iJETRM
INTERNATIONAL JOURNAL OF ENGINEERING TECHNOLOGY RESEARCH & MANAGEMENT (IJETRM)
https://ijetrm.com/

AUTOMATED ACCESSIBILITY COMPLIANCE PIPELINES FOR WCAG CONFORMANCE IN ANGULAR COMPONENT LIBRARIES

Harshika Juvvadi
Full stack Developer, USA

ABSTRACT
Shared Angular component libraries are the point of highest leverage for accessibility in an enterprise frontend, since a single defect in a widely reused component propagates into every application that depends on it, yet accessibility conformance against the Web Content Accessibility Guidelines is still checked in most organizations through periodic, manual audits that scale poorly with the size and release frequency of a modern component library. This article presents an automated accessibility compliance pipeline for Angular component libraries, built around four coordinated elements, component level automated scanning with axe-core integrated directly into the build and test cycle, structured use of Angular CDK accessibility utilities for focus management and live region announcements rather than ad hoc keyboard handling, an automated conformance check against a defined set of WAI-ARIA authoring practice patterns for each component's design pattern, and a layered program of quarterly assistive technology spot checks that catches the class of defect automated tooling cannot, backed by a continuous integration gate and a persistent regression registry that prevents a previously fixed defect from silently reappearing. Each element is presented with illustrative workflow diagrams and code, including an axe-core continuous integration configuration, an Angular CDK focus management and live announcer example, an ARIA pattern conformance check, a continuous integration gate configuration, and a regression registry schema. An illustrative case study for a single component library reports simulated indicators showing WCAG 2.1 AA conformance rising from an illustrative fifty eight percent with no automation to ninety eight percent as the pipeline's elements are adopted cumulatively, and post release accessibility defects falling from a low to high teens count per release to near zero within roughly one and a half releases of the pipeline's adoption. The article closes with a discussion of what automated tooling still misses, the cost of operating the pipeline at small scale, limitations of the illustrative evaluation, and directions for further work. The intended audience is frontend engineers, accessibility specialists, and engineering leads maintaining shared Angular component libraries, and researchers studying automated accessibility testing in 2025.

Keywords:
accessibility, WCAG, axe-core, Angular CDK, ARIA authoring practices, continuous integration, assistive technology, component library, regression testing

1. INTRODUCTION
A shared Angular component library occupies an unusual position in an enterprise frontend, it is consumed by every application team that depends on it, which means a single accessibility defect introduced in one component release does not stay contained to one product, it ships silently into every application that consumes that version of the library. This amplification effect makes the component library the highest leverage place to invest in accessibility conformance, and also the place where a manual, periodic audit process is least able to keep pace, since the library's release cadence is typically far faster than any quarterly or annual audit cycle can track.
This article treats accessibility conformance for a component library as an engineering pipeline problem rather than solely an audit problem, following the general premise that a defined, checkable subset of the Web Content Accessibility Guidelines can be verified automatically and continuously, while the remaining subset that genuinely requires human and assistive technology judgment is reserved for a smaller, better targeted manual program layered on top. Section 4 presents that pipeline in seven parts, moving from automated scanning through structured utility usage, pattern conformance, layered manual checks, a continuous integration gate, and a persistent regression registry.
Two specific consequences of relying on periodic manual audits alone motivate the pipeline's design. First, an audit that runs quarterly or annually necessarily reviews a snapshot of the library long after many releases have already shipped, which means a defect introduced early in that window may reach a substantial number of consuming applications before it is ever caught, a problem Section 4.2's continuous scanning and Section 4.6's

iJETRM
INTERNATIONAL JOURNAL OF ENGINEERING TECHNOLOGY RESEARCH & MANAGEMENT (IJETRM)
https://ijetrm.com/

gate address directly. Second, a fixed defect has no guarantee of staying fixed, since a later refactor can silently reintroduce the same problem with no automated signal that a previously resolved issue has resurfaced, a problem Section 4.7's regression registry addresses.

The remainder of the article is organized as follows. Section 2 reviews background on the Web Content Accessibility Guidelines, the axe-core testing engine, Angular's accessibility tooling, assistive technology usage data, prior accessibility regulation, component development tooling, and established usability heuristics. Section 3 states the specific problems the pipeline addresses. Section 4 presents the pipeline in seven parts with reference diagrams and illustrative code. Section 5 walks through an illustrative case study for one component library with simulated indicators. Section 6 discusses what automated tooling still misses and the cost of the pipeline at small scale, Section 7 discusses limitations, Section 8 suggests future work, and Section 9 concludes.

2. BACKGROUND AND LITERATURE REVIEW

2.1 The Web Content Accessibility Guidelines

The Web Content Accessibility Guidelines, in their 2.1 revision, organize testable success criteria under four principles, that content be perceivable, operable, understandable, and robust, at three conformance levels, A, AA, and AAA, with level AA the level most commonly adopted as an organizational target (W3C, 2018). A meaningful share of the checkpoints under these four principles, for example whether an interactive element has an accessible name, whether color contrast meets a defined ratio, or whether a form control has an associated label, can be evaluated by static or runtime analysis of rendered markup without requiring a human judgment call, which is precisely the subset the pipeline in Section 4 automates.

2.2 Automated Accessibility Testing with axe-core

The axe-core accessibility testing engine evaluates rendered HTML against a defined rule set mapped to specific WCAG success criteria and is designed to run inside a continuous integration pipeline or a component development environment rather than only as a standalone browser extension, which is what makes it practical for Section 4.2's per component scanning and Section 4.6's gate to catch a defined class of defect automatically before a component ever reaches a consuming application (Deque Systems, 2022). The Accessible Rich Internet Applications specification that axe-core partly checks against defines the roles, states, and properties that allow assistive technology to correctly interpret custom interactive widgets that native HTML elements do not already cover (W3C, 2017).

