Bridging the Detection-Correction Gap: A Prompt Engineering Framework for Automated Accessibility Repair

João Pedro Wieland
COPPE/UFRJ
Brazil
jpvbwieland@cos.ufrj.br

Claudia Werner
COPPE/UFRJ
Brazil
werner@cos.ufrj.br

Abstract

Web accessibility is a legal requirement in many jurisdictions, yet 95.9% of websites remain non-compliant per the WebAIM Million study. Existing tools detect WCAG violations but provide no mechanism to correct them, creating a critical detection-correction gap. We present a systematic framework that transforms multi-tool accessibility detection outputs into structured inputs for LLM-powered code repair, enabling fully automated correction on real production codebases without model fine-tuning or curated training data. The framework orchestrates three complementary detection tools through a six-component prompt template, routes each violation to one of two prompt-level repair conditions (a surgical minimal-edit prompt and a broad-context prompt), and validates results programmatically before any patch is committed. We evaluate on a corpus of 1,249 React/TypeScript files from 36 real-world open-source projects containing 168 WCAG violations across 33 files, crediting each repair only when an independent browser re-scan confirms the violation is gone. A preliminary ablation isolates the prompting strategy and shows that zero-shot prompting substantially outperforms few-shot and chain-of-thought for this constrained-repair task. Zero-shot is therefore used to compare three open-weight models under a 6 GB VRAM constraint across three stability-check repetitions. The three models are statistically indistinguishable in repair quality (Issue Fix Rate of 50–54%), demonstrating that scaling the model up buys no measurable quality on this task. However, they differ significantly regarding computational overhead, where the smallest model is fastest by more than an order of magnitude (0.33 s versus 5.7 s per file), making it the pragmatic choice for interactive repair. Finally, a four-layer validation pipeline rejects every behaviour-altering patch before it is committed, ensuring no regression reaches a previously clean file.

Keywords

Large Language Models, LLM-Based Code Repair, Automated Program Repair, Web Accessibility, WCAG Compliance, Prompt Engineering, Software Quality

1 Introduction

The web accessibility problem remains a persistent challenge in web development. Nearly all major websites, 95.9% [1], contain violations of the Web Content Accessibility Guidelines (WCAG), the international standard for accessible content. This is not for lack of tools: Pa11y [2], Axe [3], and Google Lighthouse [4] can scan a codebase and enumerate violations in seconds. Developers can see, in exhaustive detail, exactly what is wrong. What they cannot see is what to do about it.

This is the *detection-correction gap*: the breach that separates a structured diagnostic report from a valid, tested, production-ready code fix. The gap exists because detection tools were designed to *audit*, not to *repair*. A Pa11y report tells you that `a:nth-child(3)` has inadequate link purpose text under WCAG 2.4.4. It does not tell you what the link currently says, where it points, what framework conventions govern the component, what other screen reader announcements the surrounding code produces, or what constitutes a semantically meaningful replacement. All of that knowledge must be assembled by a developer requiring fluency in WCAG specification, familiarity with the codebase's conventions, and judgment about what preserves intent while meeting the standard.

LLM-powered code models offer a different possibility. Equipped with code-editing prompts and validation feedback, they can in principle interpret violation reports, locate the relevant source, implement fixes, and validate outcomes without human intervention. However, deploying these models effectively requires more than routing raw detection output to the model. Detection tool outputs are succinct and structured for human readers: they name violated criteria and affected elements, but provide no surrounding code context, no framework conventions, no user-impact framing, and no explicit repair constraints. Without structured transformation, models generate fixes that are locally plausible but globally incorrect, often exhibiting hallucinations (i.e., producing fabricated, unsupported, or semantically invalid content such as nonexistent or inappropriate ARIA attributes), over-engineering (introducing unnecessarily complex modifications for issues that require only minimal changes), or context-insensitive repairs (code that appears valid and may pass isolated checks, yet breaks adjacent components, existing behaviors, visual consistency, or application-wide semantics when integrated).

We leverage the fact that the information needed for repair is largely present in existing detection infrastructure. The violation code maps to a precise Web Content Accessibility Guidelines (WCAG) criterion with a full normative description. The Cascading Style Sheets (CSS) selector points to a JavaScript XML (JSX) source location recoverable through Abstract Syntax Tree (AST) analysis. The rendered context reveals the current state that must be preserved. What is missing is *systematic transformation*: a structured pipeline that enriches detection output, packages it into six purposefully designed prompt components, routes it to the appropriate repair condition, and validates the result before any code change is committed.

That pipeline is what this work presents and evaluates on *real* production codebases, under realistic hardware constraints.

Contributions

The remainder of this paper is organized around three contributions:

(1) A **detection-to-repair framework** (Section 4) that orchestrates three complementary accessibility detection tools through a six-component structured prompt, routes violations to one of two LLM-driven repair conditions (a surgical minimal-edit prompt and a broad-context prompt), and validates repairs through a four-layer programmatic pipeline. The framework requires no model fine-tuning and no curated training data.
(2) A **benchmark corpus** (Section 5.1) of 1,249 React/TypeScript files from 36 real-world open-source GitHub projects, stratified across seven application domains and three project-size and popularity tiers. The corpus contains 168 WCAG 2.1 Level AA violations across 33 files, detected by a three-tool ensemble, with each repair verified by an independent browser re-scan.
(3) A **comparative evaluation** (Section 5) of three state-of-the-art open-weight models (CodeLlama 7B, Qwen 2.5 Coder 3B, Qwen 2.5 Coder 7B) under a 6 GB VRAM deployment constraint, preceded by an ablation that selects the prompting strategy. The comparison yields a counter-intuitive result: on this task model scale does not predict repair quality (the three models are statistically indistinguishable), so the smallest model, fastest by more than an order of magnitude, is the pragmatic deployment choice.

2 Background and Motivation

Software quality assurance tools have evolved along a consistent trajectory: ever-more-precise detection, no automated correction. ESLint [5] flags semantic violations but leaves judgment-requiring fixes to the developer. SonarQube [6] identifies technical debt with rich context and stops there. Security scanners like Bandit [7] and Semgrep [8] flag CWE vulnerabilities with line-level precision, and still require manual remediation. The correction step is universally treated as a human problem.

This design reflects a reasonable historical assumption: generating correct code that preserves existing behavior in an unfamiliar codebase was genuinely beyond automated systems. LLM-powered code models mitigate these limitations: prompted with the right context and given validation feedback, they can locate relevant source, propose fixes, and iterate toward correct solutions with minimal human guidance. But raw model capability does not automatically close the gap.

Web accessibility is a *methodologically favorable* domain for studying automated repair, with properties that are rare in quality engineering. WCAG 2.1 [9] defines 78 testable success criteria with machine-checkable pass/fail semantics: WCAG 1.1.1, for instance, requires non-text content to have a text alternative, a rule that holds or does not deterministically, independently of subjective judgment. This determinism lets us verify repairs programmatically, closing the evaluation loop within automation. The W3C's Accessibility Conformance Testing (ACT) Rules [10] provide an independently validated rule corpus that Pa11y implements directly, so re-running Pa11y on a repaired component is an objective, reproducible oracle. The violation space also spans a wide complexity range, from adding a single `alt` attribute to managing focus order across a modal lifecycle.

