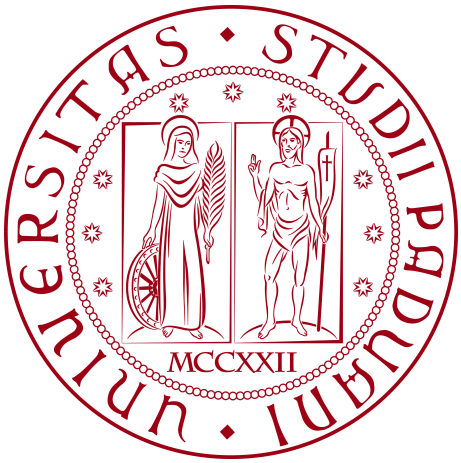
University of Padua

Department of Mathematics "Tullio Levi-Civita"

Master Degree in Computer Science



AccessibleHub - App documentation

Technical Appendix to Master's Thesis

Supervisor

Prof. Ombretta Gaggi

Candidate

Gabriel Rovesti

ID Number: 2103389

Academic Year 2024-2025

Abstract

Technical doc abstract

Table of contents

List of figures

List of tables

List of listings

Acronyms and abbreviations

API Application Programming Interface. i

ARIA Accessible Rich Internet Applications. i

LOC Lines of Code. i, 15

MCAG Mobile Content Accessibility Guidelines. i

UI User Interface. i

UX User Experience. i

W3C World Wide Web Consortium. i

WCAG Web Content Accessibility Guidelines. i, 2

Glossary

Application Programming Interface An Application Programming Interface (API) is a set of protocols, routines, and tools for building software applications. It specifies how software components should interact, allowing different software systems to communicate with each other. APIs define the methods and data structures that developers can use to interact with a system, service, or library without needing to understand the underlying implementation. They serve as a contract between different software components, enabling developers to integrate different systems, access web-based services, and create more complex and interconnected software solutions. i

ARIA Accessible Rich Internet Applications (ARIA) is a set of attributes that define ways to make web content and web applications more accessible to people with disabilities. ARIA roles, states, and properties help assistive technologies understand and interact with dynamic content and complex user interface controls. i

Flutter Flutter is an open-source UI software development kit created by Google, designed for building natively compiled applications for mobile, web, and desktop platforms from a single codebase. Launched in 2017, Flutter uses the Dart programming language and provides a comprehensive framework for creating high-performance, visually attractive applications with a focus on smooth, responsive user interfaces. Unlike traditional cross-platform frameworks that use web view rendering, Flutter compiles directly to native code, enabling near-native performance. Its key features include a rich set of pre-designed widgets, hot reload for rapid development, extensive customization capabilities, and a robust ecosystem that supports complex application development across multiple platforms. i, 1

Gray Literature Review A structured method of collecting and analyzing non-traditional

published literature, much of which is published outside conventional academic channels. This research methodology concerns conducting a review of gray literature, such as technical reports, blog postings, professional forums, and industry documentation, to gain insight from practical experience. Gray literature reviews apply most to software engineering research as they represent real practices, challenges, and solutions that have taken place during implementation that may not have been captured or documented in the academic literature. This methodology acts like a bridge that closes the gap between theoretical research and its industry application. i

Lines of Code A metric used to quantify the size or complexity of a software program by counting the number of lines in its source code. LOC is often used as an indicator of development effort, code maintenance, and project scale.. i

MCAG Mobile Content Accessibility Guidelines (MCAG) are a specialized set of accessibility recommendations specifically tailored to mobile application and mobile web content. While building upon the foundational principles of WCAG, MCAG addresses unique challenges of mobile interfaces, such as touch interactions, small screen sizes, diverse input methods, and mobile-specific assistive technologies. These guidelines provide specific considerations for creating accessible content and interfaces on smartphones, tablets, and other mobile devices, taking into account the distinct interaction patterns and technological constraints of mobile platforms. i

React Native React Native is an open-source mobile application development framework created by Facebook (now Meta) that allows developers to build mobile applications using JavaScript and React. Introduced in 2015, React Native enables developers to create native mobile apps for both iOS and Android platforms using a single codebase, leveraging the popular React web development library. Unlike hybrid app frameworks, React Native renders components using actual native platform UI elements, providing a more authentic user experience and better performance. The framework bridges the gap between web and mobile development, allowing web developers to create mobile

applications using familiar JavaScript and React paradigms, while still achieving near-native performance and user interface responsiveness. i, 1

Screen Reader A screen reader is an assistive technology software that enables people with visual impairments or reading disabilities to interact with digital devices by converting on-screen text and elements into synthesized speech or Braille output. Screen readers navigate through user interfaces, reading text, describing graphical elements, and providing auditory feedback about the computer or mobile device's content and functionality. They interpret and verbalize user interface elements, buttons, menus, and other interactive components, allowing visually impaired users to understand and interact with digital content. Popular screen readers include VoiceOver for Apple devices, TalkBack for Android, and NVDA and JAWS for desktop computers. i

TalkBack TalkBack is a screen reader developed by Google for Android devices. It provides spoken feedback and vibration to help visually impaired users navigate their devices and interact with apps. i

User Interface The User Interface refers to the space where interactions between humans and machines occur. It includes the design and arrangement of graphical elements (such as buttons, icons, and menus) that enable users to interact with software or hardware systems. The goal of a UI is to make the user's interaction simple and efficient in accomplishing tasks within a system. i

User Experience User Experience encompasses the overall experience a user has while interacting with a product or service. It includes not only usability and interface design but also the emotional response, satisfaction, and ease of use a person feels while using a system. UX design focuses on optimizing a product's interaction to provide meaningful and relevant experiences to users, ensuring that the system is intuitive, efficient, and enjoyable to use. i

VoiceOver VoiceOver is a screen reader built into Apple's macOS and iOS operating systems. It provides spoken descriptions of on-screen elements and allows users to navigate

and interact with their devices using gestures and keyboard commands. i

W3C The World Wide Web Consortium (W3C) is an international community that develops open standards to ensure the long-term growth and evolution of the web. Founded by Tim Berners-Lee in 1994, the W3C works to create universal web standards that promote interoperability and accessibility across different platforms, browsers, and devices. This non-profit organization brings together technology experts, researchers, and industry leaders to develop guidelines and protocols that form the fundamental architecture of the World Wide Web. Key contributions include HTML, CSS, accessibility guidelines (WCAG), and web standards that ensure a consistent, inclusive, and innovative web experience for users worldwide. i

WCAG The Web Content Accessibility Guidelines (WCAG) are a set of recommendations for making web content more accessible to people with disabilities. They provide a wide range of recommendations for making web content more accessible, including guidelines for text, images, sound, and more. i, 1

Chapter 1

AccessibleHub: Transforming mobile accessibility guidelines into code

This chapter introduces an accessibility learning toolkit for mobile developers. Building upon prior research, it provides a practical guide to implementing accessible mobile applications, particularly in Flutter. *AccessibleHub*, a *React Native*$_G$ toolkit, offers interactive examples and component-level guidance, comparing React Native and *Flutter*$_G$. Grounded in *WCAG*$_G$ principles, AccessibleHub aims to bridge the gap between accessibility guidelines and real-world application.

1.1 Introduction

Fancy intro to properly introduce the doc

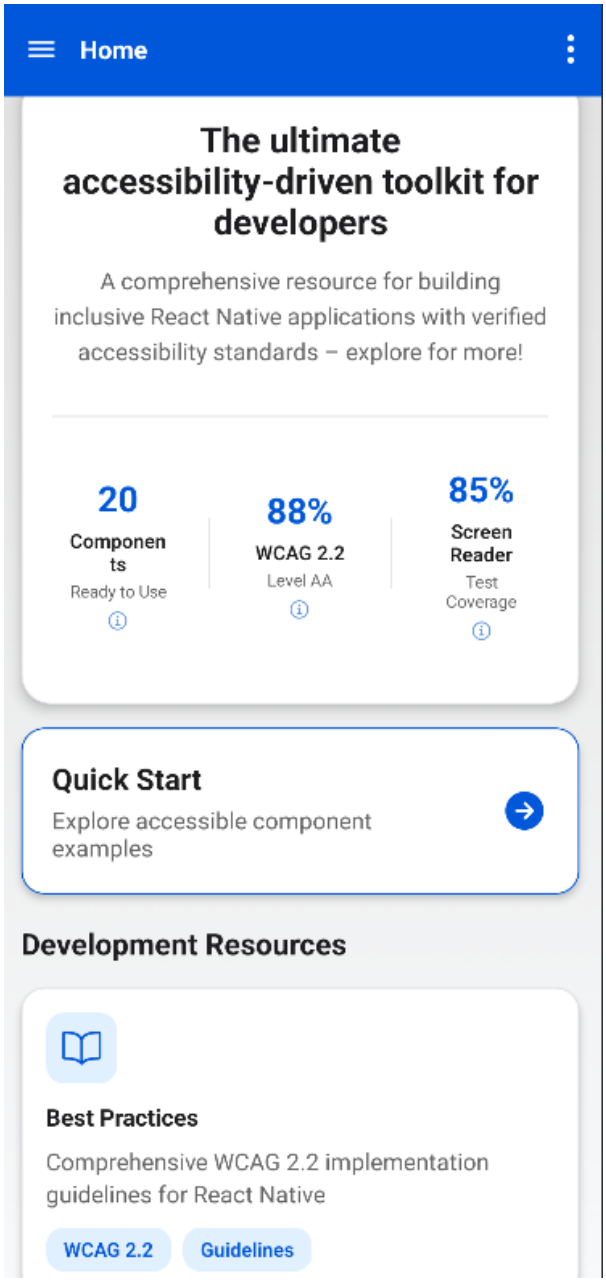
1.2 Accessibility implementation guidelines

Having established the overall architecture of *AccessibleHub* and the guiding principles from both *WCAG* and *MCAG*, we now present a screen-by-screen analysis. Each subsection highlights the key *success criteria* addressed, references relevant *mobile-specific considerations*, and demonstrates practical solutions in React Native. Where applicable, we contrast these with Flutter's approach, building upon the insights from Gaggi and Perinello's approach [4] analazying Budai's Flutter code - following guidelines and then giving advice into

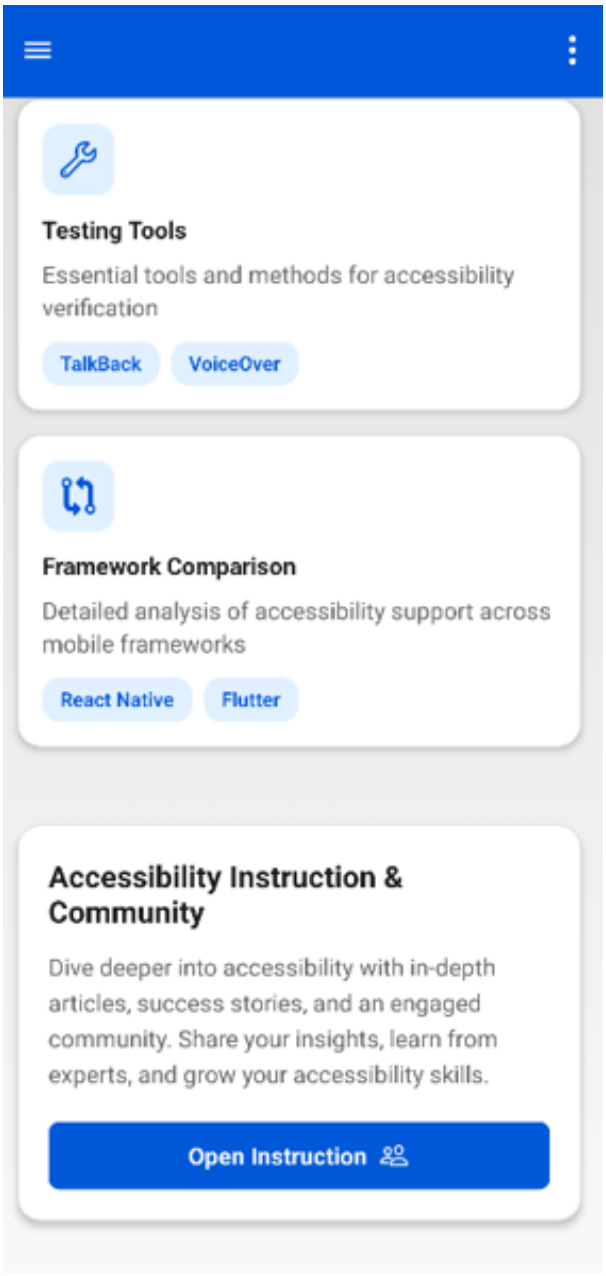
introducing new ones. These screens follow the structure presented in Section **??**, analyzed both as main screens and sections.

1.2.1 Home screen

The Home screen serves as the primary entry point of the *AccessibleHub* application. It provides key metrics on accessibility compliance (e.g., number of accessible components, $WCAG_G$ conformance level) and direct navigation to sections: *Accessible Components* (Quick Start), *Best Practices*, *Testing Tools*, and the *Framework Comparison*. A screenshot of the interface is shown in Figure 1.1.



(a) Home screen - Part 1



(b) Home screen - Part 2

Figure 1.1: Side-by-side view of the two Home sections, with metrics and navigation buttons

1.2.1.1 Component inventory and WCAG/MCAG mapping

Table 1.1 provides a formal mapping between the UI components, their semantic roles, the specific WCAG 2.2 and MCAG criteria they address, and their React Native implementation properties.

Table 1.1: Home screen component-criteria mapping

Component	**Semantic Role**	**WCAG 2.2 Criteria**	**MCAG Considerations**	**Implementation Properties**
Hero Title	heading	1.4.3 Contrast (AA) 2.4.6 Headings (AA)	Text readability on variable screen sizes	`accessibilityRole="header"`
Stats Cards	button	1.4.3 Contrast (AA) 2.5.8 Target Size (AA) 4.1.2 Name, Role, Value (A)	Touch target size	`accessibilityRole="button" accessibilityLabel="${value}% ${type}, tap for details"`
Decorative Icons	none	1.1.1 Non-text Content (A)	Reduction of unnecessary focus stops	`accessibilityElementsHidden=true importantForAccessibility="no"`
Quick Start Button	button	1.4.3 Contrast (AA) 2.5.8 Target Size (AA) 2.5.2 Pointer Cancellation (A)	One-handed operation	`accessibilityRole="button", minHeight: 48 minWidth: 150`

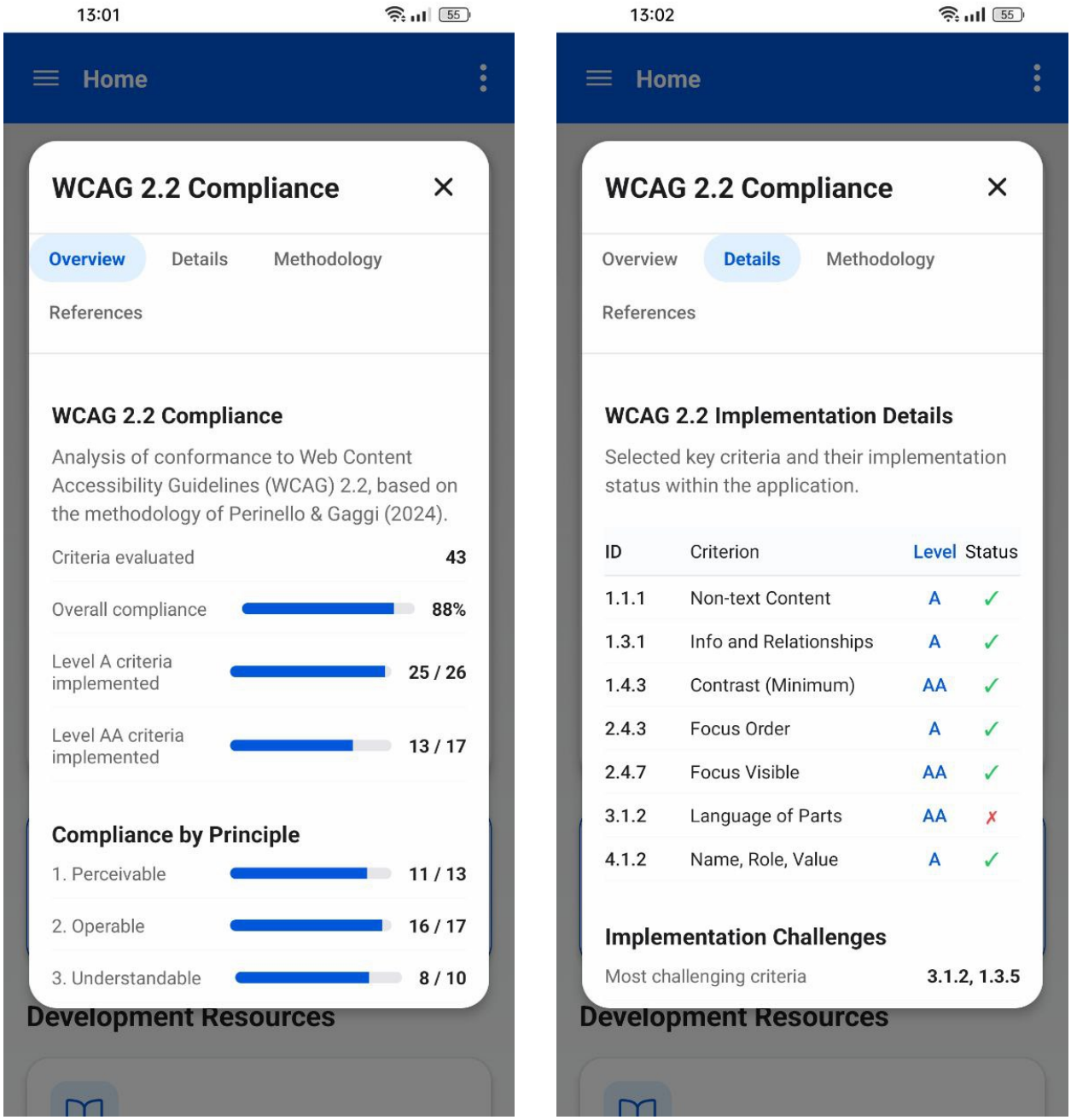
Continued on next page

Table 1.1 – continued from previous page

Component	Semantic Role	WCAG 2.2 Criteria	MCAG Considerations	Implementation Properties
Feature Cards	button	1.3.1 Info and Relationships (A) 1.4.3 Contrast (AA) 2.5.8 Target Size (AA)	Logical grouping	`accessibilityRole="button"`, `accessibilityLabel="${title}"`, `accessibilityHint="${hint}"`
Modal Dialog	dialog	2.4.3 Focus Order (A) 4.1.2 Name, Role, Value (A)	Keyboard trap prevention	`accessibilityRole="dialog"`, Focus management implementation
Modal Tabs	tablist	2.4.7 Focus Visible (AA) 4.1.2 Name, Role, Value (A)	Touch interaction	`accessibilityRole="tablist"`, `accessibility State={{ selected: isActive }}`

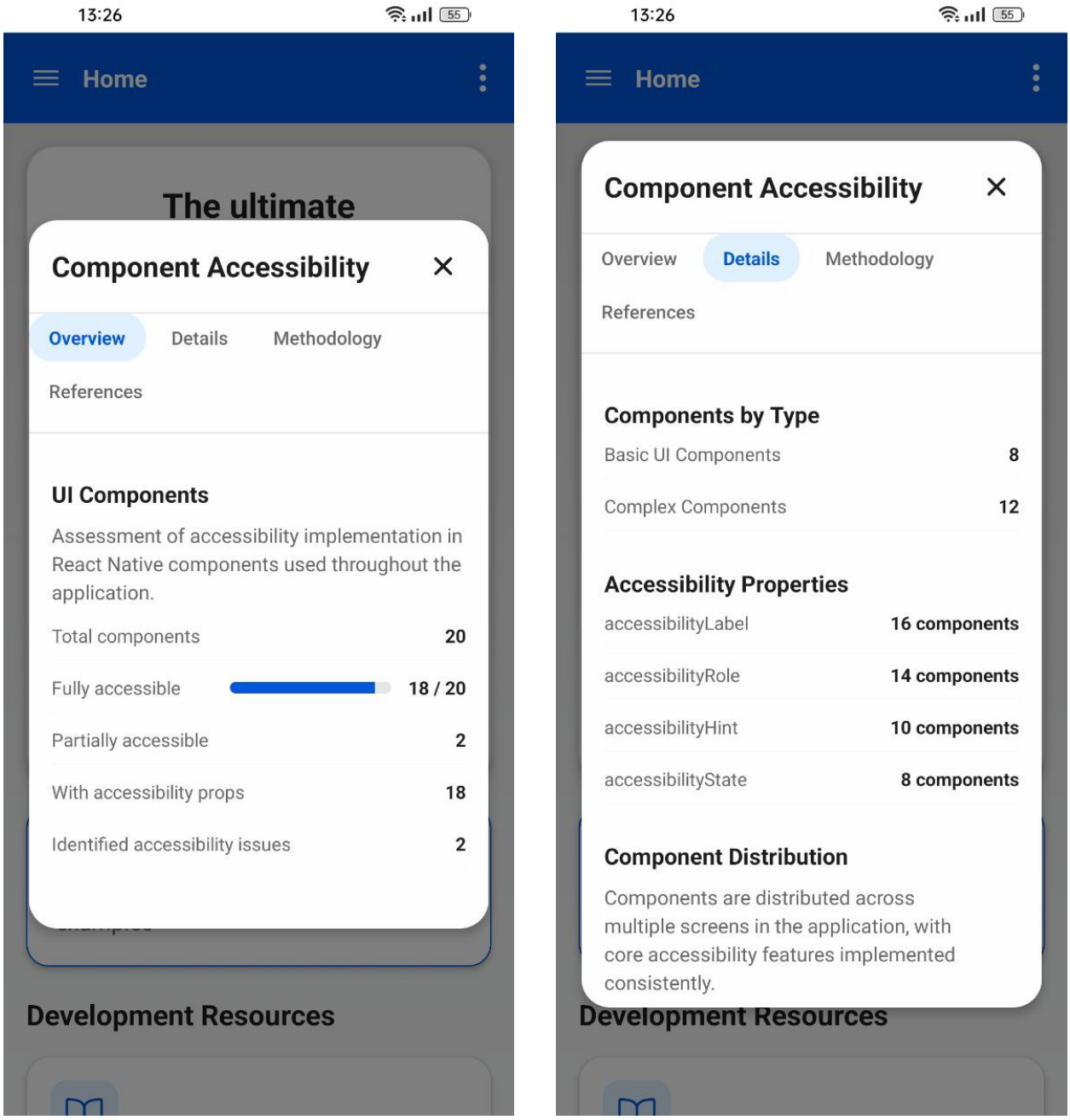
1.2.1.2 Formal metrics calculation methodology

The Home screen displays three key metrics that provide quantitative measurements of the application's accessibility. These metrics are not arbitrary but are calculated using a formal methodology defined in the `calculateAccessibilityScore` function within `index.tsx`.



(a) WCAG compliance overview **(b)** WCAG compliance details

Figure 1.2: Modal dialogs showing WCAG compliance metrics

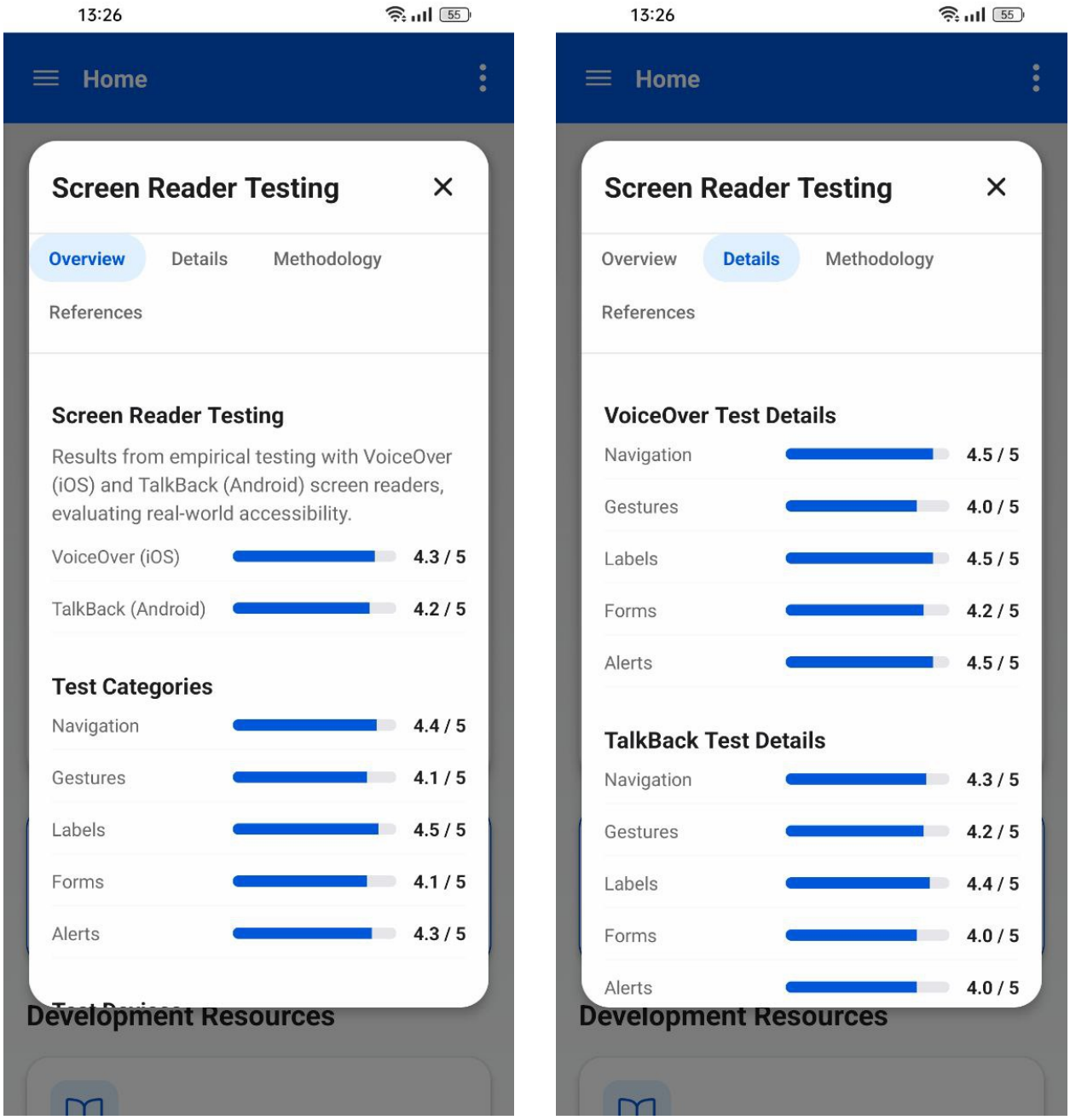


(a) Component metrics overview **(b)** Component metrics details

Figure 1.3: Modal dialogs showing component accessibility metrics

1.2.1.2.1 Component Accessibility Score The Component Accessibility Score is calculated using the following formula:

$$\text{ComponentScore} = \left(\frac{\text{AccessibleComponents}}{\text{TotalComponents}} \right) \times 100 \tag{1.1}$$

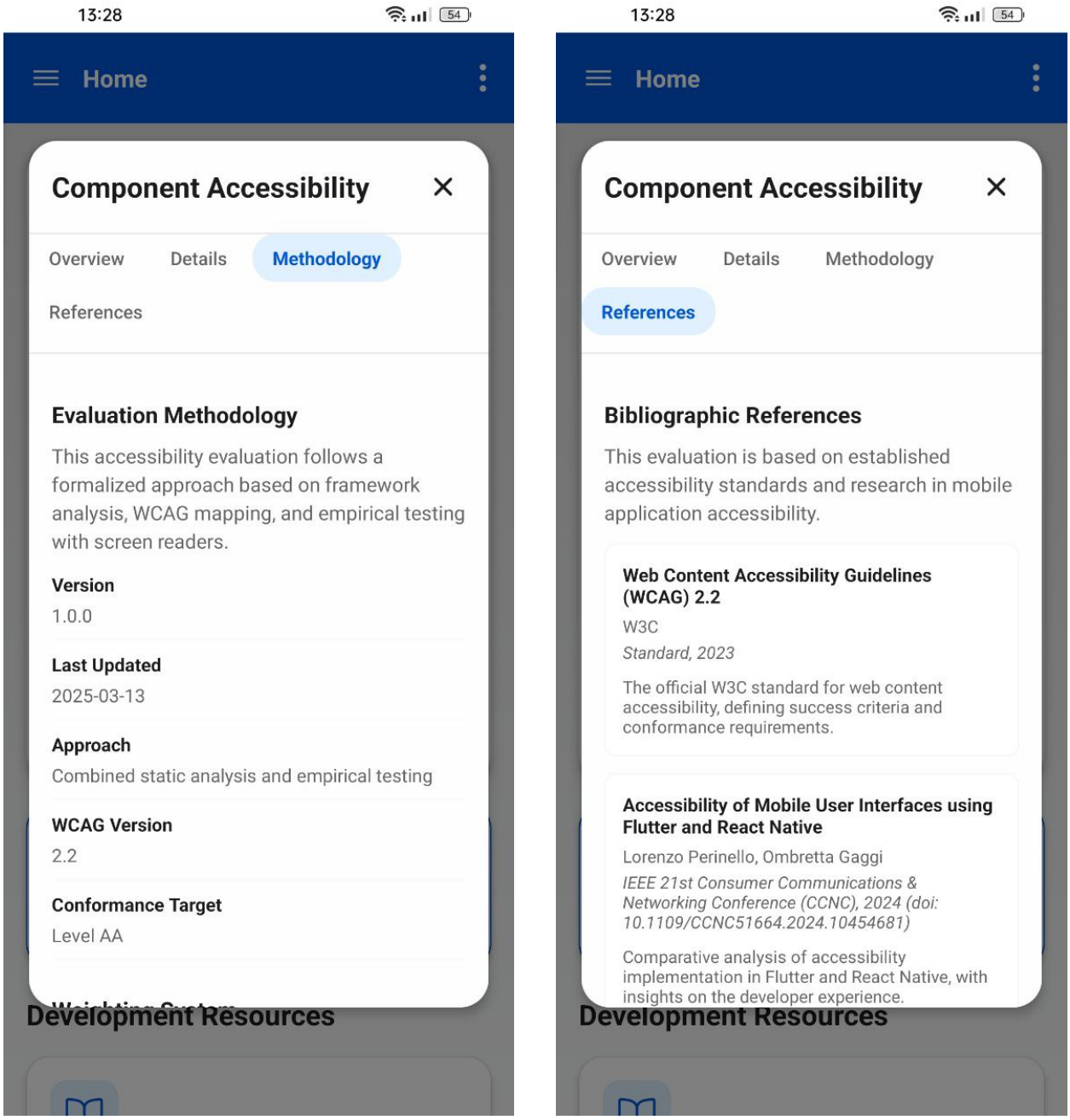


(a) Screen reader testing overview **(b)** Screen reader testing details

Figure 1.4: Modal dialogs showing screen reader testing metrics

Where:

- `AccessibleComponents` = Number of components with properly implemented accessibility attributes (18)



(a) Methodology explanation **(b)** References documentation

Figure 1.5: Modal dialogs showing methodology and references

- `TotalComponents` = Total number of UI components used in the application (20)

The implementation in `index.tsx` maintains a formal registry of all UI components, as shown by 1.1.

```
// Component registry with accessibility status tracking
const componentsRegistry = {
  'button': { implemented: true, accessible: true, screens: ['home',
     'gestures'] },
  'text': { implemented: true, accessible: true, screens: ['home',
     'guidelines'] },
  // ... other components
  'tooltip': { implemented: true, accessible: false, screens: [] },
  // Total: 20 components, 18 fully accessible
};

// Component calculation
const componentsTotal = Object.keys(componentsRegistry).length;
const accessibleComponents = Object.values(componentsRegistry)
  .filter(c => c.implemented && c.accessible).length;
const componentScore = Math.round((accessibleComponents /
   componentsTotal) * 100);
```

Listing 1.1: Component registry and calculation

1.2.1.2.2 WCAG Compliance Score

The WCAG Compliance Score represents the percentage of implemented WCAG 2.2 success criteria across four principles:

$$\text{WCAGCompliance} = \left(\frac{\text{CriteriaLevelAMet} + \text{CriteriaLevelAAMet}}{\text{TotalCriteria}} \right) \times 100 \tag{1.2}$$

Where:

- `CriteriaLevelAMet` = Number of Level A success criteria implemented (25)
- `CriteriaLevelAAMet` = Number of Level AA success criteria implemented (13)
- `TotalCriteria` = Total applicable WCAG criteria (43)

The implementation maintains a comprehensive tracking system for WCAG criteria, as shown by 1.2.