2.3 Angular Accessibility Tooling and the Component Development Kit

Angular's own accessibility guidance and the accessibility utilities in the Angular Component Development Kit provide structured primitives, a focus trap and focus monitor for managing keyboard focus within a component such as a modal dialog, and a live announcer for surfacing dynamic content changes to screen reader users, that are built and tested against the accessibility criteria in Section 2.1 rather than requiring each team to hand roll equivalent behavior from raw DOM APIs (Angular, 2022). Section 4.3's structured utility usage builds directly on these primitives, on the premise that a shared, tested utility used consistently across a component library is more reliable than a bespoke keyboard handler reimplemented separately in each component.

2.4 WAI-ARIA Authoring Practices and Design Pattern Conformance

The WAI-ARIA Authoring Practices Guide documents a defined set of accessible design patterns, for a combo box, a tab panel, a modal dialog, and similar constructs, specifying the expected roles, keyboard interactions, and focus behavior for each pattern (W3C WAI, 2022). Section 4.4's pattern conformance check treats each of a component library's constructs as an instance of one of these documented patterns and verifies the component's implementation against the pattern's specification, rather than checking only generic accessibility properties that apply regardless of the component's specific interaction model.

2.5 Assistive Technology Usage in Practice

The WebAIM Screen Reader User Survey, conducted and its ninth iteration's results published in 2021, reports on real assistive technology usage patterns, including which screen readers and browser combinations are most common and how users navigate a page, for example by heading, by landmark, or by form control, findings that inform which combinations Section 4.5's manual spot check program prioritizes rather than attempting to test every possible assistive technology and browser combination with equal depth (WebAIM, 2021). Survey based usage data of this kind is also part of why the pipeline in Section 4 does not treat automated scanning alone as sufficient, since several patterns the survey documents, particularly how users navigate rather than merely whether an element has a correct accessible name, are difficult for a static analysis rule to fully evaluate.

2.6 Accessibility Regulation and Organizational Targets

The Section 508 refresh to the Information and Communication Technology accessibility standard incorporated the Web Content Accessibility Guidelines by reference as the applicable technical standard for covered

iJETRM
INTERNATIONAL JOURNAL OF ENGINEERING TECHNOLOGY RESEARCH & MANAGEMENT (IJETRM)
https://ijetrm.com/

technology, giving organizations subject to that standard a specific, externally defined conformance target rather than an internally chosen one (U.S. Access Board, 2017). This article's pipeline is designed to be compatible with that kind of externally imposed target, since the WCAG 2.1 AA criteria referenced throughout Section 4 are the same criteria a Section 508 covered organization would already need to demonstrate conformance against.

2.7 Component Development Tooling and Automated Scoring

Storybook's isolated component development environment gives a natural point at which to run automated accessibility scans against a single component in isolation, independent of the surrounding application, which Section 4.2's scanning approach relies on directly (Storybook, 2022). Lighthouse's accessibility scoring, built on a subset of axe-core's own rule set, gives a single aggregate score intended to approximate a page's or component's overall accessibility, which this article's pipeline deliberately does not rely on as its primary signal, preferring the itemized, rule level results Section 4.2 and Section 4.6 use instead, since an aggregate score can mask a small number of severe violations behind a large number of passing checks (Google, 2022). Nielsen's usability heuristics, while not accessibility specific, share with the criteria in Section 2.1 an emphasis on visibility of system status and error prevention, principles this article's discussion of what automated tooling still misses in Section 6.1 returns to (Nielsen, 1994).

Table 1. Summary of prior work and its relationship to this article's pipeline

Prior work	Core contribution	Relationship to this pipeline
W3C (2018); W3C (2017)	Testable WCAG success criteria and ARIA roles, states, and properties	Basis for automated scanning and pattern checks (Sections 4.2, 4.4)
Deque Systems (2022)	axe-core automated testing engine mapped to WCAG criteria	Basis for the automated scanning stage (Section 4.2)
Angular (2022)	Angular CDK focus and live announcer accessibility utilities	Basis for structured utility usage (Section 4.3)
W3C WAI (2022)	Documented accessible design patterns for common widgets	Basis for pattern conformance checking (Section 4.4)
WebAIM (2021)	Real world assistive technology usage survey data	Basis for prioritizing manual spot checks (Section 4.5)
U.S. Access Board (2017)	Section 508 refresh incorporating WCAG by reference	External conformance target the pipeline targets (Section 2.6)
Storybook (2022); Google (2022)	Isolated component environment and aggregate accessibility scoring	Basis for scan placement and rejection of aggregate scores alone (Section 4.2, 6.1)
Nielsen (1994)	Usability heuristics emphasizing status visibility and error prevention	Framing for tooling limitations discussed in Section 6.1

3. PROBLEM STATEMENT

The pipeline proposed in Section 4 responds to four specific, recurring problems observed in Angular component libraries relying on periodic manual accessibility audits. Each problem is stated here in general terms and revisited concretely in the illustrative case study in Section 5.

3.1 Manual Audits Do Not Scale with Release Frequency

A component library that releases weekly or more often produces far more changes between audits than a quarterly or annual manual review can meaningfully cover, which means an audit's findings, by the time they are delivered, describe a version of the library that has already been superseded by several further releases, and any defect introduced after the audit's snapshot was taken remains uncaught until the next audit cycle at the earliest.

3.2 Automated Checkers Catch Only a Subset of WCAG Criteria

Even a comprehensive automated tool such as axe-core, discussed in Section 2.2, is explicit that it can reliably evaluate only a defined subset of WCAG success criteria, since criteria that depend on the actual meaning or context of content, such as whether alternative text accurately describes an image's content, are not mechanically

iJETRM
INTERNATIONAL JOURNAL OF ENGINEERING TECHNOLOGY RESEARCH & MANAGEMENT (IJETRM)
https://ijetrm.com/

checkable without human judgment, which means an automated scan alone will report a clean result on a component that still contains a genuine accessibility defect outside that checkable subset.

