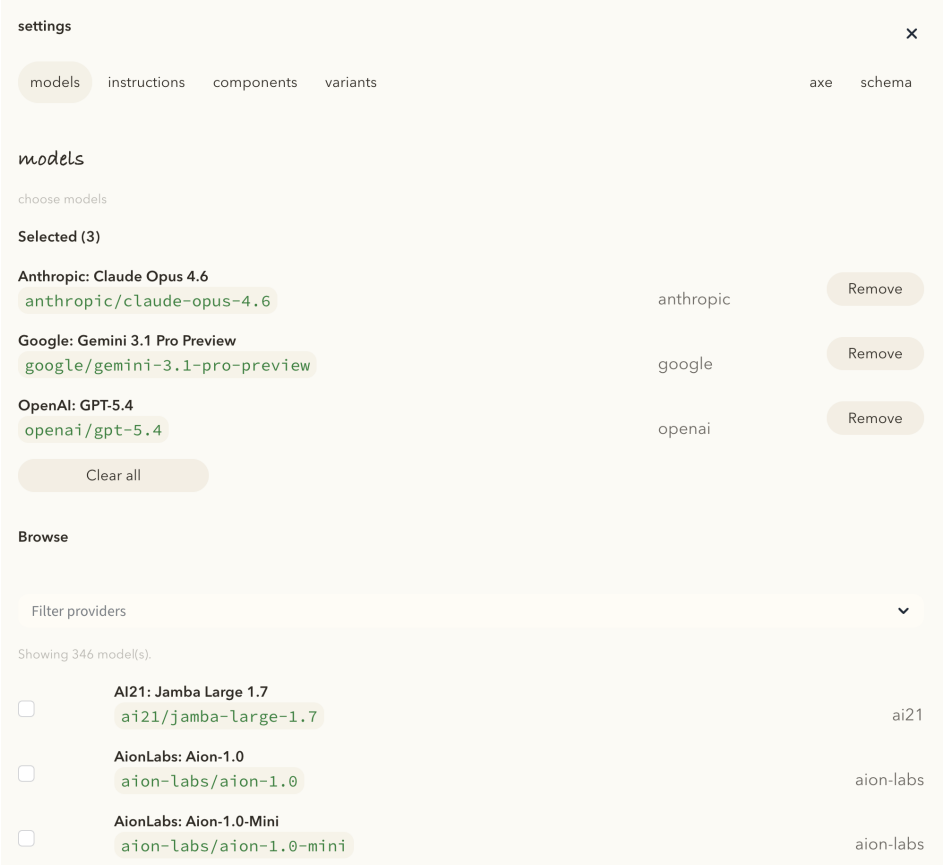


Figure 2: Settings panel in PACE. Users define models, system instructions, request variants, and component specifications, and configure accessibility scoring for Axe and Schema evaluation.

Table 2: Generation and evaluation settings used for all models and runs.

Setting	Value
Maximum tokens	600
Temperature	0.70
Top-p	0.95
Pause between requests	0.30 seconds
Repetitions per configuration	10
Axe	Enabled
Schema	Enabled

developer API, and (3) they provided at least one model that could reasonably be used for code generation.

After identifying eligible providers, we selected one model from each provider for evaluation. In each case, we chose the most recent publicly available model during data collection that could reasonably support code-generation tasks. Public model aggregation platforms like OpenRouter[6] were used only to examine which provider models were actively surfaced in developer-facing ecosystems, such as whether they were listed, accessible, and presented for practical use. We also consulted industry partners on our research team about which providers were currently relevant in practice. Final inclusion was then confirmed through provider documentation, which we used to verify each model's public availability and recency. Table 3 lists the models and their providers used in the study.

[6]https://openrouter.ai/

Table 3: Evaluated providers and models during the study window (January–March 2026).

Provider	Selected Model
Anthropic	Claude Opus 4.6
DeepSeek	DeepSeek V3.2
Google	Gemini 3.1
OpenAI	GPT 5.4
xAI	Grok 4

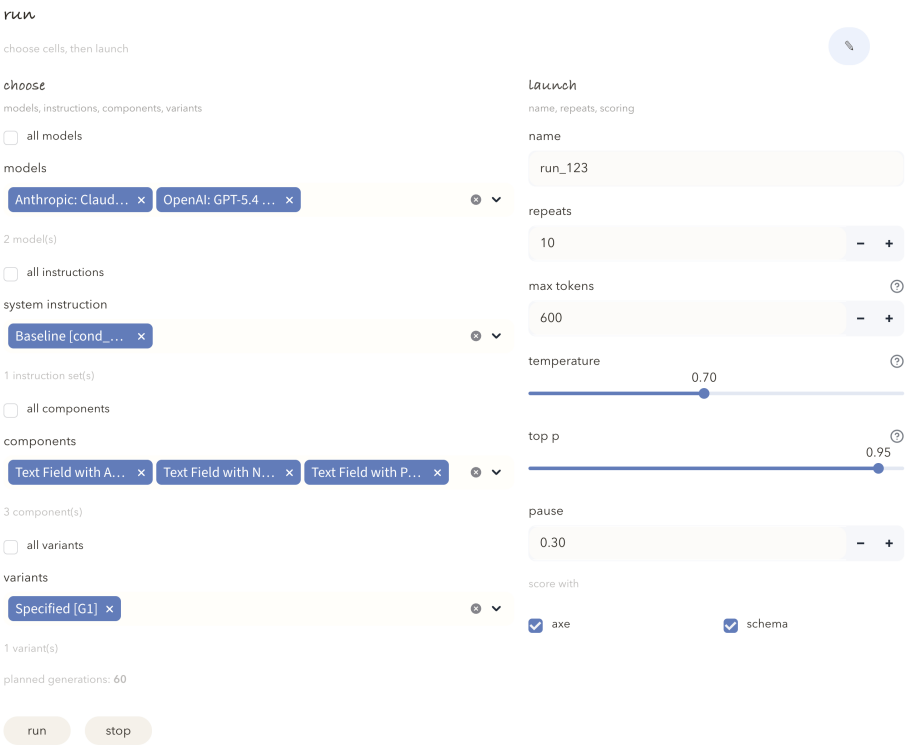


Figure 3: Run panel in PACE. Users configure factor subsets, repetition count, evaluation settings, and generation parameters, then monitor progress during execution.

All models were evaluated using their default code-generation settings without experimental features, additional reasoning, and other optional settings.

4.2 System Instructions

We took inspiration from WebAccessBench [18] and used three system instructions with stepwise increases in accessibility guidance. All three shared the same baseline constraint.

- **Unguided**: `Respond with HTML only. Do not include CSS, JavaScript, comments, or explanations.`
- **Little Guidance**: `Generate accessible HTML. Respond with HTML only. Do not include CSS, JavaScript, comments, or explanations.`
- **Expert Guidance**: `Generate accessible HTML for the requested HTML form component. Ensure that its purpose, any supporting text, and any states or feedback are conveyed through the HTML. Avoid relying on placeholder text alone to communicate meaning. Use HTML attributes so that relevant relationships and states can be interpreted by assistive technologies. Respond with HTML only. Do not include CSS, JavaScript, comments, or explanations.`

4.3 Components

We selected 58 form components as widgets that represent recurring interaction patterns commonly used across the web. We derived these form components through iterative sampling of high-traffic domains between August and December 2025. To identify popular platforms, we consulted Cloudflare Radar's publicly available domain ranking and popularity data [7] during that period, involving highly ranked domains based on 1.1.1.1 DNS resolver traffic. We then reviewed domains whose primary user flows depended on form interaction, such as signing in, creating accounts, completing purchases, making payments, booking services, searching, or submitting structured information. We excluded domains where forms appeared only incidentally or were peripheral to the main user flow.

From these sampled sites, we abstracted contexts based on the tasks that forms supported, such as authentication, e-commerce checkout, survey completion, and public-service submission. Table 4 summarizes these contexts and the role that form interaction played within them.

Within each context, we examined recurring workflows and documented repeated interaction patterns that involved multiple input types beyond free text. We continued sampling until additional sites did not yield substantively new form component patterns. The final set was then grouped into form component clusters based on similar interaction patterns, as shown in Table 5.

[7] https://radar.cloudflare.com/domains

Table 4: Contexts represented in the sampled sites during component derivation.

Context	Role of form interaction
E-commerce	Checkout, payment, shipping, and account management workflows
Authentication	Multi-step login, verification, password setup, and consent
Survey systems	Multi-question forms used to collect responses across predefined fields
Public services	Form submissions tied to applications, benefits, or licensing
Healthcare, finance	High-accuracy data entry involving sensitive information

As a final coverage check, we compared the resulting form component set against form-related controls and interaction patterns that surfaced repeatedly in the *State of HTML 2025* developer survey.[8] Specifically, we reviewed recurring form-relevant items across sections such as browser interoperability, limited functionality, commonly missed attributes, and styling issues, then checked whether those major control types and patterns were represented in our benchmark. This step served only to confirm coverage after form component derivation. Separately, we excluded patterns that fell outside the benchmark scope because they depended on task-specific logic, such as advanced phone formatting, CAPTCHA systems, and domain-specific validation workflows.

The complete list of form components, label text, and associated attributes is provided in Appendix C (Table C.1). The schema checks assigned to each form component are listed separately in Appendix B (Table D.1).

4.4 Request Variants

We used three request variants with progressively greater request detail. Each variant described the same form component, label, and attributes.

This design was informed by findings that people express tasks differently when prompting AI, with differences associated with expertise, roles, and familiarity with AI systems [38], usually relying on instructional conversation and *ad hoc* phrasing over systematic strategies [48].

- **Unspecified**: `{component} "{label}" {attribute}`
- **Specified**: `Create a {component} for "{label}" {attribute}.`
- **Detailed**: `Create a {component} for "{label}"{attribute} as it would appear within a real web form.`

These variants move from sparse requests to more specified task descriptions. The *Unspecified* variant reflects the kind of minimal prompt a developer might write quickly, with only the basic task and label. The *Specified* variant reflects a more typical instruction, where the requested form component is stated more directly but without added context. The *Detailed* variant adds further task information that more thoroughly describes the intended output. This progression let us examine whether accessibility outcomes changed as the request became more specific, apart from the system instruction.

[8] https://2025.stateofhtml.com/en-US/usage/

4.5 Design & Analysis

For code generation, `PACE` used a crossed factorial design with four manipulated factors: *model* (five levels), *system instruction* (three levels), *component specification* (58 levels), and *request variant* (three levels). Each unique factor combination was repeated 10 times, and each generated output was treated as one trial in the analysis.

We analyzed the two accessibility outcomes separately. For node-level Axe pass rate, we fit binomial generalized linear mixed-effects models (GLMMs). Each generated output was treated as one trial, with the dependent variable defined as the number of passing applicable Axe nodes versus the number of non-passing nodes, where non-passing combined violations and incompletes. For normalized schema score, we fit linear mixed-effects models (LMMs). This score ranged from 0 to 1 and reflected how completely a generated HTML form component satisfied the applicable Schema checks.

To address *RQ1-Effects*, we tested the main effects of *model*, *system instruction*, and *request variant* on each outcome. To address *RQ2-Interaction*, we tested the interaction between *system instruction* and *request variant* to examine whether the effect of request detail depended on the level of accessibility guidance in the system instruction. To address *RQ3-Component*, we fit follow-up models examining accessibility differences across both the individual form components and form component clusters.

When omnibus tests were significant, we conducted *post hoc* pairwise comparisons using Holm's sequential Bonferroni procedure [15] to determine which factor levels differed. All analyses were conducted in R using `lme4`, `car`, and `emmeans`, among other packages.

5 Results

We report the two accessibility outcomes, node-level Axe pass rate and normalized schema score, by research question. We begin with *RQ1-Effects*, which examines the main effects of *model*, *system instruction*, and *request variant*. We then report *RQ2-Interaction*, which examines the interaction between *system instruction* and *request variant*. Finally, we report *RQ3-Component*, which presents follow-up analyses of form component effects.

5.1 RQ1-Effects

5.1.1 Models. The choice of model made a large difference. Gemini 3.1 was significantly worse than all other models on both node-level Axe pass rate and normalized schema score. However, the ordering among the remaining models differed across the two measures. For node-level Axe pass rate, Gemini 3.1 scored lowest, followed by DeepSeek V3.2, while Claude Opus 4.6 and Grok 4 scored highest. For normalized schema score, Gemini 3.1 again scored lowest, followed by Claude Opus 4.6, while GPT 5.4 scored highest. Means ranged from $M = .78$ to $M = .99$ for node-level Axe pass rate and from $M = .75$ to $M = .98$ for normalized schema score. Invalid outputs were concentrated almost entirely in Gemini 3.1

Table 5: Form component clusters in the benchmark (58 total form components derived).

Form Component Cluster	Form Component
Text inputs	Text Field; Text Field (Required); Text Field with Placeholder; Text Field with Pattern; Text Field with Autocomplete Suggestion; Text Field with Accessible Description; Text Field with Non-visible Label; Text Field with Validation Error
Email inputs	Email Field; Email Field with Required; Email Field with Placeholder; Email Field with Autocomplete Suggestion; Email Field with Accessible Description; Email Field with Required, Autocomplete, and Accessible Description; Email Field with Validation Error
Specialized single-field inputs	Credit Card Field; Quantity Field; URL Field; Search Field; Color Picker; Phone Number Field
Password inputs	Password Field; Password Field (Required); Password Field with Autocomplete Suggestion; Password Field with Accessible Description; Password Field with Validation Error
Selection controls	Select Field; Select Field with Required; Select Field with Accessible Description; Radio Group; Radio Group with Accessible Description; Radio Group with Validation Error; Checkbox Group; Checkbox Group with Accessible Description; Checkbox Group with Validation Error
Confirmation and consent	Confirmation Checkbox; Confirmation Checkbox with Link
Multiline input	Textarea Field; Textarea Field with Required; Textarea Field with Accessible Description; Textarea Field with Validation Error; Textarea with Character Counter
File and media input	File Upload Field (Unrestricted); Multiple File Upload Field; Image File Upload Field; Document File Upload Field
Range and switch controls	Range Slider; Toggle Switch; Toggle Switch with Accessible Description; Toggle Switch Group with Multiple Types
Date and time inputs	Date Field; Time Field; Datetime-local Field
Submission and state	Submit Button; Submit Button with Icon; Reset Button; Disabled Field; Readonly Field

generations. Table 6 reports model means and standard deviations across both measures and output validity, Table 7 reports corrected pairwise comparisons, Table 8 reports axe violations, incomplete results, and the sensitivity analysis, and Figure 4 visualizes model differences.