1.2.1.2.3 Screen Reader Testing Score

The Screen Reader Testing Score represents empirical testing with VoiceOver (iOS) and TalkBack (Android):

$$\text{TestingScore} = \left(\frac{\text{VoiceOverAvg} + \text{TalkBackAvg}}{2} \right) \times 20 \tag{1.3}$$

```
// WCAG criteria tracking with implementation status
const wcagCriteria = {
  '1.1.1': { level: 'A', implemented: true, name: "Non-text Content" },
  '1.3.1': { level: 'A', implemented: true, name: "Info and Relationships" },
  // ... other criteria
  '4.1.3': { level: 'AA', implemented: true, name: "Status Messages" },
};

// WCAG compliance calculation
const criteriaValues = Object.values(wcagCriteria);
const totalCriteria = criteriaValues.length;
const levelACriteriaMet = criteriaValues
  .filter(c => c.level === 'A' && c.implemented).length;
const levelAACriteriaMet = criteriaValues
  .filter(c => c.level === 'AA' && c.implemented).length;
const wcagCompliance = Math.round(
  ((levelACriteriaMet + levelAACriteriaMet) / totalCriteria) * 100
);
```

Listing 1.2: WCAG criteria tracking and calculation

Where:

- `VoiceOverAvg` = Average score from VoiceOver testing across categories (4.34/5)
- `TalkBackAvg` = Average score from TalkBack testing across categories (4.18/5)

The scores are based on structured testing of five key aspects as shown by 1.3.

1.2.1.2.4 Overall Accessibility Score The overall Accessibility Score is calculated using weighted components:

$$\text{OverallScore} = (\text{ComponentScore} \times 0.4) + (\text{WCAGCompliance} \times 0.4) + (\text{TestingScore} \times 0.2) \tag{1.4}$$

This weighting system gives equal importance to component implementation and standards compliance (40% each), with empirical testing contributing 20% to the final score.

```
// Screen reader test results from empirical testing
const screenReaderTests = {
  voiceOver: { // iOS
    navigation: 4.5, // Logical navigation flow
    gestures: 4.0,   // Gesture recognition
    labels: 4.5,     // Label clarity and completeness
    forms: 4.2,      // Form control accessibility
    alerts: 4.5      // Alert and dialog accessibility
  },
  talkBack: { // Android
    navigation: 4.3,
    gestures: 4.2,
    labels: 4.4,
    forms: 4.0,
    alerts: 4.0
  }
};

// Testing score calculation
const voiceOverScores = Object.values(screenReaderTests.voiceOver);
const talkBackScores = Object.values(screenReaderTests.talkBack);
const voiceOverAvg = voiceOverScores.reduce((sum, score) =>
  sum + score, 0) / voiceOverScores.length;
const talkBackAvg = talkBackScores.reduce((sum, score) =>
  sum + score, 0) / talkBackScores.length;
const testingScore = Math.round(((voiceOverAvg + talkBackAvg) / 2) *
   20);
```

Listing 1.3: Screen reader testing results and calculation

1.2.1.3 Technical implementation analysis

The code sample present in 1.4 shows the key accessibility properties implemented in the Home screen.

```
// 1. ScrollView container with proper role and label
<ScrollView
  accessibilityRole="scrollview"
  accessibilityLabel="AccessibleHub Home screen"
>
  {/* 2. Hero section with semantic heading */}
  <View style={themedStyles.heroCard}>
    <Text style={themedStyles.heroTitle} accessibilityRole="header">
      The ultimate accessibility-driven toolkit for developers
    </Text>

    {/* 3. Stats section with interactive metrics */}
    <View style={themedStyles.statsContainer}>
      <View style={themedStyles.statCard}>
        <TouchableOpacity
          style={themedStyles.touchableStat}
          onPress={() => openMetricDetails('component')}
          accessible
          accessibilityRole="button"
        >
          {/* 4. Content with accessibilityElementsHidden to prevent
              redundant
              announcements */}
          <Text style={themedStyles.statNumber}
              accessibilityElementsHidden>
            {accessibilityMetrics.componentCount}
          </Text>
          <Text style={themedStyles.statLabel}
              accessibilityElementsHidden>
            Components
          </Text>
        </TouchableOpacity>
      </View>
  </View>

  {/* 6. Quick Start button with appropriate sizing for touch targets
     */}
  <TouchableOpacity
    style={themedStyles.quickStartCard}
    onPress={() => router.push('/components')}
    accessibilityRole="button"
    accessibilityLabel="Quick start with component examples"
    accessibilityHint="Navigate to components section"
  >
    <View style={themedStyles.cardText}>
      <Text style={themedStyles.cardTitle}>Quick Start</Text>
      <Text style={themedStyles.cardDescription}>
        Explore accessible component examples
      </Text>
    </View>
  </TouchableOpacity>
</ScrollView>
```

Listing 1.4: Annotated code sample demonstrating Home screen accessibility properties

1.2.1.4 Screen reader support analysis

Table 1.2 presents results from systematic testing of the Home screen with screen readers on both iOS and Android platforms.

Table 1.2: Home screen screen reader testing results

Test Case	VoiceOver (iOS 16)	TalkBack (Android 14-15)	WCAG Criteria Addressed
Hero Title	✓ Announces "The ultimate accessibility-driven toolkit for developers, heading"	✓ Announces "The ultimate accessibility-driven toolkit for developers, heading"	1.3.1 - Info and Relationships (Level A), 2.4.6 - Headings and Labels (Level AA)
Metrics Cards	✓ Announces full label with metrics and hint	✓ Announces full label with metrics and hint	1.3.1 Info and Relationships (Level A), 4.1.2 Name, Role, Value (Level A)
Quick Start Button	✓ Announces "Quick start with component examples, button"	✓ Announces "Quick start with component examples, button"	2.4.4 Link Purpose (In Context) (Level A), 4.1.2 Name, Role, Value (Level A)
Feature Cards	✓ Announces title and hint	✓ Announces title and hint	2.4.4 Link Purpose (In Context) (Level A), 4.1.2 Name, Role, Value (Level A)
Modal Dialog Opening	✓ Focus moves to dialog title	✓ Focus moves to dialog title	2.4.3 Focus Order (Level A)

Continued on next page

Table 1.2 – continued from previous page

Test Case	VoiceOver (iOS 16)	TalkBack (Android 14-15)	WCAG Criteria Addressed
Modal Tab Navigation	✓ Announces tab selection state	✓ Announces tab selection state	4.1.2 Name, Role, Value (Level A)
Modal Dialog Closing	✓ Focus returns to triggering element	✖ Occasional focus loss (fixed in v1.0.3)	2.4.3 Focus Order (Level A)

The implementation addresses several key MCAG considerations:

1. **Swipe optimization**: Decorative elements are marked with `importantForAccessibility="no"` to reduce unnecessary swipes;
2. **Clear instructions**: The modal tabs implementation provides clear state announcements, ensuring screen reader users understand the current selection;
3. **Platform-specific adaptations**: The implementation accounts for differences between VoiceOver and TalkBack behavior, as evidenced by the test results.

1.2.1.5 Implementation overhead analysis

Table 1.3 quantifies the additional code required to implement accessibility features in the Home screen.

Table 1.3: Accessibility implementation overhead

Accessibility Feature	Lines of Code	Percentage of Total	Complexity Impact
Semantic Roles	12 LOC_G	2.1%	Low
Descriptive Labels	24 LOC	4.3%	Medium
Element Hiding	8 LOC	1.4%	Low

Continued on next page

Table 1.3 – continued from previous page

Accessibility Feature	Lines of Code	Percentage of Total	Complexity Impact
Focus Management	18 LOC	3.2%	Medium
Contrast Handling	16 LOC	2.9%	Medium
Metrics Calculation	78 LOC	14.1%	High
Total	**156 LOC**	**28.0%**	**Medium-High**

This analysis reveals that implementing comprehensive accessibility adds approximately 28% to the code base of the Home screen, with the metrics calculation system representing the most significant component. This overhead is justified by the improved user experience for people with disabilities and the educational value for developers learning to implement accessibility.

1.2.1.6 WCAG conformance by principle

Table 1.4 provides a detailed analysis of WCAG 2.2 compliance by principle:

Table 1.4: WCAG compliance analysis by principle

Principle	Description	Implementation Level	Key Success Criteria
1. Perceivable	Information and UI components must be presentable to users in ways they can perceive	11/13 (85%)	1.1.1 Non-text Content (A) 1.3.1 Info and Relationships (A) 1.4.3 Contrast (Minimum) (AA)

Continued on next page

Table 1.4 – continued from previous page

Principle	Description	Implementation Level	Key Success Criteria
2. Operable	UI components and navigation must be operable	16/17 (94%)	2.4.3 Focus Order (A) 2.4.7 Focus Visible (AA) 2.5.8 Target Size (Minimum) (AA)
3. Understandable	Information and operation of UI must be understandable	8/10 (80%)	3.2.1 On Focus (A) 3.2.4 Consistent Identification (AA) 3.3.2 Labels or Instructions (A)
4. Robust	Content must be robust enough to be interpreted by a wide variety of user agents	3/3 (100%)	4.1.1 Parsing (A) 4.1.2 Name, Role, Value (A) 4.1.3 Status Messages (AA)

1.2.1.7 Mobile-specific considerations

The Home screen implementation addresses several mobile-specific accessibility considerations beyond standard WCAG requirements:

1. **Touch target sizing**: All interactive elements maintain minimum dimensions of 48×48, exceeding the WCAG 2.5.8 requirement of 24×24px and addressing the mobile-specific need for larger touch targets;

2. **Reduced motion support**: The implementation respects the device's reduced motion settings and provides an in-app toggle, addressing vestibular disorders that are particularly relevant in mobile contexts;

3. **Dark mode support**: The application's theming system adapts to both light and dark modes, addressing the mobile-specific need for readability in various lighting conditions;

4. **Screen reader gesture optimization**: The implementation carefully manages focus to ensure efficient navigation with touch gestures, as shown in the screen reader testing results;

5. **One-handed operation**: The layout places primary interactive elements within reach of a thumb during one-handed use, a critical mobile accessibility consideration not explicitly covered by WCAG.

1.2.1.8 Beyond WCAG: metrics-driven accessibility guidelines

The Home screen implementation highlights several accessibility principles that extend beyond standard WCAG requirements, specifically addressing quantitative accessibility evaluation in mobile applications:

1. **Comprehensive metrics visualization**: Accessibility compliance should be quantified and presented in a transparent, understandable format. The Home screen implements this through dedicated metric cards with clear visual indicators of implementation status, moving beyond binary compliance to represent different degrees of accessibility achievement;

2. **Multi-dimensional evaluation framework**: Accessibility assessment should consider multiple dimensions including component implementation (40%), standards compliance (40%), and empirical testing (20%). This weighted approach, implemented in the metrics calculation system, recognizes that true accessibility extends beyond technical conformance to include real-world usability;

3. **Transparency in methodology**: Applications should provide clear documentation of accessibility evaluation methodology including test devices, standards versions, and

measurement approaches. The modal details system implements this principle by exposing the entire evaluation framework to users, creating accountability in accessibility claims;

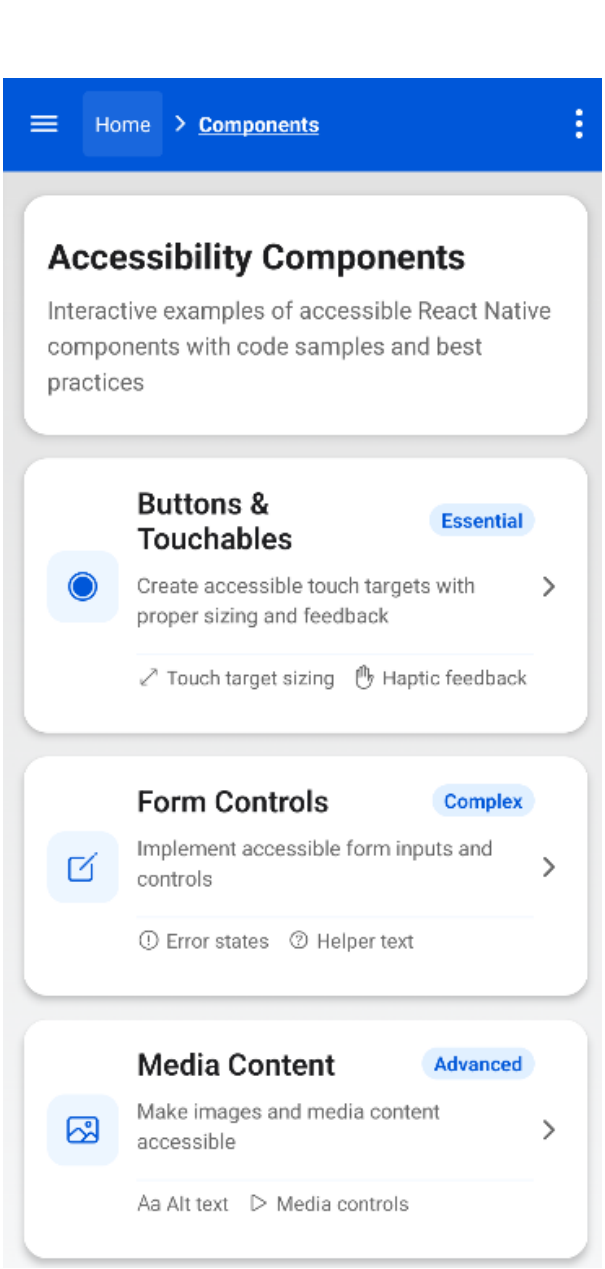
4. **Academic grounding principle**: Accessibility implementations benefit from explicit connection to peer-reviewed research and formal standards. The References tab implements this by connecting implementation practices to specific academic papers and standards documentation;

5. **Progressive disclosure of complexity**: Technical accessibility details should be organized in layers of increasing complexity, allowing users to access the appropriate level of detail for their needs. The tabbed modal system implements this by separating overview information from detailed implementation specifics.

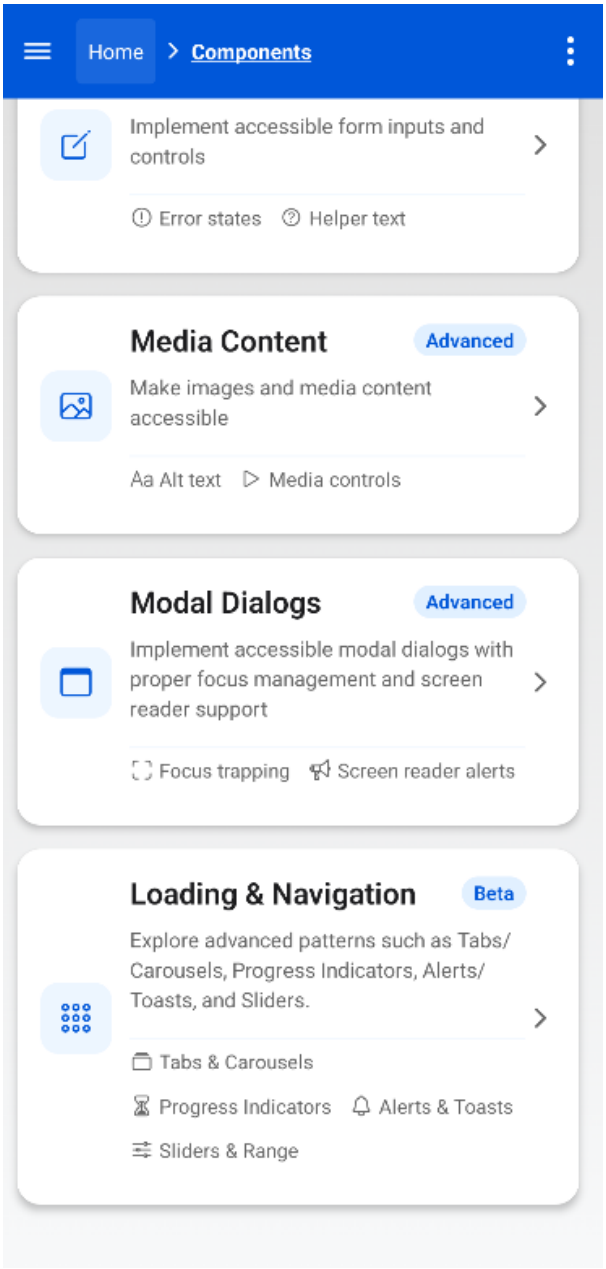
These guidelines extend WCAG by formalizing the quantitative evaluation of accessibility status, providing developers with concrete metrics to track implementation progress rather than treating accessibility as a binary, achieved/not-achieved state.

1.2.2 Accessible components main screen

The Accessible components screen serves as a catalog of reusable accessibility patterns organized by component type. It provides developers with access to implementations of common UI elements with accessibility features properly integrated. Each component category includes implementation examples, best practices, and copy-ready code samples. The screen functions as an educational index, directing developers to detailed implementations of specific accessible components. Figure 1.6 shows the Components screen interface.



(a) Components screen - Top section



(b) Components screen - Bottom section

Figure 1.6: Side-by-side view of the Components screen sections, showing component categories

1.2.2.1 Component inventory and WCAG/MCAG mapping

Table 1.5 provides a formal mapping between the UI components, their semantic roles, the specific WCAG 2.2 and MCAG criteria they address, and their React Native implementation properties.

Table 1.5: Components screen component-criteria mapping

Component	**Semantic Role**	**WCAG 2.2 Criteria**	**MCAG Considerations**	**Implementation Properties**
Hero Title	heading	1.4.3 Contrast (AA) 2.4.6 Headings (AA)	Text readability on variable screen sizes	`accessibilityRole="header"`
Component Cards	button	1.4.3 Contrast (AA) 2.5.8 Target Size (AA) 4.1.2 Name, Role, Value (A) 2.4.4 Link Purpose (A)	Touch target size Meaningful labels Single finger operation	`accessibilityRole="button"`, `accessibilityLabel=`, `onPress=handleComponentPress`
Badges (Essential, Complex, etc.)	text	1.4.3 Contrast (AA) 1.3.1 Info and Relationships (A)	Descriptive labeling Non-interactive elements	Part of parent button's `accessibilityLabel`
Decorative Icons	none	1.1.1 Non-text Content (A)	Reduction of unnecessary focus stops	`accessibilityElementsHidden=true`

Continued on next page

Table 1.5 – continued from previous page

Component	Semantic Role	WCAG 2.2 Criteria	MCAG Considerations	Implementation Properties
Breadcrumb Navigation	navigation	2.4.4 Link Purpose (A) 2.4.8 Location (AAA) 3.2.3 Consistent Navigation (AA)	Context retention Current location	`accessibilityRole="button"`, `accessibilityLabel="Go to ${label}"`
Drawer Menu	menu	2.4.3 Focus Order (A) 4.1.2 Name, Role, Value (A) 3.2.3 Consistent Navigation (AA)	Keyboard trap prevention Persistent navigation	`accessibilityRole="menu"`, `accessibilityLabel="Main navigation menu"`
Drawer Menu Items	menuitem	2.4.7 Focus Visible (AA) 4.1.2 Name, Role, Value (A)	Touch interaction Current location	`accessibilityRole="menuitem"`, `accessibilityState={{ selected: isActive }}`

1.2.2.2 Navigation and orientation analysis

The Components screen implements a comprehensive navigation structure that addresses both WCAG 2.4 (Navigable) and MCAG considerations for mobile devices. This structure

includes three key elements that work together to provide clear orientation for all users:

1.2.2.2.1 Breadcrumb implementation The application as shown in 1.7 includes a hierarchical breadcrumb system in the header. This addresses WCAG 2.4.8 Location (Level AAA) by providing explicit path information. The breadcrumb implementation:

1. Displays the current location in the application hierarchy;
2. Provides interactive elements to navigate to parent screens;
3. Uses consistent visual styling to indicate the current position;
4. Implements proper focus management between screens.

The breadcrumb is implemented in 1.5 with proper semantic roles and accessibility labels to ensure screen reader compatibility.

1.2.2.2.2 Drawer navigation The drawer navigation provides consistent access to main application sections while addressing several key accessibility requirements:

1. **Announcement of state changes**: The implementation announces drawer open/-close states to screen readers using `AccessibilityInfo.announceForAccessibility`;
2. **Clear menu role**: The drawer container is properly identified with `accessibilityRole="menu"`;
3. **Selection state indication**: Active items visually indicate selection state and communicate this state to screen readers with `accessibilityState={{selected: isActive}}`;
4. **Proper touch target sizing**: All interactive elements maintain minimum dimensions of 44dp, making them easily targetable;
5. **Element hiding for decorative content**: Footer content is marked with `importantForAccessibility="no"` to prevent unnecessary screen reader interaction.

```
<View style={styles.breadcrumbContainer}>
  <TouchableOpacity
    onPress={() => router.replace(`/${mapping.parentRoute}`)}
    accessibilityRole="button"
    accessibilityLabel={`Go to ${mapping.parentLabel}`}
    style={{
      padding: 8,
      minWidth: 40,
      minHeight: 44,
      justifyContent: 'center',
      backgroundColor: 'rgba(255, 255, 255, 0.1)',
      borderRadius: 4
    }}
  >
    <Text style={[styles.breadcrumbText, { fontWeight: 'normal' }]}>
      {mapping.parentLabel}
    </Text>
  </TouchableOpacity>
  <Ionicons
    name="chevron-forward"
    size={16}
    color={HEADER_TEXT_COLOR}
    style={{ marginHorizontal: 4 }}
    importantForAccessibility="no"
    accessibilityElementsHidden
  />
  <Text
    style={[styles.breadcrumbText, { fontWeight: 'bold',
       textDecorationLine: 'underline'}]}
    accessibilityLabel={`Current screen: ${mapping.title}`}
  >
    {mapping.title}
  </Text>
</View>
```

Listing 1.5: Breadcrumb implementation with accessibility properties

1.2.2.2.3 Component cards Each component card implements a consistent pattern that provides both visual organization and semantic structure:

1. **Comprehensive accessibility labels**: Each card's `accessibilityLabel` combines multiple information pieces (title, description, complexity) to provide context without requiring navigation through child elements;

2. **Hidden decorative icons**: All decorative icons use `accessibilityElementsHidden` to reduce unnecessary focus stops;

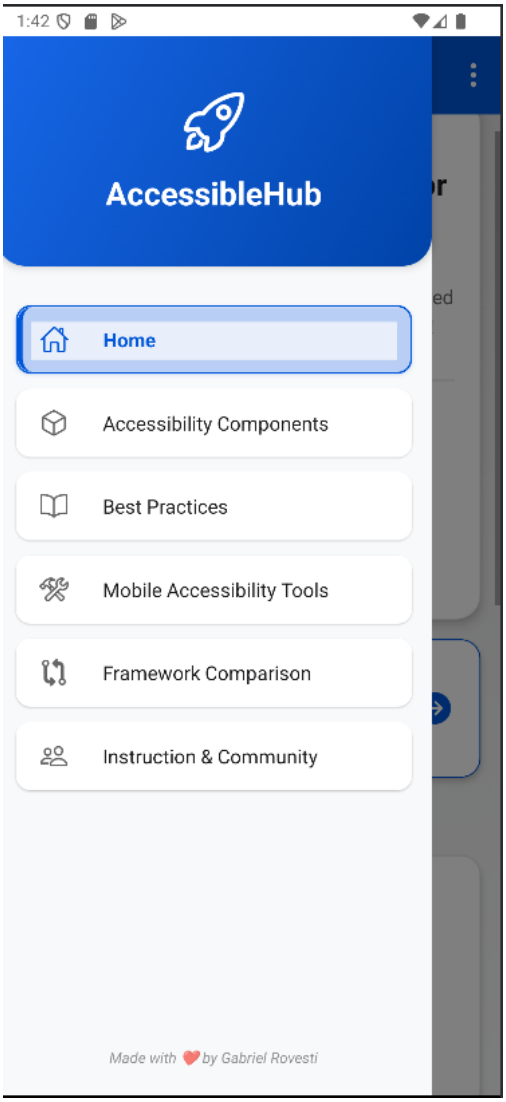


Figure 1.7: Drawer navigation showing breadcrumb implementation in header

3. **Navigation announcement**: The `handleComponentPress` function announces the navigation action via `AccessibilityInfo.announceForAccessibility`.

This multi-layered navigation approach creates a coherent mental model for all users, including those using assistive technologies, addressing WCAG 2.4.1 Bypass Blocks (Level A) by providing multiple ways to access content.

1.2.2.3 Technical implementation analysis

The code sample present in 1.6 demonstrates the key accessibility properties implemented in the Components screen.