To make that range concrete, contrast two repairs. The trivial end is an icon-only button missing an accessible name: the fix is a one-token `aria-label` addition, fully determined by the element and its context. The hard end is a modal dialog that traps keyboard focus incorrectly (WCAG 2.4.3, 2.1.2): a correct repair must move focus to the dialog on open, cycle `Tab` within it, restore focus to the triggering element on close, and wire an `Escape` handler: logic that spans the component's lifecycle, depends on refs and effects defined elsewhere, and cannot be inferred from the flagged element alone. The same detection tool reports both as a single line; the variance in repair complexity between them is exactly what an automated framework must navigate, and, as the evaluation shows, exactly where its ceiling lies.

Beyond methodology, accessibility matters in human terms. The 95.9% noncompliance rate documented by WebAIM [1] persists despite mature detection tooling precisely because of the detection-correction gap. For users who navigate by keyboard, rely on screen readers, or require sufficient color contrast, these errors are barriers, not inconveniences.

Applying LLM-powered code models to repair introduces characteristic failure modes that directly motivate the framework's design. *API hallucination*: models generate syntactically valid but nonexistent interfaces, such as `aria-role="navigation-button"`, where both attribute and value are invented. *Specification misinterpretation*: for WCAG 1.1.1, models trained on general corpora often produce `alt="decorative image"` instead of the required `alt=""`, forcing screen readers to announce a string that should be suppressed. *Over-engineering*: a hex-value contrast fix may elicit a custom theme-managing React hook, correct in principle but undeployable in practice. *Context blindness*: a model unaware of the project's styling or state conventions may introduce inline styles in a CSS-Modules project or conflicting local state. Each mode points to a specific deficiency in raw detection output as input, and to a specific component of the prompt template designed to address it.

3 Related Work

Automated program repair has progressed through generate-and-validate genetic search [18], constraint-based synthesis [19], template-based repair [20], and neural sequence-to-sequence models [21].

More recently, AlphaRepair [22] applies zero-shot CodeBERT cloze repair without task-specific fine-tuning, and ChatRepair [23] uses iterative conversational repair. All of these approaches target classical software bugs where a failing test defines the repair goal. Our setting is structurally different: detection tools define violations, domain compliance is the success criterion, and no failing tests exist to guide repair.

Accessibility evaluation and repair have seen growing research activity. Pa11y, Axe, and Lighthouse together cover 70–90% of testable WCAG criteria [15]. López-Gil and Pereira [16] establish that LLMs reach 87.18% accuracy on WCAG *detection* for criteria that conventional tools require manual evaluation for, confirming

that these models have internalized accessibility knowledge. ACCESS [17] applies GPT-3.5 to repair suggestions on static HTML pages, achieving 51% violation reduction but requiring manual intervention to apply fixes. Our work targets modern component-based React applications with fully automated repair and programmatic validation, and evaluates on real production codebases rather than curated HTML snippets.

Prompt engineering for code tasks has established that semantically relevant few-shot examples improve performance substantially over zero-shot, with diminishing returns beyond three to five examples [24, 25]. Chain-of-Thought prompting [26] benefits complex reasoning tasks at the cost of latency. What the literature lacks is a systematic study of prompt design for detection-to-repair transformation: structuring violation reports into model-consumable prompts in a domain with objective validation. That is the gap this work fills.

LLM-powered coding agents such as SWE-Agent [13] demonstrate that agents equipped with code-editing tools can resolve 12.5% of real-world GitHub issues from SWE-bench, a rate that grows with model capability, while OpenHands [14] provides a complementary architecture with broader tool access and stronger multi-step planning. We do not run these agents; rather, their contrasting design philosophies (minimal, surgical edits versus broad-context planning) motivate the two prompt-level repair conditions we evaluate.

End-to-end accessibility remediation systems represent an emerging and directly comparable research thread. FixAlly [29] demonstrates a plan-localize-fix agent for SwiftUI mobile applications, achieving plausibility rates of 77% and developer acceptance of 69.4%. Two parts of FixAlly are intrinsically tied to SwiftUI: its localization operates over the UIKit/SwiftUI accessibility tree and its fix templates encode iOS-specific APIs (`accessibilityLabel`, traits). The remaining machinery is technology-agnostic and reappears in our design: the source-localization phase (mapping an accessibility failure to component source) parallels our Babel AST traversal, the multi-stage retry confirms the value of verification-driven iteration, and FixAlly's *developer-acceptance* evaluation is a methodology our automated, re-scan oracle does not yet capture: measuring acceptance of our generated fixes is a direct line of future work. Fernández-Navarro and Chicano [30] present an end-to-end pipeline for Angular using AST-aware edits and dynamic re-rendering feedback; their approach requires deep framework-specific instrumentation, while ours targets React with a prompt abstraction that could in principle generalize to other component models. VLM-based approaches such as WebAccessVL [31] condition repair on visual screenshots and supervised fine-tuning, achieving larger reduction rates at the cost of substantial dataset curation; our framework reaches competitive rates *without fine-tuning* through its structured prompt components, and the trade-off between SFT specialisation and prompt-engineering generality is an open question this work motivates. IDE-integrated tools such as CodeA11y [32] target the same gap at development time, complementing the post-hoc repair evaluated here.

Two further works ground our feedback-driven retry design. InferFix [33] pairs a static-analysis detector with a transformer fixer and conditions generation on the diagnostic's bug type, location, and surrounding context, showing that structured diagnostic signals raise fix accuracy over diagnostic-free generation. ITER [34] iterates repair across multiple locations, re-running the analyzer after each edit and feeding the residual errors back into the next round until the diagnostics clear. Our four-layer pipeline applies the same principle at the accessibility-domain level: the Pa11y re-scan output feeds the retry prompt on a Layer 3 failure, and ESLint diagnostics are appended on Layer 4 failures. Extending this to richer localization signals (e.g., retrieving similar previously-fixed violations from prior runs) is a natural direction for future work.

4 Framework Design

The framework transforms detection tool outputs into automated code repairs through four sequential stages (Figure 1): multi-tool detection and enrichment, structured prompt construction, prompt-based repair, and multi-layer validation. We describe each stage in terms of the problem it solves and the mechanism that solves it.

4.1 Stage 1: Multi-Tool Detection and Enrichment

A fundamental design choice is the use of three complementary detection tools rather than one. Pa11y, Playwright with Axe-Core, and ESLint-jsx-a11y implement different subsets of WCAG rules and exhibit different false-positive profiles. Violations detected by two or more tools are classified as high confidence, and the redundancy also widens coverage: ESLint-jsx-a11y catches static-analysis issues in conditionally rendered paths that runtime tools miss, while Pa11y and Axe catch contrast and interaction issues that static analysis cannot evaluate without rendering.

A second challenge is bridging the DOM-source gap. Pa11y and Axe operate on the rendered HTML, reporting violations using CSS selectors like `div.card > img:nth-child(2)` that identify elements in the live DOM but carry no direct correspondence to JSX source locations. We resolve this through Babel AST traversal (Algorithm 1), matching DOM attribute fingerprints against JSX node attribute profiles to recover source file and line number. This mapping succeeds in 94.0% of cases, established by running the mapper over all 168 violations and counting the fraction meeting the cosine-similarity threshold of 0.8. That threshold was set by manual calibration rather than adopted from prior work: lowering it toward 0.7 admitted spurious matches between structurally similar but distinct elements (e.g., sibling list items sharing a class), while raising it toward 0.9 rejected valid matches differing only in rendered-only fields (computed styles, framework-injected `data-` attributes); 0.8 was where manual inspection found no false matches and few missed ones. A formal sensitivity analysis against a labelled ground-truth set is left for future work. The main failure causes are dynamically generated class names, third-party components whose JSX source is absent from the project tree, and CSS pseudo-class selectors (e.g., `:hover`, `:nth-child`) that carry no attribute fingerprint; unmatched selectors are logged and passed to the model with DOM-only context.