Table 6: Model performance across node-level Axe pass rate ($\chi^2(4)$=3513.9, p<.001), normalized schema score ($F(4, 26038)$=4336.5, p<.001), and output validity (% of 5,220).

Model	Axe	Schema	Invalid
Claude Opus 4.6	**0.99 (0.11)** ^{a} †	0.85 (0.05)^{h}	**0 (0.00%)** †
Grok 4	**0.99 (0.11)** ^{b} †	0.97 (0.07)^{f}	1 (0.02%)
GPT 5.4	0.98 (0.15)^{b}	**0.98 (0.05)** ^{e} †	**0 (0.00%)** †
DeepSeek V3.2	0.94 (0.23)^{c}	0.97 (0.08)^{g}	**0 (0.00%)** †
Gemini 3.1	0.78 (0.41)^{d}	0.75 (0.23)^{i}	125 (2.39%)

Note. Values are mean (SD). Shared superscripts within a column indicate no significant difference based on corrected pairwise tests. Highlighted boldface marked with † indicates best observed performance within each metric.

5.1.2 System Instruction. *system instruction* influenced both accessibility measures, but the overall direction differed across them. At the aggregate level, both guided conditions outperformed Unguided on node-level Axe pass rate, whereas for normalized schema score, Expert Guidance was lowest and Unguided and Little Guidance did not differ. These aggregate effects, however, masked different model-specific responses.

Table 7: Pairwise comparisons between models.

Measure	Comparison	Result
Axe	Claude Opus > GPT^{a}	p = .0003
	Claude Opus > Grok^{b}	p = .0325
	DeepSeek < Claude / GPT / Grok^{c}	all p<.0001
	Gemini < all models^{d}	all p<.0001
Schema	GPT > Grok / DeepSeek / Claude / Gemini^{e}	all p<.0001
	Grok > DeepSeek / Claude / Gemini^{f}	p = .0129; others p<.0001
	DeepSeek > Claude / Gemini^{g}	all p<.0001
	Claude > Gemini^{h}	p<.0001
	Gemini < all models^{i}	all p<.0001

Note. Only statistically significant comparisons are shown. Superscripts correspond to superscripts in Table 6.

Model-specific responses. Table 9, Table 10, and Figure 5 show that the aggregate pattern concealed different responses across models. Claude Opus 4.6 changed very little in practical terms. DeepSeek V3.2 showed the clearest gains as guidance increased:

Table 8: Axe violation, incomplete, and sensitivity reporting by model.

Model	N	Any violation	Any incomplete	Both	Pass, V only	Pass, V+I	Δ
Aggregate	26,100	1,436 (5.50%)	385 (1.48%)	2 (0.01%)	24,664 (94.50%)	24,281 (93.03%)	1.47 pp
Claude Opus 4.6	5,220	31 (0.59%)	48 (0.92%)	1 (0.02%)	5,189 (99.41%)	5,142 (98.51%)	0.90 pp
DeepSeek V3.2	5,220	304 (5.82%)	21 (0.40%)	0 (0.00%)	4,916 (94.18%)	4,895 (93.77%)	0.40 pp
Gemini 3.1	5,220	884 (16.93%)	306 (5.86%)	1 (0.02%)	4,336 (83.07%)	4,031 (77.22%)	5.84 pp
GPT 5.4	5,220	133 (2.55%)	0 (0.00%)	0 (0.00%)	5,087 (97.45%)	5,087 (97.45%)	0.00 pp
Grok 4	5,220	84 (1.61%)	10 (0.19%)	0 (0.00%)	5,136 (98.39%)	5,126 (98.20%)	0.19 pp

Note. Values are count (%) unless otherwise noted. "Any violation" counts outputs with at least one axe violation. "Any incomplete" counts outputs with at least one axe incomplete result. "Both" counts outputs with at least one violation and at least one incomplete result. "Pass, V only" treats violations as failures and reports outputs with zero violations. "Pass, V+I" treats both violations and incomplete results as failures and reports outputs with zero violations and zero incomplete results. Δ is the percentage-point change between the two pass rates.

Table 9: Aggregate and model-specific effects of *system instruction* on node-level Axe pass rate and normalized schema score. Aggregate effects: node-level Axe pass rate ($\chi^2(2)$=111.32, p<.001); normalized schema score ($F(2, 26040)$=97.36, p<.001, η_p^2=.007).

Model	System instruction	Axe	Schema
Aggregate	Unguided	0.95 (0.22)^{a}	**0.91 (0.14)**^{d†}
	Little Guidance	**0.97 (0.18)**^{bc†}	**0.91 (0.14)**^{d†}
	Expert Guidance	**0.97 (0.17)**^{c†}	0.89 (0.16)^{e}
Claude Opus 4.6	Unguided	0.98 (0.14)^{f}	0.85 (0.06)^{h}
	Little Guidance	**0.99 (0.10)**^{g†}	**0.85 (0.04)**^{i†}
	Expert Guidance	**0.99 (0.07)**^{g†}	**0.85 (0.04)**^{hi†}
DeepSeek V3.2	Unguided	0.87 (0.33)^{j}	0.94 (0.10)^{l}
	Little Guidance	0.97 (0.16)^{k}	0.98 (0.06)^{m}
	Expert Guidance	**0.99 (0.10)**^{l†}	**0.99 (0.05)**^{n†}
Gemini 3.1	Unguided	**0.84 (0.36)**^{o†}	**0.84 (0.22)**^{r†}
	Little Guidance	0.77 (0.42)^{p}	0.76 (0.22)^{s}
	Expert Guidance	0.73 (0.44)^{q}	0.65 (0.20)^{t}
GPT 5.4	Unguided	0.95 (0.23)^{u}	0.96 (0.08)^{w}
	Little Guidance	**0.99 (0.11)**^{v†}	**1.00 (0.02)**^{x†}
	Expert Guidance	**0.99 (0.08)**^{v†}	**1.00 (0.02)**^{x†}
Grok 4	Unguided	0.97 (0.17)^{y}	**0.98 (0.06)**^{aa†}
	Little Guidance	**1.00 (0.06)**^{z†}	**0.98 (0.05)**^{aa†}
	Expert Guidance	**1.00 (0.04)**^{ab†}	0.95 (0.08)^{ab}

Note. Values are mean (SD). Shared superscripts within a model and metric indicate no significant difference based on corrected pairwise comparisons. Highlighted boldface marked with † indicates the best observed performance within each model and metric.

both node-level Axe pass rate and normalized schema score rose monotonically, and all pairwise differences were significant for both measures. Gemini 3.1 moved in the opposite direction, with both outcomes declining as guidance increased. GPT 5.4 showed a similar but more bounded pattern: both measures improved from Unguided to the guided conditions, but Little Guidance and Expert Guidance did not differ. Grok 4 was the least consistent case across the two measures: guidance improved node-level Axe pass rate, but Expert Guidance reduced normalized schema score relative to the other two conditions.

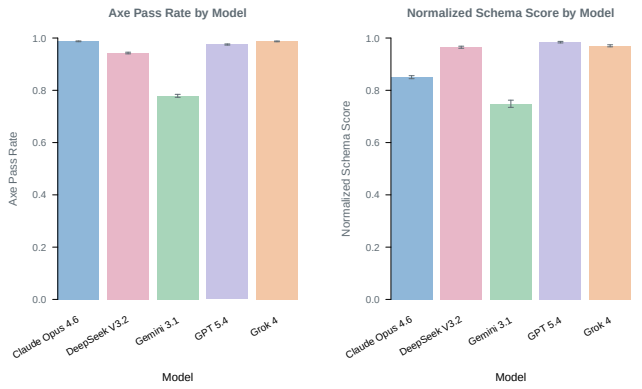
These results show that stronger guidance did not produce a uniform accessibility effect. It improved both outcomes for DeepSeek V3.2, improved GPT 5.4 up to a plateau, reduced both outcomes for Gemini 3.1, produced only minor shifts for Claude Opus 4.6, and moved in opposite directions across the two measures for Grok 4.

5.1.3 Request Variant. *Request variant* impacted both accessibility measures, but the overall direction again differed across them. At the aggregate level, Detailed produced the highest node-level Axe pass rate, whereas for normalized schema score, Detailed was lowest

Table 10: Corrected pairwise comparisons for *system instruction*.

Measure	Model	Comparison	Result
Axe	Aggregate	Unguided < Little / Expert[a]	both p<.0001
Axe	Claude Opus 4.6	Unguided < Little / Expert[b]	both p<.001
Axe	DeepSeek V3.2	Unguided < Little < Expert[c]	all p<.001
Axe	Gemini 3.1	Unguided > Little > Expert[d]	all p<.001
Axe	GPT 5.4	Unguided < Little / Expert[e]	both p<.001
Axe	Grok 4	Unguided < Little / Expert[f]	both p<.001
Axe	Grok 4	Little > Expert[g]	p=.032
Schema	Aggregate	Expert < Unguided / Little[h]	both p<.0001
Schema	Claude Opus 4.6	Unguided < Little[i]	p<.001
Schema	Claude Opus 4.6	Unguided < Expert[j]	p=.0115
Schema	DeepSeek V3.2	Unguided < Little < Expert[k]	all p<.001
Schema	Gemini 3.1	Unguided > Little > Expert[l]	all p<.001
Schema	GPT 5.4	Unguided < Little / Expert[m]	both p<.001
Schema	Grok 4	Expert < Unguided / Little[n]	both p<.001

Note. Only statistically significant comparisons are shown. Superscripts correspond to superscripts in Table 9.



(a) **(b)**

Figure 4: Mean accessibility scores for outputs generated by each model for (a) node-level Axe pass rate and (b) normalized schema score. Error bars show ±1 SE. Higher values indicate more accessible outputs.

and Unspecified and Specified did not differ. These aggregate effects again masked different model-specific responses.

Model-specific responses. Table 11, Table 12, and Figure 6 show that the aggregate pattern concealed different responses across models. Claude Opus 4.6 remained near ceiling on node-level Axe pass rate across all three variants, but normalized schema score was slightly lower for Detailed than for Unspecified and Specified. DeepSeek V3.2 showed the clearest gains from more detailed requests: Detailed increased both node-level Axe pass rate and normalized schema score, whereas Unspecified and Specified did not differ. Gemini 3.1 showed the opposite pattern, with Detailed reducing both node-level Axe pass rate and normalized schema score relative to the other two variants. GPT 5.4 showed a more favorable response to detail, with normalized schema score increasing monotonically from Unspecified to Specified to Detailed, while node-level Axe pass rate improved mainly from Unspecified to Detailed. Grok 4 showed the weakest response: Detailed slightly improved node-level Axe pass rate relative to Unspecified, but *request variant* did not significantly affect normalized schema score.

These results show that increasing request detail did not produce uniform accessibility benefit. Detailed requests generally improved node-level Axe pass rate, especially for DeepSeek V3.2 and GPT 5.4, but their effect on normalized schema score depended strongly on the model.

5.2 RQ2-Interaction

There was a *system instruction* and *request variant* interaction for both accessibility outcomes, although the interaction was more pronounced for normalized schema score.

Interaction pattern. Tables 13 and 14 show that the node-level Axe pass rate interaction was concentrated in the Unguided condition. Under Unguided, Detailed outperformed both Unspecified and Specified, but under Little Guidance and Expert Guidance, request variants did not differ. For normalized schema score, the interaction was stronger and qualitatively different. Under Unguided, Detailed was the strongest request variant. Under both Little Guidance and Expert Guidance, Detailed became the weakest. This reversal was clearest when the results were read across request variants: for Detailed, normalized schema score was highest under Unguided, lower under Little Guidance, and lowest under Expert Guidance, with all pairwise differences significant. Figure 7 visualizes these interaction patterns.

These results illustrate that the effect of *request variant* depended on the level of *system instruction*. For node-level Axe pass rate, stronger guidance largely reduced differences among request variants. For normalized schema score, stronger guidance reversed the direction of the request-variant effect itself.

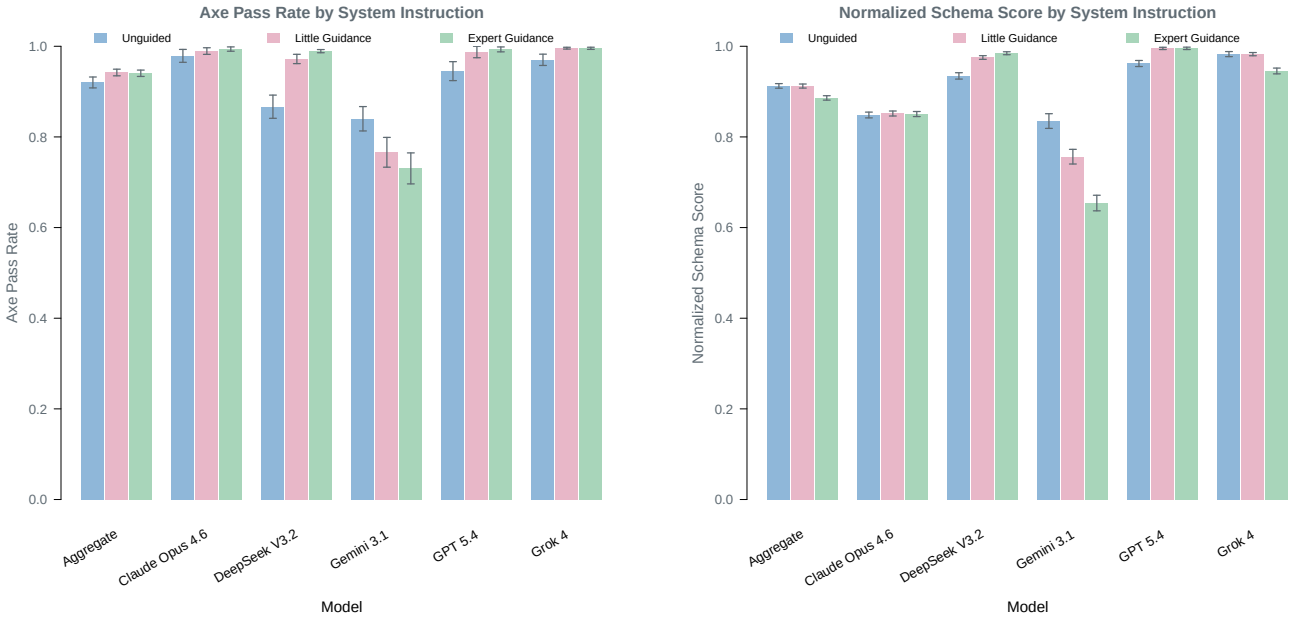


Figure 5: Aggregate and model-specific effects of *system instruction* on accessibility outcomes for (a) node-level Axe pass rate and (b) normalized schema score. Error bars show ±1 SE. Higher values indicate more accessible outputs.