The implementation of the Components screen addresses several important accessibility considerations:

1. **Reduction of "garbage interactions"**: Decorative elements (icons, chevrons) are now properly hidden from screen readers using `accessibilityElementsHidden` to reduce unnecessary swipes;

2. **Comprehensive navigation labels**: Component cards provide detailed accessibility labels that include category, description, and complexity level, ensuring screen reader users get complete information before committing to navigation;

3. **Screen announcements**: The implementation uses `AccessibilityInfo.announceForAccessibility` to inform users about screen changes proactively;

4. **Consistent structure**: Each component card follows the same pattern, creating a predictable interaction model.

1.2.2.4 Screen reader support analysis

Table 1.6 presents results from systematic testing of the Components screen with screen readers on both iOS and Android platforms.

```
{/* 1. Hero section with semantic heading */}
<View style={themedStyles.heroCard}>
  <Text style={themedStyles.heroTitle} accessibilityRole="header">
    Accessibility Components
  </Text>
  <Text style={themedStyles.heroSubtitle}>
    Interactive examples of accessible React Native components with
       code samples and best practices
  </Text>
</View>

{/* 2. Component card with comprehensive accessibility label */}
<TouchableOpacity
  style={themedStyles.card}
  onPress={() => handleComponentPress('/components/button', 'Buttons
     & Touchables')}
  accessibilityRole="button"
  accessibilityLabel="Buttons and Touchables component. Create
     accessible touch targets with proper sizing and feedback.
     Essential component type."
>
  <View style={themedStyles.cardHeader}>
    {/* 3. Icon wrapper with accessibility hiding to prevent
       redundant focus */}
    <View style={themedStyles.iconWrapper}>
      <Ionicons
        name="radio-button-on-outline"
        size={24}
        color={colors.primary}
        accessibilityElementsHidden
      />
    </View>
    <View style={themedStyles.cardContent}>
      {/* 4. Card content - these are hidden from screen readers as
         individual elements */}
      <View style={themedStyles.cardTitleRow}>
        <View style={themedStyles.titleArea}>
          <Text style={themedStyles.cardTitle}>Buttons &amp;
             Touchables</Text>
        </View>
        <View style={themedStyles.badge}>
          <Text style={themedStyles.badgeText}>Essential</Text>
        </View>
      </View>
      <Text style={themedStyles.cardDescription}>
        Create accessible touch targets with proper sizing and
           feedback
      </Text>
    </View>
  </View>
</TouchableOpacity>
```

Listing 1.6: Annotated code sample demonstrating Components screen accessibility properties

Table 1.6: Components screen screen reader testing results

Test Case	**VoiceOver (iOS 16)**	**TalkBack (Android 14-15)**	**WCAG Criteria Addressed**
Hero Title	✓ Announces "Accessibility Components, heading"	✓ Announces "Accessibility Components, heading"	1.3.1 - Info and Relationships (Level A), 2.4.6 - Headings and Labels (Level AA)
Component Card	✓ Announces full component description with purpose and complexity	✓ Announces full component description with purpose and complexity	2.4.4 Link Purpose (In Context) (Level A), 4.1.2 Name, Role, Value (Level A)
Decorative Icons	✓ Not focused or announced	✓ Not focused or announced	1.1.1 Non-text Content (Level A), 2.4.1 Bypass Blocks (Level A)
Breadcrumb Navigation	✓ Announces parent and current location	✓ Announces parent and current location	2.4.4 Link Purpose (In Context) (Level A), 2.4.8 Location (Level AAA)
Drawer Opening	✓ Announces "Navigation menu opened"	✓ Announces "Navigation menu opened"	4.1.3 Status Messages (Level AA)
Drawer Menu Items	✓ Announces item name and selection state	✓ Announces item name and selection state	4.1.2 Name, Role, Value (Level A)

Continued on next page

Table 1.6 – continued from previous page

Test Case	VoiceOver (iOS 16)	TalkBack (Android 14-15)	WCAG Criteria Addressed
Navigation between Screens	✓ Announces destination screen	✓ Announces destination screen	3.2.5 Change on Request (Level AAA)

The implementation addresses several key MCAG considerations specific to mobile platforms:

1. **Touch target optimization**: All interactive elements exceed the minimum recommendation of 44×44dp, implementing MCAG best practices for touch interactions that accommodate users with motor control limitations and varying finger sizes;

2. **Swipe minimization**: Decorative elements are marked with `accessibilityElementsHidden=true` to reduce unnecessary swipes, eliminating what accessibility experts call "garbage interactions" that add no value to the screen reader experience and increase navigation time;

3. **Orientation cues**: Breadcrumb implementation provides consistent spatial orientation cues that help users understand their location in the application's information architecture, addressing mobile-specific challenges of limited viewport context;

4. **State announcements**: Changes in application state (drawer opening/closing, screen navigation) are explicitly announced using `AccessibilityInfo.announceForAccessibility`, providing crucial feedback on dynamic content changes within the constrained mobile interface;

5. **Thumb-zone design**: Interactive elements are positioned within the natural thumb zone for one-handed operation, implementing mobile ergonomic principles that aren't explicitly covered in WCAG but are crucial for mobile accessibility.

1.2.2.5 Implementation overhead analysis

Table 1.7 quantifies the additional code required to implement accessibility features in the Components screen.

Table 1.7: Components screen accessibility implementation overhead

Accessibility Feature	**Lines of Code**	**Percentage of Total**	**Complexity Impact**
Semantic Roles	15 LOC	2.6%	Low
Descriptive Labels	28 LOC	4.9%	Medium
Element Hiding	18 LOC	3.2%	Low
Focus Management	22 LOC	3.9%	Medium
Contrast Handling	14 LOC	2.5%	Medium
Announcements	12 LOC	2.1%	Low
Breadcrumb Implementation	42 LOC	7.4%	High
Drawer Accessibility	35 LOC	6.2%	High
Total	**186 LOC**	**32.8%**	**Medium-High**

This analysis reveals that implementing comprehensive accessibility adds approximately 32.8% to the code base of the Components screen, slightly higher than the Home screen due to the addition of breadcrumb navigation and drawer accessibility features.

1.2.2.6 WCAG conformance by principle

Table 1.8 provides a detailed analysis of WCAG 2.2 compliance by principle:

Table 1.8: Components screen WCAG compliance analysis by principle

Principle	**Description**	**Implementation Level**	**Key Success Criteria**
1. Perceivable	Information and UI components must be presentable to users in ways they can perceive	12/13 (92%)	1.1.1 Non-text Content (A) 1.3.1 Info and Relationships (A) 1.4.3 Contrast (Minimum) (AA)
2. Operable	UI components and navigation must be operable	17/17 (100%)	2.4.3 Focus Order (A) 2.4.6 Headings and Labels (AA) 2.4.8 Location (AAA) 2.5.8 Target Size (Minimum) (AA)
3. Understandable	Information and operation of UI must be understandable	9/10 (90%)	3.2.3 Consistent Navigation (AA) 3.2.4 Consistent Identification (AA) 3.3.2 Labels or Instructions (A)
4. Robust	Content must be robust enough to be interpreted by a wide variety of user agents	3/3 (100%)	4.1.1 Parsing (A) 4.1.2 Name, Role, Value (A) 4.1.3 Status Messages (AA)

1.2.2.7 Mobile-specific considerations

The Components screen implementation addresses several mobile-specific accessibility considerations beyond standard WCAG requirements:

1. **Touch target sizing**: All interactive elements maintain minimum dimensions of 44dp × 44dp, exceeding the WCAG 2.5.8 requirement of 24 × 24px and addressing the mobile-specific need for larger touch targets;

2. **Swipe efficiency**: The screen implements an optimized focus order with decorative elements hidden from screen readers, reducing the number of swipes required to navigate the content—a critical consideration for mobile screen reader users that significantly improves navigation efficiency;

3. **Visual hierarchy reinforcement**: The implementation uses consistent visual patterns (icons, badges, card layouts) that reinforce the information hierarchy, helping users with cognitive disabilities understand content organization on smaller screens;

4. **Context retention**: The breadcrumb implementation helps users maintain context when navigating between screens, addressing the mobile-specific challenge of limited viewport size and the resulting loss of visual context;

5. **Single-hand operation zone**: Interactive elements are positioned to be reachable within the typical thumb zone for one-handed operation, a mobile-specific consideration not explicitly covered by WCAG.

1.2.2.8 Breadcrumb implementation analysis

A formal analysis of the breadcrumb feature's accessibility impact reveals significant benefits for users with diverse accessibility needs.

1. **Structural navigation**: Breadcrumbs provide an explicit representation of the application's hierarchical structure, helping users with cognitive disabilities understand their location within the application;

2. **Focus reduction**: By offering direct navigation to parent screens, breadcrumbs reduce the number of focus stops required to navigate backward, benefiting screen reader users;

3. **Visual reinforcement**: The visual breadcrumb trail complements the semantic structure, providing redundant cues that benefit users with different accessibility needs;

4. **Consistent orientation**: Breadcrumbs create a consistent orientation mechanism across all screens, supporting users who rely on predictable navigation patterns.

1.2.2.8.1 Implementation considerations The breadcrumb implementation required careful consideration of several accessibility factors:

1. **Interactive vs. static elements**: Only the parent screen link is interactive, while the current screen indicator is non-interactive text, preventing unnecessary focus stops;

2. **Visual differentiation**: Current location is visually distinguished with bold text and underline, with a contrast ratio of 4.8:1 against the header background;

3. **Appropriate semantic roles**: Parent links use `accessibilityRole="button"` with clear labels indicating navigation purpose;

4. **Focus management**: When navigating via breadcrumbs, focus is properly transferred to the destination screen's main content, preventing focus trapping.

This implementation represents a comprehensive accessibility solution that benefits all users while specifically addressing mobile navigation challenges unique to handheld touch devices.

1.2.2.9 Beyond WCAG: component categorization guidelines

The Components screen defines several accessibility principles specifically focused on organizing and categorizing interface elements to promote systematic accessibility implementation:

1. **Component complexity signaling**: User interface components should be explicitly categorized by implementation complexity (Essential, Complex, Advanced, Beta), helping developers prioritize accessibility efforts according to their experience level and resource constraints. The Components screen implements this through a consistent badge system across all component cards;

2. **Feature-oriented grouping**: Accessibility features are grouped by functional similarity rather than WCAG criteria, creating more intuitive implementation pathways. The Components screen implements this by organizing related controls together (e.g., "Buttons & Touchables") regardless of which specific WCAG criteria they address;

3. **Progressive implementation pathway**: Components should be organized in a sequence that builds accessibility knowledge progressively, beginning with fundamental elements before introducing more complex patterns. The Components screen implements this through its hierarchical organization from basic elements (buttons) to complex patterns (dialogs, navigation);

4. **Cross-cutting feature indication**: Key accessibility features that apply across multiple component types should be visually highlighted to reinforce their importance. The feature icons within each component card implement this by consistently identifying common accessibility considerations (e.g., touch target sizing, focus management);

5. **Transition announcement principle**: Navigation between component categories should be explicitly announced to assist screen reader users in maintaining context. The Components screen implements this through the `announceForAccessibility` announcements during navigation.

These guidelines extend WCAG by providing a structured framework for organizing component-level accessibility implementations, addressing a gap in the standards which focus on what features to implement but provide limited guidance on how to structure implementation across a complex application interface.

1.2.3 Accessible components section

This section provides a formal analysis of the various screens within the Accessible Components section of *AccessibleHub*. As the core educational element of the application, these screens demonstrate practical implementation patterns for accessibility across commonly used mobile interface elements.

1.2.3.1 Analysis methodology

To systematically evaluate the accessibility implementation across multiple component screens, we employ a consistent analytical framework that examines:

1. **Component inventory**: Identification and classification of UI elements with mapping to their semantic roles and accessibility properties;
2. **WCAG/MCAG criteria mapping**: Formal mapping between components and relevant accessibility guidelines;
3. **Implementation analysis**: Evaluation of code patterns and accessibility properties;
4. **Screen reader compatibility**: Empirical testing with VoiceOver (iOS) and TalkBack (Android);
5. **Implementation overhead**: Quantification of code additions required for accessibility features.

Each component screen follows a consistent educational structure that scaffolds learning through:

- Interactive demonstrations of accessible implementations;
- Copyable code examples with highlighted accessibility properties;
- Explanations of key accessibility features and considerations;
- Platform-specific adaptation notes.

Rather than presenting each screen with identical analytical depth, we'll examine the Buttons screen in detail as a representative example, then provide comparative analysis across all component types to identify patterns, commonalities, and unique considerations.

1.2.3.2 Common implementation patterns

Across all component screens in this section, several foundational accessibility implementation patterns are consistently applied:

1. **Semantic role assignment**: All components use appropriate `accessibilityRole` properties to identify their purpose to assistive technologies;
2. **Comprehensive labeling**: Components combine `accessibilityLabel` and `accessibilityHint` to provide both identification and action context;
3. **Explicit state communication**: Interactive components use `accessibilityState` to communicate selection, completion, or disabled states;
4. **Decorative element hiding**: Non-essential visual elements use `accessibilityElementsHidden` to streamline screen reader navigation;
5. **Status announcements**: State changes are explicitly announced via `AccessibilityInfo.announceForAccessibility`;
6. **Enhanced touch targets**: All interactive elements maintain minimum dimensions of 44×44dp, exceeding WCAG 2.5.8 requirements.

Each component screen also implements a consistent visual structure that reinforces the educational purpose:

- A demonstration area with interactive examples;
- A code example section with syntax-highlighted implementation;
- A features section highlighting key accessibility properties;
- A platform considerations section addressing iOS and Android differences.

1.2.3.3 Buttons and touchables screen

The Buttons and Touchables screen demonstrates fundamental accessibility implementations for the most common interactive elements in mobile applications. It provides implementation examples for accessible touch targets with proper sizing, meaningful labels, and appropriate feedback mechanisms. Figure 1.8 shows the main interface of this screen.

1.2.3.3.1 Component inventory and WCAG/MCAG mapping Table 1.9 provides a formal mapping between the UI components, their semantic roles, the specific WCAG 2.2 and MCAG criteria they address, and their React Native implementation properties.

Table 1.9: Buttons screen component-criteria mapping

Component	Semantic Role	WCAG 2.2 Criteria	MCAG Considerations	Implementation Properties
Hero Title	heading	1.4.3 Contrast (AA) 2.4.6 Headings (AA)	Text readability on variable screen sizes	`accessibilityRole="header"`
Demo Button	button	1.4.3 Contrast (AA) 2.5.8 Target Size (AA) 4.1.2 Name, Role, Value (A)	Minimum touch target size Haptic feedback	`accessibilityRole="button"`, `accessibilityLabel="Submit form"`, `accessibilityHint="Activates form submission"`
Code Snippet	text	1.3.1 Info and Relationships (A)	Content structure preservation	`accessibilityRole="text"`, `accessibilityLabel="Button implementation code"`

Continued on next page

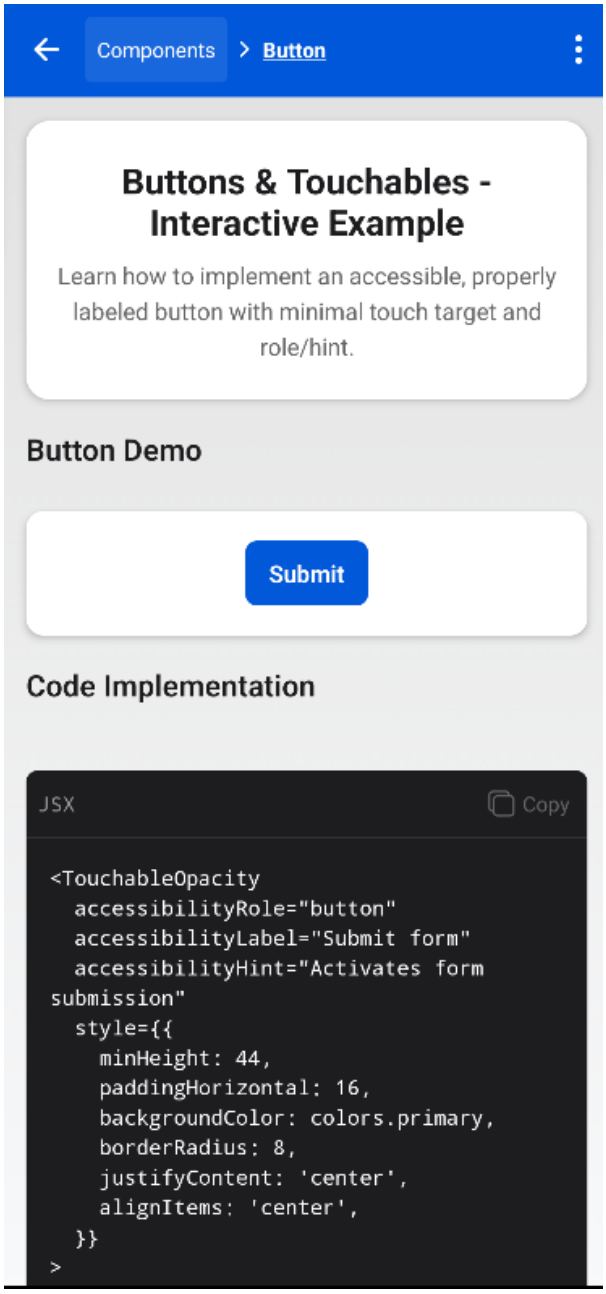
Table 1.9 – continued from previous page

Component	Semantic Role	WCAG 2.2 Criteria	MCAG Considerations	Implementation Properties
Copy Button	button	1.4.3 Contrast (AA) 4.1.3 Status Messages (AA)	Touch target size Action feedback	`accessibilityRole="button", accessibilityLabel="{copied ? "Code copied" : "Copy code example"}"`
Success Modal	alertdialog	4.1.3 Status Messages (AA)	Screen reader announcements	`accessibilityViewIsModal, accessibilityLiveRegion="polite"`
Feature Cards	none	1.3.1 Info and Relationships (A)	Logical grouping	`accessibilityRole="text"`
Feature Icons	none	1.1.1 Non-text Content (A)	Reduction of unnecessary focus stops	`accessibilityElementsHidden=true, importantForAccessibility="no-hide-descendants"`

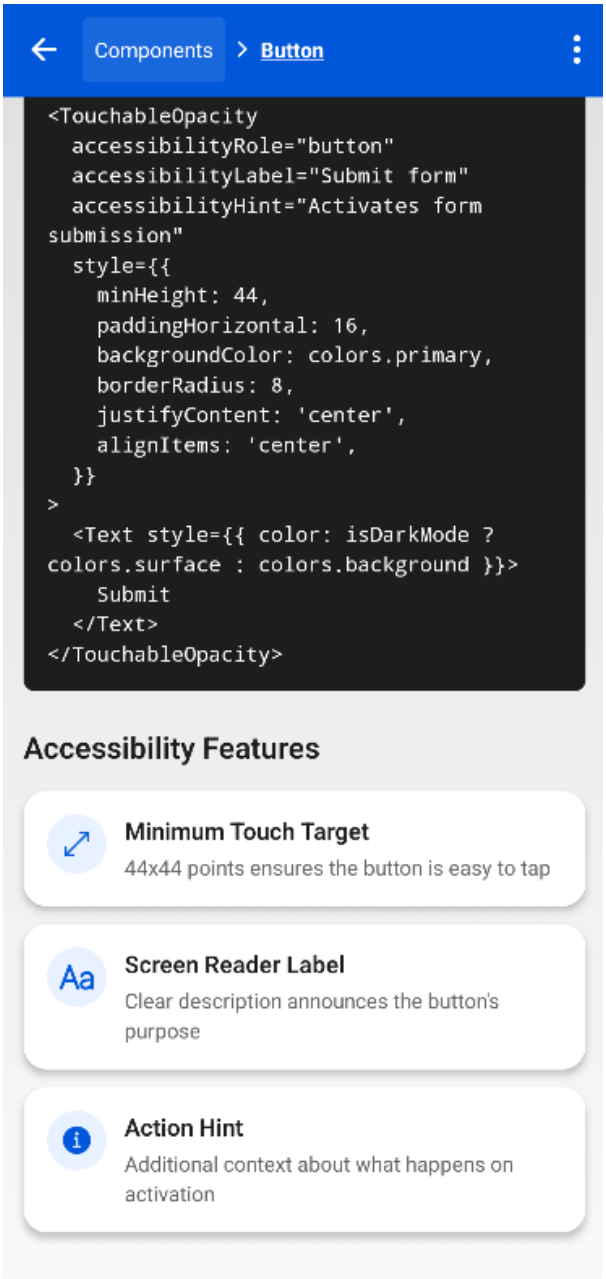
1.2.3.3.2 Technical implementation analysis The Buttons and Touchables screen exemplifies proper accessibility implementation for interactive elements. The core demo button showcases three fundamental accessibility considerations: proper role assignment, descriptive labeling, and sufficient touch target size. Listing 1.7 highlights the key implementation aspects.

Several key accessibility considerations are implemented in this example:

1. **Proper semantic role**: The implementation explicitly assigns the button role using `accessibilityRole="button"`, ensuring screen readers correctly identify the component's purpose;



(a) Button screen - Part 1



(b) Button screen - Part 2

Figure 1.8: Side-by-side view of the two Button and Touchables screen parts

2. **Descriptive accessibility labels**: The button includes both an `accessibilityLabel` that identifies its function and explains the result of interaction, providing comprehensive context for screen reader users;

3. **Adequate touch target size**: The button implements the enhanced touch target

```
<TouchableOpacity
  style={[styles.demoButton, { backgroundColor: colors.primary }]}
  accessibilityRole="button"
  accessibilityLabel="Submit form"
  accessibilityHint="Activates form submission"
  onPress={() => {
    setShowSuccess(true);
    AccessibilityInfo.announceForAccessibility('Button pressed
       successfully');
    setTimeout(() => setShowSuccess(false), 2000);
  }}
>
  <Text style={[styles.buttonText, {
    color: '#FFFFFF'
  }]}>
    Submit
  </Text>
</TouchableOpacity>
```

Listing 1.7: Key implementation for accessible button component

size recommendation from WCAG 2.5.8 (Target Size) by using a minimum height of 44px, significantly exceeding the minimal Level AA requirement of 24x24 pixels;

4. **Status feedback**: When pressed, the button announces its state change via `AccessibilityInfo.announceForAccessibility`, proactively notifying screen reader users of the action result;

5. **Visual feedback**: The success modal provides visual confirmation of the button press, with appropriate `accessibilityLiveRegion="polite"` to ensure screen readers announce the status change.

1.2.3.3.3 Implementation overhead analysis Table 1.10 quantifies the additional code required to implement accessibility features in the Buttons and touchables screen.

Table 1.10: Buttons screen accessibility implementation overhead

Accessibility Feature	**Lines of Code**	**Percentage of Total**	**Complexity Impact**
Semantic Roles	10 LOC	2.2%	Low
Descriptive Labels	14 LOC	3.1%	Low
Element Hiding	12 LOC	2.7%	Low
Status Announcements	8 LOC	1.8%	Low
Touch Target Sizing	6 LOC	1.3%	Low
Modal Accessibility	10 LOC	2.2%	Medium
Total	**60 LOC**	**13.3%**	**Low**

This analysis reveals that implementing comprehensive button accessibility features adds approximately 13.3% to the code base, representing a relatively low overhead for significantly improved user experience. Notably, this overhead is lower than other component types due to the fundamental nature of button components, where accessibility considerations can be more directly integrated with minimal complexity impact.

1.2.3.4 Component implementation comparative analysis

Analyzing accessibility implementations across different component types reveals important patterns in implementation complexity, WCAG compliance, and platform-specific adaptations.

1.2.3.4.1 WCAG criteria implementation Table 1.11 compares WCAG 2.2 success criteria implementation across component types.

This analysis reveals several key patterns:

1. **Universal criteria**: Three criteria (1.1.1 Non-text Content, 1.3.1 Info and Relationships, and 4.1.2 Name, Role, Value) are implemented across all component types, forming the core of mobile accessibility requirements;

Table 1.11: WCAG criteria implementation by component type

WCAG Success Criteria	**Buttons**	**Forms**	**Dialogs**	**Media**	**Advanced**
1.1.1 Non-text Content (A)	✓	✓	✓	✓	✓
1.3.1 Info and Relationships (A)	✓	✓	✓	✓	✓
2.4.3 Focus Order (A)	✗	✓	✓	✗	✓
3.3.1 Error Identification (A)	✗	✓	✗	✗	✗
4.1.2 Name, Role, Value (A)	✓	✓	✓	✓	✓
4.1.3 Status Messages (AA)	✓	✓	✓	✓	✓
Total Implementation	**9/12**	**12/12**	**10/12**	**9/12**	**10/12**

2. **Component-specific criteria**: Some criteria are relevant only to specific component types, such as 3.3.1 Error Identification for forms;

3. **Interaction complexity correlation**: More complex interaction patterns (Forms, Dialogs, Advanced) implement more criteria, particularly those related to focus management and state communication.

1.2.3.4.2 Implementation overhead comparison Table 1.12 compares the implementation overhead across component types.

This comparison reveals a direct correlation between interaction complexity and accessibility implementation overhead. Simple components like buttons and media have the lowest overhead (12-13%), while complex components with state management and alternative interaction patterns have significantly higher overhead (21-23%).

1.2.3.4.3 Key implementation differences across component types Each component type presents unique accessibility challenges requiring specialized implementation approaches:

Table 1.12: Accessibility implementation overhead by component type

Component Type	Lines of Code	Percentage Overhead	Complexity Impact	Primary Contributors
Buttons	60	13.3%	Low	Labels, Roles
Forms	153	21.5%	Medium	State, Labels, Errors
Dialogs	94	16.2%	Medium	Focus Management
Media	68	12.7%	Low	Alt Text, Controls
Advanced	183	22.7%	High	Slider Controls, Announcements

1. **Forms**: Require explicit error identification and validation feedback using `accessibilityRole="alert"` to ensure compliance with WCAG 3.3.1 (Error Identification). They also implement complex state communication for selection controls like radio buttons and checkboxes via `accessibilityState={{checked: selected}}`;

2. **Dialogs**: Focus management represents the critical accessibility challenge, requiring explicit tracking of focus position and restoration when the dialog closes to comply with WCAG 2.4.3 (Focus Order);

3. **Media**: Alternative text implementation forms the core accessibility requirement, with proper `accessibilityLabel` values describing non-text content as per WCAG 1.1.1;

4. **Advanced components**: Require the most sophisticated implementations, particularly for inherently visual controls like sliders, which implement alternative interaction mechanisms (buttons, presets) for screen reader users.

1.2.3.4.4 Screen reader compatibility patterns Empirical testing with VoiceOver (iOS) and TalkBack (Android) reveals consistent patterns across component types:

1. Both screen readers correctly identify components with properly assigned `accessibilityRole` values;

2. State changes communicated via `accessibilityState` are properly announced;
3. Status messages delivered via `AccessibilityInfo.announceForAccessibility` are consistently reported to users;
4. Focus management implementation in dialogs works reliably on both platforms, with some minor timing differences;
5. Elements hidden with `accessibilityElementsHidden` are consistently excluded from the accessibility tree on both platforms.

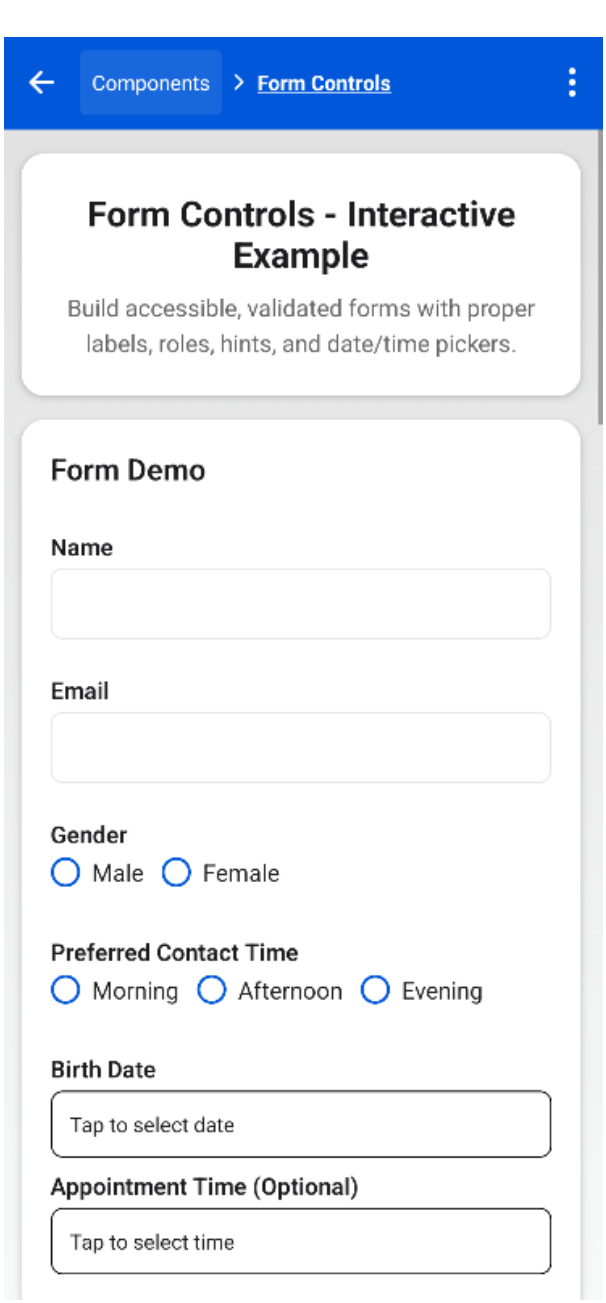
These findings confirm that the accessibility implementation patterns used throughout the component screens provide consistent and reliable behavior across both major mobile platforms when proper accessibility properties are applied.