3.3 Accessibility Regressions Introduced Silently During Refactors

A component that once passed both automated and manual review can regress during an unrelated refactor, for example a keyboard event handler removed while simplifying a component's internal state management, with no automated signal distinguishing that regression from any other code change unless the specific accessibility behavior that regressed is itself covered by an automated check that runs on every change, not only at release time.

3.4 Assistive Technology Behavior Not Captured by Static Analysis

A static analysis tool evaluates a component's markup and computed accessibility tree, but does not evaluate how an actual assistive technology, a specific screen reader and browser combination, or a specific screen magnification and speech recognition tool, behaves when a real user navigates the rendered component, which means a class of defect that only manifests in that live interaction, such as an announcement that fires but is phrased ambiguously, is invisible to automated tooling regardless of how comprehensive that tooling's rule set is.

Table 2. Summary of the four pipeline problems and where Section 4 addresses each

Problem	Root cause	Addressed in
Manual audits do not scale	Audit cadence far slower than release cadence	Section 4.2, continuous automated scanning
Automated checkers cover only a subset of WCAG	Some criteria require human or contextual judgment	Section 4.5, layered assistive technology spot checks
Silent regressions during refactors	No automated check tied to every code change	Sections 4.4 and 4.6, pattern checks and the continuous integration gate
Real assistive technology behavior uncaptured	Static analysis cannot observe live interaction	Section 4.5 and Section 4.7, spot checks and the regression registry

4. PROPOSED AUTOMATED ACCESSIBILITY COMPLIANCE PIPELINE

4.1 Pipeline Principles

The pipeline rests on four principles that map directly onto the four problems in Section 3. First, every component is scanned automatically on every change, not on a periodic cadence, so that a defect is caught at the point it is introduced rather than at the next scheduled audit. Second, common interaction behavior, focus management and live announcements in particular, is implemented once through shared, tested utilities rather than reimplemented per component, reducing the surface area where a defect can be introduced in the first place. Third, a component's conformance to its documented design pattern is checked explicitly, not only its generic accessibility properties, since a pattern's expected keyboard interaction is itself part of what makes it accessible. Fourth, a smaller, deliberately layered program of manual assistive technology spot checks and a persistent regression registry cover the class of defect the first three principles cannot, by construction, catch.

iJETRM
INTERNATIONAL JOURNAL OF ENGINEERING TECHNOLOGY RESEARCH & MANAGEMENT (IJETRM)
https://ijetrm.com/

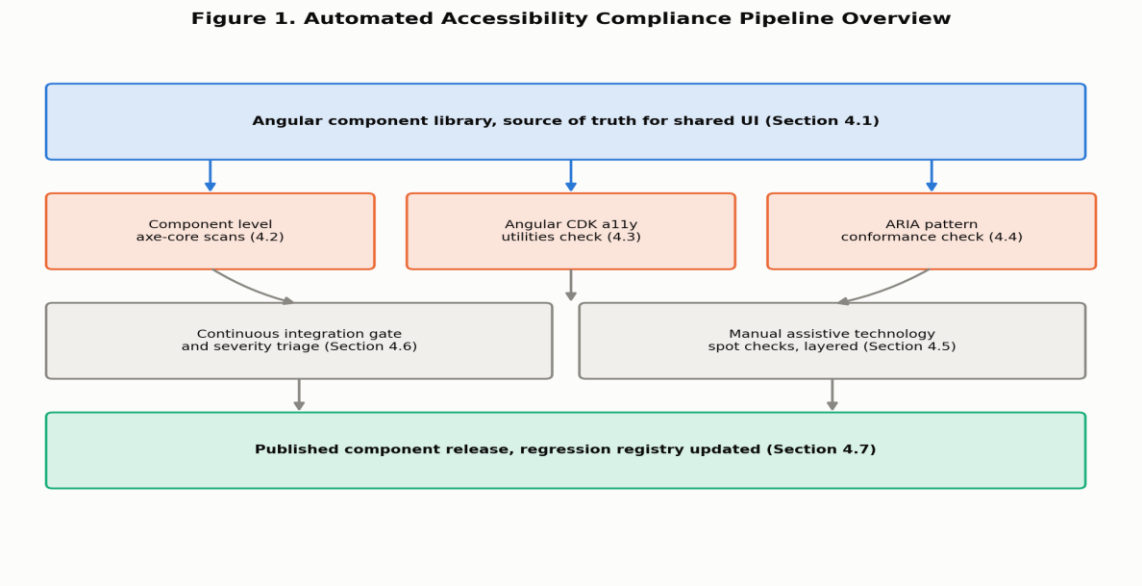

Figure 1. Automated accessibility compliance pipeline overview, from the component library through parallel automated checks to the published release.

4.2 Component Level Automated Scanning with axe-core

Every component in the library is scanned with axe-core in isolation, using the Storybook environment described in Section 2.7, on every commit rather than only at release time, so that a scan failure is attributable to a specific, recent change rather than to an accumulated set of changes since the last audit. Listing 1 shows an illustrative Storybook and axe-core integration for a single component story.

```
// button.stories.ts (illustrative excerpt)
import { Meta, StoryObj } from '@storybook/angular';
import { withA11y } from '@storybook/addon-a11y';
import { ButtonComponent } from './button.component';

const meta: Meta<ButtonComponent> = {
  title: 'Components/Button',
  component: ButtonComponent,
  decorators: [withA11y],
  parameters: {
    a11y: { config: { rules: [{ id: 'color-contrast', enabled: true }] } },
  },
};
export default meta;

export const Primary: StoryObj<ButtonComponent> = {
  args: { label: 'Submit', variant: 'primary' },
};
```

Listing 1. An illustrative Storybook story configured with the accessibility addon, running axe-core against a single component in isolation on every build (Deque Systems, 2022; Storybook, 2022).