5.3 RQ3-Component

Form components showed clear accessibility differences at both the individual form component level and the form component cluster level. Table 15 summarizes the five highest- and five lowest-scoring individual form components on each metric. A complete set of form component-level results for both measures are in Appendix E.

At the individual form component level, buttons and radio-group variants appeared near the top of the rankings, whereas toggle switches, non-visible labels, and several specialized text-entry variants appeared near the bottom. This separation was sharper for normalized schema score than for node-level Axe pass rate.

To illustrate how these individual-form component results appeared in generated markup, Figure 8 presents representative HTML excerpts for three informative cases in our data: Submit Button, Toggle Switch with Accessible Description, and Search Field.

Cluster-level patterns. Tables 16 and 17, together with Figure 9, show that the form component cluster-level pattern differed across the two measures. For node-level Axe pass rate, date and time inputs, specialized single-field inputs, and multiline input ranked highest overall, whereas selection controls, confirmation and consent, and password inputs ranked lowest. For normalized schema score, submission and state, selection controls, and date and time inputs ranked highest, whereas range and switch controls ranked lowest.

The form component cluster-level comparisons also clarify that accessibility differences were not random across the design space. For node-level Axe pass rate, several relatively straightforward single-field or native-control clusters outperformed password-related, confirmation, and grouped selection clusters. For normalized schema score, the clearest pattern was the weakness of range and switch controls, which scored lower than every other cluster.

6 Discussion

PACE began from the observation that current benchmarks of AI-generated code make it difficult to tell what produced an accessible outcome. When one generated UI scores higher in accessibility than another, the difference may be due to the model, the prompt, the component, or the benchmarking measure, since these factors are frequently entangled in the evaluation setup. Our goal in building PACE was to make those sources more interpretable. By keeping the form component fixed and allowing model choice and prompt construction to be configured under controlled conditions, PACE supports stronger interpretation of accessibility outcomes and clearer comparison across generation conditions.

Our results show why this separation is needed. Accessibility differed across *model*, *system instruction*, *request variant*, *component specification*, and *evaluation measure*. There was no general pattern across these directions, and prompting did not function as a dependable control layer. A prompt adjustment that appeared helpful under one measure could be neutral or unfavorable under another. An HTML form component that looked accessible under automated auditing could still fail to instantiate the richer semantics expected for that widget. Without a controlled setup, these patterns could

Table 11: Aggregate and model-specific effects of *request variant* on node-level Axe pass rate and normalized schema score. Aggregate effects: node-level Axe pass rate ($\chi^2(2)$=46.82, p<.001); normalized schema score ($F(2, 26040)$=19.87, p<.001, η_p^2=.002).

Model	Request variant	Axe	Schema
Aggregate	Unspecified	0.93 (0.26)^{a}	**0.91 (0.14)**^{d†}
	Specified	0.93 (0.25)^{a}	**0.91 (0.14)**^{d†}
	Detailed	**0.94 (0.23)**^{b†}	0.90 (0.16)^{e}
Claude Opus 4.6	Unspecified	**0.99 (0.11)**^{f†}	**0.85 (0.04)**^{h†}
	Specified	**0.99 (0.11)**^{f†}	**0.85 (0.04)**^{h†}
	Detailed	**0.99 (0.11)**^{f†}	0.85 (0.06)^{i}
DeepSeek V3.2	Unspecified	0.92 (0.26)^{j}	0.95 (0.09)^{l}
	Specified	0.93 (0.26)^{j}	0.95 (0.09)^{l}
	Detailed	**0.98 (0.14)**^{k†}	**0.99 (0.04)**^{m†}
Gemini 3.1	Unspecified	**0.78 (0.41)**^{n†}	0.78 (0.22)^{p}
	Specified	**0.78 (0.41)**^{n†}	**0.79 (0.23)**^{p†}
	Detailed	0.77 (0.41)^{o}	0.68 (0.21)^{q}
GPT 5.4	Unspecified	0.97 (0.18)^{r}	0.97 (0.07)^{t}
	Specified	**0.98 (0.16)**^{rs†}	0.98 (0.05)^{u}
	Detailed	**0.98 (0.12)**^{s†}	**1.00 (0.02)**^{v†}
Grok 4	Unspecified	0.98 (0.14)^{w}	**0.98 (0.06)**^{y†}
	Specified	**0.99 (0.11)**^{wy†}	**0.98 (0.05)**^{y†}
	Detailed	**0.99 (0.07)**^{x†}	**0.98 (0.05)**^{y†}

Note. Values are mean (SD). Shared superscripts within a model and metric indicate no significant difference based on corrected pairwise comparisons. Highlighted boldface marked with † indicates the best observed performance within each model and metric.

Table 12: Corrected pairwise comparisons for *request variant*

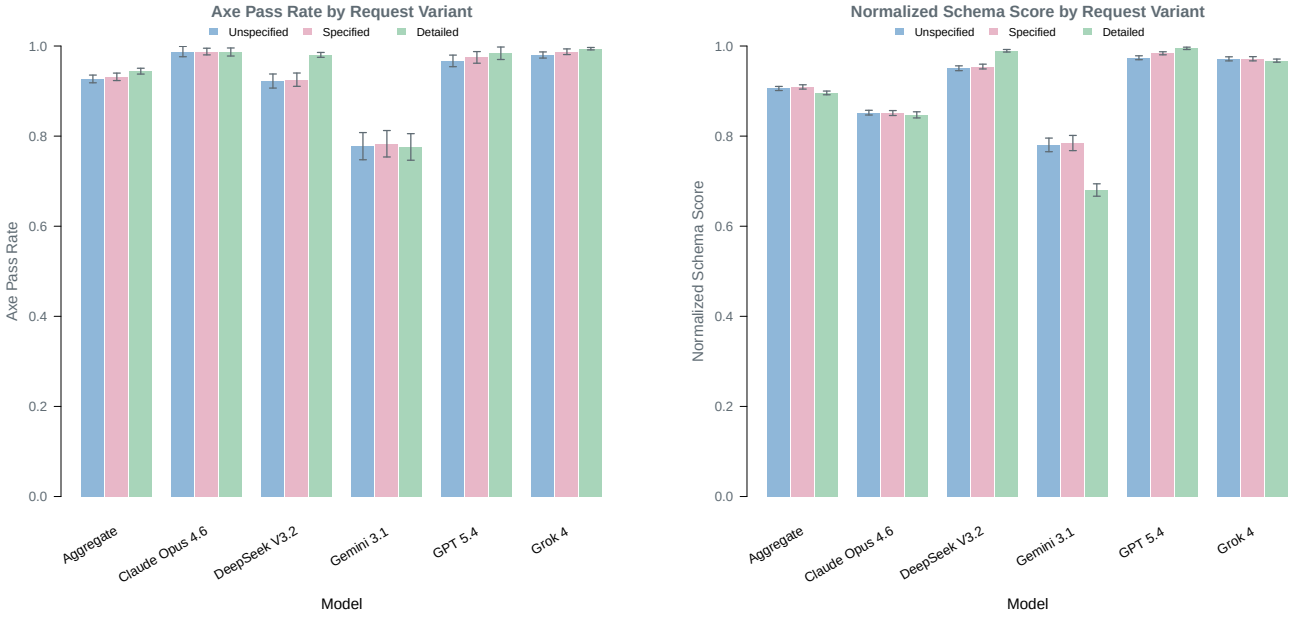
Measure	Model	Comparison	Result
Axe	Aggregate	Detailed > Unspecified / Specified^{a}	both p<.0001
Axe	DeepSeek V3.2	Detailed > Unspecified / Specified^{b}	both p<.001
Axe	Gemini 3.1	Detailed < Unspecified / Specified^{c}	both p<.001
Axe	GPT 5.4	Detailed > Unspecified^{d}	p<.001
Axe	Grok 4	Detailed > Unspecified^{e}	p=.0089
Schema	Aggregate	Detailed < Unspecified / Specified^{f}	both p<.0001
Schema	Claude Opus 4.6	Detailed < Unspecified / Specified^{g}	both p<.001
Schema	DeepSeek V3.2	Detailed > Unspecified / Specified^{h}	both p<.001
Schema	Gemini 3.1	Detailed < Unspecified / Specified^{i}	both p<.001
Schema	GPT 5.4	Unspecified < Specified < Detailed^{j}	all p<.001

Note. Only statistically significant comparisons are shown. Superscripts correspond to superscripts in Table 11.

easily be misread as evidence that one model is more accessible or that one prompting strategy produces more accessible code. PACE shows that such conclusions can collapse multiple sources of variation into one judgment. This separation is necessary if we want to improve AI-generated code meaningfully, since progress becomes difficult to interpret when evaluations do not show what drove success or failure.

6.1 Model Behavior Imposed the Strongest Constraint

Across our study, model choice remained the strongest and most persistent source of difference. Claude Opus 4.6 and Grok 4 produced the highest node-level Axe pass rate, GPT 5.4 produced the highest normalized schema score, and Gemini 3.1 performed worst on both measures. The two metrics did not rank models identically,



(a) **(b)**

Figure 6: Model effects of *request variant* on accessibility outcomes for (a) node-level Axe pass rate and (b) normalized schema score. Error bars show ±1 SE. Higher values indicate more accessible outputs.

Table 13: Interaction between *system instruction* and *request variant* for node-level Axe pass rate ($\chi^2(4)=27.07$, $p<.001$) and normalized schema score ($F(4, 26034)=13.11$, $p<.001$, $\eta_p^2=.002$).

System instruction	Request variant	Axe	Schema
Unguided	Unspecified	0.90 (0.30)^{a}	0.91 (0.12)^{g}
	Specified	0.91 (0.28)^{a}	**0.92 (0.13)**^{gh†}
	Detailed	**0.95 (0.22)**^{b†}	**0.92 (0.15)**^{h†}
Little Guidance	Unspecified	**0.94 (0.23)**^{c†}	**0.92 (0.13)**^{i†}
	Specified	**0.94 (0.23)**^{c†}	**0.92 (0.13)**^{i†}
	Detailed	**0.94 (0.23)**^{c†}	0.90 (0.16)^{j}
Expert Guidance	Unspecified	**0.94 (0.24)**^{d†}	**0.89 (0.16)**^{k†}
	Specified	**0.94 (0.24)**^{d†}	**0.89 (0.16)**^{k†}
	Detailed	**0.94 (0.23)**^{d†}	0.88 (0.17)^{l}

Note. Values are mean (SD). Shared superscripts within each *system instruction* block indicate no significant difference based on corrected pairwise comparisons for that metric. Highlighted boldface marked with † indicates the best observed performance within each *system instruction* block and metric.

which suggests that accessibility performance was not reducible to any one dimension. Even so, the larger pattern was that accessibility outcomes were strongly model-sensitive.

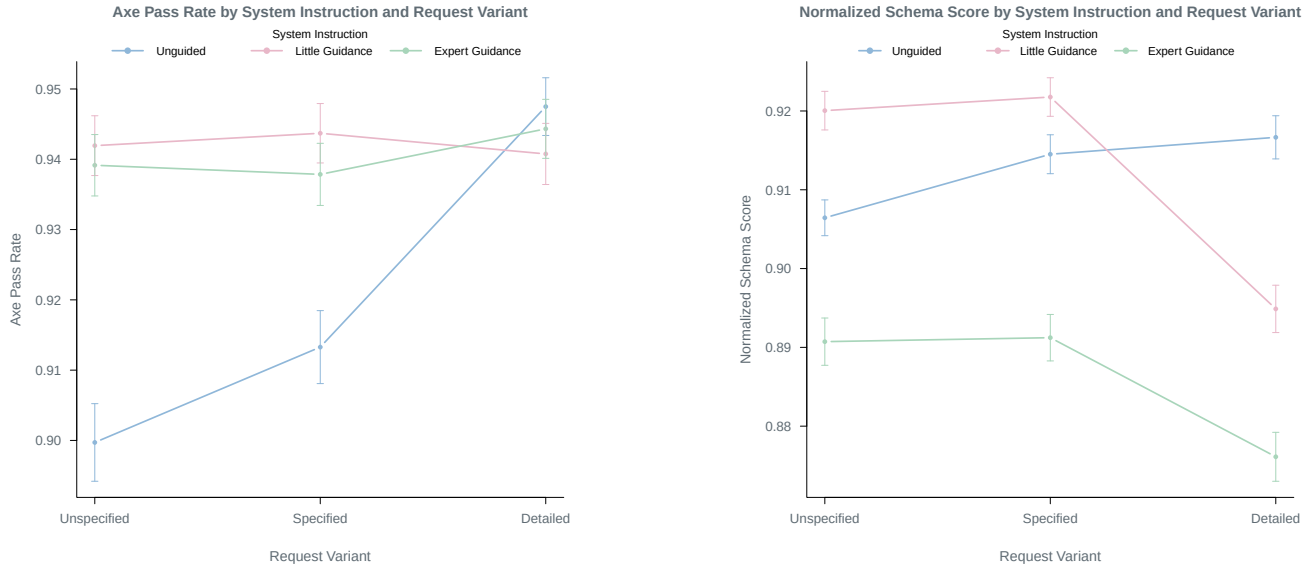
We cannot determine from these results why the models differed. The source may lie in training data, code specialization, instruction tuning, learned assumptions tied to HTML structure, or some combination of these factors. What our benchmark does show is that these differences were large enough to influence both interpretation and practical use.

This pattern places a limit on what prompt refinement can accomplish. In our study, some models improved under stronger accessibility guidance or greater request detail, some plateaued, and some declined. Prompt changes therefore operated within constraints set by the model. These constraints do not make prompt optimization unimportant, but prompt effects did not erase model

Table 14: Corrected pairwise comparisons for the interaction between *system instruction* and *request variant*. Only significant comparisons are shown.

Measure	Comparison set	Comparison	Result
Axe	Within Unguided	Detailed > Unspecified / Specified[a]	both $p<.0001$
Axe	Within Little Guidance	none	all $p\geq.5371$
Axe	Within Expert Guidance	none	all $p\geq.0941$
Schema	Within Unguided	Detailed > Unspecified[b]	$p=.0203$
Schema	Within Little Guidance	Detailed < Unspecified / Specified[c]	both $p<.0001$
Schema	Within Expert Guidance	Detailed < Unspecified / Specified[d]	both $p=.0002$
Schema	Within Unspecified	Little Guidance > Unguided > Expert Guidance[e]	all $p\leq.0003$
Schema	Within Specified	Little Guidance / Unguided > Expert Guidance[f]	both $p<.0001$
Schema	Within Detailed	Unguided > Little Guidance > Expert Guidance[g]	all $p<.0001$

Note. Non-significant comparisons are summarized only when useful for interpretation. Superscripts correspond to superscripts in Table 13.