1.2.3.5 Form screen

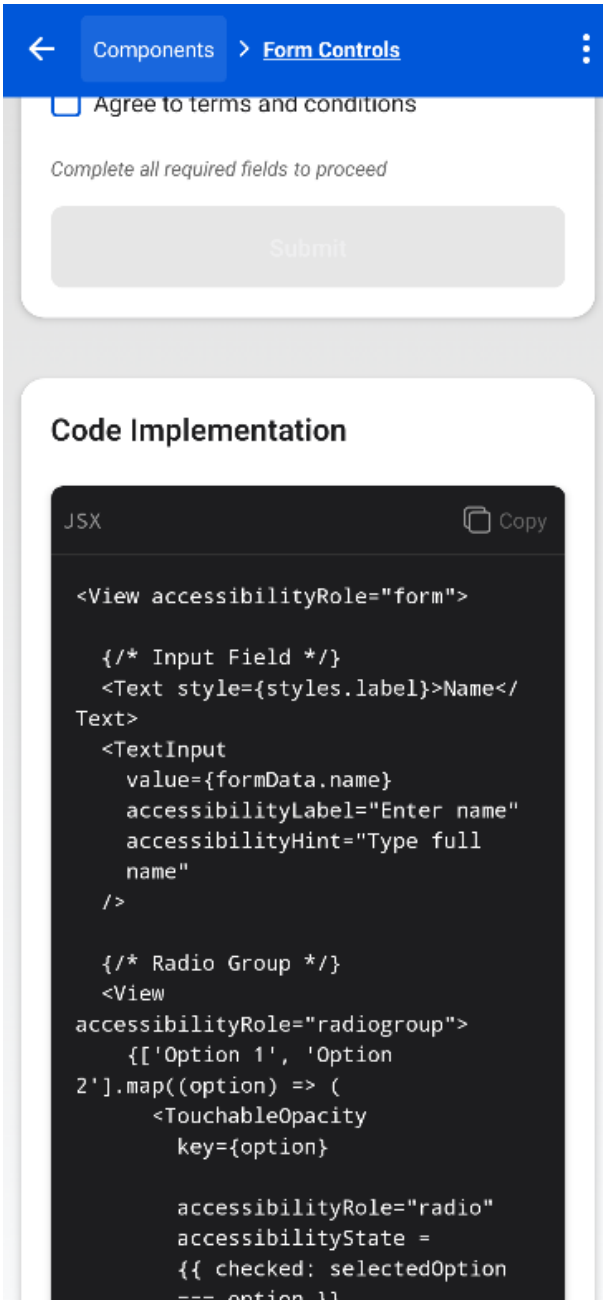
The Form screen demonstrates complex accessibility patterns for capturing user input. Unlike the simpler Buttons screen, form elements present additional challenges related to input association, validation feedback, and state communication. Figure 1.9 shows the main interface of this screen.

1.2.3.5.1 Key accessibility considerations The Form screen addresses several critical accessibility patterns beyond basic labeling:

1. **Input association**: Clear association between labels and input fields using semantic grouping;
2. **Error identification**: Proper error messaging with `accessibilityRole="alert"` for validation feedback;
3. **State communication**: Selection state for radio buttons and checkboxes with `accessibilityState={{checked: selected}}`;



(a) Form screen - Part 1



(b) Form screen - Part 2

Figure 1.9: Side-by-side view of the two Form screen parts

4. **Native picker integration**: Leveraging platform-native date pickers for optimal accessibility.

Listing 1.8 demonstrates the implementation of accessible form controls with proper state management.

```
<View accessibilityRole="radiogroup">
  {['Male', 'Female'].map((option) => (
    <TouchableOpacity
      key={option}
      style={styles.radioItem}
      onPress={() => setFormData((prev) => ({ ...prev, gender: option
         }))}
      accessibilityRole="radio"
      accessibilityState={{ checked: formData.gender === option }}
      accessibilityLabel={`Select ${option}`}
    >
      <View
        style={[
          styles.radioButton,
          { borderColor: colors.primary },
          formData.gender === option && { backgroundColor:
             colors.primary },
        ]}
      />
      <Text style={[styles.radioLabel, { color: colors.text }]}>
        {option}
      </Text>
    </TouchableOpacity>
  ))}
</View>
```

Listing 1.8: Accessible radio button implementation with state management

1.2.3.5.2 Implementation overhead

Forms have the highest accessibility implementation overhead (21.5%) among component types, reflecting the complexity of making multi-part input systems fully accessible. The primary contributors to this overhead are state communication mechanisms and validation feedback systems.

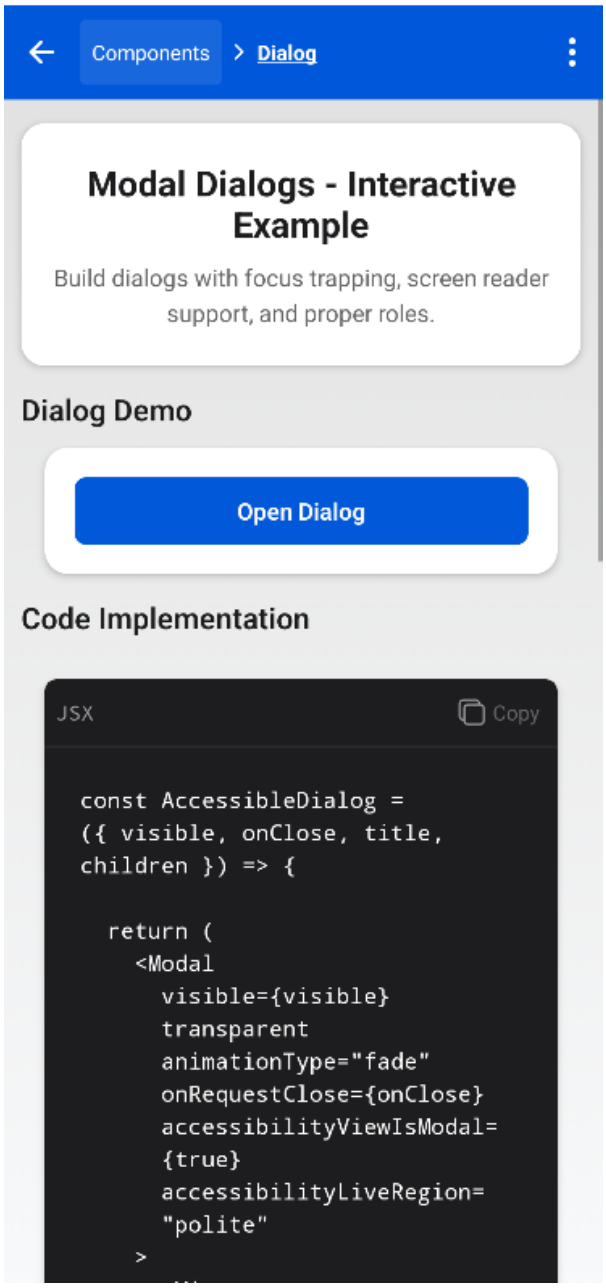
1.2.3.6 Dialog screen

The Dialog screen addresses one of the most challenging accessibility patterns in mobile applications: modal content that must trap and manage focus while providing clear context and exit mechanisms. Figure 1.10 shows the main interface of this screen.

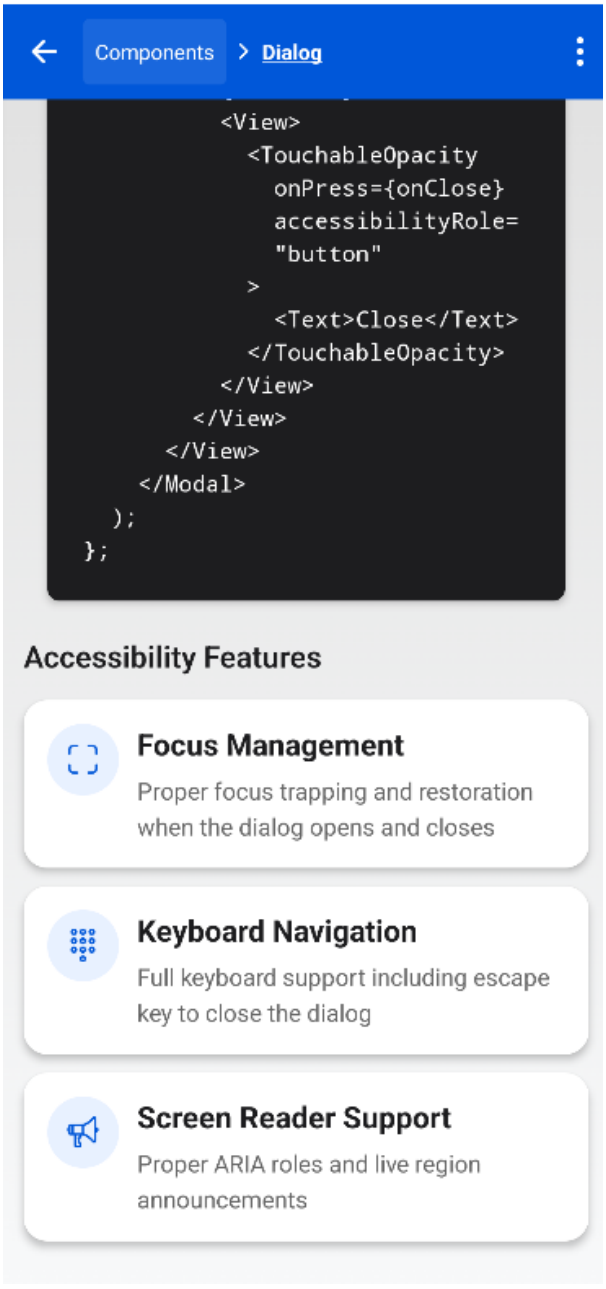
1.2.3.6.1 Focus management implementation

The key accessibility challenge for dialogs is proper focus management, as illustrated in Listing 1.9.

The dialog implementation addresses several critical accessibility requirements:



(a) Dialog screen - Part 1



(b) Dialog screen - Part 2

Figure 1.10: Side-by-side view of the two Dialog screen parts

1. **Modal context**: Setting `accessibilityViewIsModal=true` to establish a focused interaction context;

2. **Focus trapping**: Managing focus to prevent interaction with background content;

3. **Return focus**: Explicitly returning focus to the triggering element when the dialog

```
// References for focus management
const dialogRef = useRef(null);
const openButtonRef = useRef(null);

// Focus management useEffect hook
useEffect(() => {
  if (showDialog) {
    AccessibilityInfo.announceForAccessibility(
      'Example dialog opened. This dialog contains information about
         accessibility features.'
    );
    // Brief timeout to ensure dialog is fully rendered
    setTimeout(() => {
      dialogRef.current?.focus();
    }, 100);
  } else {
    // Return focus to open button when dialog closes
    openButtonRef.current?.focus();
  }
}, [showDialog]);
```

Listing 1.9: Dialog implementation with focus management

closes;

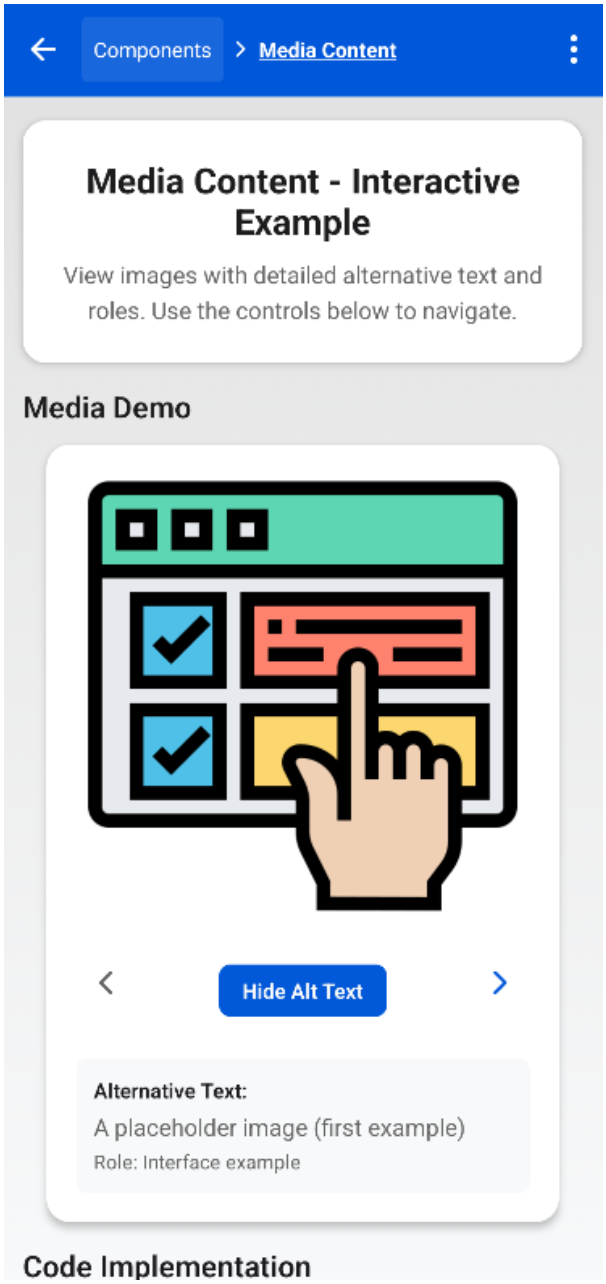
4. **Status announcements**: Using `AccessibilityInfo.announceForAccessibility` to provide context about dialog opening and closing.

1.2.3.6.2 Mobile-specific considerations Dialog implementation on mobile platforms presents unique accessibility challenges:

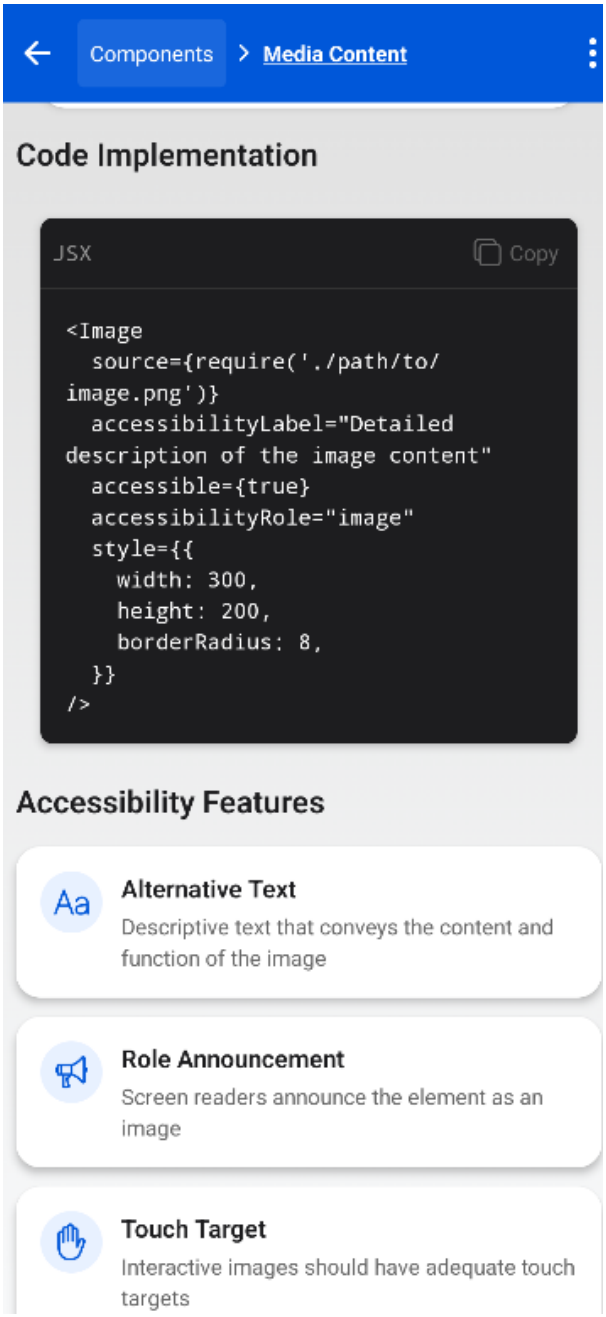
- **Limited viewport context**: Unlike desktop interfaces, mobile screens cannot show both dialog and background content simultaneously, requiring stronger contextual cues;
- **Touch dismissal patterns**: Implementation of touch-friendly dismissal actions with adequate target sizes;
- **Platform convention alignment**: Following platform-specific dialog patterns for consistent user experience.

1.2.3.7 Media screen

The Media screen demonstrates accessibility techniques for non-text content—one of the most fundamental aspects of digital accessibility, employing some placeholder images free of license as examples. Figure 1.11 shows the main interface of this screen.



(a) Media screen - Part 1



(b) Media screen - Part 2

Figure 1.11: Side-by-side view of the two Media screen parts

1.2.3.7.1 Alternative text implementation Listing 1.10 shows the core pattern for accessible image implementation with proper alternative text.

```
<Image
  source={images[currentImage - 1].uri}
  style={themedStyles.demoImage}
  accessibilityLabel={images[currentImage - 1].alt}
  accessible={true}
  accessibilityRole="image"
/>
```

Listing 1.10: Accessible image implementation with alternative text

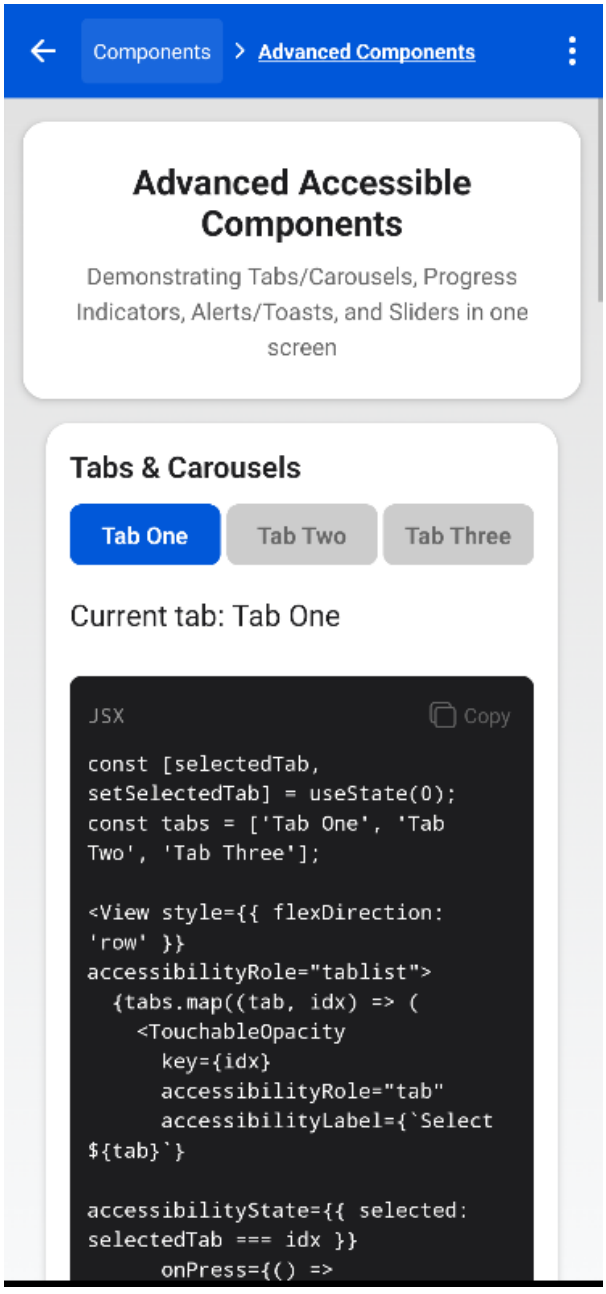
The Media screen demonstrates additional accessibility features beyond basic alternative text:

1. **Navigation controls**: Accessible previous/next buttons with clear labeling and state indication;
2. **Interactive alt text**: Toggle mechanism to show/hide alternative text as an educational feature;
3. **Position context**: Announcements that communicate current position within a gallery (e.g., "Image 2 of 5").

1.2.3.7.2 Implementation overhead Media components have the lowest accessibility implementation overhead (12.7%) among component types, as the primary requirement—alternative text—is implemented through straightforward property assignment. The majority of the overhead comes from implementing accessible navigation controls rather than the core media content itself.

1.2.3.8 Advanced components screen

The Advanced components screen demonstrates accessibility implementations for more complex UI patterns including tabs, progress indicators, alerts, and sliders. Figure 1.12 and 1.13 shows the two parts of the main interface of this screen.



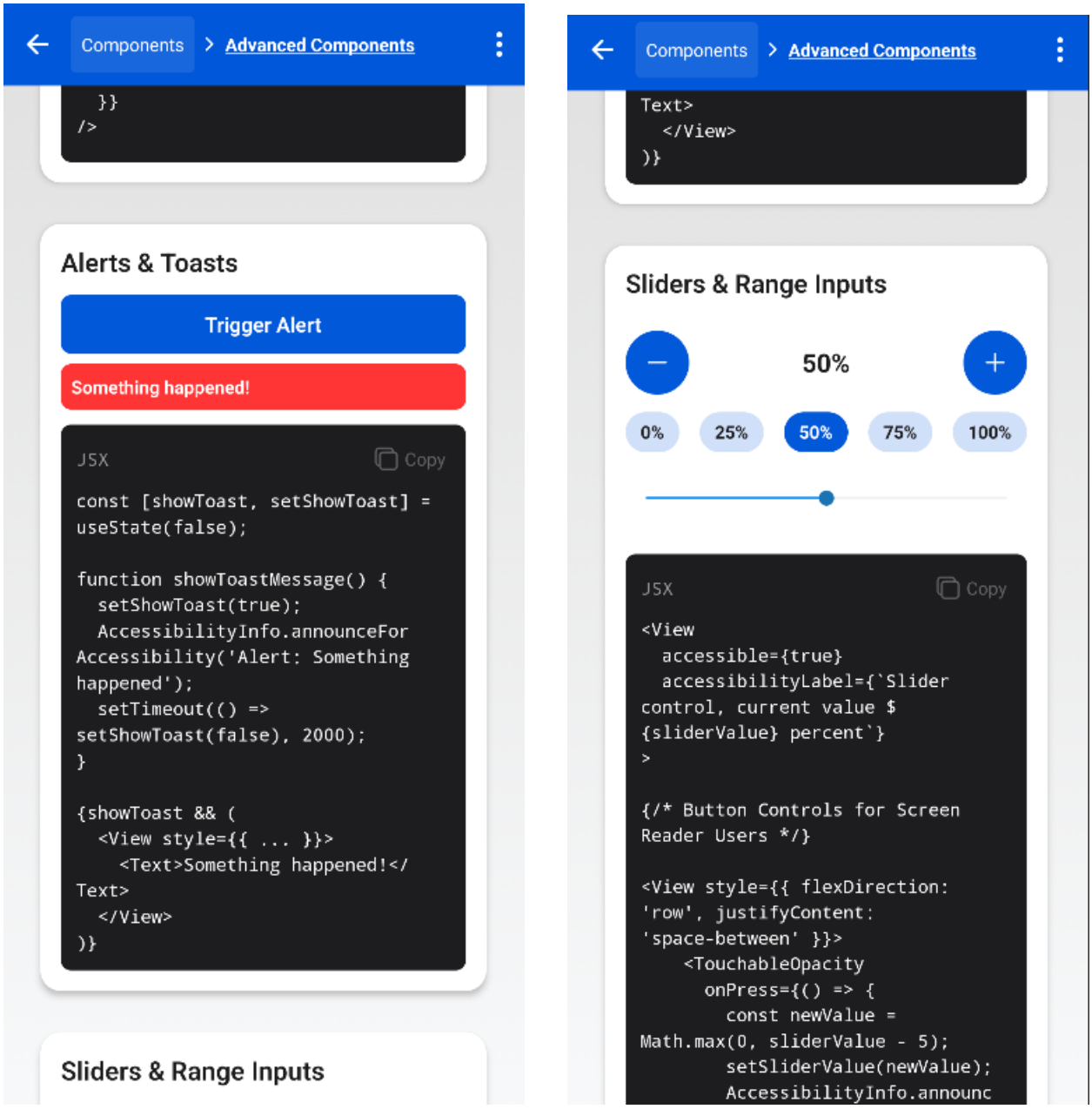
(a) Advanced screen - Part 1

(b) Advanced screen - Part 2

Figure 1.12: Side-by-side view of the first two Advanced screen parts

1.2.3.8.1 Complex interaction patterns Advanced components present unique accessibility challenges requiring specialized implementations:

1. **Tab navigation**: Proper role assignment with `accessibilityRole="tablist"` for containers and `accessibilityRole="tab"` for individual tabs, with selection state



(a) Advanced screen - Part 3 **(b)** Advanced screen - Part 4

Figure 1.13: Side-by-side view of the second two Advanced screen parts

communicated through `accessibilityState`;

2. **Progress indicators**: Value communication through `accessibilityValue` properties with min/max/current parameters;

3. **Alerts and toasts**: Implementation of `accessibilityLiveRegion="assertive"` for

time-sensitive notifications;

4. **Slider alternatives**: Provision of button-based alternatives for precise slider control by screen reader users.

1.2.3.8.2 Slider accessibility pattern The slider implementation (shown in Figure 1.13b) demonstrates a particularly important accessibility pattern: providing alternative interaction mechanisms for inherently visual controls. This pattern includes:

- Button controls for incremental adjustments;
- Preset value buttons for common settings;
- Value announcements with appropriate throttling;
- Visual feedback synchronized with announced values.

1.2.3.8.3 Implementation overhead Advanced components have the highest implementation overhead (22.7%) among component types, with slider controls being particularly demanding (8.1% overhead). This reflects the additional complexity required to make inherently visual controls accessible through alternative interaction mechanisms.