4.3 Structured Angular CDK Accessibility Utilities

Rather than each component implementing its own keyboard focus handling and dynamic announcement logic, the pipeline requires components with a focus trapping requirement, a modal dialog for example, or a dynamic content update requirement, a form validation message for example, to use the Angular CDK's FocusTrap and LiveAnnouncer utilities directly, discussed in Section 2.3, so that this behavior is implemented once, tested once, and inherited by every component that uses it rather than reimplemented with variable correctness across the library. Listing 2 shows an illustrative modal component using both utilities.

iJETRM
INTERNATIONAL JOURNAL OF ENGINEERING TECHNOLOGY RESEARCH & MANAGEMENT (IJETRM)
https://ijetrm.com/

```
// modal.component.ts (illustrative excerpt)
import { Component, ElementRef, ViewChild, AfterViewInit } from '@angular/core';
import { FocusTrapFactory, FocusTrap } from '@angular/cdk/a11y';
import { LiveAnnouncer } from '@angular/cdk/a11y';

@Component({ selector: 'app-modal', templateUrl: './modal.component.html' })
export class ModalComponent implements AfterViewInit {
 @ViewChild('modalRoot') modalRoot!: ElementRef;
 private focusTrap!: FocusTrap;

 constructor(
  private focusTrapFactory: FocusTrapFactory,
  private liveAnnouncer: LiveAnnouncer
 ) {}

 ngAfterViewInit(): void {
  this.focusTrap = this.focusTrapFactory.create(this.modalRoot.nativeElement);
  this.focusTrap.focusInitialElementWhenReady();
  this.liveAnnouncer.announce('Dialog opened', 'polite');
 }
}
```

Listing 2. An illustrative modal component using the Angular CDK FocusTrap and LiveAnnouncer utilities rather than a hand rolled equivalent (Angular, 2022).

4.4 ARIA Pattern Conformance Checking

Beyond generic accessibility properties, the pipeline maps each component to one of the documented design patterns discussed in Section 2.4, a combo box, a tab panel, or a modal dialog for example, and runs a check specific to that pattern's expected roles and keyboard interactions rather than only axe-core's generic rule set. Listing 3 shows an illustrative pattern conformance check for a tab panel component.

```
// tabpanel-pattern.spec.ts (illustrative excerpt)
describe('TabPanelComponent ARIA pattern conformance', () => {
 it('exposes tablist, tab, and tabpanel roles', () => {
  const { getByRole, getAllByRole } = render(TabPanelComponent);
  expect(getByRole('tablist')).toBeTruthy();
  expect(getAllByRole('tab').length).toBeGreaterThan(0);
  expect(getByRole('tabpanel')).toBeTruthy();
 });

 it('moves focus with arrow keys per the APG tab pattern', () => {
  const { getAllByRole } = render(TabPanelComponent);
  const tabs = getAllByRole('tab');
  tabs[0].focus();
  fireEvent.keyDown(tabs[0], { key: 'ArrowRight' });
  expect(document.activeElement).toBe(tabs[1]);
 });
});
```

Listing 3. An illustrative pattern conformance test verifying a tab panel component against the WAI-ARIA Authoring Practices tab pattern (W3C WAI, 2022).

4.5 Layered Assistive Technology Spot Checks

Automated scanning and pattern conformance checks in Sections 4.2 and 4.4 are complemented, not replaced, by a quarterly program of manual spot checks conducted with real assistive technology, prioritized using the usage data discussed in Section 2.5, so that the combination the survey identifies as most common among actual users receives coverage even though it cannot be fully automated. The spot check program is deliberately layered, meaning it targets a rotating subset of the library's highest usage components each quarter rather than attempting

iJETRM
INTERNATIONAL JOURNAL OF ENGINEERING TECHNOLOGY RESEARCH & MANAGEMENT (IJETRM)
https://ijetrm.com/

exhaustive manual coverage of every component every cycle, which would not scale any better than the periodic full audit discussed in Section 3.1.

Table 3. Illustrative assistive technology spot check prioritization

Priority tier	Assistive technology and browser combination	Rationale
Tier 1	NVDA with Firefox	Most commonly reported desktop combination in survey data (WebAIM, 2021)
Tier 1	JAWS with Chrome	Second most commonly reported desktop combination
Tier 2	VoiceOver with Safari	Primary combination for macOS and iOS users
Tier 3	TalkBack with Chrome	Primary combination for Android users
Tier 3	Keyboard only, no screen reader	Covers motor impairment and power user navigation patterns

4.6 Continuous Integration Gate and Severity Triage

The automated scans and pattern checks from Sections 4.2 and 4.4 feed into a single continuous integration gate that blocks a pull request on any violation above a defined severity threshold, following a severity triage that distinguishes a blocking defect, one that prevents an assistive technology user from completing a task, from a lower severity finding logged for later attention but not blocking the current change. Listing 4 shows an illustrative continuous integration configuration for the gate.

```
# .github/workflows/a11y-gate.yml (illustrative excerpt)
jobs:
  accessibility-gate:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - name: Build Storybook
        run: npx storybook build
      - name: Run axe-core scans
        run: npx test-storybook --url http://localhost:6006 --testTimeout 15000
      - name: Run ARIA pattern conformance tests
        run: npx jest --config jest.a11y-patterns.config.js
      - name: Triage and gate on severity
        run: node scripts/triage-a11y-results.js --fail-on=blocking
```

Listing 4. An illustrative continuous integration configuration running the automated scanning and pattern checks from Sections 4.2 and 4.4, gating on severity from the triage script (Deque Systems, 2022).