(a) **(b)**

Figure 7: Interaction between *system instruction* and *request variant* for (a) node-level Axe pass rate and (b) normalized schema score. Error bars show ±1 SE. Higher values indicate more accessible outputs.

differences or bring models into the same accessibility profile under the same tasks and scoring procedures.

In practice, this means that teams using LLMs in front-end workflows should treat model selection as an accessibility decision, not just a productivity or cost decision. For research, it means benchmark conclusions should avoid implying that prompt optimization alone can account for accessible code. Prompt design can influence code generation, but model behavior still sets the range within which those effects unfold.

6.2 There Was No General Prompting Formula

Stronger accessibility guidance and greater request detail did influence outputs, but they did not yield a dependable accessibility recipe. Across the aggregate analyses, stronger accessibility guidance and more detailed requests sometimes improved node-level Axe pass rate, yet they did not reliably improve normalized schema score and in some cases reduced it. The interaction analyses also aligned. Under Unguided conditions, greater request detail could help. Under stronger accessibility guidance, that same increase in detail could weaken normalized schema score performance.

This finding challenges a common assumption in AI-assisted coding workflows, namely that accessibility can be improved in a

Table 15: Most- and least-accessible form components on node-level Axe pass rate ($\chi^2(57)=937.50$, $p<.001$) and normalized schema score ($\chi^2(57)=2151.58$, $p<.001$, $\eta_p^2=.08$).

Axe				Schema			
Rank	**Component**	**M**	**SD**	**Rank**	**Component**	**M**	**SD**
Highest 1	Submit Button	1.00	0.00	Highest 1	Submit Button	0.96	0.08
Highest 2	Reset Button	1.00	0.06	Highest 2	Reset Button	0.96	0.10
Highest 3	Text Field with Accessible Description	0.99	0.06	Highest 3	Radio Group	0.95	0.09
Highest 4	Textarea Field with Accessible Description	0.99	0.08	Highest 4	Submit Button with Icon	0.95	0.12
Highest 5	Search Field	0.99	0.08	Highest 5	Radio Group with Validation Error	0.95	0.08
Lowest 5	Confirmation Checkbox	0.90	0.30	Lowest 5	Text Field with Pattern	0.85	0.21
Lowest 4	Toggle Switch with Accessible Description	0.90	0.31	Lowest 4	Search Field	0.84	0.17
Lowest 3	Confirmation Checkbox with Link	0.89	0.31	Lowest 3	Text Field with Non-visible Label	0.83	0.25
Lowest 2	Text Field with Non-visible Label	0.77	0.42	Lowest 2	Toggle Switch with Accessible Description	0.83	0.16
Lowest 1	Password Field with Autocomplete Suggestion	0.74	0.44	Lowest 1	Toggle Switch	0.79	0.17

Table 16: Cluster-level accessibility outcomes for node-level Axe pass rate ($\chi^2(10)=372.11$, $p<.001$) and normalized schema score ($\chi^2(10)=1017.8$, $p<.001$).

Cluster	Axe	Schema
Text inputs	0.92 (0.27)^{a}	0.88 (0.16)^{j}
Email inputs	0.94 (0.24)^{ab}	0.91 (0.14)^{k}
Specialized single-field inputs	0.96 (0.18)^{bc}	0.88 (0.14)^{j}
Password inputs	0.90 (0.31)^{d}	0.86 (0.16)^{l}
Selection controls	0.89 (0.31)^{d}	**0.93 (0.12)**^{m†}
Confirmation and consent	0.89 (0.31)^{d}	0.89 (0.15)^{jk}
Multiline input	0.96 (0.18)^{bc}	0.88 (0.15)^{jl}
File and media input	0.94 (0.24)^{ab}	0.88 (0.14)^{jl}
Date and time inputs	**0.97 (0.18)**^{c†}	**0.93 (0.14)**^{m†}
Range and switch controls	0.93 (0.25)^{ab}	0.85 (0.16)^{n}
Submission and state	0.94 (0.24)^{ab}	**0.93 (0.13)**^{m†}

Note. Values are mean (SD). Shared superscripts within a column indicate no significant difference based on corrected pairwise comparisons. Highlighted boldface marked with † indicates the best observed performance within each metric.

Table 17: Corrected pairwise comparisons for form component clusters. Only significant comparisons are shown.

Measure	Cluster	Comparison	Result
Axe	Text inputs	< Email / Specialized / Multiline / Date–Time / Submission–State^{a}	$p=.0189$, all others $p\leq.0336$
Axe	Text inputs	> Password / Selection / Confirmation^{b}	$p<.0001$, $p<.0001$, $p=.0004$
Axe	Specialized single-field inputs	> Password / Selection / Confirmation^{c}	all $p<.0001$
Axe	Multiline input	> Password / Selection / Confirmation^{d}	all $p<.0001$
Axe	Date and time inputs	> Password / Selection / Confirmation^{e}	all $p<.0001$
Schema	Range and switch controls	< all other clusters^{f}	all $p<.0001$
Schema	Submission and state	> Text / Specialized / Password / Multiline / File–Media^{g}	all $p<.0001$
Schema	Submission and state	> Confirmation and consent^{h}	$p=.0041$
Schema	Selection controls	> Text / Specialized / Password / Multiline / File–Media / Confirmation^{i}	all $p\leq.0001$
Schema	Email inputs	> Text / Specialized / Password / Multiline^{j}	all $p\leq.0002$

Note. Non-significant comparisons are omitted for clarity. Several mid-ranked clusters did not differ reliably from one another on node-level Axe pass rate, including email inputs, file and media input, range and switch controls, and submission and state. For normalized schema score, text inputs, specialized single-field inputs, multiline input, and file and media input did not differ significantly from one another.

dependable way by making prompts more careful, more descriptive, or more accessibility-focused [13, 25]. Our results do not support that interpretation. Prompting behaved as a circumstantial intervention whose effect changed across models, shifted across measures, and depended on the *component specification* being generated.

The consequence is that prompt recommendations can appear portable because they are easy to describe and adopt, but apparent success in one setting does not guarantee the same effect in another. In accessibility work, that portability is risky because a prompt may improve automatically auditable checks while leaving deeper structural expectations unmet. Prompting therefore should

Figure 8: Illustrative excerpts from generated HTML outputs. Snippets are drawn from actual outputs in our dataset. The examples were selected to reflect three result patterns from Table 15: Submit Button as a consistently high-scoring component, Toggle Switch with Accessible Description as a consistently low-scoring component, and Search Field as a mismatch case that scored strongly on node-level Axe pass rate but weakly on normalized schema score.

A. Submit Button

Representative high-scoring output

```
<button type="submit">Submit</button>
```

Model: GPT 5.4
System instruction: Unguided
Request variant: Unspecified
node-level Axe pass rate: 1.00
normalized schema score: 1.00

This output uses native button markup with an action label. Submit Button scored highly on both measures in our dataset.

B. Toggle Switch with Accessible Description

Lower-accessibility output

```
<label for="notifications-toggle">Notifications</label>
<p>Turn alerts on or off.</p>
<input type="checkbox" id="notifications-toggle"
  name="notifications" checked>
```

Model: Claude Opus 4.6
System instruction: Unguided
Request variant: Detailed
node-level Axe pass rate: 0.00
normalized schema score: 0.42

This output has a visible label and descriptive text, but it does not programmatically associate the description with the checkbox.

Higher-accessibility output

```
<label for="notifications-toggle">Notifications</label>
<input type="checkbox" id="notifications-toggle"
  name="notifications-toggle"
  aria-describedby="notifications-note">
<p id="notifications-note">Turn alerts on or off.</p>
```

Model: GPT 5.4
System instruction: Unguided
Request variant: Specified
node-level Axe pass rate: 1.00
normalized schema score: 1.00

This output has a visible label and programmatically associates the description with the checkbox through `aria-describedby`.

C. Search Field

Lower-accessibility output

```
<input type="search" id="search" name="search"
  placeholder="Search"
  aria-label="Search">
```

Model: Gemini 3.1
System instruction: Unguided
Request variant: Specified
node-level Axe pass rate: 1.00
normalized schema score: 0.43

This output passed node-level Axe checks, but it consisted only of a bare search input with an `aria-label`. It lacked the visible label and surrounding search structure captured by the schema rules, which lowered its normalized schema score.

Higher-accessibility output

```
<form          action="##"          method="get">            <label
for="search">Search</label>
  <input type="search" id="search" name="search" />
</form>
```

Model: GPT 5.4
System instruction: Unguided
Request variant: Detailed
node-level Axe pass rate: 1.00
normalized schema score: 1.00

This output provides a label and a more complete search structure, which helped it score well on both measures.

be evaluated as an experimental condition that can shift accessibility outcomes in different directions, not treated as a reliable accessibility intervention on its own.

6.3 Accessibility Depends on What Is Measured

A central finding was that the two accessibility measures did not always move together. Node-level Axe pass rate captured whether generated nodes passed applicable automated checks, whereas normalized schema score captured whether an HTML form component realized the labels, descriptions, states, and relationships expected for that widget. These measures were related, but they captured different accessibility properties, and that difference was consequential for interpretation.

This distinction helps explain why prompt configuration and model effects did not align across outcomes. A model could appear strong under one measure and less strong under another because the two metrics rewarded different aspects of accessible code generation. This divergence indicates that accessibility conclusions depend in part on what an evaluation is designed to detect.

This observation also clarifies why schema scoring was necessary in PACE. Axe can detect many important automated accessibility failures, but it cannot fully assess whether a generated HTML form

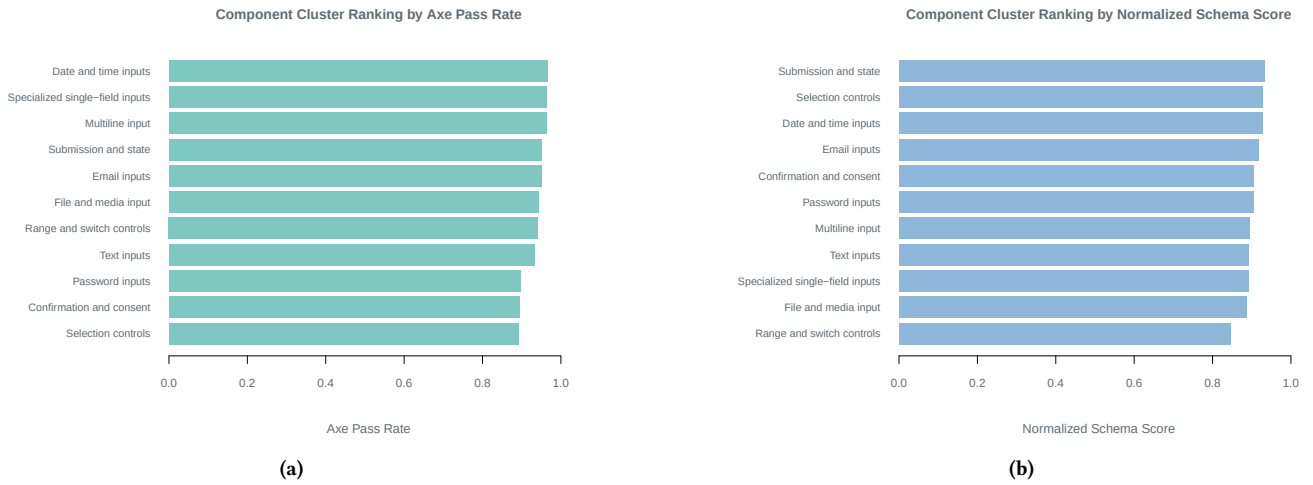


Figure 9: Form component clusters ranked by mean accessibility outcome for (a) node-level Axe pass rate and (b) normalized schema score.

component instantiates the richer accessible structure expected for that specific widget. Normalized schema score was designed to capture that larger layer by evaluating whether the output realized the semantic relationships and structural expectations appropriate for the widget.

Evaluation claims therefore need to remain tied to the measure that produced them. Passing more automated checks is not equivalent to generating semantically complete accessible HTML form components. PACE contributes by making that distinction visible in a controlled benchmark setting, which end-to-end evaluations can easily blur.

Even so, these two measures do not provide a complete account of accessibility in use. Some issues resist automated auditing, while others depend on downstream interaction, assistive technology behavior, and human interpretation. We do not claim that PACE captures accessibility in full, but accessibility evaluation becomes more informative when it incorporates complementary views, since one measure can present a misleadingly narrow picture.

6.4 Some Components Were Harder to Generate Accessibly

Component specification effects were also systematic. Simpler controls such as submit buttons and radio-group variants appeared more robust, whereas toggle switches, non-visible labels, and several specialized text-entry variants were more difficult to generate accessibly. At the form component cluster level, range and switch controls were especially weak on normalized schema score. These results suggest that AI-generated accessibility was not evenly distributed across interface patterns. Some widgets placed greater demands on a model's ability to produce precise semantic structure, state representation, and node relationships.

This distinction appeared at both the individual-*component specification* and form component cluster levels. Search Field provides an example at the individual-*component specification* level: it ranked among the top five *component specification*s on node-level Axe pass rate but among the bottom five on normalized schema score. At the form component cluster level, selection controls showed a similar split, ranking among the weakest groups on node-level Axe pass rate but among the strongest on normalized schema score. These mismatches may arise because some widgets or form component clusters are simple enough to avoid obvious automated violations, yet still remain semantically underspecified in ways that automated auditing does not penalize strongly. In our benchmark, the schema expected more specific accessible structures and relationships for these widgets, meaning that models could generate markup that looked acceptable under automated auditing while still failing to instantiate richer semantic patterns that PACE treated as appropriate for a given widget or form component cluster.

This unevenness is consequential for evaluation and practice. Researchers should avoid drawing conclusions from a narrow set of easier widgets that may inflate performance. Practitioners should not assume that all generated UI widgets require the same level of review. Some form component clusters appeared substantially more fragile than others, even when the prompt requested accessible HTML.

6.5 PACE as a Benchmarking Contribution

These findings clarify the value of PACE as n accessibility focused benchmarking testbed. It compares models under controlled prompts, but more importantly, it makes accessibility outcomes easier to interpret by separating sources of variation that benchmark setups can otherwise collapse together. Without this decomposition, it would have been easy to conclude that more accessibility guidance or more request detail improves accessibility in a general sense. Our results support a more careful interpretation. Prompt factors interacted in ways that could weaken, reverse, or concentrate the apparent effects of detail or guidance depending on the model and metric.