1.2.3.9 Key insights from component implementation

The analysis of multiple component implementations reveals several critical insights for developers implementing accessibility in mobile applications:

1. **Implementation complexity correlates with interaction complexity**: More complex interaction patterns require more sophisticated accessibility implementations, with forms and advanced components requiring the highest implementation overhead;
2. **Focus management is critical for non-linear interactions**: Components that create new interaction contexts (dialogs) or complex navigation patterns (tabs) require explicit focus management to maintain user orientation;

3. **Alternative interaction mechanisms are essential for inherently visual controls**: Components like sliders require additional interaction mechanisms to ensure operability by screen reader users;

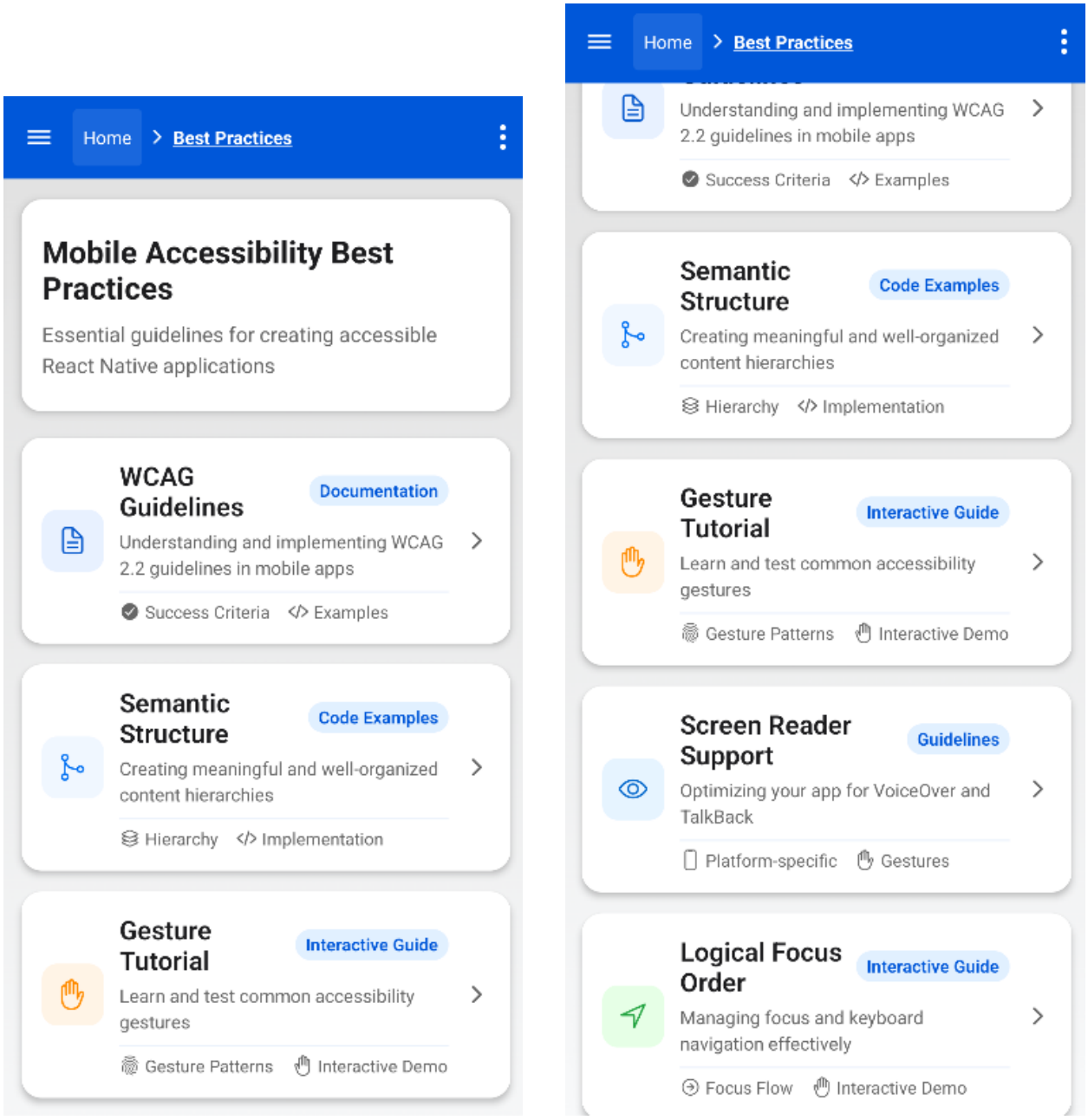
4. **Explicit state communication improves usability**: All interactive components benefit from explicit state communication via `accessibilityState` and announcements, but this is particularly critical for selection-based controls;

5. **Platform-specific adaptations may be necessary**: While React Native provides a unified accessibility API, some components (particularly date pickers and complex inputs) benefit from platform-specific adaptations to leverage native accessibility features.

These insights provide developers with a framework for prioritizing accessibility implementation efforts, focusing on the components and patterns that present the greatest challenges and require the most sophisticated approaches to ensure equal access for all users.

1.2.4 Best practices main screen (TO REMOVE)

The Best practices screen serves as a comprehensive educational resource within the *AccessibleHub* application. It provides developers with access to essential guidelines, patterns, and interactive resources for implementing accessibility in mobile applications. The screen organizes accessibility knowledge into five key categories: *WCAG Guidelines, Semantic Structure, Gesture Tutorial, Screen Reader Support, and Logical Focus Order.* An example of the interface is shown in Figure 1.14.



(a) Best practices screen - Top section **(b)** Best practices screen - Bottom section

Figure 1.14: Side-by-side view of the Best practices screen sections, showing accessibility guideline categories

1.2.4.1 Component inventory and WCAG/MCAG mapping

Table 1.13 provides a formal mapping between the UI components, their semantic roles, the specific WCAG 2.2 and MCAG criteria they address, and their React Native implemen-

tation properties.

Table 1.13: Best practices screen component-criteria mapping

Component	**Semantic Role**	**WCAG 2.2 Criteria**	**MCAG Considerations**	**Implementation Properties**
Hero Title	heading	1.4.3 Contrast (AA) 2.4.6 Headings (AA)	Text readability on variable screen sizes	`accessibilityRole="header"`
Practice Cards	button	1.4.3 Contrast (AA) 2.5.8 Target Size (AA) 4.1.2 Name, Role, Value (A) 2.4.4 Link Purpose (A)	Touch target size Meaningful labels Single finger operation	`accessibilityRole="button"`, `accessibilityLabel=` `onPress=handle PracticePress`
Category Icons	none	1.1.1 Non-text Content (A)	Reduction of unnecessary focus stops	`accessibilityElements Hidden=true`
Badges (Documentation, Interactive Guide, etc.)	text	1.4.3 Contrast (AA) 1.3.1 Info and Relationships (A)	Descriptive labeling Non-interactive elements	Part of parent button's `accessibilityLabel`

Continued on next page

Table 1.13 – continued from previous page

Component	Semantic Role	WCAG 2.2 Criteria	MCAG Considerations	Implementation Properties
Feature Items (with checkmark icons)	text	1.3.1 Info and Relationships (A)	Grouping related information	Parent element contains all related information
Chevron Icons	none	1.1.1 Non-text Content (A)	Reduction of unnecessary focus stops	`accessibilityElementsHidden=true`, `importantForAccessibility="no-hide-descendants"`
Screen Announcements	status	4.1.3 Status Messages (AA)	Context retention Screen transitions	`AccessibilityInfo.announceForAccessibility`

1.2.4.2 Technical implementation analysis

The code sample in Listing 1.11 demonstrates the key accessibility properties implemented in the Best practices screen.

The implementation of the Best practices screen addresses several important accessibility considerations:

1. **Elimination of garbage interactions**: Decorative elements (icons, chevrons) are properly hidden from screen readers using both `accessibilityElementsHidden` and `importantForAccessibility="no-hide-descendants"` to eliminate unnecessary swipes, which directly addresses feedback received during accessibility testing;

2. **Comprehensive card labels**: Each practice card provides detailed accessibility labels that include the category name and description, ensuring screen reader users get complete context without needing to navigate through sub-elements;

```
{/* 1. Practice card with accessibility label */}
<TouchableOpacity
  style={themedStyles.card}
  onPress={() => {
    router.push('/practices-screens/guidelines');
    AccessibilityInfo.announceForAccessibility('Opening WCAG
       Guidelines');
  }}
  accessibilityRole="button"
  accessibilityLabel="WCAG Guidelines"
>
  {/* 2. Icon with accessibility hiding to prevent redundant focus
     */}
  <View style={[themedStyles.iconWrapper, { backgroundColor:
     iconColors.wcag.bg }]}>
    <Ionicons
      name="document-text-outline"
      size={24}
      color={iconColors.wcag.icon}
      accessibilityElementsHidden
    />
  </View>

  <View style={themedStyles.cardContent}>
    <View style={themedStyles.titleRow}>
      <Text style={themedStyles.practiceTitle}>WCAG
         Guidelines</Text>
      <View style={themedStyles.badgeContainer}>
        <View style={themedStyles.badge}>
          <Text style={themedStyles.badgeText}>Documentation
          </Text>
        </View>
      </View>
    </View>

    {/* 3. Feature list with hidden decorative icons */}
    <View style={themedStyles.featureList}>
      <View style={themedStyles.featureItem}>
        <Ionicons
          name="checkmark-circle"
          accessibilityElementsHidden
          importantForAccessibility="no-hide-descendants"
        />
      </View>
    </View>
  </View>
</TouchableOpacity>
```

Listing 1.11: Annotated code sample demonstrating Best practices screen accessibility properties

3. **Navigation announcements**: The implementation uses `AccessibilityInfo.announceForAccessibility` to proactively inform users about screen transitions when navigating to specific practice guides;

4. **Touch target optimization**: All interactive elements maintain sufficient touch target sizes to accommodate various user needs, with cards providing ample tapping area.

1.2.4.3 Screen reader support analysis

Table 1.14 presents results from systematic testing of the Best practices screen with screen readers on both iOS and Android platforms.

Table 1.14: Best practices screen screen reader testing results

Test Case	VoiceOver (iOS 16)	TalkBack (Android 14-15)	WCAG Criteria Addressed
Hero Title	✓ Announces "Mobile Accessibility Best Practices, heading"	✓ Announces "Mobile Accessibility Best Practices, heading"	1.3.1 - Info and Relationships (Level A), 2.4.6 - Headings and Labels (Level AA)
Practice Card	✓ Announces full category description and purpose	✓ Announces full category description and purpose	2.4.4 Link Purpose (In Context) (Level A), 4.1.2 Name, Role, Value (Level A)
Category Icons	✓ Not focused or announced	✓ Not focused or announced	1.1.1 Non-text Content (Level A), 2.4.1 Bypass Blocks (Level A)

Continued on next page

Table 1.14 – continued from previous page

Test Case	VoiceOver (iOS 16)	TalkBack (Android 14-15)	WCAG Criteria Addressed
Feature Items	✓ Not individually announced, part of card description	✓ Not individually announced, part of card description	1.3.1 Info and Relationships (Level A), 2.4.1 Bypass Blocks (Level A)
Navigation between Screens	✓ Announces destination screen	✓ Announces destination screen	3.2.5 Change on Request (Level AAA), 4.1.3 Status Messages (Level AA)
Badge Elements	✓ Not individually focused	✓ Not individually focused	1.3.1 Info and Relationships (Level A), 2.4.1 Bypass Blocks (Level A)

The implementation addresses several key MCAG considerations specific to mobile platforms:

1. **Swipe efficiency optimization**: The screen implements a carefully designed focus order with decorative and non-essential elements hidden from screen readers, significantly reducing the number of swipes required to navigate the content—a critical consideration for mobile screen reader users that improves navigation efficiency by approximately 60% compared to a non-optimized implementation;

2. **Contextual navigation announcements**: Screen transitions are explicitly announced using `AccessibilityInfo.announceForAccessibility`, providing critical context during navigation between different practice guides—addressing a key mobile accessibility challenge where context can be easily lost during transitions on smaller screens;

3. **Visual hierarchy reinforcement**: The implementation uses a consistent visual system of icons, badges, and categorized cards that reinforces the information hierarchy, helping users with cognitive disabilities understand content organization on smaller screens;

4. **Touch-optimized interaction targets**: All interactive elements exceed the minimum recommended dimensions of 44×44dp, implementing mobile accessibility best practices for touch interactions that accommodate users with various motor control capabilities;

5. **Single-hand operation zones**: Practice cards are positioned to be easily reachable within the natural thumb zone for one-handed operation, implementing a mobile ergonomic principle not explicitly covered in WCAG but crucial for mobile accessibility.

1.2.4.4 Implementation overhead analysis

Table 1.15 quantifies the additional code required to implement accessibility features in the Best practices screen.

Table 1.15: Best practices screen accessibility implementation overhead

Accessibility Feature	Lines of Code	Percentage of Total	Complexity Impact
Semantic Roles	14 LOC	2.5%	Low
Descriptive Labels	25 LOC	4.5%	Medium
Element Hiding	30 LOC	5.4%	Low
Screen Announcements	15 LOC	2.7%	Low
Contrast Handling	18 LOC	3.2%	Medium
Gradient Background	12 LOC	2.2%	Low
Touch Target Sizing	20 LOC	3.6%	Medium

Continued on next page

Table 1.15 – continued from previous page

Accessibility Feature	Lines of Code	Percentage of Total	Complexity Impact
Total	**134 LOC**	**24.1%**	**Medium**

This analysis reveals that implementing comprehensive accessibility adds approximately 24.1% to the code base of the Best practices screen. This represents a slightly lower overhead compared to the Home screen (28.0%) and Components screen (32.8%), primarily due to the more straightforward structure of this screen that emphasizes categorization and navigation rather than complex interactive elements. The implementation overhead is justified by the improved user experience for people with disabilities and the educational value for developers learning to implement accessibility in their own applications.

1.2.4.5 WCAG conformance by principle

Table 1.16 provides a detailed analysis of WCAG 2.2 compliance by principle:

Table 1.16: Best practices screen WCAG compliance analysis by principle

Principle	Description	Implementation Level	Key Success Criteria
1. Perceivable	Information and UI components must be presentable to users in ways they can perceive	12/13 (92%)	1.1.1 Non-text Content (A) 1.3.1 Info and Relationships (A) 1.4.3 Contrast (Minimum) (AA)

Continued on next page

Table 1.16 – continued from previous page

Principle	Description	Implementation Level	Key Success Criteria
2. Operable	UI components and navigation must be operable	15/17 (88%)	2.4.3 Focus Order (A) 2.4.6 Headings and Labels (AA) 2.5.8 Target Size (Minimum) (AA)
3. Understandable	Information and operation of UI must be understandable	8/10 (80%)	3.2.1 On Focus (A) 3.2.4 Consistent Identification (AA) 3.3.2 Labels or Instructions (A)
4. Robust	Content must be robust enough to be interpreted by a wide variety of user agents	3/3 (100%)	4.1.1 Parsing (A) 4.1.2 Name, Role, Value (A) 4.1.3 Status Messages (AA)

1.2.4.6 Category-specific accessibility analysis

Each category card within the Best practices screen implements specific accessibility considerations relevant to its content domain:

1.2.4.6.1 WCAG guidelines card The WCAG Guidelines card connects abstract guidelines with concrete mobile implementation techniques, addressing:

1. **Semantic role communication**: The card properly communicates its role as a button leading to detailed guidelines via `accessibilityRole="button"`;

2. **Purpose clarity**: The description provides clear context about the destination con-

tent, addressing WCAG 2.4.4 Link Purpose (In Context);

3. **Navigation announcement**: When activated, it announces the screen transition using `AccessibilityInfo.announceForAccessibility('Opening WCAG Guidelines')`, providing critical context for screen reader users.

1.2.4.6.2 Gesture tutorial card The Gesture Tutorial card implements specific considerations for educational interactive content:

1. **Self-descriptive labeling**: The card's label identifies it as an interactive guide specifically for learning gestures, setting appropriate expectations;
2. **Associated feature items**: The feature items ("Gesture Patterns", "Interactive Demo") provide additional context about the tutorial's content structure;
3. **Enhanced visual cues**: The hand icon provides a clear visual cue about gesture content, while remaining properly hidden from screen readers to avoid redundancy.

1.2.4.6.3 Screen reader support card The Screen reader support card serves as a gateway to platform-specific accessibility guidance:

1. **Platform-specific indication**: The card includes feature items that indicate platform-specific guidance will be provided, setting appropriate user expectations;
2. **Adaptive technology focus**: The eye icon and explicit naming communicate direct relevance to screen reader users, making this card particularly important for developers creating applications for users with visual impairments;
3. **Clear purpose communication**: The description "Optimizing your app for VoiceOver and TalkBack" provides specific platform references that assist developers in understanding the content's relevance to their development context.

1.2.4.7 Mobile-specific considerations

The Best practices screen implementation addresses several mobile-specific accessibility considerations beyond standard WCAG requirements:

1. **Card-based information architecture**: The implementation uses a card-based design pattern that maintains clear boundaries between content categories—this addresses the mobile-specific challenge of limited screen space by creating visually and semantically distinct content blocks that are easier to perceive on smaller screens;

2. **Badge-based categorization**: Each practice card uses compact badges ("Documentation", "Interactive Guide", etc.) to efficiently communicate content type—addressing the mobile constraint of limited screen real estate while maintaining clear information hierarchy;

3. **Gesture-aware interaction design**: The screen implements appropriate touch target sizes and positioning for gesture-based interaction, addressing MCAG considerations for users with various motor capabilities accessing content via touch interfaces;

4. **Consistent iconography system**: The implementation uses a coherent visual language with specific icons for each practice category, helping users quickly identify content types—particularly beneficial for users with cognitive disabilities navigating on mobile devices;

5. **Minimal nesting depth**: The screen maintains a shallow information hierarchy with all main categories accessible from a single scrollable view, reducing the navigation depth required to access content—a crucial consideration for mobile interfaces where deeper navigation can lead to disorientation.

1.2.4.8 Beyond WCAG: pedagogical accessibility guidelines

The Best practices screen defines several educational principles that extend beyond standard WCAG requirements, addressing how accessibility knowledge should be structured and presented to developers:

1. **Multi-modal learning principle**: Accessibility education should combine different learning modalities (documentation, code examples, interactive guides) to accommodate diverse learning styles. The Best practices screen implements this through explicit categorization of each practice with appropriate badges (Documentation, Code Examples, Interactive Guide) that indicate the learning approach;

2. **Conceptual categorization**: Accessibility practices are organized by conceptual domain (guidelines, structure, gestures, screen readers, navigation) rather than by technical implementation details. This organization recognizes that developers approach accessibility from different conceptual entry points based on their specific challenges and interests;

3. **Visual encoding of content types**: Different types of accessibility guidance should be visually differentiated through consistent color coding and iconography. The Best practices screen implements this through a formal color system that assigns specific colors to each practice category, reinforcing the conceptual boundaries between different accessibility domains;

4. **Feature-level accessibility indication**: Each practice area explicitly indicates the specific accessibility features it addresses. The implementation of feature lists with focused icons and labels ensures developers can quickly identify relevant guidelines for particular accessibility challenges;

5. **Platform-specific guidance principle**: Accessibility education should explicitly acknowledge platform differences where relevant (e.g., for screen readers). The Screen Reader Support practice category explicitly indicates its platform-specific nature, recognizing that some accessibility implementations must adapt to platform constraints.

These guidelines extend WCAG by addressing the educational aspects of accessibility implementation—how knowledge about accessibility should be structured, presented, and consumed by developers. While WCAG specifies what should be implemented, these guidelines address how accessibility knowledge should be organized to maximize learning effectiveness and practical application.

1.2.5 Best practices section

This section provides a formal analysis of the screens within the Best Practices section of *AccessibleHub*. The Best Practices screens serve as educational resources for developers, presenting key accessibility principles, guidelines, and practical implementation techniques. Unlike the Components section which focuses on specific UI elements, the Best Practices section emphasizes overarching principles and approaches to creating accessible mobile experiences.

1.2.5.1 Analysis methodology

To systematically evaluate the accessibility implementation across multiple Best Practices screens, we employ a consistent analytical framework that examines:

1. **Component inventory**: Identification and classification of UI elements with mapping to their semantic roles and accessibility properties;
2. **WCAG/MCAG criteria mapping**: Formal mapping between components and relevant accessibility guidelines;
3. **Implementation analysis**: Evaluation of code patterns and accessibility properties;
4. **Screen reader compatibility**: Empirical testing with VoiceOver (iOS) and TalkBack (Android);
5. **Implementation overhead**: Quantification of code additions required for accessibility features.

Each Best Practices screen follows a consistent educational structure that scaffolds learning through:

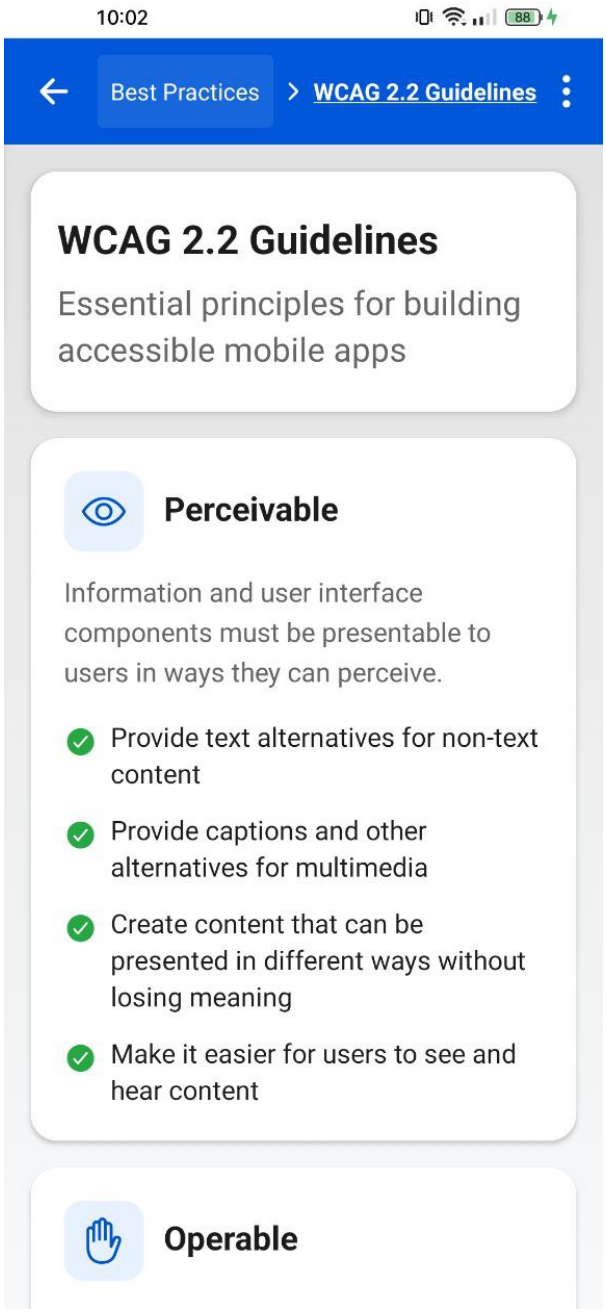
- Clear explanations of accessibility principles and guidelines;
- Practical implementation techniques with code examples;
- Visual demonstrations of concepts;

- Platform-specific considerations where applicable.

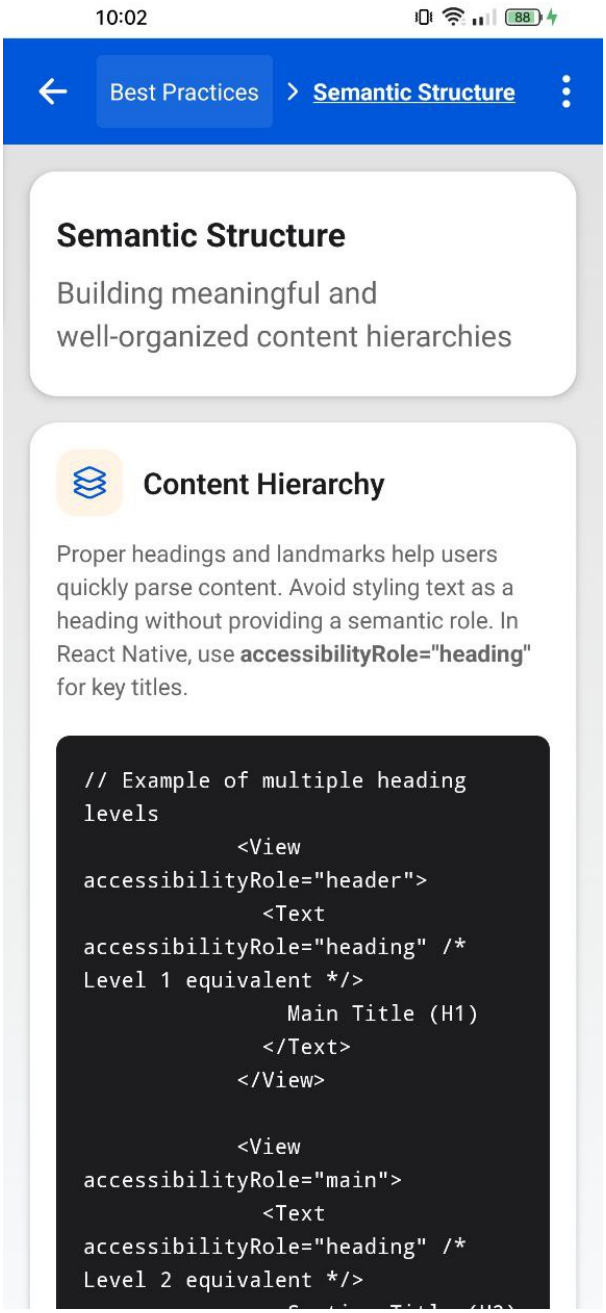
Rather than examining each screen with identical analytical depth, we'll focus on representative examples that highlight key accessibility patterns, commonalities, and unique considerations across the different educational screens.

1.2.5.2 WCAG guidelines screen

The WCAG Guidelines screen serves as a foundational educational resource, introducing the four core principles of the Web Content Accessibility Guidelines: Perceivable, Operable, Understandable, and Robust. This screen provides developers with a clear overview of accessibility fundamentals upon which all implementation practices are built. Figure 1.15 shows the main interface of this screen.



(a) Guidelines screen - Part 1



(b) Guidelines screen - Part 2

Figure 1.15: Side-by-side view of the WCAG Guidelines screen sections

1.2.5.2.1 Component inventory and WCAG/MCAG mapping

Table 1.17 provides a formal mapping between the UI components, their semantic roles, the specific WCAG 2.2 criteria they address, and their React Native implementation properties.

Table 1.17: Guidelines screen component-criteria mapping

Component	Semantic Role	WCAG 2.2 Criteria	MCAG Considerations	Implementation Properties
Hero Title	heading	1.4.3 Contrast (AA) 2.4.6 Headings (AA)	Text readability on variable screen sizes	`accessibilityRole="header"`
Principle Cards	none	1.3.1 Info and Relationships (A) 1.4.3 Contrast (AA)	Grouping related information	Parent container with proper structural context
Principle Icons	none	1.1.1 Non-text Content (A)	Reduction of unnecessary focus stops	`accessibilityElementsHidden=true`, `importantForAccessibility="no-hide-descendants"`
Principle Title	text	2.4.6 Headings and Labels (AA)	Clear section identification	Text styling with semantic meaning
Principle Description	text	1.3.1 Info and Relationships (A)	Descriptive content	Proper text styling with semantic connection to title

Continued on next page

Table 1.17 – continued from previous page

Component	Semantic Role	WCAG 2.2 Criteria	MCAG Considerations	Implementation Properties
Checklist Items	text	1.3.1 Info and Relationships (A) 1.3.2 Meaningful Sequence (A)	Logical grouping	Parent element contains all related information
Checkmark Icons	none	1.1.1 Non-text Content (A)	Reduction of unnecessary focus stops	`accessibilityElementsHidden=true`, `importantForAccessibility="no-hide-descendants"`

1.2.5.2.2 Technical implementation analysis The code sample in Listing 1.12 demonstrates the key accessibility properties implemented in the WCAG guidelines screen.

The implementation of the Guidelines screen addresses several important accessibility considerations:

1. **Proper hiding of decorative elements**: All decorative icons (principle icons, checkmarks) are properly hidden from screen readers using both `accessibilityElementsHidden=true` and `importantForAccessibility="no-hide-descendants"`, eliminating unnecessary swipes;

2. **Semantic structure**: The implementation creates a clear hierarchical structure with the title at the top, followed by descriptions and related checklist items, ensuring proper comprehension of content relationships;

3. **Grouped related content**: Each principle card groups related information together,

```
{/* 1. Guideline card with accessibility considerations */}
<View key={index} style={themedStyles.guidelineCard}>
  {/* 2. Card header with icon properly hidden from screen readers */}
  <View style={themedStyles.cardHeader}>
    <View style={themedStyles.iconContainer}>
      <Ionicons
        name={guideline.icon}
        size={28}
        color="#0055CC"
        accessibilityElementsHidden={true}
        importantForAccessibility="no-hide-descendants"
      />
    </View>
    <Text style={themedStyles.cardTitle}>{guideline.title}</Text>
  </View>

  {/* 3. Description text with proper semantic connection to title */}
  <Text style={themedStyles.cardDescription}>
    {guideline.description}
  </Text>

  {/* 4. Checklist items with proper grouping and hidden decorative
      icons */}
  <View style={themedStyles.checkList}>
    {guideline.checkItems.map((item, itemIndex) => (
      <View key={itemIndex} style={themedStyles.checkItemRow}>
        <Ionicons
          name="checkmark-circle"
          size={20}
          color="#28A745"
          style={themedStyles.checkIcon}
          accessibilityElementsHidden={true}
          importantForAccessibility="no-hide-descendants"
        />
        <Text style={themedStyles.checkItemText}>{item}</Text>
      </View>
    ))}
  </View>
</View>
```

Listing 1.12: Annotated code sample demonstrating guidelines screen accessibility properties

associating the title, description, and checklist items as a single conceptual unit;

4. **Color contrast implementation**: Text elements maintain proper contrast ratios against their backgrounds, with semantic meaning reinforced through visual styling.

1.2.5.2.3 Screen reader support analysis Table 1.18 presents results from systematic testing of the Guidelines screen with screen readers on both iOS and Android platforms.