Once the source location is resolved, we extract the component context: ±10 lines of code, component type (functional or class), props interface, relevant state variables, and the styling approach

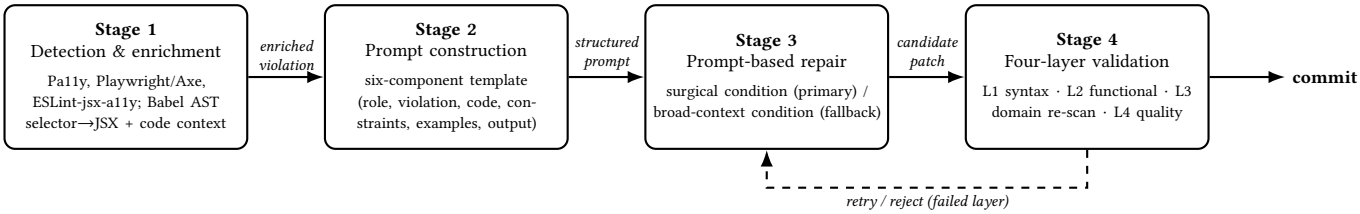

Figure 1: The four-stage pipeline. Each stage transforms its input into the artifact consumed by the next: detection outputs become an enriched violation, which is packaged into a structured prompt, repaired under one of two prompt conditions, and validated before commit. A failed validation layer feeds its error back as context for one retry.

Algorithm 1 Map CSS Selector to JSX Source Location

Require: CSS selector s, React project directory D
Ensure: Source file f, line number l

```
target ← BrowserDOM.querySelector(s)
attrs ← ExtractAttributes(target) {id, className, src, href, ...}
for each file f in D do
    ast ← Babel.parse(f)
    for each JSXElement n in ast do
        sim ← CosineSimilarity(Embed(n.attrs), Embed(attrs))
        if sim > 0.8 then
            return f, n.loc.start.line
        end if
    end for
end for
return NULL
```

(CSS Modules, styled-components, inline). This enriched violation object feeds the prompt construction stage.

4.2 Stage 2: The Six-Component Prompt Template

The prompt template transforms enriched violation objects into model-consumable instructions. Six components work in concert, each addressing a specific failure mode identified in Section 2. The components were not adopted wholesale from a single prompting framework; each was added in response to a concrete failure observed during development, and the resulting set aligns with the building blocks established in the prompt-engineering literature (role priming, in-context grounding, explicit constraint, demonstration, and output contracting [24, 27]), of which the CRISPE schema (Capacity/Role, Insight, Statement, Personality, Experiment) [28] is one widely used instance. The organising principle is the explicit mapping from failure mode to component, detailed component by component below, rather than adherence to any one template.

4.2.1 Component 1: Role Definition. LLMs exhibit measurably different behavior based on persona specification [27]: the role activates relevant knowledge patterns and sets the register for subsequent content. The role string ("You are an expert React accessibility engineer…fix accessibility violations while preserving all existing functionality") primes React-specific knowledge (JSX patterns, hooks, ref-based focus management) rather than generic HTML repair strategies.

4.2.2 Component 2: Violation Context. This is the most semantically loaded component. Raw detection output is intentionally terse: it names the criterion and the element, nothing more. This component expands the Pa11y/Axe violation into the full WCAG normative description, the severity level, the current rendered HTML, and a statement of *user impact*: who is affected and how, as shown in Listing 1.

```
VIOLATION DETAILS:
- WCAG Criterion: {WCAG_CODE} ({LEVEL})
- Rule: {RULE_NAME}
- Description: {FULL_NORMATIVE_DESCRIPTION}
- Confidence: {high|standard} ({N_TOOLS} tools
    detected)
- Element: {CSS_SELECTOR}
- Current HTML: {HTML_CONTEXT}

This violation prevents {USER_IMPACT}.
```

Listing 1. Component 2: the violation-context block of the prompt template.

The `USER_IMPACT` field is semantically critical. "This violation prevents screen reader users from understanding the link's destination" steers the model toward repairs that address the *intent* of the criterion, not just its letter. We repeatedly observed the model generating technically compliant but semantically empty fixes as `alt="image"` instead of a descriptive alternative until this framing was added. Confidence level is also surfaced here, signaling to the model when two independent tools agree on the violation and higher confidence in the diagnosis is warranted.

4.2.3 Component 3: Code Context. Without architectural context, the model may generate fixes that are locally correct but inconsistent with the codebase: `document.getElementById()` in a `useRef` project, inline styles in a CSS Modules codebase. This component injects the current component's source code alongside framework and styling metadata, as shown in Listing 2.

```
CODEBASE CONTEXT:
- Framework: React {VERSION} (functional + hooks)
- Styling: {CSS_APPROACH}
- File: {RELATIVE_FILE_PATH}
- Parent component: {PARENT} (props: {PROPS})

Current component code:
```jsx
{COMPONENT_CODE (+/-10 lines)}
```
```

Listing 2. Component 3: the code-context block of the prompt template.

4.2.4 Component 4: Constraints. Without explicit constraints, models favor comprehensive solutions. This component establishes boundaries through positive directives (preserve existing props,

state, handlers, and the public interface; follow the codebase's React conventions; use the simplest solution) and negative ones (do not modify unrelated code, introduce dependencies, or add abstraction). "Simplest solution" directly targets over-engineering, and "do not modify unrelated code" prevents scope creep that passes validation while introducing unintended changes.

4.2.5 Component 5: Few-Shot Examples. Semantically relevant examples provide the model with format demonstration and strategy grounding. We select the k most similar examples from a curated library of validated violation-fix pairs by cosine similarity over sentence embeddings of the violation description; each example carries before-code, after-code, and a brief rationale, modelling the reasoning process rather than just the input-output transformation. Whether these exemplars help is an empirical question. As the ablation in Section 5.4 shows, for the small models under our 6 GB budget the few-shot exemplars *hurt* repair quality by crowding scarce context, so the deployed configuration disables this component and prompts zero-shot; we retain it in the template description because it is the part the ablation isolates.

4.2.6 Component 6: Output Format Specification. The response is constrained to a structured JSON object (fields `fixed_code`, `explanation`, `wcag_criteria_addressed`, and `confidence`), which enables automated extraction and reduces generative degrees of freedom in ways that improve repair coherence [27]. The `confidence` field provides a self-assessment signal: fixes below 0.7 are routed to human review rather than auto-applied.

4.3 Stage 3: Prompt-Level Repair Conditions

The structured prompt is routed to one of two repair conditions that differ only in how the prompt is shaped; both run as direct calls to the local model, and the framework's routing and validation logic is identical across them.

The *surgical condition* is the primary one. It applies a minimal-edit prompt (FIND/REPLACE patches) and addresses up to four issues per file in a single session, which is efficient for files with clustered violations.

The *broad-context condition* serves as a fallback for cases where the primary condition exhausts its retry budget without resolving the violation. It applies the full six-component template and is intended for complex semantic violations (ARIA relationships, focus management).