PACE therefore contributes both a benchmark and a way of studying AI-generated code under controlled prompt constructions. This contribution has practical implications for workflows in which LLMs are used to generate reusable UI code. In design-system and front-end settings, accessibility failures in shared HTML form components can affect many interfaces if they are not identified early. PACE offers one way to compare models, prompting strategies, and form component clusters before those outputs are reused widely. It can also support regression testing across model updates or prompt changes. Research on large-scale web accessibility evaluation has shown the value of scalable measurement for tracking page-level accessibility patterns over time while reducing the cost of repeated evaluation [27]. Because our benchmark fixes *component specifications* while varying generation conditions, it now becomes possible to examine when component-level accessibility improves, plateaus, or declines as these systems and workflows change over time.

6.6 Limitations and Future Work

Like any study, ours has several limitations. First, PACE evaluates isolated HTML form components, not full pages or interactive systems. That choice was necessary to achieve experimental control, but it also limits the scope of the findings. Component-level accessibility does not establish page-level accessibility, and component-level performance should not be assumed to predict accessibility in larger composed interfaces. Models may behave differently in tightly controlled component-generation tasks than in full webpage generation, in which additional context, layout structure, and cross-node dependencies come into play. Future work should examine how component-level findings relate to accessibility in more complete interface settings.

Second, our study did not compare LLM outputs against matched human-authored HTML. Recent work comparing LLM-generated web code with human-written code addresses this related question [40]. However, our current study prioritizes controlled comparisons across a large generation set, which made collecting matched human-authored HTML at the same scale difficult. Future work can add matched human-authored baselines for the same form component specifications.

Third, although node-level Axe pass rate and normalized schema score captured complementary accessibility properties, they do not provide a complete account of accessibility in use. Axe may miss structural weaknesses that Schema detects, while Schema is itself a designed scoring framework and cannot represent all accessibility experiences in practice (e.g., they do not replace empirical evaluation with assistive technologies or assistive technology users.). Prior work has likewise noted that automated accessibility testing is only one part of accessibility evaluation [41]. Future work should continue developing evaluation strategies that connect automated auditing, structural assessment, and human-centered evaluation.

Fourth, our study examined initial outputs under one particular decomposition of prompting, namely a fixed *system instruction* paired with a task request. This design let us isolate two important prompt dimensions, but it does not cover the full "sPACE" of model instruction. Other forms of prompting, such as persona or role framing, demonstrations, attached reference materials, example-driven guidance, or structured external documents describing accessibility best practices, such as Github Copilot's Accessibility Auditor,[9] may shape generated accessibility in different ways. Current coding practice also includes more agentic workflows in which users iterate with models inside a harness, invoke tools, inspect intermediate outputs, and revise requests across multiple steps [20, 42]. These workflows may change accessibility outcomes substantially, including through tool-supported checking or synthetic user testing. Our study also focused on relatively short prompts. In real development settings, prompt contexts can become much larger, including very large prompt files or instruction documents that span many pages. It remains unclear whether embedding much more extensive accessibility guidance would improve results, leave them unchanged, or introduce new downsides. Relatedly, future work could test prompt materials grounded in established accessibility acceptance criteria, such as Atomica11y[10] or MagentaA11y,[11] to examine whether accessibility outcomes differ when models are guided by richer and more operationalized accessibility requirements. Because PACE is configurable and extensible, future work can use the same framework to examine whether the model-specific and metric-specific patterns observed here persist under richer prompting configurations, long-context prompts, iterative agentic workflows, and instruction styles that better reflect deployment settings.

Finally, the present study focused on five development-centered models during a specific study window. Our goal was to evaluate a contemporary set that could plausibly support HTML generation in practice, rather than to claim that recency predicts code-generation quality. Newer models may not be more efficient or effective for code generation, and accessibility behavior and prompt sensitivity may shift across model versions. An important future direction is to automate and repeat benchmarking over time across updated models, richer prompting configurations, agentic workflows, and more complete interface settings. PACE provides a foundation for this kind of longitudinal and multi-level accessibility evaluation.

7 Conclusion

We addressed a persistent problem in evaluating accessibility in AI-generated code. In many end-to-end settings, accessibility differences are visible, but their source is not. Model behavior, prompt construction, form component, and evaluation choice can shift together, making it difficult to tell what produced a stronger or weaker result. PACE was designed to separate these influences through controlled evaluation of LLM-generated HTML form components.

Across 26,100 generated outputs from five development-centered LLMs, our results showed model differences were pronounced. Prompt effects changed across conditions and did not yield a dependable recipe. Form components varied in difficulty across the design sPACE. Node-level Axe pass rate and normalized schema score diverged in meaningful ways, showing that evaluation choices can support different conclusions about the same code.

Our findings show that accessibility in AI-generated code is not well captured by one prompt or one score. They also show that evaluation design informs what can be concluded about accessible code generation. PACE contributes as one way to study these

[9] https://docs.github.com/en/copilot/tutorials/customization-library/custom-instructions/accessibility-auditor
[10] https://www.atomica11y.com/
[11] https://www.magentaa11y.com/

dependencies under controlled conditions. As LLMs become more embedded in front-end workflows and reusable UI generation, accessibility evaluation will need to identify when accessibility holds, when it weakens, and which conditions produced that outcome.

Acknowledgments

The authors thank Ed Summers, Carie Fisher, Nandita Gupta, and Mary Bellard for conversations related to this work. This work was supported in part by Microsoft, Github, ServiceNow, the University of Washington Center for Research and Education on Accessible Technology and Experiences (CREATE), the National Institute on Disability, Independent Living, and Rehabilitation Research (grant number 90REGE0026), and a National Science Foundation Graduate Research Fellowship. Any opinions, findings, conclusions or recommendations expressed in our work are those of the authors and do not necessarily reflect those of any supporter.

References

[1] Mahdieh Abazar, Reyhaneh Farahmand, Gouri Ginde, Benjamin Tan, and Lorenzo De Carli. 2026. Behavioral Analysis of AI Code Generation Agents: Edit, Rewrite, and Repetition. (2026).

[2] Iyad Abu Doush and Reem Kassem. 2025. Can generative AI create accessible web code? A benchmark analysis of AI-generated HTML against accessibility standards. *Universal Access in the Information Society* 24, 4 (2025), 3483–3506.

[3] Wajdi Aljedaani, Abdulrahman Habib, Ahmed Aljohani, Marcelo Eler, and Yunhe Feng. 2024. Does chatgpt generate accessible code? investigating accessibility challenges in llm-generated source code. In *Proceedings of the 21st International Web for All Conference*. 165–176.

[4] Shraddha Barke, Michael B James, and Nadia Polikarpova. 2023. Grounded copilot: How programmers interact with code-generating models. *Proceedings of the ACM on Programming Languages* 7, OOPSLA1 (2023), 85–111.

[5] Sean Bechhofer, Simon Harper, and Darren Lunn. 2006. Sadie: Semantic annotation for accessibility. In *International Semantic Web Conference*. Springer, 101–115.

[6] Brett A Becker, Paul Denny, James Finnie-Ansley, Andrew Luxton-Reilly, James Prather, and Eddie Antonio Santos. 2023. Programming is hard-or at least it used to be: Educational opportunities and challenges of ai code generation. In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1*. 500–506.

[7] Joel Becker, Nate Rush, Elizabeth Barnes, and David Rein. 2025. Measuring the impact of early-2025 AI on experienced open-source developer productivity. *arXiv preprint arXiv:2507.09089* (2025).

[8] Carlos Rafael Catalan, Lheane Marie Dizon, Patricia Nicole Monderin, and Emily Kuang. 2026. I'm Not Reading All of That: Understanding Software Engineers' Level of Cognitive Engagement with Agentic Coding Assistants. *arXiv preprint arXiv:2603.14225* (2026).

[9] Anwoy Chatterjee, HSVNS Kowndinya Renduchintala, Sumit Bhatia, and Tanmoy Chakraborty. 2024. POSIX: A prompt sensitivity index for large language models. In *Findings of the Association for Computational Linguistics: EMNLP 2024*. 14550–14565.

[10] Parmit K. Chilana, Celena Alcock, Shruti Dembla, Anson Ho, Ada Hurst, Brett Armstrong, and Philip J. Guo. 2015. Perceptions of non-CS majors in intro programming: The rise of the conversational programmer. In *2015 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 251–259. doi:10.1109/VLHCC.2015.7357224

[11] Iyad Abu Doush and Reem Kassem. 2024. Evaluating ai-generated web code for accessibility compliance: A metric-driven approach. In *Proceedings of the 11th International Conference on Software Development and Technologies for Enhancing Accessibility and Fighting Info-exclusion*. 338–344.

[12] Javedul Ferdous, Sami Uddin, and Vikas Ashok. 2021. Semantic table-of-contents for efficient web screen reading. In *Proceedings of the 36th Annual ACM Symposium on Applied Computing*. 1941–1949.

[13] Alexandra-Elena Gurita. 2025. Understanding and Improving Accessibility in AI-Generated Interfaces through Interactive Prompt Engineering Methods. In *Companion Proceedings of the 30th International Conference on Intelligent User Interfaces (IUI '25 Companion)*. Association for Computing Machinery, New York, NY, USA, 101–104. doi:10.1145/3708557.3716347

[14] Alexandra-Elena Gurita and Radu-Daniel Vatavu. 2025. 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*. 124–134.

[15] Sture Holm. 1979. A simple sequentially rejective multiple test procedure. *Scandinavian journal of statistics* (1979), 65–70.

[16] Elizabeth Barnes David Rein Joel Becker, Nate Rush. 2025. Measuring the Impact of Early-2025 AI on Experienced Open-Source Developer Productivity. https://metr.org/blog/2025-07-10-early-2025-ai-experienced-os-dev-study/.

[17] Mansi Khemka and Brian Houck. 2024. Toward Effective AI Support for Developers: A survey of desires and concerns. *Commun. ACM* 67, 11 (2024), 42–49.

[18] Casey Kreer. 2026. WebAccessBench: Digital Accessibility Reliability in LLM-Generated Web Interfaces. https://conesible.de/wab/whitepaper_webaccessbench.pdf. Whitepaper.

[19] Arina Kudriavtseva, Nisar Ahmad Hotak, and Olga Gadyatskaya. 2025. My Code Is Less Secure with Gen AI: Surveying Developers' Perceptions of the Impact of Code Generation Tools on Security. In *Proceedings of the 40th ACM/SIGAPP Symposium on Applied Computing*. 1637–1646.

[20] Aayush Kumar, Yasharth Bajpai, Sumit Gulwani, Gustavo Soares, and Emerson Murphy-Hill. 2025. Why AI Agents Still Need You: Findings from Developer-Agent Collaborations in the Wild. *arXiv preprint arXiv:2506.12347* (2025).

[21] Jenny T Liang, Chenyang Yang, and Brad A Myers. 2024. A large-scale survey on the usability of ai programming assistants: Successes and challenges. In *Proceedings of the 46th IEEE/ACM international conference on software engineering*. 1–13.

[22] Chenxu Liu, Yingjie Fu, Wei Yang, Ying Zhang, and Tao Xie. 2026. WebCoder-Bench: Benchmarking Web Application Generation with Comprehensive and Interpretable Evaluation Metrics. *arXiv preprint arXiv:2601.02430* (2026).

[23] Peya Mowar, Yi-Hao Peng, Aaron Steinfeld, and Jeffrey P Bigham. 2024. Tab to autocomplete: The effects of ai coding assistants on web accessibility. In *Proceedings of the 26th International ACM SIGACCESS Conference on Computers and Accessibility*. 1–6.

[24] Peya Mowar, Yi-Hao Peng, Jason Wu, Aaron Steinfeld, and Jeffrey P Bigham. 2025. Codea11y: Making ai coding assistants useful for accessible web development. In *Proceedings of the 2025 CHI Conference on Human Factors in Computing Systems*. 1–15.

[25] Peya Mowar, Yi-Hao Peng, Jason Wu, Aaron Steinfeld, and Jeffrey P Bigham. 2025. CodeA11y: Making AI Coding Assistants Useful for Accessible Web Development. In *Proceedings of the 2025 CHI Conference on Human Factors in Computing Systems (CHI '25)*. Association for Computing Machinery, New York, NY, USA, Article 45, 15 pages. doi:10.1145/3706598.3713335

[26] Lilian Ngweta, Kiran Kate, Jason Tsay, and Yara Rizk. 2025. Towards llms robustness to changes in prompt format styles. In *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 4: Student Research Workshop)*. 529–537.

[27] Luís P. Carvalho, Tiago Guerreiro, Shaun Lawson, and Kyle Montague. 2023. Towards real-time and large-scale web accessbility. In *Proceedings of the 25th International ACM SIGACCESS Conference on Computers and Accessibility* (New York, NY, USA) *(ASSETS '23)*. Association for Computing Machinery, New York, NY, USA, Article 35, 9 pages. doi:10.1145/3597638.3608403

[28] PD Parthasarathy and Swaroop Joshi. 2024. Exploring the need of accessibility education in the Software Industry: insights from a Survey of Software Professionals in India. In *Proceedings of the 46th International Conference on Software Engineering: Software Engineering Education and Training*. 212–220.

[29] Laxmisha Rai, Smriti Khatiwada, Chunrao Deng, and Fasheng Liu. 2024. Cross-language code development with generative ai: A source-to-source translation perspective. In *2024 IEEE 7th International Conference on Electronic Information and Communication Technology (ICEICT)*. IEEE, 562–565.

[30] Amirhossein Razavi, Mina Soltangheis, Negar Arabzadeh, Sara Salamat, Morteza Zihayat, and Ebrahim Bagheri. 2025. Benchmarking prompt sensitivity in large language models. In *European Conference on Information Retrieval*. Springer, 303–313.

[31] Anne Spencer Ross, Xiaoyi Zhang, James Fogarty, and Jacob O Wobbrock. 2017. Epidemiology as a framework for large-scale mobile application accessibility assessment. In *Proceedings of the 19th international ACM SIGACCESS conference on computers and accessibility*. 2–11.

[32] Pradeep Kumar Saraswathi. 2024. Ai-enhanced web form development: Tackling accessibility barriers with generative technologies. In *2024 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*. IEEE, 1195–1198.

[33] Mirjam Seckler, Silvia Heinz, Javier A Bargas-Avila, Klaus Opwis, and Alexandre N Tuch. 2014. Designing usable web forms: empirical evaluation of web form improvement guidelines. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. 1275–1284.

[34] Raphael Serafini, Asli Yardim, and Alena Naiakshina. 2025. Exploring the impact of intervention methods on developers' security behavior in a manipulated chatgpt study. In *Proceedings of the 2025 CHI Conference on Human Factors in Computing Systems*. 1–26.

[35] Anna Sokol, Elizabeth Daly, Michael Hind, David Piorkowski, Xiangliang Zhang, Nuno Moniz, and Nitesh Chawla. 2024. BenchmarkCards: Standardized Documentation for Large Language Model Benchmarks. *arXiv preprint arXiv:2410.12974* (2024).