4.7 Regression Registry and Release Tracking

Every accessibility defect found through any of the pipeline's four checking mechanisms, automated scanning, pattern conformance, a spot check, or a defect reported after release, is recorded in a persistent regression registry tagged with the component, the specific criterion or pattern violated, and the release in which it was first found and the release in which it was resolved, so that a later regression of the same defect is recognizable as a regression rather than appearing to be a new, unrelated finding.

iJETRM
INTERNATIONAL JOURNAL OF ENGINEERING TECHNOLOGY RESEARCH & MANAGEMENT (IJETRM)
https://ijetrm.com/

Table 4. Illustrative regression registry fields

Field	Purpose
defectId	Stable identifier, referenced from the originating scan, check, or spot check report
componentId	Component in which the defect was found, following the library's component registry
criterionOrPattern	WCAG success criterion or ARIA pattern violated, per Sections 2.1 and 2.4
firstFoundRelease	Release version in which the defect was first identified
resolvedRelease	Release version in which the defect was resolved, blank if still open
status	Open, scheduled, resolved, or regressed

4.8 Common Pitfalls and Anti Patterns

Teams adopting the pipeline in Section 4 tend to encounter a small, recurring set of pitfalls, each of which quietly reintroduces one of the problems in Section 3 if left unaddressed. This subsection lists the four most common ones observed while designing the illustrative case study in Section 5.

The first pitfall is treating an aggregate accessibility score, discussed in Section 2.7, as sufficient evidence of conformance on its own, without reviewing the itemized rule level results that produced it. A high aggregate score can coexist with a small number of severe, blocking violations, and a team that gates only on the aggregate score rather than on itemized severity, as Section 4.6 recommends, can ship a defect that a more granular gate would have caught.

The second pitfall is a continuous integration gate, described in Section 4.6, configured to warn rather than block on a violation, on the reasoning that a warning is less disruptive to release velocity. This quietly reintroduces the problem in Section 3.3, since a warning a team has learned to routinely dismiss provides no more protection against a silent regression than having no gate at all. Listing 5 contrasts a gate configured to warn against one configured to block.

```
// Anti pattern: gate warns but does not block
- name: Run axe-core scans
  run: npx test-storybook || echo 'Accessibility issues found, continuing anyway'

// Correct: gate blocks the pipeline on a blocking severity violation
- name: Triage and gate on severity
  run: node scripts/triage-a11y-results.js --fail-on=blocking
```

Listing 5. A continuous integration gate configured to warn rather than block, an anti pattern that reintroduces the silent regression problem in Section 3.3 (Section 4.8).

The third pitfall is a spot check program, described in Section 4.5, that always targets the same small set of familiar, well tested components rather than rotating coverage across the library's actual usage distribution, which leaves lower profile but still widely consumed components permanently outside the manual program's reach. A rotation that is not actually enforced tends to drift toward whichever components are easiest to test rather than the ones prioritization in Table 3 would actually select.

The fourth pitfall is a regression registry, described in Section 4.7, that is updated only for defects found through the automated gate and not for defects reported after release by a consuming application team, which leaves the registry blind to precisely the class of defect Section 3.4 identifies as hardest to catch automatically. A registry that only reflects automated findings understates the pipeline's true defect rate and can give a false impression that the manual program in Section 4.5 is unnecessary.

5. ILLUSTRATIVE CASE STUDY

To make the pipeline’s expected effect concrete, this section walks through an illustrative case study for a single Angular component library over twelve releases, with the pipeline’s elements adopted cumulatively in the order presented in Section 4. All figures in this section report simulated indicators constructed for illustration, not measurements from a production library, and should be read as a worked example of the kind of effect the pipeline is designed to produce rather than an empirical result.

5.1 Conformance Rate by Cumulative Adoption Stage

iJETRM
INTERNATIONAL JOURNAL OF ENGINEERING TECHNOLOGY RESEARCH & MANAGEMENT (IJETRM)
https://ijetrm.com/

The first indicator tracks the share of components in the library passing both automated and manual accessibility checks at level WCAG 2.1 AA, at each cumulative adoption stage, starting from a baseline with no automation in place.

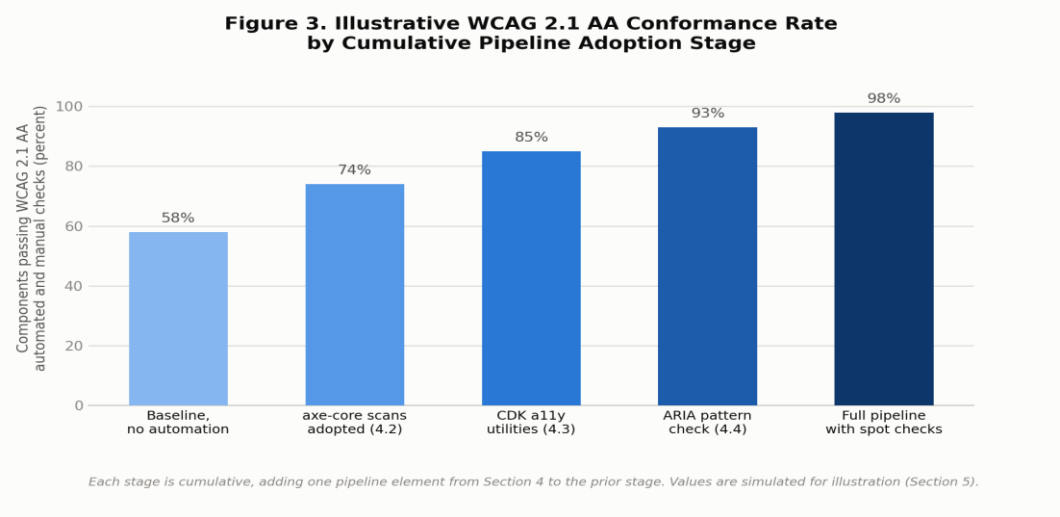

Figure 3. Illustrative WCAG 2.1 AA conformance rate by cumulative pipeline adoption stage.

In the illustrative scenario in Figure 3, adopting axe-core scanning produces the first substantial increase by catching the checkable subset of criteria discussed in Section 2.2 across every component rather than only the subset an infrequent manual audit happened to sample, while adding the CDK utilities, the ARIA pattern check, and finally the layered spot checks each produce further gains by addressing progressively narrower classes of remaining defect.