Temperature is set to T=0.1 for both conditions. Accessibility repairs typically have a single correct form (a missing label has one right value, not many), so low-temperature decoding favours deterministic, on-target edits over exploratory variation, and also makes the repetitions comparable. The retry budget is three attempts per file, chosen during development as the point of diminishing returns: a second attempt recovers a substantial share of Layer-1 and Layer-3 rejections when the failure context is fed back, whereas attempts beyond the third rarely convert and multiply latency without improving the fix rate.

4.4 Stage 4: The Four-Layer Validation Pipeline

Generated fixes pass through four sequential validation layers. Failed layers feed back as error context for one additional retry, transforming the pipeline from a passive quality gate into an active correction mechanism.

Layer 1: Syntactic Validation. Babel parses the generated JSX. Syntax errors trigger regeneration with the error message appended as context.

Layer 2: Functional Preservation. AST differencing verifies that props interfaces, state variables, and event handler signatures are unchanged unless the fix explicitly requires modification. Any functional signature change is escalated for human review.

Layer 3: Domain Verification. Pa11y reruns against the repaired component, confirming that the original violation is resolved and no new violations were introduced, and its error message feeds the retry prompt on failure. Pa11y is the sole domain oracle here by design: it covers the runtime WCAG 2.1 Level AA checks relevant to the corpus, so adding Axe would double the Layer 3 cost with no extra corrective signal. ESLint-jsx-a11y is a source-level checker reserved for Layer 4.

Layer 4: Code Quality Assessment. ESLint validates the generated code against the project's configuration, and cyclomatic complexity is checked against a threshold of 10. Quality failures generate warnings without blocking application, reflecting the prioritization: functional correctness and domain compliance are hard constraints; code quality is a soft one.

5 Empirical Evaluation

5.1 Building the Benchmark Corpus

A central methodological decision was to evaluate on *real-world production code* rather than synthetic benchmarks constructed from test cases. Accessibility violations in synthetic snippets, isolated HTML fragments or toy components, do not represent the entangled, contextually complex violations that developers actually encounter in production React applications. We therefore constructed a benchmark corpus from 36 open-source GitHub projects.

The selection was designed to avoid evaluation on a convenience sample. Projects were stratified across three independent dimensions: *application domain* (e-commerce, government, healthcare, education, developer tools, dashboard, social), *project size* (small: 10–50 TSX files; medium: 51–300; large: over 300), and *popularity tier* (emerging: 100–999 GitHub stars; established: 1K–10K; popular: over 10K). Across the resulting grid, projects were sampled to achieve balanced coverage, subject to inclusion criteria requiring React/TypeScript composition, active maintenance, and no existing accessibility automation in CI pipelines. We operationalise *active maintenance* as at least one commit to the default branch within the twelve months preceding collection, together with an open-source licence; projects that were archived, read-only, or dormant beyond that window were excluded. Each project was pinned to a specific git commit to ensure complete reproducibility.

Each project was scanned by three complementary tools: Pa11y version 9.1.1, Playwright with Axe-Core, and ESLint-jsx-a11y version 10.0.3. The three tools are intentionally complementary: ESLint-jsx-a11y performs static analysis on JSX source, catching violations in conditionally rendered paths invisible to runtime tools; Pa11y and Axe evaluate the live DOM, detecting contrast and interaction violations that static analysis cannot assess. Together they provide coverage across WCAG categories that no single tool achieves.

Table 1: Benchmark corpus statistics

Attribute	Value
Total files	1,249
Files with violations	33 (2.6%)
Total violations	168
Multi-tool (high-confidence) violations	0
Violations per affected file (mean / median / mode)	5.09 / 1 / 1
Max violations in a single file	42
Projects (GitHub repos)	36

Table 2: Violation distribution across WCAG categories. Single-attribute *structural* categories (`aria`, `alt_text`, `contrast`, `label`, target-size) admit higher repair rates; `semantic` structure (1.3.1), despite being the largest category, concentrates the framework's residual failures (Section 5.5).

Category	WCAG Criteria	Count	%
`semantic`	1.3.1	55	32.7
`aria`	4.1.2	43	25.6
`other` (target size)	2.5.8	28	16.7
`label`	2.4.4, 3.3.2	18	10.7
`contrast`	1.4.3	16	9.5
`alt_text`	1.1.1	7	4.2
`keyboard`	2.1.1	1	0.6
Total		**168**	**100**

Violations appearing in two or more tool reports for the same element are classified as high confidence. In this corpus, no violation reached that multi-tool consensus: all 168 confirmed violations were reported by a single tool, a consequence of the three tools covering largely non-overlapping rule subsets on this code. The framework targets all 168 violations and surfaces confidence level to the model as context; the absence of consensus cases is a limitation we return to in Section 6.

All React/TypeScript component files in the 36 projects were scanned and evaluated: the 1,249 files reported are the *complete* set of eligible component files across the corpus, not a sampled subset. Of these, 33 files (2.6%) contain at least one accessibility violation, for a total of 168 violations, summarized in Table 1. The per-affected-file count is heavily skewed (mean 5.09, but *median and mode of 1*) because 19 of the 33 files carry a single violation while a handful of large component files carry many (up to 42). Detected issues span seven WCAG violation categories, broken down in Table 2: `semantic` (WCAG 1.3.1), `aria` (4.1.2), `other`/target-size (2.5.8), `label` (2.4.4, 3.3.2), `contrast` (1.4.3), `alt_text` (1.1.1), and `keyboard` (2.1.1). No single category dominates: `semantic` structure (32.7%) and `aria` attributes (25.6%) are the most frequent, followed by a long tail of target-size, label, contrast, and alt-text issues. As the per-category analysis in Section 5.5 shows, this spread is favourable because aggregate performance is not an artefact of one easy or one hard category dominating the corpus.

5.2 Experimental Protocol

We evaluate three open-weight models under a 6 GB VRAM deployment constraint, reflecting the hardware reality of academic and small-organization settings. CodeLlama 7B [11] is an instruction-tuned code model from Meta. Qwen 2.5 Coder 3B and 7B [12] are instruction-tuned code models from Alibaba with a focus on code generation and repair tasks. All models run under 4-bit quantization (bitsandbytes) on a single NVIDIA GPU with 6 GB VRAM available for inference.

Each model processes all 1,249 files three times (three stability-check repetitions). Each repetition starts from the unmodified pinned snapshots and shares no state with the others. With the seed and temperature fixed (below), the repetitions probe environmental stability (scheduler jitter, browser-rendering timing) rather than draws from a high-entropy distribution; we therefore report means and standard deviations to characterise stability, not as a basis for parametric significance testing.

The surgical condition is the primary repair condition (max 4 issues per file session, 3 retries per file). The broad-context condition is invoked as a fallback when the primary condition exhausts its budget without resolution. Both use T=0.1.

All experiments were executed on a commodity laptop (Acer Nitro ANV15-52) equipped with an NVIDIA RTX 4050 Mobile GPU (6 GB VRAM), an Intel i7-13620H CPU (16 threads), and 32 GB of RAM, running Ubuntu 24.04 LTS. Models were loaded at 4-bit quantization via bitsandbytes [35]. A dedicated headless Chromium process is launched *per file session* and not reused, so the browser cold-start cost (≈0.5–0.8 s) is fully included in MTTR; files are processed sequentially, so MTTR reflects true per-file latency. The sub-second MTTR for Qwen 2.5 Coder 3B (0.33 s) is dominated by the detection-only path: only 33 of 1,249 files invoke the LLM at all, so the mean is pulled toward the cost of scanning a clean file. Per-stage timing was not logged separately.