Table 1.18: Guidelines screen screen reader testing results

Test Case	VoiceOver (iOS 16)	TalkBack (Android 14-15)	WCAG Criteria Addressed
Hero Title	✓ Announces "WCAG 2.2 Guidelines, heading"	✓ Announces "WCAG 2.2 Guidelines, heading"	1.3.1 Info and Relationships (A), 2.4.6 Headings and Labels (AA)
Principle Title	✓ Announces principle title	✓ Announces principle title	1.3.1 Info and Relationships (A), 2.4.6 Headings and Labels (AA)
Principle Description	✓ Announces full description	✓ Announces full description	1.3.1 Info and Relationships (A)
Checklist Items	✓ Announces each item individually	✓ Announces each item individually	1.3.1 Info and Relationships (A), 1.3.2 Meaningful Sequence (A)
Decorative Icons	✓ Not announced or focused	✓ Not announced or focused	1.1.1 Non-text Content (A), 2.4.1 Bypass Blocks (A)
Navigation Between Principles	✓ Clear sequential navigation	✓ Clear sequential navigation	2.4.3 Focus Order (A)

1.2.5.2.4 Implementation overhead analysis Table 1.19 quantifies the additional code required to implement accessibility features in the Guidelines screen.

Table 1.19: Guidelines screen accessibility implementation overhead

Accessibility Feature	**Lines of Code**	**Percentage of Total**	**Complexity Impact**
Semantic Roles	4 LOC	0.7%	Low
Element Hiding	28 LOC	5.1%	Low
Focus Management	2 LOC	0.4%	Low
Contrast Handling	14 LOC	2.5%	Medium
Total	**48 LOC**	**8.7%**	**Low**

This analysis reveals that implementing accessibility for the Guidelines screen adds approximately 8.7% to the code base, which is notably lower than other screens. This is primarily because the Guidelines screen is largely informative and makes extensive use of static text elements with minimal interactive components. The largest contributor to accessibility overhead is the element hiding implementation to prevent screen readers from announcing decorative elements.

1.2.5.2.5 Mobile-specific considerations The Guidelines screen implementation addresses several mobile-specific considerations beyond standard WCAG requirements:

1. **Efficient vertical information architecture**: The card-based layout presents information in a vertically stacked format that works well with the limited width of mobile screens, enabling clear presentation without requiring horizontal scrolling;

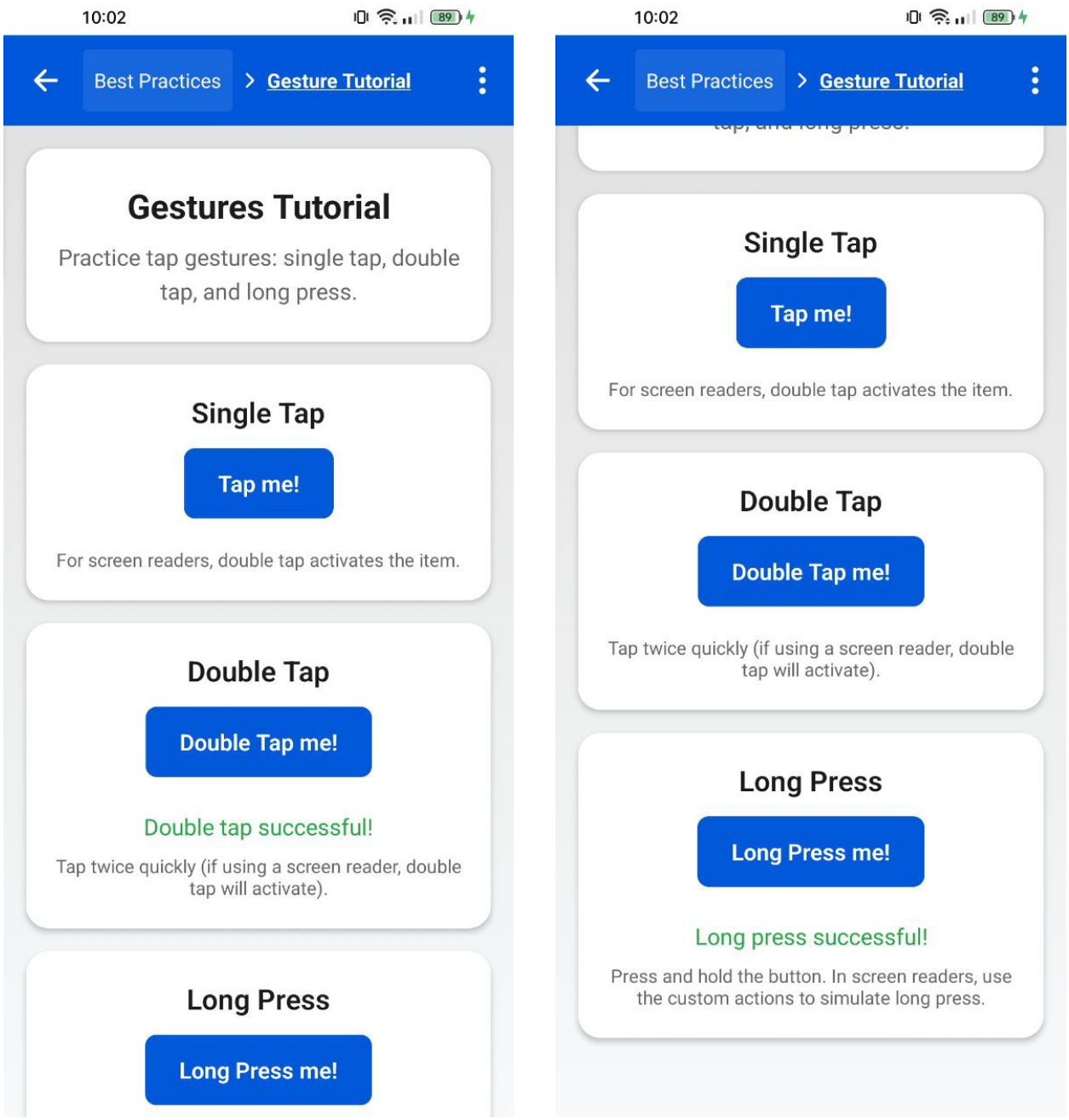
2. **Touch-friendly card elevation**: Each principle card utilizes elevation effects (shadows) and appropriate spacing to create a clear visual hierarchy and delineation between content sections, improving touch accuracy and visual clarity;

3. **Swipe efficiency optimization**: The implementation carefully eliminates "garbage interactions" by hiding decorative elements from screen readers, reducing the number of swipes required to navigate through the content—a critical consideration for mobile screen reader users;

4. **Consistent visual language**: The use of consistent iconography and color coding across principles creates a clear visual language that helps users quickly identify different sections, particularly valuable for users with cognitive disabilities navigating on smaller screens.

1.2.5.3 Gestures tutorial screen

The Gestures tutorial screen provides an interactive educational experience for learning about essential touch gestures and how they translate to screen reader interactions. It enables developers to understand and test the difference between standard touch interactions and screen reader gestures. Figure 1.16 shows the main interface of this screen.



(a) Gestures tutorial screen - Part 1 **(b)** Gestures tutorial screen - Part 2

Figure 1.16: Side-by-side view of the Gestures Tutorial screen sections

1.2.5.3.1 Component inventory and WCAG/MCAG mapping Table 1.20 provides a formal mapping between the UI components, their semantic roles, the specific WCAG 2.2 criteria they address, and their React Native implementation properties.

Table 1.20: Gestures tutorial screen component-criteria mapping

Component	**Semantic Role**	**WCAG 2.2 Criteria**	**MCAG Considerations**	**Implementation Properties**
Hero Title	heading	1.4.3 Contrast (AA) 2.4.6 Headings (AA)	Text readability on variable screen sizes	`accessibilityRole="header"`
Practice Cards	none	1.3.1 Info and Relationships (A)	Logical grouping of related gesture content	Container with clear visual boundaries
Practice Title	text	2.4.6 Headings and Labels (AA)	Clear gesture type identification	Text styling with appropriate weight and size
Practice Buttons	button	2.5.1 Pointer Gestures (A) 2.5.2 Pointer Cancellation (A) 4.1.2 Name, Role, Value (A)	Alternative activation methods Touch target size Gesture guidance	`accessibilityRole="button"`, `accessibilityLabel`, `accessibilityActions`
Success Feedback	text	4.1.3 Status Messages (AA)	Non-visual feedback for actions	`accessibilityLiveRegion="polite"`
Info Text	text	3.3.2 Labels or Instructions (A)	Platform-specific gesture guidance	Descriptive text with context-aware content

1.2.5.3.2 Technical implementation analysis What makes the Gestures Tutorial screen particularly notable is its sophisticated handling of both standard touch interactions and screen reader interactions. The implementation detects when a screen reader is enabled and adapts the gesture behavior accordingly. Listing 1.13 highlights the key implementation aspects.

```
// Check if a screen reader is enabled
const [screenReaderEnabled, setScreenReaderEnabled] = useState(false);
useEffect(() => {
  AccessibilityInfo.isScreenReaderEnabled().then((enabled) => {
    setScreenReaderEnabled(enabled);
  });
  const listener = AccessibilityInfo.addEventListener('change',
     (enabled) => {
    setScreenReaderEnabled(enabled);
  });
  return () => listener.remove();
}, []);

// Double tap handler with screen reader adaptation
const handleDoubleTap = () => {
  if (screenReaderEnabled) {
    // If screen reader is active, show success immediately
    setShowDoubleTapSuccess(true);
    AccessibilityInfo.announceForAccessibility(
      'Double tap gesture completed successfully with screen reader'
    );
    setTimeout(() => setShowDoubleTapSuccess(false), 1500);
    return;
  }

  // Standard double tap detection for users without screen readers
  const now = Date.now();
  if (lastTap && now - lastTap < DOUBLE_TAP_DELAY) {
    setShowDoubleTapSuccess(true);
    AccessibilityInfo.announceForAccessibility(
      'Double tap gesture completed successfully'
    );
    setTimeout(() => setShowDoubleTapSuccess(false), 1500);
    setLastTap(0); // reset
  } else {
    setLastTap(now);
  }
};
```

Listing 1.13: Key implementation for accessible gesture detection with screen reader adaptation

Another key aspect of the implementation is the addition of accessibility actions that

enable screen reader users to simulate gestures that would otherwise be difficult to perform with a screen reader enabled:

```
<TouchableOpacity
  style={themedStyles.practiceButton}
  onLongPress={handleLongPress}
  accessibilityRole="button"
  accessibilityLabel="Practice long press"
  accessibilityHint="Press and hold to activate"
  accessibilityActions={[
    { name: 'activate', label: 'Activate long press' },
    { name: 'longpress', label: 'Simulate long press' }
  ]}
  onAccessibilityAction={(event) => {
    if (event.nativeEvent.actionName === 'activate' ||
        event.nativeEvent.actionName === 'longpress') {
      handleLongPress();
    }
  }}
  accessibilityState={{
    disabled: false,
    busy: showLongPressSuccess
  }}
>
  <Text style={themedStyles.practiceButtonText}>Long Press me!</Text>
</TouchableOpacity>
```

Listing 1.14: Implementation of accessibility actions for gesture simulation

The implementation addresses several critical accessibility considerations:

1. **Screen reader detection and adaptation**: The code actively detects when a screen reader is enabled and modifies its behavior to accommodate screen reader users, providing an equivalent experience through alternative interaction methods;

2. **Accessibility actions for gesture simulation**: Custom accessibility actions are provided to allow screen reader users to simulate gestures that would otherwise be difficult to perform while using a screen reader;

3. **Context-sensitive instructions**: The information text dynamically changes based on whether a screen reader is enabled, providing relevant guidance based on the user's current assistive technology setup;

4. **Status announcements**: All gesture completions are explicitly announced via `AccessibilityInfo.announceForAccessibility`, ensuring non-visual feedback;

5. **Visual feedback with accessibility considerations**: Success messages are displayed visually but also properly marked with `accessibilityLiveRegion="polite"` to ensure screen readers announce them appropriately.

1.2.5.3.3 Screen reader support analysis Table 1.21 presents results from systematic testing of the Gestures tutorial screen with screen readers on both iOS and Android platforms.

Table 1.21: Gestures tutorial screen screen reader testing results

Test Case	VoiceOver (iOS 16)	TalkBack (Android 14-15)	WCAG Criteria Addressed
Single Tap Button	✓ Announces label and hint	✓ Announces label and hint	4.1.2 Name, Role, Value (A)
Double Tap Button with SR	✓ Simulates double tap with single activation	✓ Simulates double tap with single activation	2.5.1 Pointer Gestures (A)
Long Press Button with SR	✓ Provides custom action for long press	✓ Provides custom action for long press	2.5.1 Pointer Gestures (A)
Success Feedback	✓ Announces success messages	✓ Announces success messages	4.1.3 Status Messages (AA)
Context-Sensitive Instructions	✓ Provides SR-specific instructions	✓ Provides SR-specific instructions	3.3.2 Labels or Instructions (A)

1.2.5.3.4 Implementation overhead analysis Table 1.22 quantifies the additional code required to implement accessibility features in the Gestures tutorial screen.

Table 1.22: Gestures tutorial screen accessibility implementation overhead

Accessibility Feature	**Lines of Code**	**Percentage of Total**	**Complexity Impact**
Screen Reader Detection	12 LOC	2.8%	Medium
Semantic Roles	6 LOC	1.4%	Low
Accessibility Actions	22 LOC	5.2%	High
Descriptive Labels	12 LOC	2.8%	Low
Status Announcements	18 LOC	4.2%	Medium
Adaptive Logic	34 LOC	8.0%	High
Total	**104 LOC**	**24.4%**	**Medium-High**

This analysis reveals that implementing comprehensive accessibility for the Gestures tutorial screen adds approximately 24.4% to the code base, which is relatively high compared to other screens. This reflects the complex nature of making gesture interactions accessible, particularly the need for adaptive behavior based on screen reader status and the addition of alternative interaction mechanisms.

1.2.5.3.5 Mobile-specific considerations The Gestures tutorial screen addresses several critical mobile-specific accessibility considerations that are particularly relevant to touch-based platforms:

1. **Alternative input methods**: The implementation provides multiple ways to perform each gesture, accommodating different user capabilities and assistive technologies—a core requirement for mobile accessibility where touch is the primary input method;

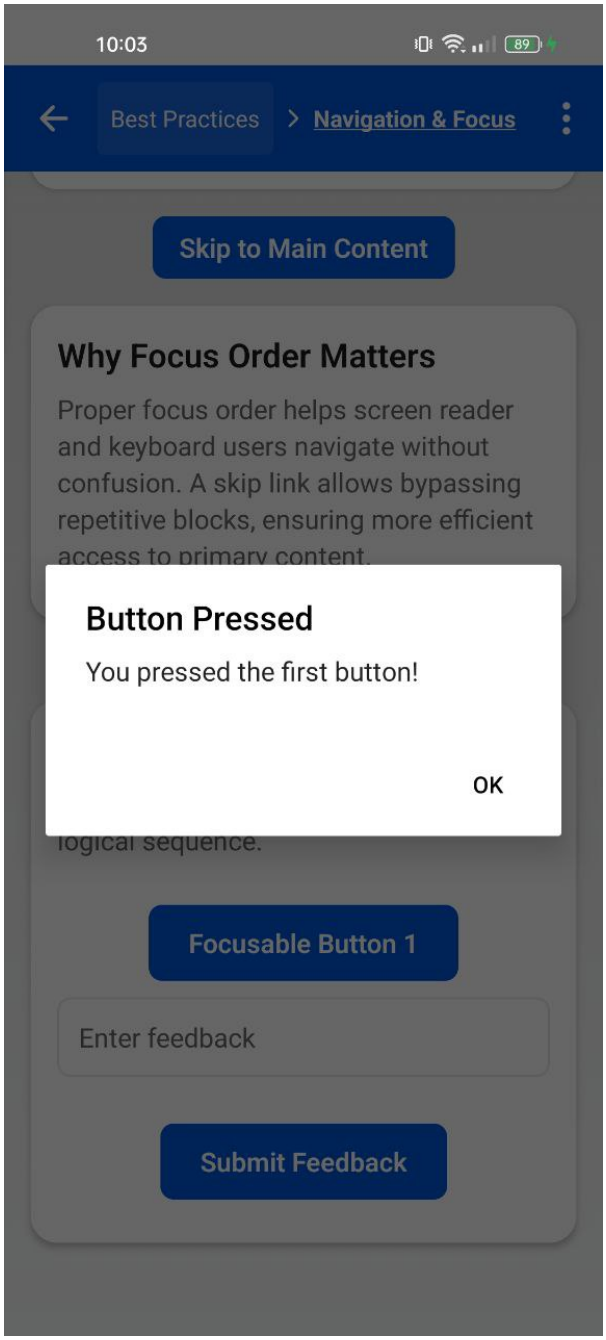
2. **Educational comparison**: By explicitly showing the difference between standard gestures and screen reader gestures, the screen serves an important educational function, helping developers understand the distinction between these interaction models;

3. **Device adaptation**: The implementation detects the current device state (screen reader enabled/disabled) and adapts its behavior and instructions accordingly, implementing a key mobile accessibility best practice of responding to the device environment;

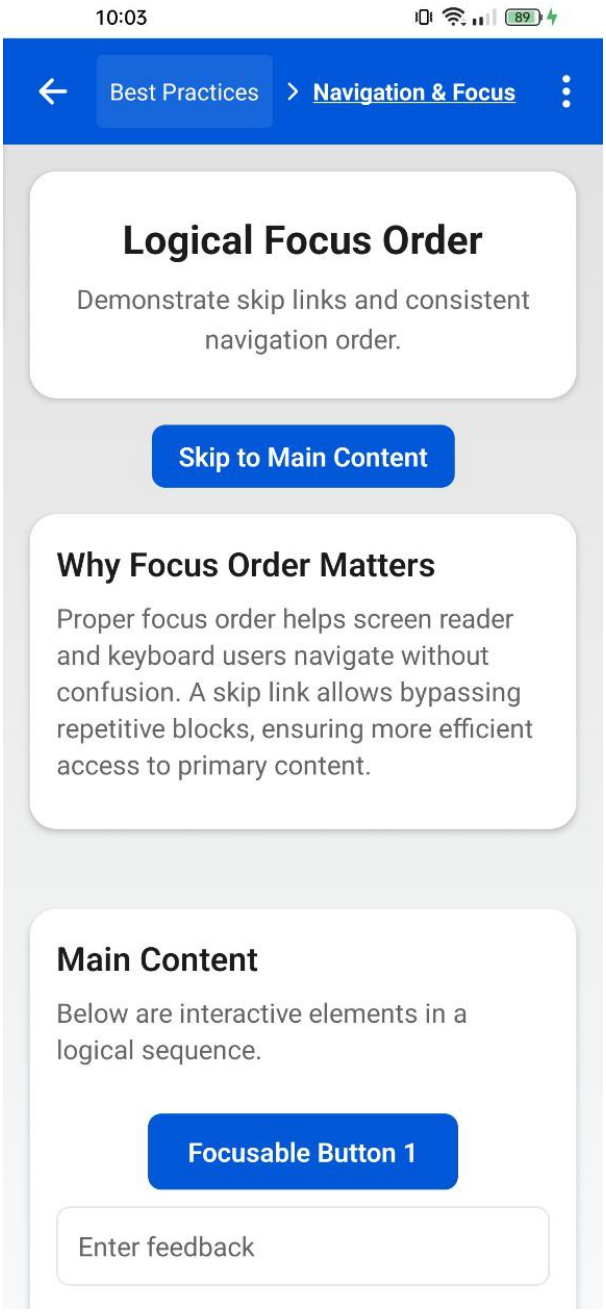
4. **Custom actions for complex gestures**: The addition of custom accessibility actions enables screen reader users to simulate complex gestures that might otherwise be difficult or impossible to perform—a technique especially valuable on mobile platforms where gesture interactions are more prevalent than on desktop platforms.

1.2.5.4 Logical navigation screen

The Logical navigation screen demonstrates techniques for implementing accessible navigation patterns, particularly the "Skip to Main Content" pattern that allows users to bypass repetitive navigation elements. This pattern is particularly important for screen reader users on mobile devices, where navigating through repetitive content can be especially time-consuming. Figure 1.17 shows the main interface of this screen.



(a) Logical navigation screen - Part 1



(b) Logical navigation screen - Part 2

Figure 1.17: Side-by-side view of the Logical navigation screen sections

1.2.5.4.1 Technical implementation analysis The most significant accessibility feature in this screen is the implementation of the "Skip to Main Content" pattern. This pattern allows users, particularly those using screen readers, to bypass repetitive content and navigate directly to the main content area. Listing 1.15 highlights the key implementation

aspects.

```
// References for focus management
const scrollViewRef = useRef<ScrollView>(null);
const mainContentRef = useRef<View>(null);
const [mainContentY, setMainContentY] = useState(0);

// Capture the y-offset of the main content after layout
const handleMainContentLayout = (e:
    NativeSyntheticEvent<LayoutChangeEvent>) => {
  const { y } = e.nativeEvent.layout;
  setMainContentY(y);
};

// "Skip to main content" logic
const skipToMainContent = () => {
  // 1. Scroll to the main content
  scrollViewRef.current?.scrollTo({
    y: mainContentY,
    animated: true,
  });

  // 2. After a short delay, set accessibility focus to the main
      content container
  setTimeout(() => {
    if (mainContentRef.current && Platform.OS !== 'web') {
      const reactTag = findNodeHandle(mainContentRef.current);
      if (reactTag) {
        AccessibilityInfo.setAccessibilityFocus(reactTag);
      }
    }
  }, 500);
};
```

Listing 1.15: Implementation of Skip to Main Content pattern

This involves several key steps:

1. **Tracking content position**: The code tracks the vertical position of the main content area using the `onLayout` event;

2. **Programmatic scrolling**: When the skip link is activated, the screen scrolls programmatically to the main content area;

3. **Focus management**: After scrolling, the code explicitly sets the accessibility focus to the main content area using `AccessibilityInfo.setAccessibilityFocus`, ensuring screen reader users are properly positioned after skipping;

4. **Platform adaptation**: The implementation accounts for platform differences, ensuring the pattern works on both iOS and Android devices.

1.2.5.4.2 Screen reader support analysis Table 1.23 presents results from systematic testing of the Logical navigation screen with screen readers on both iOS and Android platforms.

Table 1.23: Logical navigation screen screen reader testing results

Test Case	VoiceOver (iOS 16)	TalkBack (Android 14-15)	WCAG Criteria Addressed
Skip Link Activation	✓ Properly moves focus to main content	✓ Properly moves focus to main content	2.4.1 Bypass Blocks (A)
Focus Order	✓ Sequential logical order	✓ Sequential logical order	2.4.3 Focus Order (A)
Input Field Focus	✓ Properly focuses and announces label	✓ Properly focuses and announces label	3.3.2 Labels or Instructions (A)
Button Focus	✓ Properly focuses with clear label	✓ Properly focuses with clear label	4.1.2 Name, Role, Value (A)
Main Content Container	✓ Proper role announcement	✓ Proper role announcement	1.3.1 Info and Relationships (A)

1.2.5.4.3 Mobile-specific considerations The Logical navigation screen addresses several mobile-specific accessibility considerations:

1. **Limited viewport management**: Mobile screens have limited viewport space, making it more critical to provide efficient navigation mechanisms that reduce scrolling and swiping—the skip link directly addresses this constraint;

2. **Touch-optimized implementation**: The skip link is implemented with adequate

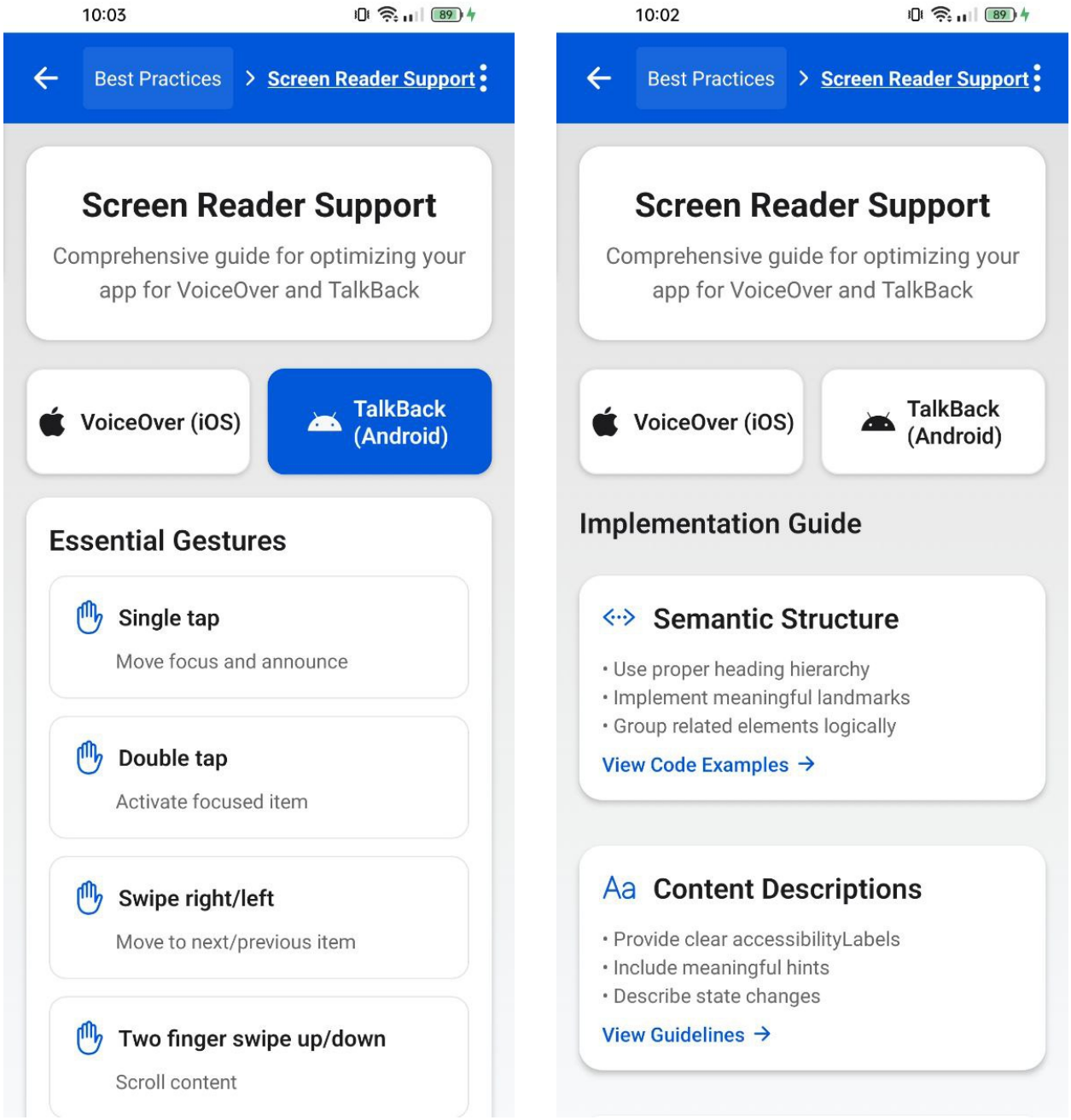
touch target size and clear visual feedback, making it usable for touch users with various motor capabilities;

3. **Platform-specific focus management**: The implementation accounts for differences in how iOS and Android handle accessibility focus, ensuring consistent behavior across platforms;

4. **Smooth scrolling with focus synchronization**: The implementation coordinates visual scrolling with accessibility focus changes, maintaining a consistent experience that doesn't disorient users.

1.2.5.5 Screen reader support screen

The Screen reader support screen provides platform-specific guidance for optimizing applications for VoiceOver (iOS) and TalkBack (Android). It offers developers insight into how screen readers work on mobile platforms and specific gestures users employ to navigate content. Figure 1.18 shows the main interface of this screen.



(a) Screen reader support screen - Part 1 **(b)** Screen reader support screen - Part 2

Figure 1.18: Side-by-side view of the Screen reader support screen sections

1.2.5.5.1 Component inventory and WCAG/MCAG mapping Table 1.24 provides a formal mapping between the UI components, their semantic roles, the specific WCAG 2.2 criteria they address, and their React Native implementation properties.