5.2 Illustrative Post Release Defect Trend

The second indicator tracks accessibility defects reported after a release has shipped, meaning issues a consuming application team or an end user reports that the pipeline's checks did not catch before release, reported per release across the same twelve release period.

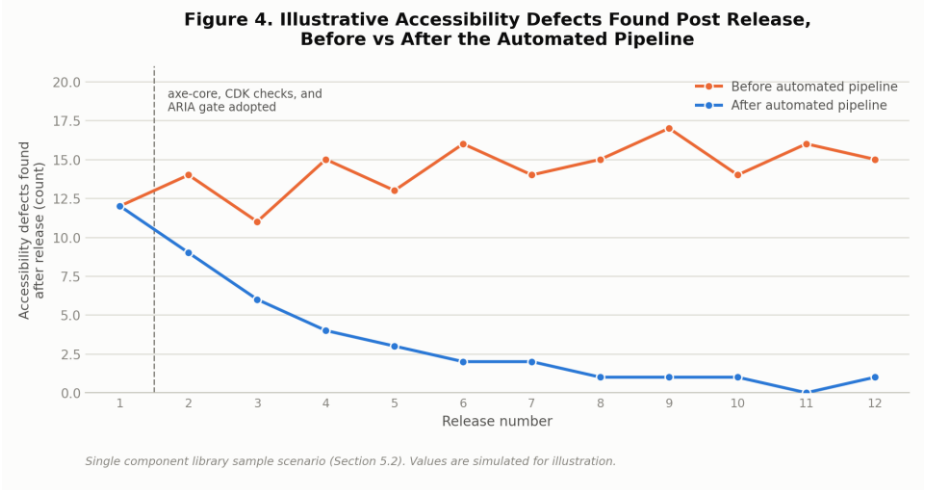

Figure 4. Illustrative accessibility defects found post release, before versus after the automated pipeline.

The illustrative before series in Figure 4 fluctuates in the low to high teens with no clear downward trend, consistent with defects arising from ordinary component development absent any structural prevention, while the after series drops sharply following adoption and settles near zero within a few releases, consistent with the automated gate in Section 4.6 and the layered spot checks in Section 4.5 together catching nearly all defects before a release reaches consuming applications.

iJETRM
INTERNATIONAL JOURNAL OF ENGINEERING TECHNOLOGY RESEARCH & MANAGEMENT (IJETRM)
https://ijetrm.com/

Table 5. Illustrative case study summary by adoption stage

Stage	Pipeline element added (Section 4)	WCAG 2.1 AA conformance, percent (illustrative)	Note
Stage 1	None, baseline	58	No automation, periodic manual audit only
Stage 2	4.2 axe-core scans adopted	74	Checkable WCAG subset caught on every change
Stage 3	4.3 CDK a11y utilities adopted	85	Focus and announcement defects reduced structurally
Stage 4	4.4 ARIA pattern check adopted	93	Pattern specific keyboard behavior verified
Stage 5	4.5 Full pipeline with spot checks	98	Layered manual coverage closes remaining gap

6. RESULTS AND DISCUSSION

6.1 What Automated Tooling Still Misses

Even at full adoption in Table 5, the illustrative conformance rate does not reach one hundred percent, and the remaining gap is instructive about the limits automated tooling has regardless of how much of the pipeline in Section 4 is adopted. Two classes of defect are especially persistent, the accuracy of textual content such as alternative text or an error message's wording, which requires a human judgment axe-core and the ARIA pattern check cannot make, echoing the general point about visibility of system status and clear error messaging in Nielsen's heuristics discussed in Section 2.7 (Nielsen, 1994), and the actual experience of a specific assistive technology combination in practice, which only Section 4.5's spot checks can observe directly.

This gap is not a flaw specific to this article's pipeline, it follows from the more general point in Section 3.2 that automated checkers are explicit about covering only a defined subset of WCAG criteria; the practical implication for a team adopting the pipeline is that reaching a high conformance rate through automation alone should not be read as a signal to reduce the layered spot check program in Section 4.5, since that program is precisely what closes the remaining gap rather than being redundant with the automated stages.

6.2 Cost of Pipeline Overhead at Small Scale

The scanning, structured utilities, pattern checks, spot checks, gate, and registry described in Section 4 are not free to operate, and a small component library with few consuming applications or infrequent releases is unlikely to recover that cost through improved conformance or fewer post release defects at the same rate the illustrative case study in Section 5 reports, since the amplification effect described in Section 1, a defect propagating to many consuming applications, scales with the number of consumers rather than being a fixed property of the library itself.

A useful, if informal, signal for when the overhead in Table 5 is worth paying in full is the number of distinct applications currently consuming the library, since that number is what determines how far a single undetected defect actually propagates; a library still consumed by only one or two applications can reasonably start with axe-core scanning alone, described in Section 4.2, and add the remaining elements as the consumer count and release frequency grow, following a similar staged adoption path to the one Table 5 already illustrates.

6.3 Cost of False Positives and Alert Fatigue

A gate that blocks on every automated finding, as Section 4.6 recommends, is only sustainable if the underlying rule set has a low false positive rate, since a team that repeatedly encounters a blocked pull request for a finding that turns out not to be a genuine defect will eventually lose confidence in the gate and begin routing around it, arriving at the same warn rather than block anti pattern described in Section 4.8 through a different path. The severity triage step in Section 4.6 is partly a defense against this outcome, since a rule that produces a disproportionate share of false positives can be reclassified to a lower severity that logs a finding without blocking,

iJETRM
INTERNATIONAL JOURNAL OF ENGINEERING TECHNOLOGY RESEARCH & MANAGEMENT (IJETRM)
https://ijetrm.com/

while the itemized, rule level review recommended in Section 6.1 gives a team the visibility it needs to identify which specific rules are producing that disproportionate share in the first place.
The illustrative case study in Section 5 does not model false positive rates directly, but a team applying the pipeline in practice should expect to spend some ongoing effort tuning the rule set described in Listing 1, particularly in the months immediately following initial adoption, before the gate's blocking behavior and the team's trust in that behavior converge. Table 6 sketches an illustrative rule tuning cadence a team might follow during that period.