5.3 Measurements

Evaluating automated code repair requires metrics that capture independent quality dimensions: correctness, domain compliance, and cost. Our metric suite makes each independently visible.

Success Rate (SR) is a file-level binary metric: a file is "successful" if Pa11y detects no unresolved violations after repair. For the 97.4% of files with no initial violations, SR is trivially 1, so they serve as regression guards; because they dominate the corpus and inflate the aggregate, we report *complete-repair SR*, restricted to the 33 violation-bearing files, as the file-level effectiveness measure. SR is conservative: a file with one of two violations fixed counts as failure.

Issue Fix Rate (IFR) is the issue-level metric: a file with two violations where one is resolved contributes 0.5 to the numerator. A violation counts as resolved only when an independent browser re-scan of the patched component no longer reports it, so IFR measures re-scan-verified repair rather than self-reported success. It is unaffected by the large proportion of violation-free files and is our primary measure of repair effectiveness.

Mean Time to Repair (MTTR) is the average wall-clock time per file, from job start to validated result; for the 33 violation-bearing files it includes detection, repair invocation, retries, and

validation, and for the rest it reflects detection overhead only. It is critical for deployability: matching IFR at an order of magnitude more latency imposes very different operational costs.

Token Efficiency is the total output tokens generated by the LLM backend over the corpus. Combined with IFR it yields *tokens per resolved issue*, the computational price of each successful repair.

Regression Rate (ρ) is the fraction of repair attempts that produce a patch altering a component's structural interface. It is measured at Layer 2 (AST interface differencing), and every flagged patch is rejected before it is written, so ρ quantifies regressions *prevented*, not shipped.

Condition Selection tracks how often the surgical versus the broad-context condition handled each file. A high broad-context rate signals that violations are systematically harder than the primary condition can resolve in its retry budget, itself diagnostic of the violation complexity distribution.

5.4 Strategy Ablation: Selecting the Prompting Strategy

Before comparing models, we isolate one design choice that would otherwise confound the comparison: the prompting strategy. Holding the model (Qwen 2.5 Coder 3B), corpus, and pipeline fixed, we run the framework under three strategies: zero-shot (the six-component template without few-shot exemplars), few-shot (with the exemplars of Component 5), and chain-of-thought, each over three repetitions, and measure IFR, regression rate, and output tokens per resolved violation. This one-factor-at-a-time design isolates, with a controlled test, the contribution of the template's most contentious component. Because the same 168 violations are repaired under all three strategies, the comparison is a repeated-measures design; we report the Friedman test as the primary omnibus result for this reason, and the Kruskal–Wallis test alongside it as a robustness check that treats each repetition as an independent draw. The two agree in both direction and significance.

The strategies are not close (Table 3). Zero-shot resolves 56.3% of violations, roughly four times the few-shot rate (14.2%) and nearly six times chain-of-thought (9.6%); the omnibus tests reject equality strongly (Friedman and Kruskal–Wallis, $p \approx 0.0003$), and the pairwise contrasts are substantial (zero-shot vs. few-shot $p = 0.019$, Cliff's $\delta = -0.34$, medium; vs. chain-of-thought $p = 0.0004$, $\delta = 0.47$, large). The same ordering holds on cost: zero-shot spends about 1,100 output tokens per resolved violation against roughly 10,700 for few-shot and on the order of 136,000 for chain-of-thought. This direction is explained by the setting. For a 3B model under a 6 GB budget, retrieved exemplars and a reasoning preamble crowd the limited context and draw attention away from the violation in its surrounding code, while chain-of-thought's long generations tend to introduce edits the file did not need (its 0.34 regression rate is more than triple zero-shot's). For the short, tightly bounded edits this task requires, a leaner prompt performs better. We therefore use *zero-shot* for the model comparison that follows; the ablation's own zero-shot IFR (56.3%) and the Qwen 2.5 Coder 3B zero-shot IFR in that later comparison (52.6%, Table 4) come from separate sets of three stability-check repetitions and sit within one another's run-to-run spread (Section 5.5), not from a change in strategy or model.

Table 3: Strategy ablation (Qwen 2.5 Coder 3B, 168 violations, mean over 3 repetitions). Zero-shot dominates on both effectiveness and cost.

Strategy	IFR (%)	ρ (%)	Output tokens/fix
Zero-shot	**56.3**	**10**	≈1,100
Few-shot	14.2	21	≈10,700
Chain-of-thought	9.6	34	≈136,000

A companion axis varied the repair scaffolding (a direct call, a surgical minimal-edit prompt, and a broad-context prompt) with the strategy held fixed, and no scaffold beat the baseline (McNemar $p > 0.05$). This axis compares prompt shapes rather than external agent frameworks: all conditions run as direct calls to the local model, not as separate agent tools.

5.5 Results

Table 4 reports all metrics for three models over three repetitions. Standard deviations quantify result stability across repetitions.

The comparison shows no difference in quality between models but a large difference in cost. The three models post mean IFRs of 50.4%, 54.4%, and 52.6% (CodeLlama 7B, Qwen 7B, Qwen 3B); every pairwise comparison is non-significant after correction (Kruskal–Wallis with Bonferroni $p = 1.0$, Cliff's $|\delta| \leq 0.02$). At 168 violations the study cannot resolve a few-point gap, so we read this as equivalence within noise rather than a ranking. The practical consequence is that the 7B models are no better than the 3B: scaling the model up bought no measurable repair quality on this task, which argues against the common practice of defaulting to the largest model that fits in VRAM.

Where the models do separate is cost, shown in Figure 2. Mean time per file spans more than an order of magnitude: 0.33 s for Qwen 3B, 0.99 s for Qwen 7B, and 5.73 s for CodeLlama, which additionally incurred per-file generation timeouts and by far the widest run-to-run spread (IFR std 15.5 pp, versus 6.9 and 7.4 for the Qwen models). Because quality is statistically tied, these secondary axes (latency and stability) are what should drive model selection. Qwen 2.5 Coder 3B is the fastest by a wide margin and is as stable as Qwen 7B across runs (IFR std 7.4 vs 6.9 pp, and the lowest latency and complete-repair SR variance of the three), while CodeLlama is far less stable; this makes the 3B the pragmatic default. A full-corpus pass takes roughly 7 min for Qwen 3B, 20 min for Qwen 7B, and 2 h for CodeLlama.

Complete-repair SR. Restricting SR to the 33 violation-bearing files (the fraction in which *every* issue was resolved at once) gives CodeLlama 7B 59.6%, Qwen 7B 54.5%, and Qwen 3B 50.5% (Table 4). As with IFR these differences sit within the run-to-run spread; the gap from IFR (partial credit) to complete-repair SR reflects multi-issue files where one violation was fixed while another remained pending.

Per-category IFR: which violations get resolved. Aggregate IFR hides a sharp structure, detailed in Table 5. Repairs that insert or correct a single attribute on a known element resolve well across all three models: ARIA names, alt-text, contrast, target-size, and label associations sit between roughly 55% and 76%. Repairs that require

Table 4: Model comparison (means ± std over 3 repetitions, n=1,249 files, 168 violations). IFR is re-scan-verified; complete-repair SR is restricted to the 33 violation-bearing files; ρ is the functional-regression rate (all flagged patches rejected before commit). The three IFR values are statistically indistinguishable (Kruskal–Wallis with Bonferroni $p = 1.0$, Cliff's $|\delta| \leq 0.02$).