[36] Siddhant Sonkar. 2025. Boosting Developer Efficiency with AI: A Technical Review. *Journal Of Engineering And Computer Sciences* 4, 9 (2025), 500–508.
[37] Viktoria Stray, Nils Brede Moe, Nivethika Ganeshan, and Simon Kobbenes. 2025. Generative ai and developer workflows: How github copilot and chatgpt influence solo and pair programming. In *Proceedings of the 58th Hawaii International Conference on System Sciences*.
[38] Hari Subramonyam, Divy Thakkar, Andrew Ku, Juergen Dieber, and Anoop K Sinha. 2025. Prototyping with prompts: Emerging approaches and challenges in generative ai design for collaborative software teams. In *Proceedings of the 2025 CHI Conference on Human Factors in Computing Systems*. 1–22.
[39] Hyunjae Suh, Mahan Tafreshipour, Sam Malek, and Iftekhar Ahmed. 2025. Human or llm? a comparative study on accessible code generation capability. *arXiv preprint arXiv:2503.15885* (2025).
[40] Hyunjae Suh, Mahan Tafreshipour, Sam Malek, and Iftekhar Ahmed. 2025. Human or LLM? A Comparative Study on Accessible Code Generation Capability. arXiv:2503.15885 [cs.SE] https://arxiv.org/abs/2503.15885
[41] Amanda Swearngin, Jason Wu, Xiaoyi Zhang, Esteban Gomez, Jen Coughenour, Rachel Stukenborg, Bhavya Garg, Greg Hughes, Adriana Hilliard, Jeffrey P. Bigham, and Jeffrey Nichols. 2024. Towards Automated Accessibility Report Generation for Mobile Apps. *ACM Trans. Comput.-Hum. Interact.* 31, 4, Article 54 (Sept. 2024), 44 pages. doi:10.1145/3674967
[42] Ningzhi Tang, Chaoran Chen, Zihan Fang, Gelei Xu, Maria Dhakal, Yiyu Shi, Collin McMillan, Yu Huang, and Toby Jia-Jun Li. 2026. Programming by Chat: A Large-Scale Behavioral Analysis of 11,579 Real-World AI-Assisted IDE Sessions. *arXiv preprint arXiv:2604.00436* (2026).
[43] Masoumeh Taromirad and Per Runeson. 2025. Assertions in software testing: survey, landscape, and trends. *International Journal on Software Tools for Technology Transfer* 27, 1 (2025), 117–135.
[44] Michele Tufano, Dawn Drain, Alexey Svyatkovskiy, and Neel Sundaresan. 2022. Generating accurate assert statements for unit test cases using pretrained transformers. In *Proceedings of the 3rd ACM/IEEE International Conference on Automation of Software Test*. 54–64.
[45] Renata Vinadé and Sabrina Marczak. 2024. Educating for accessibility: insights into knowledge gaps and practical challenges in software engineering. In *Proceedings of the XXIII Brazilian Symposium on Software Quality*. 646–656.
[46] Tao Xiao, Youmei Fan, Fabio Calefato, Christoph Treude, Raula Gaikovina Kula, Hideaki Hata, and Sebastian Baltes. 2025. Self-admitted GenAI usage in open-source software. *arXiv preprint arXiv:2507.10422* (2025).
[47] Janghan Yoon, Jaegwan Cho, Junhyeok Kim, Jiwan Chung, Jaehyun Jeon, and Youngjae Yu. 2025. A11YN: aligning LLMs for accessible web UI code generation. *arXiv preprint arXiv:2510.13914* (2025).
[48] J Diego Zamfirescu-Pereira, Richmond Y Wong, Bjoern Hartmann, and Qian Yang. 2023. Why Johnny can't prompt: how non-AI experts try (and fail) to design LLM prompts. In *Proceedings of the 2023 CHI conference on human factors in computing systems*. 1–21.

A Axe-core Scoring

Table A.1: Subset of 26 Axe-core rules relevant to form components grouped by semantic category.

Category	Axe-core Rules
Accessible name / labeling	`label`, `label-title-only`, `form-field-multiple-labels`, `aria-input-field-name`, `button-name`, `input-button-name`, `select-name`, `textarea-name`, `input-image-alt`
ARIA correctness	`aria-valid-attr`, `aria-allowed-attr`, `aria-required-attr`, `aria-valid-attr-value`, `aria-hidden-focus`, `aria-required-children`, `aria-required-parent`, `aria-roles`, `aria-roles-required-attr`
Error state semantics	`aria-errormessage`, `aria-invalid`
ID / reference integrity	`duplicate-id`, `duplicate-id-aria`, `duplicate-id-active`
Form attribute validity	`autocomplete-valid`
Group semantics	`fieldset`
Toggle / switch	`aria-toggle-field-name`

B Schema Scoring

Table B.1: Complete schema check list with deterministic decision rules.

Check	Score = 2	Score = 1	Score = 0	WCAG
DOC_VALID_HTML	Contains ≥ 1 HTML element after parsing. No markdown fences (‘ ‘ ‘, ~~~). No <script> or <style> elements.	Parses to ≥ 1 element, but includes <script> or <style> when meta.forbid_script_style=false.	Any markdown fence appears, or parses to 0 elements, or meta.forbid_script_style=true and contains <script> or <style>.	N/A *Used to ensure valid parsing; hard-zeroing applies only when no parseable HTML is found.*
ID_UNIQUENESS	All id values are unique, or no id attributes exist.	A duplicated id exists, but none of the duplicated values appear in association references (labelfor, aria-labelledby, aria-describedby, aria-errormessage).	A duplicated id exists and at least one duplicated value appears in an association reference set.	N/A *Used to ensure valid parsing*
REF_TARGET_EXISTS	All labelfor, aria-labelledby, aria-describedby, and aria-errormessage references point to existing elements. If aria-labelledby or aria-errormessage is used, referenced text is non-empty after trimming.	Some references resolve while others do not, or referenced text resolves but is empty or whitespace-only.	All references point to missing elements.	1.3.1, 4.1.2
NAME_ROLE_VALUE_NATIVE	Primary control is native: input, select, textarea, or button. If meta.expected_primary_kind is set, it matches. If meta.expected_input_types is set, it matches. No role override on the primary native control.	Primary control is native, but essential attributes are missing or mismatched, or a role override appears, or type does not align with meta.expected_input_types.	No native primary control, or main control is non-native with ARIA role, or native control encodes the wrong semantics.	4.1.2
LABEL_ASSOCIATION	Programmatic label association exists via one mechanism: <labelfor> + matching id, label wrapping, or aria-labelledby to existing elements with non-empty text.	Label text exists but association is broken: missing for, missing id, missing targets, or aria-labelledby targets contribute no text.	No programmatic label association exists.	1.3.1, 3.3.2, 4.1.2
LABEL_VISIBLE_OR_EQUIV	Non-empty accessible name provided by <label> or aria-labelledby. Placeholder text is not the sole naming source.	Accessible name provided only via aria-label with no visible label element.	No <label> or aria-labelledby exists and the only naming source is a placeholder, or the accessible name resolves to empty text.	3.3.2, 1.3.1
DESCRIPTION_ASSOCIATION	If meta.description_ids is non-empty: a description element exists with non-empty text, the relevant target includes it in aria-describedby, and all referenced ids exist.	aria-describedby exists but is incomplete: missing ids, missing expected ids, or description text is empty or whitespace-only.	meta.description_ids is non-empty and aria-describedby is missing.	1.3.1, 3.3.2
REQUIRED_STATE	If meta.required_expected=true: required encoded via required or aria-required="true" with no conflict.	Required intent appears in text (for example, required or *), but encoding is missing, or conflicting encoding appears.	No required encoding and no required intent text when meta.required_expected=true.	3.3.2, 4.1.2
REQUIRED_CUE_EXPLAINED	If meta.required_expected=true: either no * appears, or * is hidden from AT via aria-hidden="true". If meta.required_symbol_explanation_required=true, explicit explanation text mentions required.	* appears but is not hidden from AT, or explanation is required but missing.	Reserved; unused.	3.3.2
AUTOCOMPLETE_PRESENT	If meta.expected_autocomplete_tokens is non-empty: autocomplete exists and is non-empty after trimming.	autocomplete exists but is empty or whitespace-only, or equals on/off.	Expected tokens provided but autocomplete is missing.	1.3.5

Table B.1: Complete schema check library with deterministic decision rules (continued).

Check	Score = 2	Score = 1	Score = 0	WCAG
AUTOCOMPLETE_VALID_TOKEN	If `meta.expected_autocomplete_tokens` is non-empty: `autocomplete` matches one expected token exactly.	`autocomplete` is empty or whitespace-only, or equals `on`/`off`.	`autocomplete` missing, or value not in expected tokens.	1.3.5
INPUT_TYPE_MATCHES_PURPOSE	If `meta.expected_input_types` is non-empty: primary control is `<input>` and `type` matches an expected type.	Primary control is `<input>` but uses `type="text"` when `text` is not expected.	Not `<input>`, missing `type`, or `type` is a clear mismatch and not a reasonable fallback.	4.1.2
CONSTRAINT_PATTERN_EXPLAINED	If `meta.pattern_expected=true`: `pattern` exists, `meta.pattern_guidance_ids` is non-empty, guidance text is non-empty, and `aria-describedby` references at least one guidance id, with all referenced ids existing.	`pattern` exists and guidance ids are provided, but `aria-describedby` is missing or incorrect, or referenced ids are missing, or guidance text is empty or whitespace-only.	Missing `pattern`, or guidance ids are missing or empty, or no guidance element with non-empty text exists.	3.3.2, 3.3.3
ERROR_PRESENT	If `meta.error_ids` is non-empty: at least one error element exists with non-empty text more specific than generic `Error`.	Error element exists but text is empty or whitespace-only, or overly generic (for example, `Error`).	No element with id in `meta.error_ids` exists.	3.3.1, 3.3.3
ERROR_ASSOCIATED	If `meta.error_ids` is non-empty: the relevant target includes at least one error id in `aria-describedby` or `aria-errormessage`, and all referenced ids exist.	Association attribute exists but is incomplete: missing referenced ids, or includes none of the ids in `meta.error_ids`.	No `aria-describedby` and no `aria-errormessage` on the relevant target when errors are expected.	1.3.1, 3.3.1
ERROR_INVALID_STATE	If `meta.error_ids` is non-empty: `aria-invalid="true"` appears on the relevant target.	`aria-invalid` present but not `true`, or `aria-invalid="true"` appears on the wrong element.	`aria-invalid` missing on the relevant target when errors are expected.	3.3.1, 4.1.2
STATUS_ANNOUNCE	If `meta.status_ids` is non-empty: at least one status element exists and has `role="status"`, `role="alert"`, or non-empty `aria-live`.	Status element exists but lacks live semantics (no `role="status"`, no `role="alert"`, and `aria-live` missing or empty).	No element with id in `meta.status_ids` exists.	4.1.3
GROUP_FIELDSET_LEGEND	If `meta.group_required=true`: `fieldset` exists with non-empty `legend` text.	`fieldset` exists but `legend` is missing or empty, or `role="group"` is used and the group has a non-empty accessible name.	No `fieldset` and no usable named `role="group"` fallback when grouping is required.	1.3.1, 2.4.6
RADIO_SAME_NAME	If `meta.radio_group_expected=true`: all radios share the same non-empty `name`.	At least two radios share a non-empty `name`, but at least one radio differs or is missing `name`.	No consistent shared non-empty `name`, or fewer than two radios exist.	4.1.2
EACH_OPTION_LABELED	If `meta.option_inputs_expected=true`: every radio or checkbox has a valid name source via label association, `aria-labelledby` with non-empty text, or non-empty `aria-label`.	At least one option is labeled, but at least one option lacks a valid association or has empty or broken label text.	No options have a valid programmatic label, or no option inputs exist.	1.3.1, 3.3.2
DISABLED_STATE	If `meta.disabled_expected=true`: primary control has `disabled`.	`aria-disabled="true"` used without `disabled`, or disabled intent appears elsewhere while the primary control lacks `disabled`.	No disabled encoding on the primary control and no other element indicates disabled.	4.1.2
READONLY_STATE	If `meta.readonly_expected=true`: primary control has `readonly`.	`aria-readonly="true"` used without `readonly`, or readonly intent appears elsewhere while the primary control lacks `readonly`.	No readonly encoding on the primary control and no other element indicates readonly.	4.1.2
FILE_INPUT_SEMANTICS	If `meta.file_input_expected=true`: `input[type=file]` exists. All attributes listed in `meta.file_required_attributes` are present. If `accept` is required, its value is non-empty and contains comma-separated tokens (for example, `image/png`, `image/*`, `.pdf`).	`input[type=file]` exists but one or more required attributes are missing or empty, or the `accept` attribute is present but empty.	`input[type=file]` element is missing when a file input is expected.	4.1.2

Table B.1: Complete schema check library with deterministic decision rules (continued).

Check	Score = 2	Score = 1	Score = 0	WCAG
BUTTON_TYPE_AND_NAME	If `meta.button_expected=true`: a native button exists (`<button>` or `input[type=submit|reset|button]`). Accessible name is non-empty from button text, `value`, or `aria-label`. Button type matches `meta.expected_button_type` if specified.	Button exists but type does not match the expected type, or the accessible name is provided only via `aria-label` with no visible text or `value`.	No native button exists, or the button has no accessible name source.	4.1.2, 2.4.6

C Component Specifications Used in Our Study

Table C.1: Component specifications with label text and attributes used in the evaluation.