Table 6. Illustrative rule tuning cadence during initial pipeline adoption

Period after adoption	Typical activity	Owner
Weeks 1 to 4	Review every gate failure to confirm it reflects a genuine defect, not a false positive	Accessibility specialist or designated reviewer
Weeks 5 to 12	Reclassify any rule with a recurring false positive pattern to a lower, non blocking severity	Accessibility specialist, following Section 4.6
Quarter 2 onward	Periodic audit of severity classifications against Table 4's regression data	Engineering lead

7. LIMITATIONS

The case study in Section 5 uses simulated indicators constructed for illustration and does not report measurements from a production component library, so the specific conformance and defect figures shown in Figures 3 and 4 should be read as illustrative of the kind of effect the pipeline is designed to produce rather than as an empirical claim. The pipeline also assumes a Storybook or equivalent isolated component development environment is already part of the team's workflow, as discussed in Section 2.7, and assumes the Angular CDK's accessibility utilities discussed in Section 2.3 are compatible with the library's existing component architecture, an assumption that would need revisiting for a library built on an older Angular version predating the CDK's current accessibility module. Finally, the spot check prioritization in Table 3 is based on published survey data rather than the specific consuming applications' actual user population, which a library serving a substantially different user population, for example one with a higher proportion of users relying on switch access devices, would need to adjust for directly.

8. FUTURE WORK

Three directions follow from the limitations in Section 7. First, an empirical study measuring the pipeline's effect on a production component library, rather than through simulated indicators, would test whether the illustrative conformance and defect trends in Section 5 hold in practice and at what adoption cost. Second, extending the pattern conformance checks in Section 4.4 to a broader set of the documented patterns in the WAI-ARIA Authoring Practices Guide, beyond the tab panel example in Listing 3, would reduce the class of pattern specific defect not yet covered by an explicit check.
Third, the informal small scale guidance offered in Section 6.2 would benefit from a more rigorous treatment, for example a decision framework that takes a library's consumer count, release frequency, and observed defect history as input and recommends which of the pipeline's elements to adopt first, rather than presenting the full pipeline as a single all or nothing decision, giving the checklist in Appendix A a natural sequence to follow rather than a flat list to work through in one pass.
Fourth, the rule tuning cadence sketched in Section 6.3 was presented informally rather than derived from measured false positive rates for specific axe-core rules; a follow up study cataloging which rules most commonly produce false positives against Angular specific markup patterns, as opposed to markup patterns generic to the web platform, would let a team skip much of the weeks one to four review period in Table 6 by starting from an Angular specific severity classification rather than deriving one from scratch.

9. CONCLUSION

This article presented an automated accessibility compliance pipeline for Angular component libraries, built around four coordinated elements, continuous component level scanning with axe-core, structured use of Angular CDK accessibility utilities, an explicit ARIA pattern conformance check, and a layered program of manual assistive technology spot checks backed by a continuous integration gate and a persistent regression registry. An illustrative case study for one component library suggested the combined elements can substantially improve WCAG 2.1 AA conformance and reduce post release accessibility defects, though these results are illustrative

iJETRM
INTERNATIONAL JOURNAL OF ENGINEERING TECHNOLOGY RESEARCH & MANAGEMENT (IJETRM)
https://ijetrm.com/

rather than empirical. The pipeline's core structural idea, that a defined, checkable subset of accessibility conformance can be automated continuously while a smaller, deliberately targeted manual program covers what automation genuinely cannot, is not specific to Angular or to component libraries, and Section 8's proposed extensions suggest it generalizes to other shared frontend infrastructure where a single defect propagates broadly to many consumers.

Appendix A: Accessibility Pipeline Adoption Checklist

This checklist summarizes the practical steps a team can use when adopting the pipeline described in Section 4, organized by the same four elements from Section 4.1. It is offered as a starting point rather than an exhaustive audit.

A.1 Automated scanning (Section 4.2)

1. Confirm every component has a Storybook story configured with the accessibility addon, following Listing 1.
2. Confirm the scan runs on every commit, not only at release time, so a defect is attributable to a specific recent change.
3. Review the axe-core rule set periodically against Section 2.1's four principles to confirm coverage has not drifted.
4. Confirm scan results are visible in the pull request itself, not only in a separate dashboard reviewers may not check.
5. Pilot the scan on a small number of components before rolling it out library wide, and resolve any pre existing violations surfaced by the pilot before enforcing the gate in Section 4.6.

A.2 Structured CDK utilities (Section 4.3)

1. Identify every component with a focus trapping or dynamic announcement requirement, following the examples in Listing 2.
2. Confirm those components use the Angular CDK's FocusTrap and LiveAnnouncer utilities rather than a hand rolled equivalent.
3. Audit existing components for hand rolled focus or announcement logic that should be migrated to the shared utilities.
4. Document the expected utility usage pattern for each component archetype, a modal, a form, a live status region, in the library's own contribution guidelines.
5. Confirm the CDK version in use matches the version the accessibility utilities were verified against.

A.3 Pattern conformance checks (Sections 4.4, 4.6)

1. Map every interactive component to its corresponding WAI-ARIA Authoring Practices pattern, following Table 3's tier logic where relevant.
2. Write a pattern specific conformance test for each mapped component, following the structure in Listing 3.
3. Confirm the continuous integration gate blocks on a blocking severity pattern violation, following the correction in Listing 5.
4. Add a documented exception process for a gate failure that reflects a deliberate, reviewed deviation from the standard pattern.
5. Audit pattern coverage periodically to identify components not yet mapped to an explicit pattern check.