Model	IFR (%)	Compl.-rep. SR (%)	ρ (%)	MTTR (s)	Tokens/Fix
CodeLlama 7B	50.4 ± 15.5	59.6 ± 12.6	5.2	5.73 ± 0.60	954
Qwen 2.5 Coder 7B	54.4 ± 6.9	54.5 ± 5.3	7.6	0.99 ± 0.64	1,260
Qwen 2.5 Coder 3B	52.6 ± 7.4	50.5 ± 1.8	11.9	**0.33 ± 0.03**	1,326

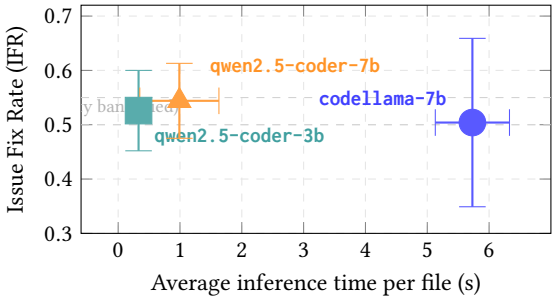

Figure 2: IFR versus average inference time per file (means over 3 repetitions; error bars show ±1 standard deviation on both axes, from Table 4). The three models fall within a tied quality band (Kruskal–Wallis $p = 1.0$; their IFR intervals overlap broadly), but spread across more than an order of magnitude on latency: Qwen 2.5 Coder 3B is fastest, CodeLlama 7B slowest and the most variable on quality.

Table 5: Per-category IFR (mean across the three models, pooled over repetitions). Structural, single-attribute categories repair well; semantic structure is the residual. † small cell, descriptive.

Category	n	Mean IFR (%)
`alt_text` (1.1.1)	7	76.2†
`other`/target-size (2.5.8)	28	66.5
`aria` (4.1.2)	43	61.7
`contrast` (1.4.3)	16	55.7
`label` (2.4.4, 3.3.2)	18	55.5
`semantic` (1.3.1)	55	12.0
`keyboard` (2.1.1)	1	0.0†

meaning the source file does not contain do not: `semantic`-structure violations (WCAG 1.3.1), despite being the largest category at a third of the corpus, resolve at only 12%, and the lone keyboard case at zero. This structural-versus-semantic split is what holds the aggregate IFR in the low fifties; a corpus with fewer semantic-structure violations would report a higher headline number. The ceiling is therefore epistemic, not architectural: how a heading hierarchy should be reorganised or what a link conveys depends on design intent that no prompt can recover from the JSX alone.

A repair example. To make the framework's output concrete, consider `index.tsx` from a project, whose header renders a repository link, shown before repair in Listing 3.

```
<a href="https://github.com/F-star/suika" target="
    _blank">
  {value}
</a>
```

Listing 3. A repository link before repair.

Pa11y flags WCAG 2.4.4 (*link has no discernible text*): the link's only content is the dynamic prop `{value}`, which the scanner cannot guarantee is non-empty meaningful text. A developer skimming the file might dismiss this as a false alarm; the framework instead enriches the finding with the criterion's intent, and the model returns a minimal, correct patch (Listing 4) that adds a stable accessible name without touching the dynamic content.

```
<a href="https://github.com/F-star/suika" target="
    _blank"
   aria-label="GitHub repository">
```

Listing 4. The same link after the framework's repair.

The re-scan confirms the violation is resolved, and the identical fix was produced in all three repetitions. This is the structural, single-attribute case the framework handles best; the semantic-structure cases above are where it does not.

Regression analysis. A central question for any automated-repair tool is whether it breaks what it touches. Across the three models, between 5.2% and 11.9% of repair attempts produced a patch that altered a component's structural interface (the regression rate ρ, Table 4). Every such patch was rejected at Layer 2 before being written, so none reached the codebase: the 1,216 files that began violation-free remained so in every run. The regression rate therefore measures regressions prevented rather than shipped: a non-trivial fraction of surface-plausible patches would have broken the component's contract had the validation layer not intercepted them, which is a direct argument for keeping that layer in any deployment. Routing remained dominated by the primary surgical condition, with the broad-context fallback invoked only on a small minority of files; its session count was too low to support a per-condition success-rate analysis, which we leave to future work with a larger corpus.

Per-repetition IFR values (provided in the replication package) reinforce that quality is not separable across models and reverse a tempting assumption about the largest model: CodeLlama is the *least* stable, swinging from 0.68 to 0.38 across runs (15.5 pp std), while the two Qwen models stay within a tighter band (≈7 pp). No model shows a monotonic trend that survives the small sample; the run-to-run movement is consistent with environmental noise at the fixed seed and temperature.

With quality tied, the recommendation collapses to a single model. Qwen 2.5 Coder 3B is the fastest (0.33 s per file), is as stable as the 7B across runs, and carries no quality penalty. Its sub-second MTTR is essential for interactive tooling (IDE plugins, pre-commit

hooks), and its low latency variance and absence of timeouts suit it equally for unattended batch runs, a role CodeLlama's instability and per-file timeouts now argue against. Qwen 7B matches the 3B on quality but is three times slower, and CodeLlama is an order of magnitude slower still; neither offers a quality gain to justify the cost.

On token economy, the three models are again close, at roughly 950–1,330 output tokens per resolved violation (Table 4), with no correlation between token cost and the (indistinguishable) fix rates. These figures cover output tokens only; the six-component prompt adds an estimated 1,200–1,800 input tokens per violation, so total cost per fix remains comfortably sub-cent at current cloud-inference pricing while being dominated by the input side: a point production cost projections should account for.

Because no violation in this corpus reached multi-tool consensus (Section 5.1), the framework's high-confidence path was never exercised here. Whether multi-tool agreement makes a violation systematically easier or harder to repair, and how its precision varies across WCAG categories, therefore remains an open question for future work on a corpus with overlapping-tool coverage.

When a repair attempt fails to resolve a violation, the most frequent outcome reported in the repair log is "no valid code or patches found in response." In this terminal failure mode the model cannot formulate a syntactically valid patch, and it accounts for the majority of the roughly 75–90 unresolved issues per model (about half of the 168 violations). It is most prevalent in violations requiring multi-file edits (ARIA relationships spanning parent and child components), violations requiring semantic inference about content that is not encoded in the source file (link purpose, image descriptions), and violations where the repair requires understanding of interaction flow across component lifecycle (focus management in modals and dialogs).

A secondary failure mode is superficial addressing: the model modifies the right attribute but with an incorrect value. WCAG 1.1.1 requires `alt=""` for decorative images: the empty string signals to screen readers that the element carries no informational content. Models trained on general corpora frequently generate `alt="decorative image"`, which causes screen readers to announce the string and worsens the user experience despite technically providing an attribute. This class of failure passes Layer 2 (functional preservation) but fails Layer 3 (domain verification by Pa11y), triggering one retry. When the retry also fails on this semantic nuance, the violation is left pending.

The two failure modes together characterize where the automation ceiling lies for the current framework: *structural* repairs (attribute addition, attribute value correction, heading level adjustment) have high resolution rates; *semantic* repairs (content generation for descriptive alternatives, intent inference for link purpose, interaction flow management) require reasoning about user experience that static code analysis cannot fully inform.