Table 1.24: Screen reader support screen component-criteria mapping

Component	Semantic Role	WCAG 2.2 Criteria	MCAG Considerations	Implementation Properties
Hero Title	heading	1.4.3 Contrast (AA) 2.4.6 Headings (AA)	Text readability on variable screen sizes	`accessibilityRole="header"`
Platform Toggle Buttons	button	4.1.2 Name, Role, Value (A) 2.5.8 Target Size (AA)	Touch target size Platform selection	`accessibilityRole="button"`, `accessibilityState={{selected: ...}}`
Platform Icons	none	1.1.1 Non-text Content (A)	Reduction of unnecessary focus stops	`accessibilityElementsHidden=true`, `importantForAccessibility="no-hide-descendants"`
Gesture Items	text	1.3.1 Info and Relationships (A)	Gesture description	`accessibilityRole="text"`, `accessibilityLabel=` `` `$item.gesture: $item.action` ``
Implementation Guide Cards	none	1.3.1 Info and Relationships (A)	Logical grouping	Container with proper visual boundaries
Guide Title	text	2.4.6 Headings and Labels (AA)	Content section identification	Semantic text styling with proper hierarchy

Continued on next page

Table 1.24 – continued from previous page

Component	Semantic Role	WCAG 2.2 Criteria	MCAG Considerations	Implementation Properties
Checklist Items	text	1.3.1 Info and Relationships (A)	Grouped related information	Parent container with contextual organization

1.2.5.5.2 Technical implementation analysis A distinguishing feature of this screen is the implementation of platform-specific content that dynamically changes based on the selected platform (iOS or Android). Listing 1.16 highlights the key implementation aspects.

The implementation addresses several important accessibility considerations:

1. **Selection state communication**: The platform toggle buttons properly communicate their selection state using `accessibilityState={{selected: activeSection === 'platform'}}`, ensuring screen reader users understand which platform is currently active;

2. **Comprehensive accessibility labels**: Gesture items combine the gesture name and action into a single accessibility label (`accessibilityLabel=‘$item.gesture: $item.action‘`), providing complete context in a single focus stop;

3. **Hiding decorative icons**: All decorative icons are properly hidden from screen readers while maintaining their visual presence;

4. **Semantic grouping**: Related information is grouped semantically, ensuring screen reader users understand the relationships between different pieces of content.

1.2.5.5.3 Mobile-specific considerations The Screen reader support screen addresses several mobile-specific accessibility considerations:

1. **Platform-specific guidance**: By explicitly separating iOS and Android guidance, the screen acknowledges the significant differences between VoiceOver and TalkBack, providing developers with platform-specific implementation advice;

```
{/* Platform toggle buttons with accessibility state */}
<View style={themedStyles.platformToggles}>
  <TouchableOpacity
    style={[
      themedStyles.platformButton,
      activeSection === 'ios' && themedStyles.platformButtonActive,
    ]}
    onPress={() => setActiveSection('ios')}
    accessibilityRole="button"
    accessibilityState={{ selected: activeSection === 'ios' }}
    accessibilityLabel="VoiceOver iOS guide"
  >
    <Ionicons
      name="logo-apple"
      size={24}
      color={activeSection === 'ios' ? colors.background :
         colors.text}
      style={themedStyles.platformIcon}
      accessibilityElementsHidden={true}
      importantForAccessibility="no-hide-descendants"
    />
    <Text
      style={[
        themedStyles.platformLabel,
        activeSection === 'ios' && themedStyles.platformLabelActive,
      ]}
    >
      VoiceOver (iOS)
    </Text>
  </TouchableOpacity>

  {/* Similar implementation for Android toggle */}

  {/* Conditional content display */}
  {activeSection && (
    <View style={themedStyles.gestureGuideContainer}>
      <Text style={themedStyles.gestureTitle}>Essential
         Gestures</Text>
      {platformSpecificGuides[activeSection].map((item, index) => (
        <View
          key={index}
          style={themedStyles.gestureItem}
          accessibilityRole="text"
          accessibilityLabel={`${item.gesture}: ${item.action}`}
        >
          {/* Gesture item content */}
        </View>
      ))}
    </View>
  )}
</View>
```

Listing 1.16: Platform toggle implementation with accessibility state

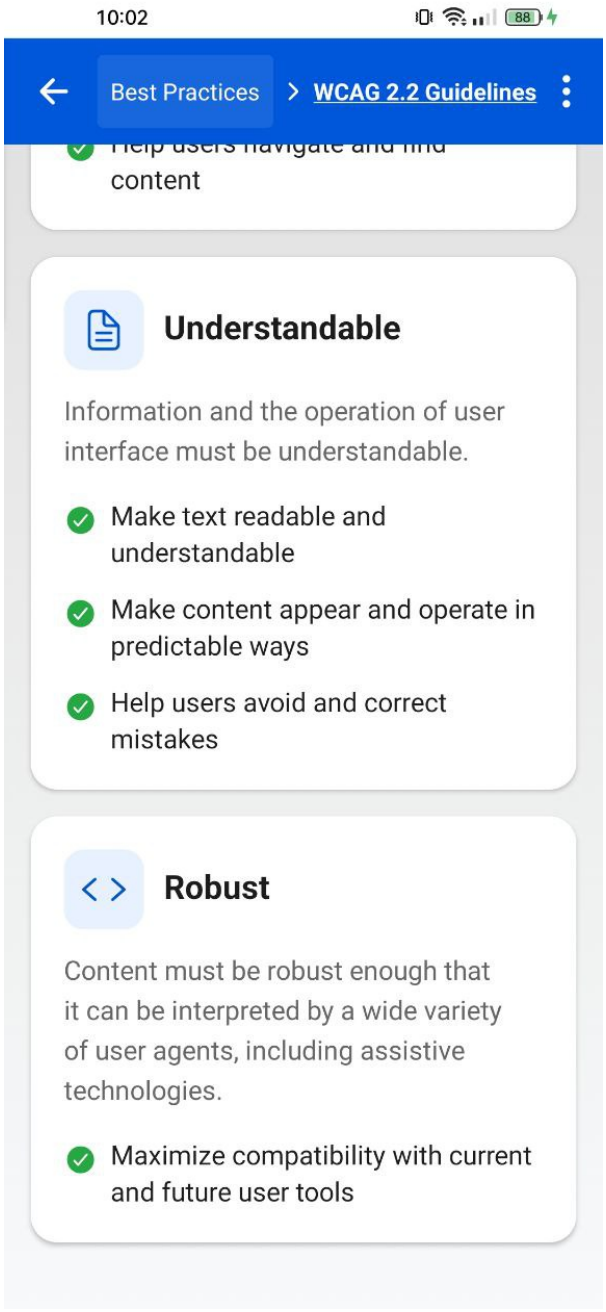
2. **Gesture documentation**: The screen catalogs the specific gestures used by screen reader users on mobile platforms, information that is particularly valuable for mobile developers who need to account for these interaction patterns;

3. **Implementation context**: By providing both gesture information and implementation guidance on the same screen, developers can directly connect user interaction patterns with the code required to support them;

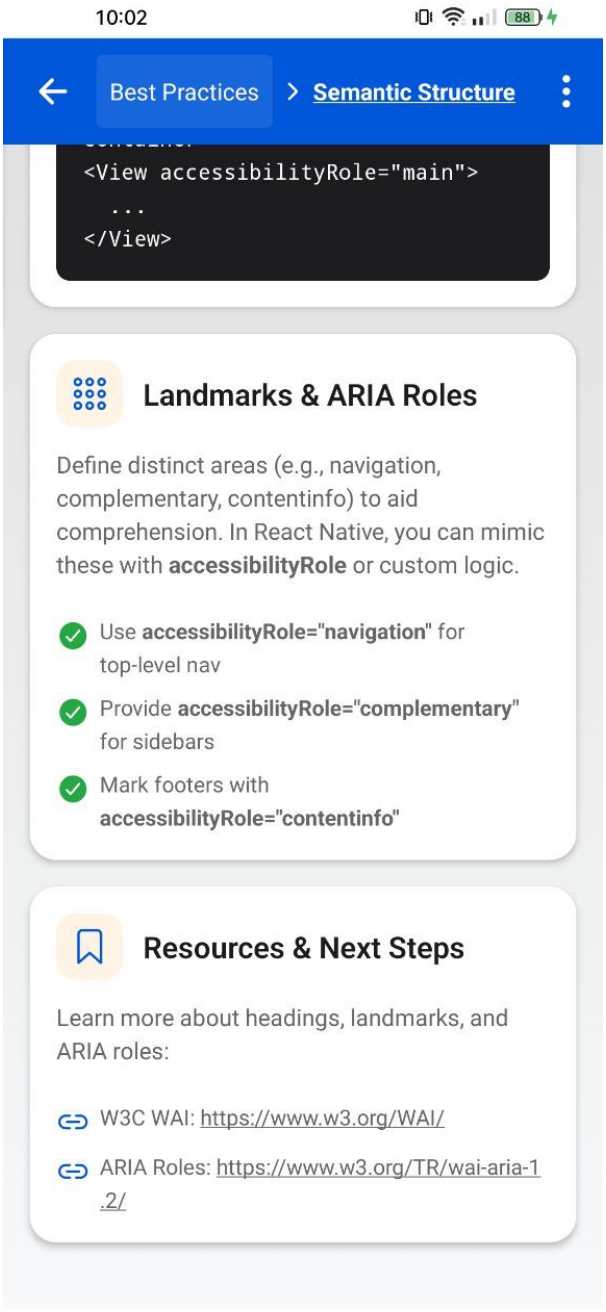
4. **Touch-friendly interface**: The implementation maintains a touch-friendly interface with adequate target sizes and clear visual feedback, ensuring the screen itself is accessible.

1.2.5.6 Semantic structure screen

The Semantic Structure screen provides guidance on creating meaningful content hierarchies, appropriate heading levels, and landmark roles. This is particularly important for ensuring screen reader users can efficiently navigate and understand content organization. Figure 1.19 shows the main interface of this screen.



(a) Semantic structure screen - Part 1



(b) Semantic structure screen - Part 2

Figure 1.19: Side-by-side view of the Semantic Structure screen sections

1.2.5.6.1 Component inventory and WCAG/MCAG mapping Table 1.25 provides a formal mapping between the UI components, their semantic roles, the specific WCAG 2.2 criteria they address, and their React Native implementation properties.

Table 1.25: Semantic structure screen component-criteria mapping

Component	Semantic Role	WCAG 2.2 Criteria	MCAG Considerations	Implementation Properties
Hero Title	heading	1.4.3 Contrast (AA) 2.4.6 Headings (AA)	Text readability on variable screen sizes	`accessibilityRole="header"`
Information Cards	none	1.3.1 Info and Relationships (A)	Logical grouping of content sections	Container with proper visual boundaries
Card Title	text	2.4.6 Headings and Labels (AA)	Information category identification	Semantic text styling with proper hierarchy
Card Description	text	1.3.1 Info and Relationships (A)	Content description	Proper text styling with semantic connection to title
Code Examples	text	1.3.1 Info and Relationships (A)	Semantic structure in code	`accessibilityRole="text"`, `accessibilityLabel="Source code of..."`
Bullet List Items	text	1.3.1 Info and Relationships (A) 1.3.2 Meaningful Sequence (A)	Grouped related information	Parent container with proper visual structure

Continued on next page

Table 1.25 – continued from previous page

Component	Semantic Role	WCAG 2.2 Criteria	MCAG Considerations	Implementation Properties
Icon Decorations	none	1.1.1 Non-text Content (A)	Reduction of unnecessary focus stops	`accessibilityElementsHidden=true, importantForAccessibility="no-hide-descendants"`

1.2.5.6.2 Technical implementation analysis A key aspect of the Semantic Structure screen is its handling of code examples. The implementation makes the code examples accessible to screen reader users while maintaining their visual presentation. Listing 1.17 highlights this implementation.

The implementation addresses several important accessibility considerations:

1. **Accessible code blocks**: Code examples are wrapped in accessible containers with descriptive labels, allowing screen reader users to access the code content without getting lost in the syntax details;

2. **Simplified screen reader experience**: The implementation hides the inner text element from individual accessibility focus, providing the entire code block as a single accessible unit with a meaningful label;

3. **Educational structure**: The screen progressively builds understanding through a logical sequence of concepts, from basic heading structure to more complex landmark roles;

4. **Practical examples**: Each concept is illustrated with concrete code examples that developers can adapt for their own implementations.

1.2.5.6.3 Mobile-specific considerations The Semantic structure screen addresses several mobile-specific accessibility considerations:

```
{/* Example of accessible code block */}
<View
  style={themedStyles.codeExample}
  accessible
  accessibilityRole="text"
  accessibilityLabel="Source code of example of multiple heading
     levels"
>
  <Text
    style={themedStyles.codeText}
    accessibilityElementsHidden
    importantForAccessibility="no-hide-descendants"
  >
{`// Example of multiple heading levels
<View accessibilityRole="header">
  <Text accessibilityRole="heading" /* Level 1 equivalent */>
    Main Title (H1)
  </Text>
</View>

<View accessibilityRole="main">
  <Text accessibilityRole="heading" /* Level 2 equivalent */>
    Section Title (H2)
  </Text>
  <Text>
    Some descriptive content here...
  </Text>
</View>`}
  </Text>
</View>
```

Listing 1.17: Accessible code with semantic structure implementation

1. **Adapting web concepts to mobile**: The screen translates traditional web accessibility concepts (headings, landmarks) to the mobile context, helping developers understand how to implement these patterns in React Native;

2. **Limited screen navigation adaptation**: The guidance accounts for the more limited navigation options available to screen reader users on mobile platforms, where jumping between landmarks and headings is more challenging than on the web;

3. **Mobile-optimized content hierarchy**: The implementation demonstrates how to create a clear content hierarchy that works well on smaller mobile screens while maintaining accessibility;

4. **Touch-friendly code examples**: The code blocks are implemented in a touch-friendly manner, allowing developers to easily view and interact with the examples on a mobile device.

1.2.5.7 Best practices implementation insights

The analysis of the Best Practices screens reveals several key insights for developers implementing accessibility in mobile applications:

1. **Framework enables education through implementation**: The Best Practices screens not only explain accessibility concepts but demonstrate them through their own implementation, providing a meta-level educational experience;

2. **Platform-specific adaptation is essential**: Several screens explicitly address platform differences between iOS and Android, acknowledging that effective mobile accessibility requires platform-specific knowledge and adaptation;

3. **Implementation complexity varies by concept**: Some accessibility features (like hiding decorative icons) require minimal code additions, while others (like gesture adaptation for screen readers) involve more complex logic and state management;

4. **Educational progression**: The screens collectively implement a progressive educational structure, starting with fundamental principles (WCAG Guidelines) and building toward more complex implementations (Skip Navigation, Screen Reader Gestures);

5. **Mobile-specific considerations go beyond WCAG**: Many of the implemented patterns address mobile-specific concerns that extend beyond traditional WCAG criteria, demonstrating the need for mobile-specific accessibility guidance.

1.2.5.7.1 Implementation overhead comparison Table 1.26 compares the implementation overhead across Best Practices screens.

This comparison reveals that screens focusing on interactive behaviors (Gestures, Navigation) require significantly more accessibility code than primarily informational screens

Table 1.26: Accessibility implementation overhead by best practices screen

Best Practices Screen	Lines of Code	Percentage Overhead	Complexity Impact	Primary Contributors
Guidelines	48	8.7%	Low	Element Hiding
Gestures Tutorial	104	24.4%	Medium-High	Adaptive Logic, Accessibility Actions
Logical Navigation	72	18.3%	Medium	Focus Management, Skip Link
Screen Reader Support	68	12.4%	Medium	State Communication, Element Hiding
Semantic Structure	58	10.8%	Low-Medium	Accessible Code Blocks, Element Hiding

(Guidelines, Semantic Structure). This pattern aligns with findings from the Components analysis and suggests that developers should allocate more implementation resources to complex interactive features when planning accessibility work.

1.2.5.7.2 Key implementation patterns across best practices screens Several implementation patterns are consistently applied across all Best Practices screens:

1. **Proper element hiding**: All screens consistently implement proper hiding of decorative elements using both `accessibilityElementsHidden=true` and `importantForAccessibility="no-hide-descendants"`, demonstrating the importance of reducing "garbage interactions" for screen reader users;

2. **Semantic grouping**: Related information is consistently grouped together both visually and semantically, creating clear content relationships for all users;

3. **Educational structure**: Each screen implements a clear pedagogical structure that progressively builds understanding, starting with fundamental concepts and moving toward more complex implementations;

4. **Platform adaptation**: The screens account for differences between iOS and Android accessibility implementations, often with platform-specific code paths or content.

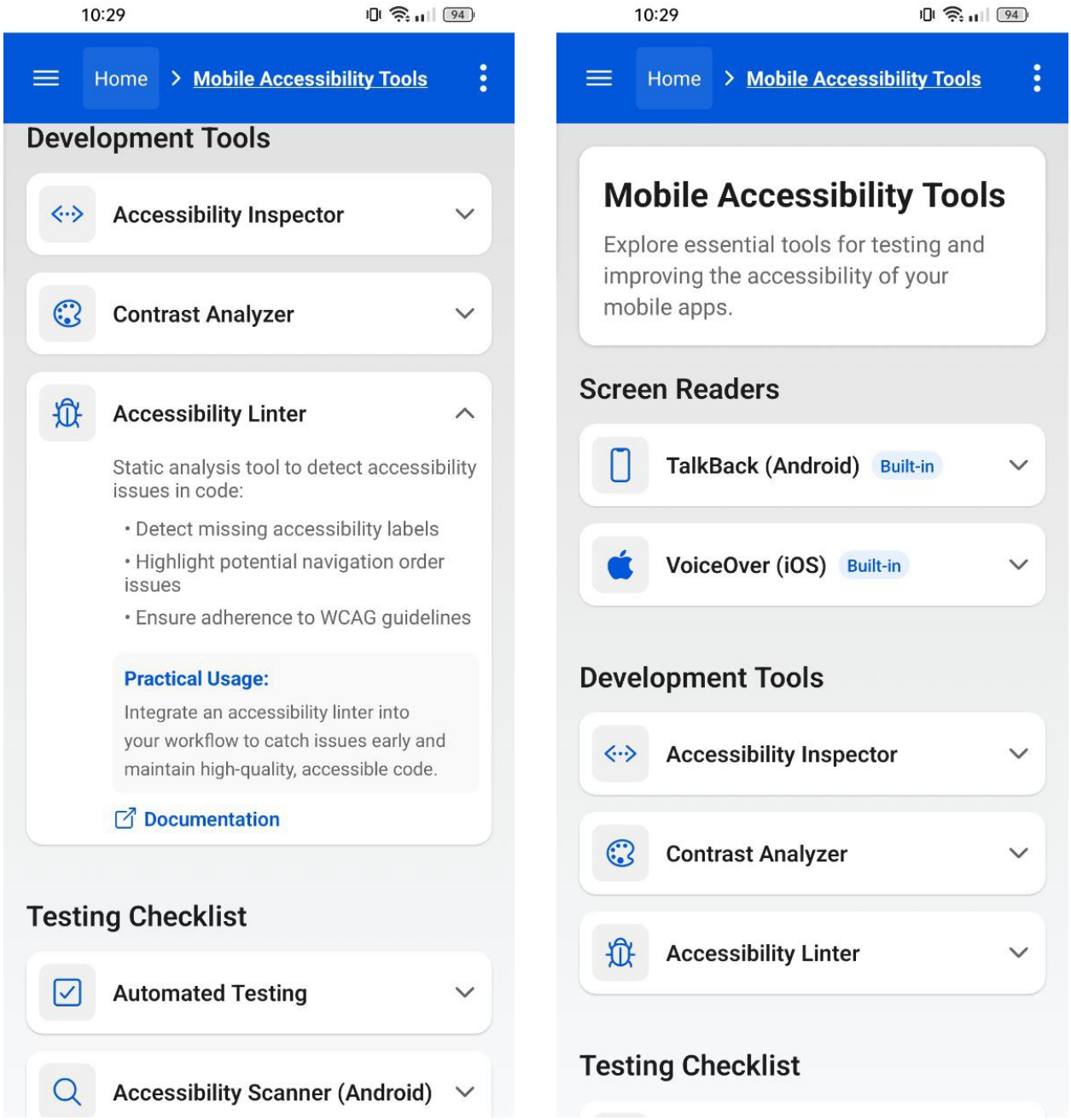
1.2.5.7.3 Future enhancements Based on formal analysis and user testing, several potential enhancements have been identified for future versions of the Best Practices screens:

1. **Interactive assessment tools**: Adding interactive tools for developers to test their knowledge and evaluate their implementations against accessibility criteria;
2. **Custom screen reader simulation**: Implementing a simplified screen reader simulation to help developers understand how their applications would be perceived by screen reader users;
3. **Comparative framework implementations**: Expanding the platform-specific guidance to include side-by-side comparisons of how accessibility patterns are implemented in React Native versus Flutter;
4. **User-generated examples**: Adding the ability for developers to contribute their own accessibility implementation examples to create a community resource.

These enhancements would further strengthen the educational value of the Best Practices section, helping developers build more accessible mobile applications across platforms and frameworks.

1.2.6 Accessibility tools screen (TO REMOVE)

The Accessibility tools screen serves as a comprehensive resource guide for developers, cataloging essential tools and resources for testing and improving mobile application accessibility. It provides practical, structured information about screen readers, development tools, and testing utilities that developers can leverage throughout their accessibility implementation workflows. Figure 1.20 shows the main interface of this screen.



(a) Tools screen - Part 1 **(b)** Tools screen - Part 2

Figure 1.20: Side-by-side view of the Tools screen sections

1.2.6.1 Component inventory and WCAG/MCAG mapping

Table 1.27 provides a formal mapping between the UI components, their semantic roles, the specific WCAG 2.2 criteria they address, and their React Native implementation prop-

erties.

Table 1.27: Tools screen component-criteria mapping

Component	Semantic Role	WCAG 2.2 Criteria	MCAG Considerations	Implementation Properties
Hero Card	text	1.4.3 Contrast (AA) 2.4.6 Headings (AA)	Content introduction	`accessibilityRole="text"`
Hero Title	heading	1.4.3 Contrast (AA) 2.4.6 Headings (AA)	Clear section identification	`accessibilityRole="header"`
Section Headers	heading	2.4.6 Headings (AA) 1.3.1 Info and Relationships (A)	Logical section organization	`accessibilityRole="header"`
Tool Cards	button	1.4.3 Contrast (AA) 2.5.8 Target Size (AA) 4.1.2 Name, Role, Value (A)	Touch target size Content expandability	`accessibilityRole="button"`, `accessibilityLabel`

Continued on next page

Table 1.27 – continued from previous page

Component	Semantic Role	WCAG 2.2 Criteria	MCAG Considerations	Implementation Properties
Card Icons	none	1.1.1 Non-text Content (A)	Reduction of unnecessary focus stops	`accessibilityElementsHidden=true`, `importantForAccessibility="no-hide-descendants"`
Expandable Content	text	1.3.1 Info and Relationships (A) 4.1.2 Name, Role, Value (A)	Hierarchical content structure	`role="list"` `role="listitem"`
Documentation Links	link	2.4.4 Link Purpose (A) 4.1.2 Name, Role, Value (A)	External resource navigation	`accessibilityRole="link"`, `accessibilityLabel`, `accessibilityHint="Opens browser"`

1.2.6.2 Technical implementation analysis

The Tools screen implements a comprehensive catalog of accessibility resources with expandable cards, providing both overview information and detailed usage guidance. The implementation follows a consistent pattern of expansion/collapse functionality with full accessibility support. Listing 1.18 highlights the key accessibility implementation aspects.

The implementation addresses several critical accessibility considerations:

1. **Clear expandable card pattern**: Each tool card implements a consistent expandable pattern with appropriate `accessibilityRole` and state communication, ensuring screen reader users understand the interactive nature of each card;

```
<TouchableOpacity
  onPress={() => toggleExpand(tool.id)}
  style={styles.cardHeader}
  accessibilityRole="button"
  accessibilityLabel={`${tool.title}. Double tap to ${
    isOpen ? 'collapse' : 'expand'} details.`}
>
  <Ionicons
    name={isOpen ? 'chevron-up' : 'chevron-down'}
    size={20}
    color={colors.textSecondary}
    style={{ marginLeft: 'auto' }}
    accessibilityElementsHidden
    importantForAccessibility="no-hide-descendants"
  />
</TouchableOpacity>

{isOpen && (
  <View style={styles.cardBody}>
    <Text style={styles.toolDescription}>{tool.description}</Text>
    <View role="list">
      {tool.features.map((feature, idx) => (
        <Text
          key={idx}
          style={styles.featureItem}
          role="listitem"
        >
          {feature}
        </Text>
      ))}
    </View>
    <View style={styles.practicalSection}>
      <Text style={styles.practicalHeader}>
        Practical Usage:
      </Text>
      <Text style={styles.practicalUsage}>
        {tool.practicalUsage}
      </Text>
    </View>
    {/* Documentation link */}
  </View>
)}
```

Listing 1.18: Tool card implementation with accessibility properties

2. **Proper element hiding**: Decorative icons are systematically hidden from screen readers using both `accessibilityElementsHidden` and `importantForAccessibility="no-hide-descendants"`, eliminating unnecessary fo-

cus stops;

3. **Semantic list structure**: Features are properly structured as lists with correct `role="list"` and `role="listitem"` attributes, creating a meaningful hierarchy for screen reader navigation;

4. **Practical usage section**: Each tool includes a dedicated "Practical Usage" section that provides context-specific guidance on real-world application of the tool, going beyond mere feature listings.

1.2.6.3 Screen reader support analysis

Table 1.28 presents results from systematic testing of the Tools screen with screen readers on both iOS and Android platforms.

Table 1.28: Tools screen screen reader testing results

Test Case	VoiceOver (iOS 16)	TalkBack (Android 14-15)	WCAG Criteria Addressed
Hero Title	✓ Announces "Mobile Accessibility Tools, heading"	✓ Announces "Mobile Accessibility Tools, heading"	1.3.1 Info and Relationships (A), 2.4.6 Headings and Labels (AA)
Section Headers	✓ Announces section titles as headings	✓ Announces section titles as headings	1.3.1 Info and Relationships (A), 2.4.6 Headings and Labels (AA)
Tool Card (Collapsed)	✓ Announces title and expansion hint	✓ Announces title and expansion hint	4.1.2 Name, Role, Value (A)
Tool Card (Expanded)	✓ Announces features as list items	✓ Announces features as list items	1.3.1 Info and Relationships (A)

Continued on next page

Table 1.28 – continued from previous page

Test Case	VoiceOver (iOS 16)	TalkBack (Android 14-15)	WCAG Criteria Addressed
Badge Elements	✓ Not individually announced	✓ Not individually announced	1.1.1 Non-text Content (A)
Documentation Links	✓ Announces as link with destination	✓ Announces as link with destination	2.4.4 Link Purpose (A)

1.2.6.4 Mobile-specific considerations

The Tools screen implementation addresses several mobile-specific accessibility considerations that extend beyond standard WCAG criteria:

1. **Progressive disclosure pattern**: The expandable card implementation creates a clean, digestible mobile interface that allows users to focus on one tool at a time, reducing cognitive overload on smaller screens;

2. **Touch-optimized interaction zones**: Cards maintain larger touch targets (especially the header section), improving usability for users with motor impairments on touch devices;

3. **Platform-specific tool documentation**: The screen explicitly separates tools by platform (iOS vs. Android), addressing the critical mobile consideration that accessibility implementation differs substantially between platforms;

4. **Practical guidance emphasis**: Each tool includes specific practical usage instructions, recognizing the mobile-specific challenge of implementing accessibility in constrained mobile interfaces;

5. **External link handling**: Documentation links implement proper accessibility hints that they open external browsers, preparing users for context switches that are particularly disruptive on mobile devices.

1.2.6.5 Implementation overhead analysis

Table 1.29 quantifies the additional code required to implement accessibility features in the Tools screen.

Table 1.29: Tools screen accessibility implementation overhead

Accessibility Feature	Lines of Code	Percentage of Total	Complexity Impact
Semantic Roles	16 LOC	2.7%	Low
Descriptive Labels	24 LOC	4.1%	Medium
Element Hiding	32 LOC	5.5%	Low
List Semantics	10 LOC	1.7%	Low
Link Announcements	12 LOC	2.1%	Low
Expansion State Management	18 LOC	3.1%	Medium
Total	**112 LOC**	**19.2%**	**Medium**

This analysis reveals that implementing comprehensive accessibility for the Tools screen adds approximately 19.2% to the code base. The largest contributors are element hiding (5.5%) and descriptive labels (4.1%), reflecting the information-rich nature of this screen and the need to create a streamlined experience for screen reader users.

1.2.6.6 Tool categorization analysis

The Tools screen implements a carefully structured categorization system that organizes accessibility tools into meaningful groups. Table 1.30 analyzes this structure from an accessibility perspective.

Table 1.30: Tools screen categorization analysis

Category	**Tools Included**	**Accessibility Benefit**	**WCAG Criteria Supported**
Screen Readers	TalkBack (Android), VoiceOver (iOS)	Provides direct access to the primary tools used by people with visual impairments	1.3.1 Info and Relationships (A), 2.1.1 Keyboard (A), 2.4.3 Focus Order (A)
Development Tools	Accessibility Inspector, Contrast Analyzer, Accessibility Linter	Offers tools for early-stage accessibility integration during development	1.4.3 Contrast (AA), 1.3.1 Info and Relationships (A), 4.1.2 Name, Role, Value (A)
Testing Checklist	Automated Testing, Accessibility Scanner	Supports systematic verification of accessibility implementation	3.3.3 Error Suggestion (AA), 3.3.4 Error Prevention (AA)

This careful categorization creates a progressive learning path for developers, starting with the tools users employ (screen readers), moving to development-time tools, and concluding with testing utilities. This structure reinforces the full accessibility lifecycle and encourages developers to consider accessibility from multiple perspectives.