ID	Component	Label Text	Attributes
C01	Text Field	What color is an orange	–
C02	Text Field (Required)	What color is an orange?	required
C03	Text Field with Placeholder	Who is your favorite dinosaur?	placeholder=Rex
C04	Text Field with Pattern	Enter two letters followed by 3 numbers	pattern
C05	Text Field with Autocomplete Suggestion	Your first name	autocomplete=given-name
C06	Text Field with Accessible Description	What is your favorite U.S. city?	description
C07	Text Field with Non-visible Label	Search for	hidden label
C08	Text Field with Validation Error	What color is an orange?	validation error
C09	Email Field	Desired email address	–
C10	Email Field with Required	Desired email address	required
C11	Email Field with Placeholder	Desired email address	placeholder=email
C12	Email Field with Autocomplete Suggestion	Email address	autocomplete=email
C13	Email Field with Accessible Description	Email Address	description
C14	Email Field with Required, Autocomplete, and Accessible Description	Email Address	required, autocomplete=email, description
C15	Email Field with Validation Error	Email Address	validation error
C16	Credit Card Field	Credit Card Number	–
C17	Quantity Field	Item Quantity	–
C18	Select Field	Theme	options
C19	Select Field with Required	Theme	required, options
C20	Select Field with Accessible Description	Theme	options, description
C21	Radio Group	Theme	options
C22	Radio Group with Accessible Description	Theme	options, description
C23	Radio Group with Validation Error	Theme	options, validation error
C24	Checkbox Group	What fruits do you like?	options
C25	Checkbox Group with Accessible Description	What fruits do you like?	options, description
C26	Checkbox Group with Validation Error	What fruits do you like?	options, validation error
C27	Confirmation Checkbox	I agree to the Terms and Conditions	–
C28	Confirmation Checkbox with Link	I agree to the Terms and Conditions	link
C29	Textarea Field	Message	–
C30	Textarea Field with Required	Message	required
C31	Textarea Field with Accessible Description	Message	description
C32	Textarea Field with Validation Error	Message	validation error
C33	Submit Button	Submit	–
C34	Submit Button with Icon	Submit	icon
C35	Password Field	New password	–
C36	Password Field (Required)	New password	required
C37	Password Field with Autocomplete Suggestion	new-password	autocomplete=new-password
C38	Password Field with Accessible Description	New password	description
C39	Password Field with Validation Error	New password	validation error
C40	Range Slider	Volume	range
C41	URL Field	Website URL	–
C42	Search Field	Search	–
C43	Color Picker	Pick a color	–
C44	Date Field	Birthdate	–
C45	Time Field	Meeting Time	–
C46	Datetime-local Field	Event Time	–
C47	File Upload Field (Unrestricted)	Upload a File	unrestricted
C48	Multiple File Upload Field	Upload Files	multiple, file types
C49	Image File Upload Field	Upload Photo	image only
C50	Document File Upload Field	Upload Document	document only
C51	Reset Button	Reset	–
C52	Disabled Field	Promo Code	disabled
C53	Readonly Field	User ID	readonly
C54	Toggle Switch	Notifications	–
C55	Toggle Switch with Accessible Description	Notifications	description
C56	Toggle Switch Group with Multiple Types	Notification Types	multiple toggles
C57	Textarea with Character Counter	Message	character counter
C58	Phone Number Field	Phone number	–

D Schema Checks Assigned to Each Element

Table D.1: Schema checks assigned to each component specification.

Component	Assigned Checks
Text Field	DOC_VALID_HTML, ID_UNIQUENESS, NAME_ROLE_VALUE_NATIVE, INPUT_TYPE_MATCHES_PURPOSE, LABEL_ASSOCIATION, LABEL_VISIBLE_OR_EQUIV, REF_TARGET_EXISTS
Text Field (Required)	DOC_VALID_HTML, ID_UNIQUENESS, NAME_ROLE_VALUE_NATIVE, INPUT_TYPE_MATCHES_PURPOSE, LABEL_ASSOCIATION, LABEL_VISIBLE_OR_EQUIV, REQUIRED_STATE, REQUIRED_CUE_EXPLAINED, REF_TARGET_EXISTS
Text Field with Placeholder	DOC_VALID_HTML, ID_UNIQUENESS, NAME_ROLE_VALUE_NATIVE, INPUT_TYPE_MATCHES_PURPOSE, LABEL_ASSOCIATION, LABEL_VISIBLE_OR_EQUIV, REF_TARGET_EXISTS
Text Field with Pattern	DOC_VALID_HTML, ID_UNIQUENESS, NAME_ROLE_VALUE_NATIVE, INPUT_TYPE_MATCHES_PURPOSE, LABEL_ASSOCIATION, LABEL_VISIBLE_OR_EQUIV, CONSTRAINT_PATTERN_EXPLAINED, DESCRIPTION_ASSOCIATION, REF_TARGET_EXISTS
Text Field with Autocomplete Suggestion	DOC_VALID_HTML, ID_UNIQUENESS, NAME_ROLE_VALUE_NATIVE, INPUT_TYPE_MATCHES_PURPOSE, LABEL_ASSOCIATION, LABEL_VISIBLE_OR_EQUIV, AUTOCOMPLETE_PRESENT, AUTOCOMPLETE_VALID_TOKEN, REF_TARGET_EXISTS
Text Field with Accessible Description	DOC_VALID_HTML, ID_UNIQUENESS, NAME_ROLE_VALUE_NATIVE, INPUT_TYPE_MATCHES_PURPOSE, LABEL_ASSOCIATION, LABEL_VISIBLE_OR_EQUIV, DESCRIPTION_ASSOCIATION, REF_TARGET_EXISTS
Text Field with Non-visible Label	DOC_VALID_HTML, ID_UNIQUENESS, NAME_ROLE_VALUE_NATIVE, INPUT_TYPE_MATCHES_PURPOSE, LABEL_ASSOCIATION, REF_TARGET_EXISTS
Text Field with Validation Error	DOC_VALID_HTML, ID_UNIQUENESS, NAME_ROLE_VALUE_NATIVE, INPUT_TYPE_MATCHES_PURPOSE, LABEL_ASSOCIATION, LABEL_VISIBLE_OR_EQUIV, ERROR_PRESENT, ERROR_ASSOCIATED, ERROR_INVALID_STATE, STATUS_ANNOUNCE, REF_TARGET_EXISTS
Email Field	DOC_VALID_HTML, ID_UNIQUENESS, NAME_ROLE_VALUE_NATIVE, INPUT_TYPE_MATCHES_PURPOSE, LABEL_ASSOCIATION, LABEL_VISIBLE_OR_EQUIV, REF_TARGET_EXISTS
Email Field with Required	DOC_VALID_HTML, ID_UNIQUENESS, NAME_ROLE_VALUE_NATIVE, INPUT_TYPE_MATCHES_PURPOSE, LABEL_ASSOCIATION, LABEL_VISIBLE_OR_EQUIV, REQUIRED_STATE, REQUIRED_CUE_EXPLAINED, REF_TARGET_EXISTS
Email Field with Placeholder	DOC_VALID_HTML, ID_UNIQUENESS, NAME_ROLE_VALUE_NATIVE, INPUT_TYPE_MATCHES_PURPOSE, LABEL_ASSOCIATION, LABEL_VISIBLE_OR_EQUIV, REF_TARGET_EXISTS
Email Field with Autocomplete Suggestion	DOC_VALID_HTML, ID_UNIQUENESS, NAME_ROLE_VALUE_NATIVE, INPUT_TYPE_MATCHES_PURPOSE, LABEL_ASSOCIATION, LABEL_VISIBLE_OR_EQUIV, AUTOCOMPLETE_PRESENT, AUTOCOMPLETE_VALID_TOKEN, REF_TARGET_EXISTS
Email Field with Accessible Description	DOC_VALID_HTML, ID_UNIQUENESS, NAME_ROLE_VALUE_NATIVE, INPUT_TYPE_MATCHES_PURPOSE, LABEL_ASSOCIATION, LABEL_VISIBLE_OR_EQUIV, DESCRIPTION_ASSOCIATION, REF_TARGET_EXISTS
Email Field with Required, Autocomplete, and Accessible Description	DOC_VALID_HTML, ID_UNIQUENESS, NAME_ROLE_VALUE_NATIVE, INPUT_TYPE_MATCHES_PURPOSE, LABEL_ASSOCIATION, LABEL_VISIBLE_OR_EQUIV, REQUIRED_STATE, REQUIRED_CUE_EXPLAINED, AUTOCOMPLETE_PRESENT, AUTOCOMPLETE_VALID_TOKEN, DESCRIPTION_ASSOCIATION, REF_TARGET_EXISTS
Email Field with Validation Error	DOC_VALID_HTML, ID_UNIQUENESS, NAME_ROLE_VALUE_NATIVE, INPUT_TYPE_MATCHES_PURPOSE, LABEL_ASSOCIATION, LABEL_VISIBLE_OR_EQUIV, ERROR_PRESENT, ERROR_ASSOCIATED, ERROR_INVALID_STATE, STATUS_ANNOUNCE, REF_TARGET_EXISTS
Credit Card Field	DOC_VALID_HTML, ID_UNIQUENESS, NAME_ROLE_VALUE_NATIVE, INPUT_TYPE_MATCHES_PURPOSE, LABEL_ASSOCIATION, LABEL_VISIBLE_OR_EQUIV, REF_TARGET_EXISTS
Quantity Field	DOC_VALID_HTML, ID_UNIQUENESS, NAME_ROLE_VALUE_NATIVE, INPUT_TYPE_MATCHES_PURPOSE, LABEL_ASSOCIATION, LABEL_VISIBLE_OR_EQUIV, REF_TARGET_EXISTS
Select Field	DOC_VALID_HTML, ID_UNIQUENESS, NAME_ROLE_VALUE_NATIVE, LABEL_ASSOCIATION, REF_TARGET_EXISTS
Select Field with Required	DOC_VALID_HTML, ID_UNIQUENESS, NAME_ROLE_VALUE_NATIVE, LABEL_ASSOCIATION, REQUIRED_STATE, REQUIRED_CUE_EXPLAINED, REF_TARGET_EXISTS
Select Field with Accessible Description	DOC_VALID_HTML, ID_UNIQUENESS, NAME_ROLE_VALUE_NATIVE, LABEL_ASSOCIATION, DESCRIPTION_ASSOCIATION, REF_TARGET_EXISTS
Radio Group	DOC_VALID_HTML, ID_UNIQUENESS, NAME_ROLE_VALUE_NATIVE, GROUP_FIELDSET_LEGEND, RADIO_SAME_NAME, EACH_OPTION_LABELED, REF_TARGET_EXISTS
Radio Group with Accessible Description	DOC_VALID_HTML, ID_UNIQUENESS, NAME_ROLE_VALUE_NATIVE, GROUP_FIELDSET_LEGEND, RADIO_SAME_NAME, EACH_OPTION_LABELED, DESCRIPTION_ASSOCIATION, REF_TARGET_EXISTS
Radio Group with Validation Error	DOC_VALID_HTML, ID_UNIQUENESS, NAME_ROLE_VALUE_NATIVE, GROUP_FIELDSET_LEGEND, RADIO_SAME_NAME, EACH_OPTION_LABELED, ERROR_PRESENT, ERROR_ASSOCIATED, ERROR_INVALID_STATE, STATUS_ANNOUNCE, REF_TARGET_EXISTS
Checkbox Group	DOC_VALID_HTML, ID_UNIQUENESS, NAME_ROLE_VALUE_NATIVE, GROUP_FIELDSET_LEGEND, EACH_OPTION_LABELED, REF_TARGET_EXISTS
Checkbox Group with Accessible Description	DOC_VALID_HTML, ID_UNIQUENESS, NAME_ROLE_VALUE_NATIVE, GROUP_FIELDSET_LEGEND, EACH_OPTION_LABELED, DESCRIPTION_ASSOCIATION, REF_TARGET_EXISTS
Checkbox Group with Validation Error	DOC_VALID_HTML, ID_UNIQUENESS, NAME_ROLE_VALUE_NATIVE, GROUP_FIELDSET_LEGEND, EACH_OPTION_LABELED, ERROR_PRESENT, ERROR_ASSOCIATED, ERROR_INVALID_STATE, STATUS_ANNOUNCE, REF_TARGET_EXISTS
Confirmation Checkbox	DOC_VALID_HTML, ID_UNIQUENESS, NAME_ROLE_VALUE_NATIVE, LABEL_ASSOCIATION, REF_TARGET_EXISTS
Confirmation Checkbox with Link	DOC_VALID_HTML, ID_UNIQUENESS, NAME_ROLE_VALUE_NATIVE, LABEL_ASSOCIATION, REF_TARGET_EXISTS

Table D.1: Schema checks assigned to each component specification (continued).