A.4 Spot checks and regression tracking (Sections 4.5, 4.7)

1. Confirm the quarterly spot check rotation actually covers the highest usage components, following the prioritization in Table 3, rather than defaulting to the most familiar ones.
2. Confirm every defect found through any channel, automated, pattern check, spot check, or post release report, is recorded in the registry described in Table 4.
3. Review open registry entries at the start of each release cycle to confirm none have silently stalled.
4. Track the conformance and defect indicators in Table 5 each release to confirm the pipeline is holding, not only at initial adoption.
5. Revisit the small scale guidance in Section 6.2 whenever the library's consumer count changes materially.

iJETRM
INTERNATIONAL JOURNAL OF ENGINEERING TECHNOLOGY RESEARCH & MANAGEMENT (IJETRM)
https://ijetrm.com/

Appendix B: Illustrative Defect Lifecycle Example

This appendix works through a single illustrative accessibility defect end to end, from initial detection through triage to its eventual resolution, to make the abstract description in Section 4 concrete.

Table 7. Illustrative defect, from initial detection to resolution

Stage	Illustrative content
Initial detection (Section 4.2)	axe-core scan flags a data table component for missing a table header association, WCAG criterion 1.3.1
Severity triage (Section 4.6)	Classified as blocking, since a screen reader user cannot associate a cell with its column header
Regression registry entry (Section 4.7)	Recorded with componentId, criterion 1.3.1, and firstFoundRelease set to the current release
Resolution	Header association added using scope attributes, verified against the pattern check in Section 4.4
Outcome	Registry status updated to resolved, resolvedRelease recorded, defect remains visible for future regression comparison

Listing 6 shows the illustrative defect as it would appear in the regression registry from Table 4, both before and after resolution.

```
// Illustrative regression registry entry, following the fields in Table 4,
// before resolution
{
 defectId: 'a11y_2025_0142',
 componentId: 'data-table',
 criterionOrPattern: 'WCAG 1.3.1',
 firstFoundRelease: 'v4.7.0',
 resolvedRelease: null,
 status: 'open',
}

// After resolution
{
 defectId: 'a11y_2025_0142',
 componentId: 'data-table',
 criterionOrPattern: 'WCAG 1.3.1',
 firstFoundRelease: 'v4.7.0',
 resolvedRelease: 'v4.7.2',
 status: 'resolved',
}
```

Listing 6. An illustrative regression registry entry worked through end to end, before and after resolution (Sections 4.2, 4.6, 4.7).

Appendix C: Illustrative Continuous Integration Gate Severity Matrix

Table 2's problem summary and Section 4.6's gate description both refer to a severity threshold without stating, for a given class of finding, which severity level applies and what action follows. This appendix sketches that assignment explicitly for the four checking mechanisms described in Section 4, as a starting point for a team adapting the severity matrix to its own component library.

iJETRM
INTERNATIONAL JOURNAL OF ENGINEERING TECHNOLOGY RESEARCH & MANAGEMENT (IJETRM)
https://ijetrm.com/

Table 8. Illustrative severity matrix by finding source

Finding source (Section 4)	Example finding	Severity	Gate action
axe-core scan (4.2)	Interactive control missing an accessible name	Blocking	Pull request blocked until resolved
axe-core scan (4.2)	Minor color contrast shortfall on a disabled state	Should fix	Logged in registry, does not block
ARIA pattern check (4.4)	Tab panel does not respond to arrow key navigation	Blocking	Pull request blocked until resolved
CDK utility audit (4.3)	Modal opens without trapping keyboard focus	Blocking	Pull request blocked until resolved
Spot check (4.5)	Announcement wording ambiguous to screen reader users	Should fix	Logged in registry, scheduled for next release

A team that lacks a distinct blocking versus should fix distinction in its existing tooling should not treat that absence as permission to gate on every finding equally, but should instead adopt at least a two level severity scheme along the lines of Table 8, following the false positive guidance in Section 6.3, so that a genuinely blocking defect and a lower priority improvement are not competing for the same urgency.

Statements and Declarations

Funding. This work received no specific grant from any funding agency in the public, commercial, or not for profit sectors. [Funding statement placeholder, to be completed with actual funding information prior to submission.]

Conflicts of interest. The author declares no known competing financial interests or personal relationships that could have appeared to influence the work reported in this article. [Conflicts of interest statement placeholder.]

Data availability. All numerical results presented in Section 5 and Section 6 are simulated for illustration and are not derived from a production dataset. [Data availability statement placeholder, to be completed if empirical data is added in a future revision.]

Author contributions. [Author contributions placeholder, to be completed prior to submission, for example following the CRediT taxonomy for conceptualization, methodology, and writing.]

Acknowledgments. [Acknowledgments placeholder for colleagues, reviewers, or institutions who supported this work but do not meet authorship criteria.]

REFERENCES

1) Angular. (2022). Accessibility. Angular documentation. https://angular.io/guide/accessibility
2) Deque Systems. (2022). axe-core Documentation. https://www.deque.com/axe/core-documentation/
3) Google. (2022). Lighthouse Accessibility Scoring. Chrome Developers documentation. https://developer.chrome.com/docs/lighthouse/accessibility/scoring
4) Nielsen, J. (1994). 10 Usability Heuristics for User Interface Design. Nielsen Norman Group. https://www.nngroup.com/articles/ten-usability-heuristics/
5) Storybook. (2022). Storybook Documentation. https://storybook.js.org/docs
6) U.S. Access Board. (2017). Section 508 Refresh: Information and Communication Technology (ICT) Final Rule. https://www.access-board.gov/ict/
7) W3C. (2017). Accessible Rich Internet Applications (WAI-ARIA) 1.1. https://www.w3.org/TR/wai-aria-1.1/
8) W3C. (2018). Web Content Accessibility Guidelines (WCAG) 2.1. https://www.w3.org/TR/WCAG21/
9) W3C WAI. (2022). WAI-ARIA Authoring Practices Guide (APG), Version 1.2. https://www.w3.org/TR/2022/NOTE-wai-aria-practices-1.2-20220519
10) WebAIM. (2021). Screen Reader User Survey 9 Results. https://webaim.org/projects/screenreadersurvey9/