These failure categories were identified through qualitative log inspection; the proportion of "no valid code" failures attributable specifically to multi-file coordination requirements (as opposed to within-file semantic failures) was not quantified separately. The broad-context fallback condition was invoked on only a small minority of files, too few to support a per-condition success-rate analysis on specific failure types, which we leave to future work on a larger corpus.

6 Discussion

Our most counter-intuitive finding is that scaling the model up did not improve repair quality. The three models are statistically indistinguishable on IFR (Kruskal–Wallis $p = 1.0$, Cliff's $\delta \approx 0$), and at 168 violations the study is not powered to resolve a few-point gap, so we read the result as equivalence within noise rather than a ranking. In practical terms, a 3B model matches its 7B counterparts on this task. The likely explanation is task calibration: the repairs the framework generates are short, single-component edits, the regime where a smaller code-tuned model is already competent and a larger model's extra capacity for multi-step reasoning has little to contribute. The ablation reinforces this. The dominant performance lever was the prompting strategy rather than the model, with zero-shot beating few-shot and chain-of-thought by a wide, significant margin. How the model was prompted mattered more than which model was used.

Because quality is tied, deployment turns on cost and stability, where Qwen 2.5 Coder 3B dominates: its 0.33 s MTTR is the only interactive latency of the three (a pre-commit hook or IDE plugin must return in under two seconds, which CodeLlama at 5.7 s misses by nearly threefold), and its low variance and absence of timeouts make it the safer batch choice as well. The budget freed by not running a larger model is better spent on a human-review queue for the semantic cases the framework cannot close.

The two failure modes locate the automation ceiling. Invalid patch synthesis concentrates on violations needing multi-file coordination or interaction-flow reasoning that a single component's source cannot supply; superficial addressing concentrates on semantic nuances the general code corpus does not encode (that `alt=""` and `alt="decorative image"` have opposite effects, or that link text must reflect destination, not element type). Layer 3 catches the latter and triggers one retry, but when the model's distribution lacks the correct semantic pattern, retries do not help. These failures are epistemically bounded by what the models have internalized about accessibility, not a compute problem.

The detection-correction gap is not a niche concern: even after filtering out scanner artefacts, confirmed WCAG violations appear in real, actively maintained open-source React projects across every domain sampled, and developers do not correct them because correction is costly: the framework addresses cost, not awareness. That framework is also domain-agnostic by construction: any quality domain with structured detection outputs, programmatic validation, and a documented repair knowledge base admits the same treatment: security (Bandit [7] and a CWE corpus), code style (ESLint and project configuration), and API migrations (deprecation warnings and migration guides), at an estimated two-to-three developer-day adaptation cost per domain.

Direct numerical comparison with FixAlly [29], WebAccessVL [31], or Fernández-Navarro and Chicano [30] is impossible without shared benchmarks, but the contrast is clear: VLM approaches reach higher raw reduction rates through visual conditioning and fine-tuning, at the cost of curation effort, whereas this framework achieves competitive rates with none.

7 Threats to Validity

Construct validity. IFR and SR are defined entirely in terms of automated scanner output: a fix is "successful" if Pa11y no longer flags the element. This creates a circularity: the oracle used to verify repair is the same class of tool used for detection. Violations that scanners systematically misclassify will be counted as fixed when they are not, and genuine improvements that satisfy the criterion without satisfying the rule implementation will be counted as failures. No assistive-technology validation (NVDA, VoiceOver, JAWS) or user study was conducted; the reported IFR approximates scanner-compliance improvement, not accessibility improvement as experienced by users of these technologies, and should be treated as an upper bound on true accessibility benefit.

Per-category IFR (Table 5) shows a clear structural-versus-semantic split, but its limitation is statistical power: two categories (keyboard, n=1; alt-text, n=7) are too small for inference, and even the well-populated categories rest on 168 violations, so the rates are characterisation rather than precise estimates.

Regression detection is bounded by Layer 2 (AST interface differencing) and Layer 4 (ESLint). Behaviour regressions that preserve the component's public interface (reordered tab stops, broken event delegation, altered ARIA live-region sequencing) would not be detected; no runtime component-mounting tests (e.g., React Testing Library) were applied.

Internal validity. The framework's failure-mode attributions (multi-file scope, semantic inference gaps) rest on qualitative log inspection. The strategy ablation (Section 5.4) provides controlled evidence for one factor, but it was run on a single model, so its magnitude may not transfer to the larger models; a component-level ablation isolating each of the six prompt components, and a stricter raw-Pa11y-JSON baseline, remain future work.

External validity. The corpus is restricted to React/TypeScript; Angular, Vue, and vanilla JavaScript differ in component model, templating, and static-analysis tooling, so results may not transfer. The 36-project sample, though stratified across domain, size, and popularity, is a single point in time (each project pinned to a commit) and may not capture violation distributions in commercial or government codebases. The 6 GB VRAM constraint deliberately excludes larger models; the performance of GPT-4, Claude 3, or unconstrained open-weight models on this corpus is unknown.

Cross-framework portability. The pipeline separates into technology-agnostic and React-specific parts, which bounds the porting effort. The agnostic parts dominate: the detection ensemble, the four-layer validation logic, and four of the six prompt components (role, violation context, constraints, output contract) operate on WCAG semantics, not React syntax. The React-bound machinery is confined to the Babel AST traversal that maps a CSS selector to a JSX location (Stage 1) and the code-context extraction (Component 3). Porting to another component-based web framework (Angular, Vue) is therefore mostly replacing the AST parser and context extractor with framework-native equivalents (the Angular template compiler, the Vue SFC parser), reusing the template and validation unchanged (an estimated two-to-three developer-days). Mobile frameworks (SwiftUI, Flutter) are a larger step: with no DOM or CSS selectors, the entire selector-to-source mapping must be rebuilt against a platform-specific accessibility tree, and the re-scan oracle (Pa11y) has no direct analogue, so adaptation effort and achievable fix rates depend more on the availability of a programmatic accessibility oracle for the platform than on the model.

Conclusion validity. The corpus of 168 violations is the binding constraint on the model comparison. The between-model IFR differences (under four points) are smaller than the within-model run-to-run variance, and the formal tests (Kruskal–Wallis with Bonferroni correction, Cliff's δ) find no significant difference for any pair, so we report the models as indistinguishable rather than ranked: a non-significant result at this sample size is absence of evidence for a difference, not evidence of equivalence. The strategy ablation, by contrast, cleared significance even at this scale, and is the study's firm quantitative finding.

8 Conclusion

Software quality assurance has long automated detection but left correction manual. This work shows that for web accessibility (and, by extension, any quality domain with structured detection outputs and programmatic validation) that asymmetry is addressable through a systematic detection-to-repair framework, without fine-tuning or curated data.

Evaluated on 1,249 real-world React/TypeScript files from 36 open-source projects, with every repair verified by an independent browser re-scan, the framework resolves roughly half of the 168 confirmed violations (IFR 50–54%), and its four-layer validation rejects every behaviour-altering patch before commit. Two findings stand out: the dominant performance lever is the prompting strategy, not the model (zero-shot beats few-shot and chain-of-thought by a wide, significant margin), and scaling the model up buys no measurable quality (the three models are statistically indistinguishable), so the smallest and fastest model is the pragmatic deployment choice, its sub-second latency making interactive use (IDE plugins, pre-commit hooks) viable at all.