Component	Assigned Checks
Textarea Field	DOC_VALID_HTML, ID_UNIQUENESS, NAME_ROLE_VALUE_NATIVE, LABEL_ASSOCIATION, REF_TARGET_EXISTS
Textarea Field with Required	DOC_VALID_HTML, ID_UNIQUENESS, NAME_ROLE_VALUE_NATIVE, LABEL_ASSOCIATION, REQUIRED_STATE, REQUIRED_CUE_EXPLAINED, REF_TARGET_EXISTS
Textarea Field with Accessible Description	DOC_VALID_HTML, ID_UNIQUENESS, NAME_ROLE_VALUE_NATIVE, LABEL_ASSOCIATION, DESCRIPTION_ASSOCIATION, REF_TARGET_EXISTS
Textarea Field with Validation Error	DOC_VALID_HTML, ID_UNIQUENESS, NAME_ROLE_VALUE_NATIVE, LABEL_ASSOCIATION, ERROR_PRESENT, ERROR_ASSOCIATED, ERROR_INVALID_STATE, STATUS_ANNOUNCE, REF_TARGET_EXISTS
Submit Button	DOC_VALID_HTML, ID_UNIQUENESS, NAME_ROLE_VALUE_NATIVE, BUTTON_TYPE_AND_NAME, REF_TARGET_EXISTS
Submit Button with Icon	DOC_VALID_HTML, ID_UNIQUENESS, NAME_ROLE_VALUE_NATIVE, BUTTON_TYPE_AND_NAME, REF_TARGET_EXISTS
Password Field	DOC_VALID_HTML, ID_UNIQUENESS, NAME_ROLE_VALUE_NATIVE, INPUT_TYPE_MATCHES_PURPOSE, LABEL_ASSOCIATION, LABEL_VISIBLE_OR_EQUIV, REF_TARGET_EXISTS
Password Field (Required)	DOC_VALID_HTML, ID_UNIQUENESS, NAME_ROLE_VALUE_NATIVE, INPUT_TYPE_MATCHES_PURPOSE, LABEL_ASSOCIATION, LABEL_VISIBLE_OR_EQUIV, REQUIRED_STATE, REQUIRED_CUE_EXPLAINED, REF_TARGET_EXISTS
Password Field with Autocomplete Suggestion	DOC_VALID_HTML, ID_UNIQUENESS, NAME_ROLE_VALUE_NATIVE, INPUT_TYPE_MATCHES_PURPOSE, LABEL_ASSOCIATION, LABEL_VISIBLE_OR_EQUIV, AUTOCOMPLETE_PRESENT, AUTOCOMPLETE_VALID_TOKEN, REF_TARGET_EXISTS
Password Field with Accessible Description	DOC_VALID_HTML, ID_UNIQUENESS, NAME_ROLE_VALUE_NATIVE, INPUT_TYPE_MATCHES_PURPOSE, LABEL_ASSOCIATION, LABEL_VISIBLE_OR_EQUIV, DESCRIPTION_ASSOCIATION, REF_TARGET_EXISTS
Password Field with Validation Error	DOC_VALID_HTML, ID_UNIQUENESS, NAME_ROLE_VALUE_NATIVE, INPUT_TYPE_MATCHES_PURPOSE, LABEL_ASSOCIATION, LABEL_VISIBLE_OR_EQUIV, ERROR_PRESENT, ERROR_ASSOCIATED, ERROR_INVALID_STATE, STATUS_ANNOUNCE, REF_TARGET_EXISTS
Range Slider	DOC_VALID_HTML, ID_UNIQUENESS, NAME_ROLE_VALUE_NATIVE, INPUT_TYPE_MATCHES_PURPOSE, LABEL_ASSOCIATION, LABEL_VISIBLE_OR_EQUIV, REF_TARGET_EXISTS
URL Field	DOC_VALID_HTML, ID_UNIQUENESS, NAME_ROLE_VALUE_NATIVE, INPUT_TYPE_MATCHES_PURPOSE, LABEL_ASSOCIATION, LABEL_VISIBLE_OR_EQUIV, REF_TARGET_EXISTS
Search Field	DOC_VALID_HTML, ID_UNIQUENESS, NAME_ROLE_VALUE_NATIVE, INPUT_TYPE_MATCHES_PURPOSE, LABEL_ASSOCIATION, LABEL_VISIBLE_OR_EQUIV, REF_TARGET_EXISTS
Color Picker	DOC_VALID_HTML, ID_UNIQUENESS, NAME_ROLE_VALUE_NATIVE, INPUT_TYPE_MATCHES_PURPOSE, LABEL_ASSOCIATION, LABEL_VISIBLE_OR_EQUIV, REF_TARGET_EXISTS
Date Field	DOC_VALID_HTML, ID_UNIQUENESS, NAME_ROLE_VALUE_NATIVE, INPUT_TYPE_MATCHES_PURPOSE, LABEL_ASSOCIATION, LABEL_VISIBLE_OR_EQUIV, REF_TARGET_EXISTS
Time Field	DOC_VALID_HTML, ID_UNIQUENESS, NAME_ROLE_VALUE_NATIVE, INPUT_TYPE_MATCHES_PURPOSE, LABEL_ASSOCIATION, LABEL_VISIBLE_OR_EQUIV, REF_TARGET_EXISTS
Datetime-local Field	DOC_VALID_HTML, ID_UNIQUENESS, NAME_ROLE_VALUE_NATIVE, INPUT_TYPE_MATCHES_PURPOSE, LABEL_ASSOCIATION, LABEL_VISIBLE_OR_EQUIV, REF_TARGET_EXISTS
File Upload Field (Unrestricted)	DOC_VALID_HTML, ID_UNIQUENESS, NAME_ROLE_VALUE_NATIVE, LABEL_ASSOCIATION, FILE_INPUT_SEMANTICS, REF_TARGET_EXISTS
Multiple File Upload Field	DOC_VALID_HTML, ID_UNIQUENESS, NAME_ROLE_VALUE_NATIVE, LABEL_ASSOCIATION, FILE_INPUT_SEMANTICS, REF_TARGET_EXISTS
Image File Upload Field	DOC_VALID_HTML, ID_UNIQUENESS, NAME_ROLE_VALUE_NATIVE, LABEL_ASSOCIATION, FILE_INPUT_SEMANTICS, REF_TARGET_EXISTS
Document File Upload Field	DOC_VALID_HTML, ID_UNIQUENESS, NAME_ROLE_VALUE_NATIVE, LABEL_ASSOCIATION, FILE_INPUT_SEMANTICS, REF_TARGET_EXISTS
Reset Button	DOC_VALID_HTML, ID_UNIQUENESS, NAME_ROLE_VALUE_NATIVE, BUTTON_TYPE_AND_NAME, REF_TARGET_EXISTS
Disabled Field	DOC_VALID_HTML, ID_UNIQUENESS, NAME_ROLE_VALUE_NATIVE, INPUT_TYPE_MATCHES_PURPOSE, LABEL_ASSOCIATION, LABEL_VISIBLE_OR_EQUIV, DISABLED_STATE, REF_TARGET_EXISTS
Readonly Field	DOC_VALID_HTML, ID_UNIQUENESS, NAME_ROLE_VALUE_NATIVE, INPUT_TYPE_MATCHES_PURPOSE, LABEL_ASSOCIATION, LABEL_VISIBLE_OR_EQUIV, READONLY_STATE, REF_TARGET_EXISTS
Toggle Switch	DOC_VALID_HTML, ID_UNIQUENESS, NAME_ROLE_VALUE_NATIVE, LABEL_ASSOCIATION, REF_TARGET_EXISTS
Toggle Switch with Accessible Description	DOC_VALID_HTML, ID_UNIQUENESS, NAME_ROLE_VALUE_NATIVE, LABEL_ASSOCIATION, DESCRIPTION_ASSOCIATION, REF_TARGET_EXISTS
Toggle Switch Group with Multiple Types	DOC_VALID_HTML, ID_UNIQUENESS, NAME_ROLE_VALUE_NATIVE, GROUP_FIELDSET_LEGEND, EACH_OPTION_LABELED, REF_TARGET_EXISTS
Textarea with Character Counter	DOC_VALID_HTML, ID_UNIQUENESS, NAME_ROLE_VALUE_NATIVE, LABEL_ASSOCIATION, STATUS_ANNOUNCE, REF_TARGET_EXISTS
Phone Number Field	DOC_VALID_HTML, ID_UNIQUENESS, NAME_ROLE_VALUE_NATIVE, INPUT_TYPE_MATCHES_PURPOSE, LABEL_ASSOCIATION, LABEL_VISIBLE_OR_EQUIV, REF_TARGET_EXISTS

Table D.2: Worked example of schema-based scoring in PACE. This example uses the *Email Field with Required* component specification and shows one generated HTML output, the assigned schema checks, the per-check scores, and the resulting normalized schema score.

Example component	Example generated HTML output	Assigned schema checks and scores	Output normalized schema score
Email Field with Required *Label text:* Desired email address *Attributes:* required	`<label for="email">Desired email address *</label>` `<input id="email"` `type="email"` `required>`	DOC_VALID_HTML → 2 ID_UNIQUENESS → 2 NAME_ROLE_VALUE_NATIVE → 2 INPUT_TYPE_MATCHES_PURPOSE → 2 LABEL_ASSOCIATION → 2 LABEL_VISIBLE_OR_EQUIV → 2 REQUIRED_STATE → 2 REQUIRED_CUE_EXPLAINED → 1 REF_TARGET_EXISTS → 2 *Reason:* the output parses as valid HTML, uses a native email input, provides a visible associated label, and encodes the required state programmatically. The only partial score appears for REQUIRED_CUE_EXPLAINED because the asterisk is not hidden from assistive technology.	$\frac{2+2+2+2+2+2+2+1+2}{2\times 9} = \frac{17}{18} = 0.944$

E Supplementary Figures for RQ3

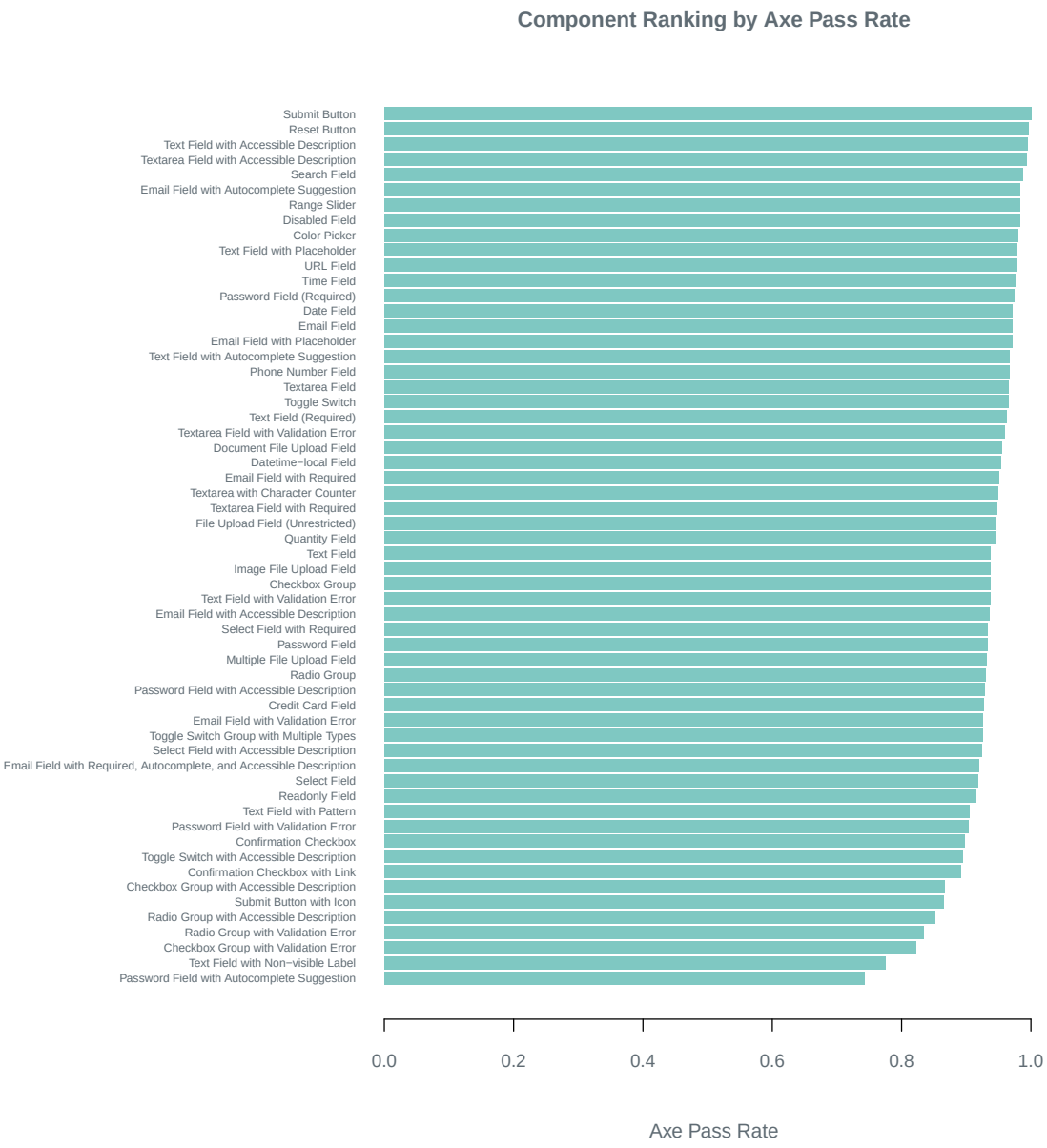


Figure E.1: Overall component ranking by axe pass rate. Higher values indicate more accessible outputs. Simpler text-entry and button components generally ranked near the top, whereas grouped controls with validation, non-visible labels, and some specialized input variants ranked lower.

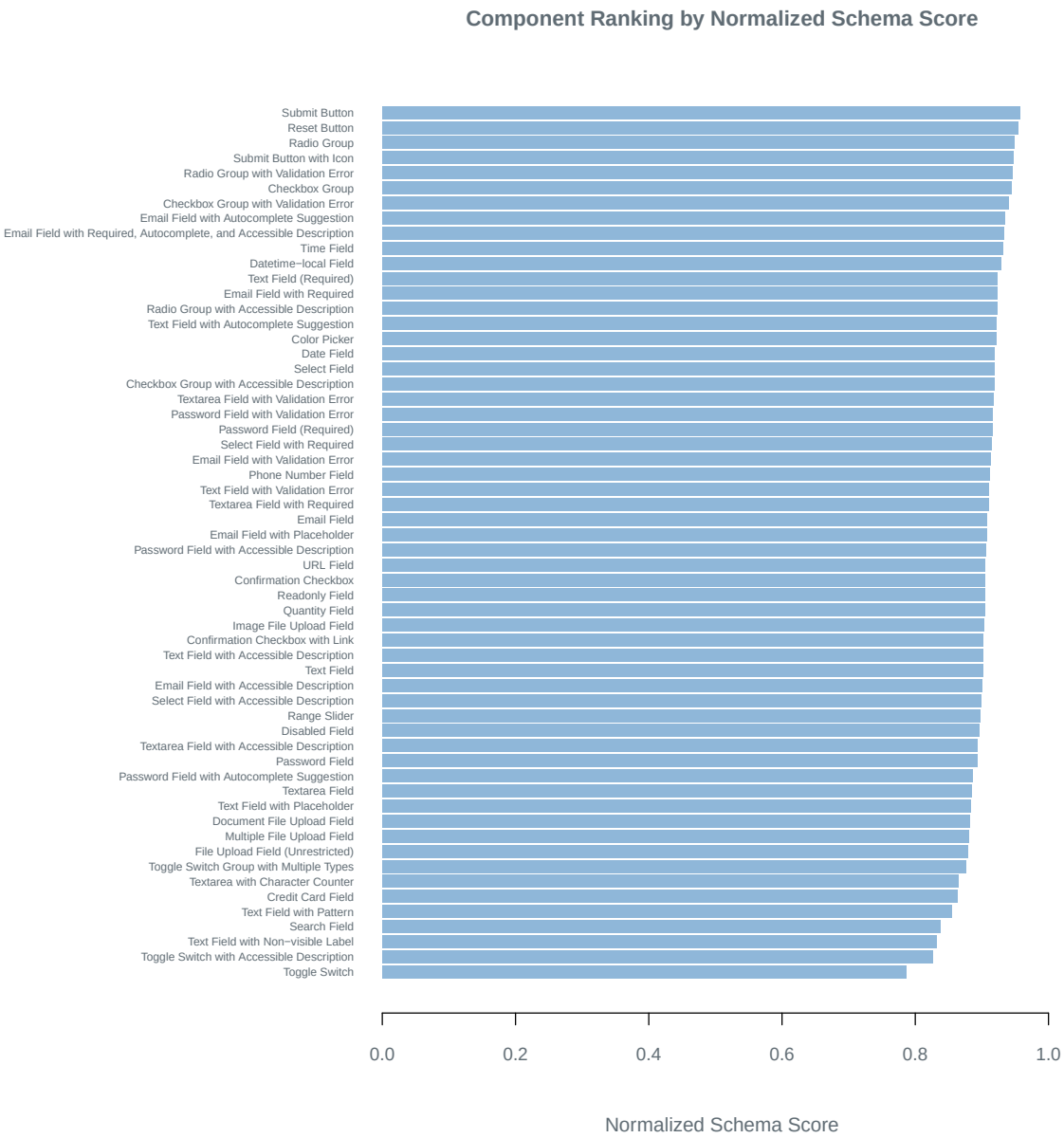


Figure E.2: Overall component ranking by normalized schema score. Higher values indicate more accessible outputs. Simpler submission and selection controls ranked highest overall, whereas more specialized or structurally demanding components tended to rank lower.