Automated repair will not replace accessibility expertise: the roughly half of violations left unresolved (semantic-structure inference, multi-component relationships, interaction-flow management) are the domain's human core. The framework provides everything else, the mechanical, rule-driven repairs that consume developer time without requiring judgment.

Artifact Availability

The framework, benchmark corpus, prompt templates, and replication scripts are at https://github.com/jpwieland/a11y_experiment.

Acknowledgments

The authors thank the anonymous reviewers for their constructive feedback. Generative AI tools assisted with language refinement and clarity; all technical content, experimental design, analysis, and results are the authors' own.

References

[1] WebAIM. *The WebAIM Million: Annual Accessibility Analysis.* https://webaim.org/projects/million/, 2024.
[2] Pa11y. *Pa11y: Automated Accessibility Testing.* https://pa11y.org/, 2024.
[3] Deque Systems. *axe DevTools: Accessibility Testing for Development Teams.* https://www.deque.com/axe/, 2024.

[4] Google. *Lighthouse: Automated Auditing for Web Pages*. https://developers.google.com/web/tools/lighthouse, 2024.
[5] Nicholas C. Zakas. *ESLint: The Pluggable Linting Utility for JavaScript*. https://eslint.org/, 2013.
[6] SonarSource. *SonarQube: Code Quality and Security*. https://www.sonarsource.com/products/sonarqube/, 2024.
[7] PyCQA. *Bandit: Security Linting for Python*. https://github.com/PyCQA/bandit, 2024.
[8] Semgrep. *Semgrep: Lightweight Static Analysis for Many Languages*. https://semgrep.dev/, 2024.
[9] W3C. *Web Content Accessibility Guidelines (WCAG) 2.1*. https://www.w3.org/TR/WCAG21/, 2018.
[10] W3C. *Accessibility Conformance Testing (ACT) Rules*. https://www.w3.org/WAI/standards-guidelines/act/, 2024.
[11] Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, et al. Code Llama: Open Foundation Models for Code. *arXiv preprint arXiv:2308.12950*, 2023.
[12] Qwen Team. *Qwen2.5-Coder Technical Report. arXiv preprint arXiv:2409.12186*, 2024.
[13] John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering. In *Proceedings of NeurIPS '24*, 2024. https://arxiv.org/abs/2405.15793
[14] Xingyao Wang, Boxuan Li, Yufan Song, et al. OpenHands: An Open Platform for AI Software Developers as Generalist Agents. In *Proceedings of ICLR '25*, 2025. https://arxiv.org/abs/2407.16741
[15] Markel Vigo, Justin Brown, and Vivienne Conway. Benchmarking Web Accessibility Evaluation Tools. In *Proceedings of W4A '13*, pages 1–10, 2013.
[16] Juan-Miguel López-Gil and Juanan Pereira. Turning Manual Web Accessibility Success Criteria into Automatic: An LLM-Based Approach. *Universal Access in the Information Society*, 24:837–852, 2025. https://doi.org/10.1007/s10209-024-01108-z
[17] Calista Huang, Alyssa Ma, Suchir Vyasamudri, et al. ACCESS: Prompt Engineering for Automated Web Accessibility Violation Corrections. *arXiv preprint arXiv:2401.16450*, 2024. https://arxiv.org/abs/2401.16450
[18] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. GenProg: A Generic Method for Automatic Software Repair. *IEEE Transactions on Software Engineering*, 38(1):54–72, 2012.
[19] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. SemFix: Program Repair via Semantic Analysis. In *Proceedings of ICSE '13*, pages 772–781, 2013.
[20] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. Automatic Patch Generation Learned from Human-Written Patches. In *Proceedings of ICSE '13*, pages 802–811, 2013.
[21] Michele Tufano, Cody Watson, Gabriele Bavota, et al. An Empirical Study on Learning Bug-Fixing Patches in the Wild via Neural Machine Translation. *ACM Transactions on Software Engineering and Methodology*, 28(4):1–29, 2019.
[22] Chunqiu Steven Xia and Lingming Zhang. Less Training, More Repairing Please: Revisiting Automated Program Repair via Zero-shot Learning. In *Proceedings of ESEC/FSE '22*, pages 959–971, 2022. https://arxiv.org/abs/2207.08281
[23] Chunqiu Steven Xia and Lingming Zhang. Keep the Conversation Going: Fixing 162 out of 337 Bugs for $0.42 Each Using ChatGPT. *arXiv preprint arXiv:2304.00385*, 2023.
[24] Tom B. Brown, Benjamin Mann, Nick Ryder, et al. Language Models are Few-Shot Learners. In *Proceedings of NeurIPS '20*, pages 1877–1901, 2020.
[25] Sewon Min, Xinxi Lyu, Ari Holtzman, et al. Rethinking the Role of Demonstrations: What Makes In-Context Learning Work? In *Proceedings of EMNLP '22*, pages 11048–11064, 2022.
[26] Jason Wei, Xuezhi Wang, Dale Schuurmans, et al. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. In *Proceedings of NeurIPS '22*, pages 24824–24837, 2022.
[27] DAIR.AI. *Prompt Engineering Guide*. https://www.promptingguide.ai/, 2023.
[28] Rui Zhong, Yang Xu, Chao Zhang, and Jun Yu. Leveraging Large Language Model to Generate a Novel Metaheuristic Algorithm with CRISPE Framework. *Cluster Computing*, 27(10):13835–13869, 2024.
[29] Forough Mehralian, Titus Barik, Jeff Nichols, and Amanda Swearngin. Automated Code Fix Suggestions for Accessibility Issues in Mobile Apps. *ACM Transactions on Computer-Human Interaction*, 2024. https://arxiv.org/abs/2408.03827
[30] Carla Fernández-Navarro and Francisco Chicano. Automated LLM-Based Accessibility Remediation: From Conventional Websites to Angular Single-Page Applications. *arXiv preprint arXiv:2602.17887*, 2026.
[31] Amber Yijia Zheng, Jae Joong Lee, Bedrich Benes, and Raymond A. Yeh. WebAccessVL: Violation-Aware VLM for Web Accessibility. *arXiv preprint arXiv:2602.03850*, 2026. https://arxiv.org/abs/2602.03850
[32] Peya Mowar, Yi-Hao Peng, Jason Wu, Aaron Steinfeld, and Jeffrey P. Bigham. CodeA11y: Making AI Coding Assistants Useful for Accessible Web Development. In *Proceedings of CHI '25*, 2025. https://doi.org/10.1145/3706598.3713335
[33] Matthew Jin, Syed Shahriar, Michele Tufano, Xin Shi, Shuai Lu, Neel Sundaresan, and Alexey Svyatkovskiy. InferFix: End-to-End Program Repair with LLMs. In *Proceedings of ESEC/FSE '23*, pages 1646–1656, 2023. https://arxiv.org/abs/2303.07263
[34] He Ye and Martin Monperrus. ITER: Iterative Neural Repair for Multi-Location Patches. In *Proceedings of ICSE '24*, pages 10:1–10:13, 2024. https://arxiv.org/abs/2304.12015
[35] Tim Dettmers, Mike Lewis, Sam Shleifer, and Luke Zettlemoyer. 8-bit Optimizers via Block-wise Quantization. In *Proceedings of ICLR '22*, 2022.