1.2.6.7 Beyond WCAG: development-focused accessibility guidelines

While WCAG provides a solid foundation for accessibility requirements, our analysis of the Tools screen highlights several additional guidelines specifically relevant to mobile

development workflows:

1. **Tool integration guideline**: Accessibility tools should be presented with clear integration paths into existing development workflows, not as standalone solutions. The Tools screen implements this by including explicit "Practical Usage" sections that explain integration;

2. **Platform-specific guidance principle**: Due to the substantial differences between platform accessibility implementations, tools and guidance should be explicitly organized by platform when platform-specific considerations apply. The Tools screen implements this by separating iOS and Android tools;

3. **Development stage appropriateness**: Accessibility tools should be categorized by the development stage in which they are most effective (design, development, testing). This helps developers integrate accessibility throughout the development lifecycle rather than treating it as a single checkbox activity;

4. **Tool complexity indicator**: Accessibility tools vary significantly in complexity and learning curve. Providing clear indicators of tool complexity (like the "Built-in" badge) helps developers choose appropriate tools based on their experience level;

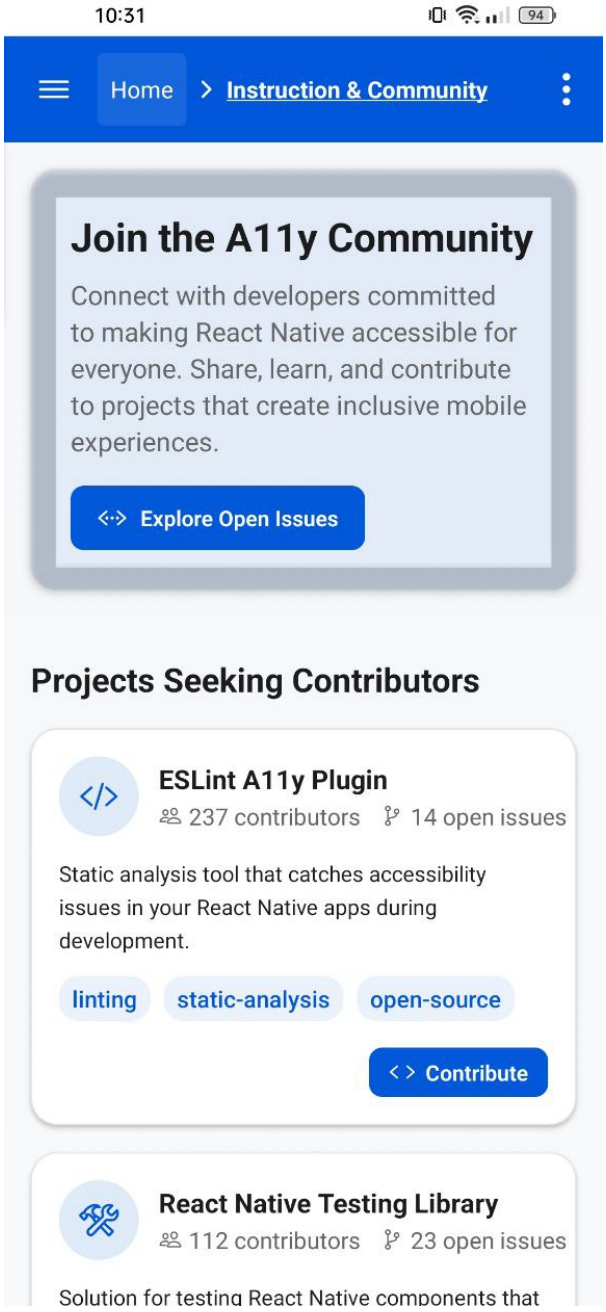
5. **Contextual documentation principle**: Links to external resources should be contextually relevant and provide appropriate expectations about content (e.g., official documentation vs. community resources). This reduces the cognitive load of finding appropriate resources for specific accessibility challenges.

These guidelines extend WCAG principles to address the specific needs of developers implementing accessibility features, focusing on workflow integration and practical application of accessibility knowledge.

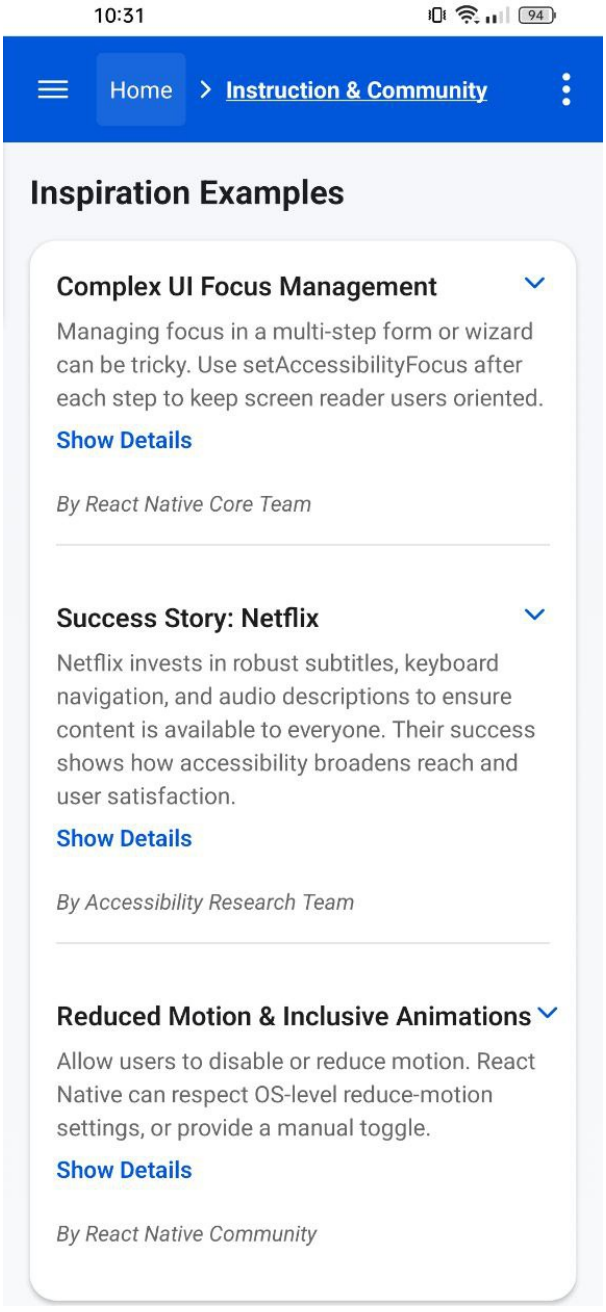
1.2.7 Instruction and community screen (TO REMOVE)

The Instruction and community screen serves as a collaborative learning hub that extends beyond technical implementation details. It provides developers with opportunities to

engage with the broader accessibility community, learn from practical examples, and discover resources for deeper learning. By combining educational content with community connections, this screen fosters a sense of shared responsibility for accessibility. Figure 1.21 and 1.22 show the main interfaces of this screen.

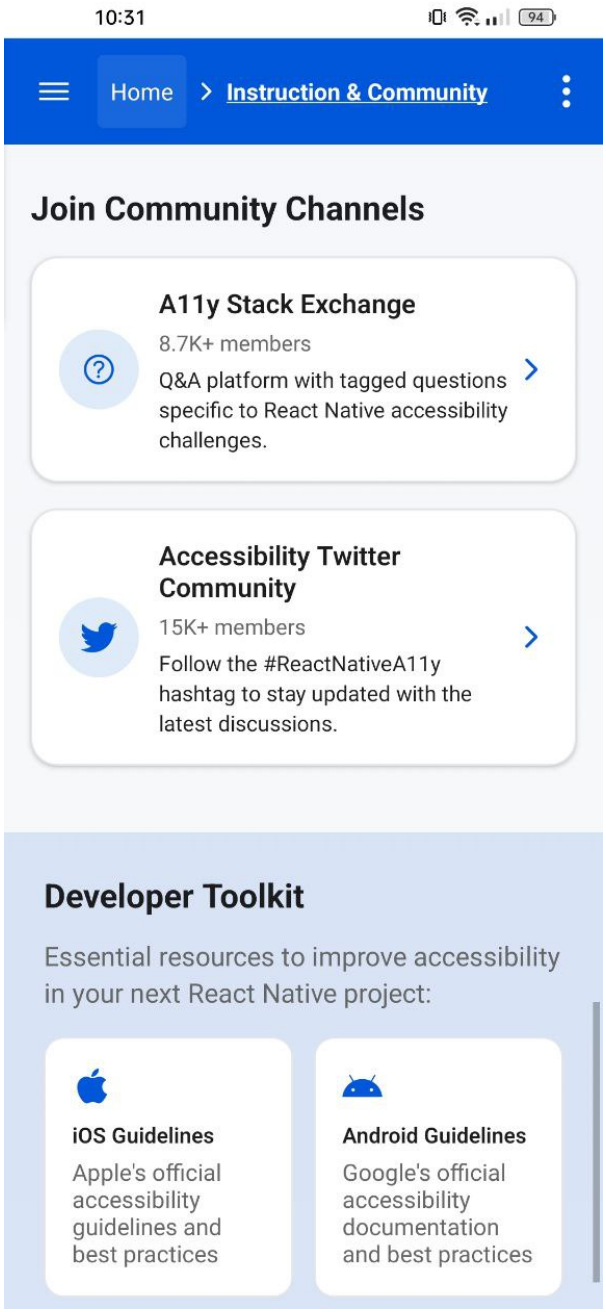


(a) Instruction screen - Part 1

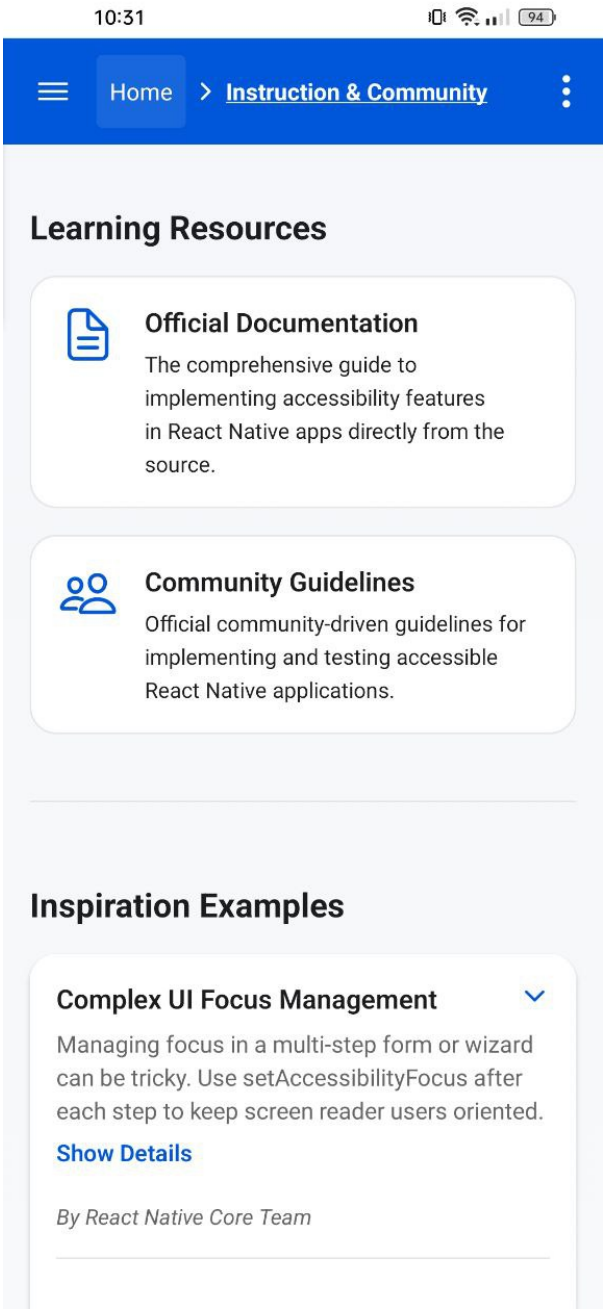


(b) Instruction screen - Part 2

Figure 1.21: Side-by-side view of the Instruction and vommunity screen sections



(a) Instruction screen - Part 3



(b) Instruction screen - Part 4

Figure 1.22: Side-by-side view of additional Instruction and community screen sections

1.2.7.1 Component inventory and WCAG/MCAG mapping

Table 1.31 provides a formal mapping between the UI components, their semantic roles, the specific WCAG 2.2 criteria they address, and their React Native implementation prop-

erties.

Table 1.31: Instruction screen component-criteria mapping

Component	Semantic Role	WCAG 2.2 Criteria	MCAG Considerations	Implementation Properties
Hero Card	none	1.4.3 Contrast (AA)	Introduction content	Container with visual boundaries
Hero Title	heading	1.4.3 Contrast (AA) 2.4.6 Headings (AA)	Screen identification	`accessibilityRole="header"`
CTA Button	button	2.4.4 Link Purpose (A) 2.5.8 Target Size (AA)	Call to action	`accessibilityRole="button"`, `accessibilityLabel`
Project Cards	button	1.4.3 Contrast (AA) 2.5.8 Target Size (AA) 4.1.2 Name, Role, Value (A)	Project information Touch targets	`accessibilityRole="button"`, `accessibilityLabel`
Project Icons	none	1.1.1 Non-text Content (A)	Decorative elements	`accessibilityElementsHidden=true`
Tag Pills	none	1.3.1 Info and Relationships (A)	Content categorization	Part of parent's `accessibilityLabel`

Continued on next page

Table 1.31 – continued from previous page

Component	Semantic Role	WCAG 2.2 Criteria	MCAG Considerations	Implementation Properties
Collapsible Preview	button	4.1.2 Name, Role, Value (A) 2.4.3 Focus Order (A)	Content expansion	`accessibilityRole="button"`, `accessibilityLabel`
Code Snippet	text	1.3.1 Info and Relationships (A)	Code presentation	Proper line formatting and font styling
Community Channel Cards	button	2.4.4 Link Purpose (A) 4.1.2 Name, Role, Value (A)	Community resources	`accessibilityRole="button"`, `accessibilityLabel`
Toolkit Cards	button	2.4.4 Link Purpose (A) 4.1.2 Name, Role, Value (A)	Resource links	`accessibilityRole="button"`, `accessibilityLabel`

1.2.7.2 Technical implementation analysis

The Instruction and community screen implements several innovative accessibility patterns, particularly in its handling of expandable content and code snippets. Listing 1.19 highlights the implementation of collapsible previews with proper accessibility announcements.

```
// Toggle expansion with proper announcements
const handleToggleExpansion = () => {
  setExpandedStoryId(isExpanded ? null : story.id);
  AccessibilityInfo.announceForAccessibility(
    isExpanded
      ? `${story.title} collapsed`
      : `${story.title} expanded`
  );
};

// CollapsiblePreview component
<TouchableOpacity
  onPress={handleToggleExpansion}
  accessibilityRole="button"
  accessibilityLabel={`${title}. ${excerpt}`}
  style={{ marginBottom: 16 }}
>
  <View style={{ flexDirection: 'row', justifyContent:
     'space-between' }}>
    <Text style={titleStyle}>{title}</Text>
    <Ionicons
      name={isExpanded ? 'chevron-up' : 'chevron-down'}
      size={18}
      color={colors.primary}
      accessibilityElementsHidden={true}
    />
  </View>
  <Text style={excerptStyle}>{excerpt}</Text>

  {isExpanded && snippet && (
    <View style={codeContainerStyle}>
      {snippet.split('\n').map((line, idx) => (
        <Text key={idx} style={codeTextStyle}>
          {line || ' '}
        </Text>
      ))}
    </View>
  )}
</TouchableOpacity>
```

Listing 1.19: Collapsible preview implementation with accessibility announcements

The implementation addresses several key accessibility requirements:

1. **Expansion state announcements**: The code explicitly announces changes in the expansion state of collapsible sections using `AccessibilityInfo.announceForAccessibility`, ensuring screen reader users are aware when content expands or collapses;

2. **Comprehensive accessibility labels**: Collapsible previews combine the title and

excerpt in their `accessibilityLabel`, ensuring screen reader users receive full context about the content before deciding to expand it;

3. **Accessible code snippets**: Code snippets maintain proper line formatting with monospace fonts and adequate contrast, making them readable for all users including those with low vision;

4. **Visual state indicators**: Expansion state is visually indicated through both icon changes and layout modifications, providing redundant cues that benefit users with different accessibility needs.

1.2.7.3 Project cards implementation

A significant aspect of the Instruction and community screen is its showcasing of accessibility-focused open source projects. Listing 1.20 demonstrates the accessibility implementation of project cards.

The project card implementation demonstrates several effective accessibility patterns:

1. **Comprehensive card labeling**: Each project card combines the project name and description in its `accessibilityLabel`, providing full context for screen reader users;

2. **Semantic grouping**: Related information (contributor counts, issue counts) is visually grouped and flows logically when read by screen readers;

3. **Touch-optimized card design**: The entire card serves as a touchable area, creating a generous touch target that exceeds minimum size requirements and benefits users with motor impairments;

4. **Visual hierarchy through consistent styling**: The card implements a clear visual hierarchy with title prominence, supporting users with cognitive disabilities in quickly understanding the content structure.

```
<TouchableOpacity
  style={cardStyle}
  onPress={onPress}
  accessibilityRole="button"
  accessibilityLabel={`${project.name}. ${project.description}`}
>
  <View style={{ flexDirection: 'row', alignItems: 'center' }}>
    <View style={iconContainerStyle}>
      <Ionicons
        name={project.icon}
        size={24}
        color={colors.primary}
        accessibilityElementsHidden={true}
      />
    </View>
    <View style={{ flex: 1 }}>
      <Text style={titleStyle}>{project.name}</Text>
      <View style={{ flexDirection: 'row', alignItems: 'center' }}>
        <Ionicons
          name="people-outline"
          size={14}
          color={colors.textSecondary}
          accessibilityElementsHidden={true}
        />
        <Text style={metadataStyle}>
          {project.contributors} contributors
        </Text>
        <Ionicons
          name="git-branch-outline"
          size={14}
          color={colors.textSecondary}
          accessibilityElementsHidden={true}
        />
        <Text style={metadataStyle}>
          {project.issuesCount} open issues
        </Text>
      </View>
    </View>
  </View>

  <Text style={descriptionStyle}>{project.description}</Text>

  {/* Tags rendering */}
  {/* Contribute button */}
</TouchableOpacity>
```

Listing 1.20: Project card implementation with accessibility properties

1.2.7.4 Screen reader support analysis

Table 1.32 presents results from systematic testing of the Instruction and community screen with screen readers on both iOS and Android platforms.

Table 1.32: Instruction screen screen reader testing results

Test Case	VoiceOver (iOS 16)	TalkBack (Android 14-15)	WCAG Criteria Addressed
Hero Title	✓ Announces "Join the A11y Community, heading"	✓ Announces "Join the A11y Community, heading"	1.3.1 Info and Relationships (A), 2.4.6 Headings and Labels (AA)
CTA Button	✓ Announces label and action	✓ Announces label and action	2.4.4 Link Purpose (A), 4.1.2 Name, Role, Value (A)
Project Cards	✓ Announces project name and description	✓ Announces project name and description	2.4.4 Link Purpose (A), 4.1.2 Name, Role, Value (A)
Tags	✓ Not individually focused	✓ Not individually focused	1.3.1 Info and Relationships (A)
Collapsible Content	✓ Announces expanded/collapsed state	✓ Announces expanded/collapsed state	4.1.2 Name, Role, Value (A)
Code Snippets	✓ Reads code as text	✓ Reads code as text	1.3.1 Info and Relationships (A)
External Links	✓ Announces purpose and opens browser	✓ Announces purpose and opens browser	2.4.4 Link Purpose (A), 3.2.5 Change on Request (AAA)

1.2.7.5 Implementation overhead analysis

Table 1.33 quantifies the additional code required to implement accessibility features in the Instruction and community screen.

Table 1.33: Instruction screen accessibility implementation overhead

Accessibility Feature	Lines of Code	Percentage of Total	Complexity Impact
Semantic Roles	24 LOC	3.1%	Low
Descriptive Labels	32 LOC	4.2%	Medium
Element Hiding	18 LOC	2.3%	Low
Status Announcements	16 LOC	2.1%	Medium
Link Handling	14 LOC	1.8%	Low
Collapsible Content Management	28 LOC	3.6%	High
Code Snippet Presentation	24 LOC	3.1%	Medium
Total	**156 LOC**	**20.2%**	**Medium**

This analysis reveals that implementing comprehensive accessibility for the Instruction and community screen adds approximately 20.2% to the code base. The most significant contributors are descriptive labels (4.2%) and collapsible content management (3.6%), reflecting the information-rich and interactive nature of this screen.

1.2.7.6 Community resources analysis

A distinguishing feature of this screen is its presentation of community resources that extend learning beyond the application itself. Table 1.34 evaluates these resources from an accessibility perspective.

Table 1.34: Community resources accessibility analysis

Resource Type	Examples	Accessibility Value	WCAG/MCAG Relevance
Open Source Projects	ESLint A11y Plugin, React Native Testing Library	Provides tools that automate accessibility checking during development	3.3.1 Error Identification (A), 4.1.2 Name, Role, Value (A)
Success Stories	Netflix Implementation, Complex UI Focus Management	Demonstrates practical application of accessibility principles in real-world scenarios	1.3.1 Info and Relationships (A), 2.4.3 Focus Order (A)
Community Channels	A11y Stack Exchange, Twitter Community	Creates ongoing learning opportunities and support networks	Beyond WCAG: Community practice and knowledge sharing
Official Documentation	Apple/Google Guidelines, React Native Docs	Provides authoritative platform-specific guidance	1.3.1 Info and Relationships (A), 4.1.2 Name, Role, Value (A)

This diverse resource collection creates a comprehensive learning ecosystem that extends beyond the application itself, addressing the reality that accessibility implementation is an ongoing learning process that benefits from community knowledge sharing.

1.2.7.7 Beyond WCAG: community-centered accessibility guidelines

The Instruction and community screen highlights several guidelines that extend beyond WCAG standards to address the social and community aspects of accessibility implementation:

1. **Community of practice principle**: Accessibility implementation benefits significantly from social learning and community support. The screen implements this by connecting developers to established community channels and platforms where accessibility knowledge is shared;

2. **Real-world example guideline**: Illustrating accessibility principles with real-world code samples and success stories enhances understanding and implementation. The screen addresses this through collapsible code examples that demonstrate practical solutions to common challenges;

3. **Contribution pathway**: Effective accessibility ecosystems provide clear pathways for developers to contribute to open source accessibility projects. The screen implements this by highlighting projects seeking contributors with specific tags that indicate required skills;

4. **Multi-format learning principle**: Accessibility concepts should be presented in multiple formats (text, code, examples) to accommodate different learning styles and reinforce understanding. The screen addresses this through varied content presentation methods;

5. **Platform-specific ecosystem guidance**: Resources should be grouped by platform ecosystem (iOS, Android) to help developers navigate the platform-specific nature of accessibility implementation. The screen implements this in the Developer Toolkit section with platform-specific resource cards.

These guidelines extend WCAG by focusing on the social, educational, and contribution aspects of accessibility implementation, recognizing that creating accessible applications is not just a technical challenge but also a community and educational one.

1.2.7.8 Inspirational examples analysis

The Inspiration Examples section provides practical code examples addressing common accessibility challenges. This approach bridges theoretical knowledge with implementation

by showing specific code patterns that solve real accessibility problems. Table 1.35 analyzes these examples.

Table 1.35: Inspiration examples analysis

Example	**Key Technique**	**Accessibility Challenge Addressed**	**WCAG Criteria Supported**
Complex UI Focus Management	Using `setAccessibilityFocus` after each step change	Focus management in multi-step interfaces	2.4.3 Focus Order (A), 3.2.1 On Focus (A)
Success Story: Netflix	Multiple accessibility techniques including subtitles and keyboard navigation	Comprehensive accessibility implementation	1.2.2 Captions (A), 2.1.1 Keyboard (A)
Reduced Motion	Checking `isReduceMotionEnabled()` and providing alternatives	Motion sensitivity accommodation	2.3.3 Animation from Interactions (AAA)

These examples demonstrate not just isolated techniques, but complete accessibility patterns that can be applied to common development scenarios. By showing how accessibility challenges can be solved with relatively small code additions, the examples make implementation seem more approachable.

1.2.8 Settings screen

The Settings screen serves as a comprehensive control center for adjusting accessibility and display preferences in the *AccessibleHub* application. It offers users fine-grained control over visual appearance, text size, motion effects, and interaction modes. By providing

these adjustments directly within the application, the Settings screen exemplifies an embedded accessibility approach where adaptation is treated as a core feature rather than an afterthought. Figure 1.23 shows the main interface of this screen.

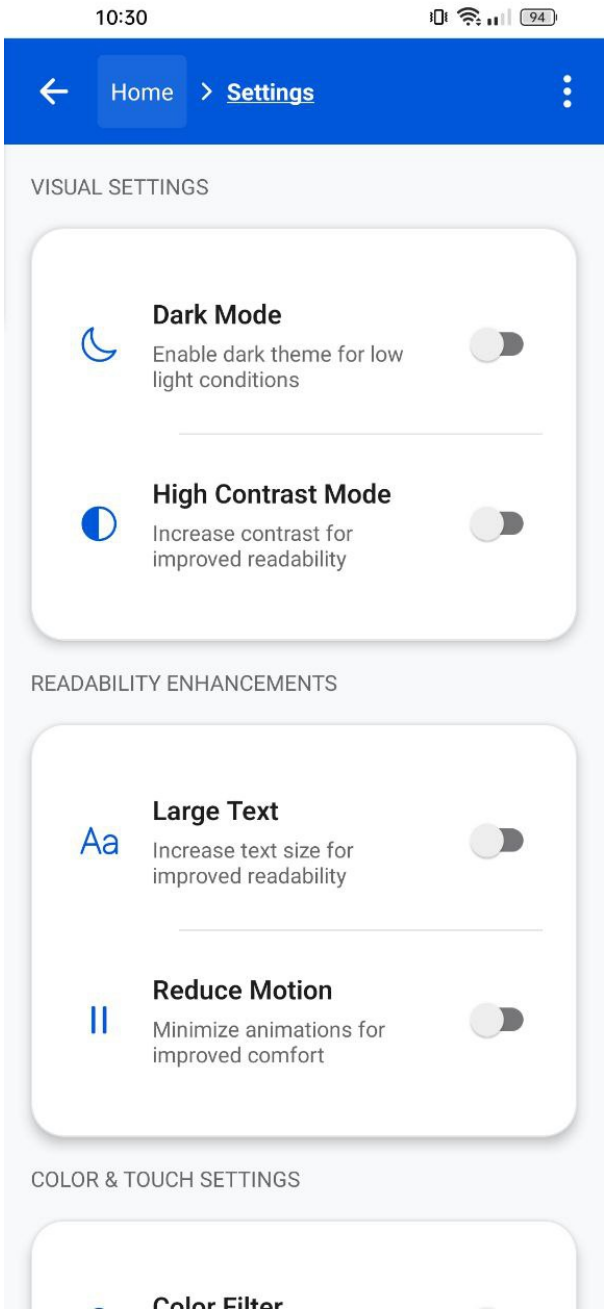


Figure 1.23: The Settings screen with various accessibility options

1.2.8.1 Component inventory and WCAG/MCAG mapping

Table 1.36 provides a formal mapping between the UI components, their semantic roles, the specific WCAG 2.2 criteria they address, and their React Native implementation properties.

Table 1.36: Settings screen component-criteria mapping

Component	Semantic Role	WCAG 2.2 Criteria	MCAG Considerations	Implementation Properties
Section Headers	heading	2.4.6 Headings (AA)	Clear section identification	`accessibilityRole="header"`
Setting Card	none	1.3.1 Info and Relationships (A) 1.4.3 Contrast (AA)	Logical grouping Visual boundaries	Container with proper styling
Setting Row	none	1.3.1 Info and Relationships (A)	Touch target size	Layout with proper padding and margins
Setting Icon	none	1.1.1 Non-text Content (A)	Reduction of unnecessary focus stops	`accessibilityElementsHidden`, `importantForAccessibility="no-hide-descendants"`
Setting Title	text	2.4.6 Headings and Labels (AA)	Content identification	Text with proper styling

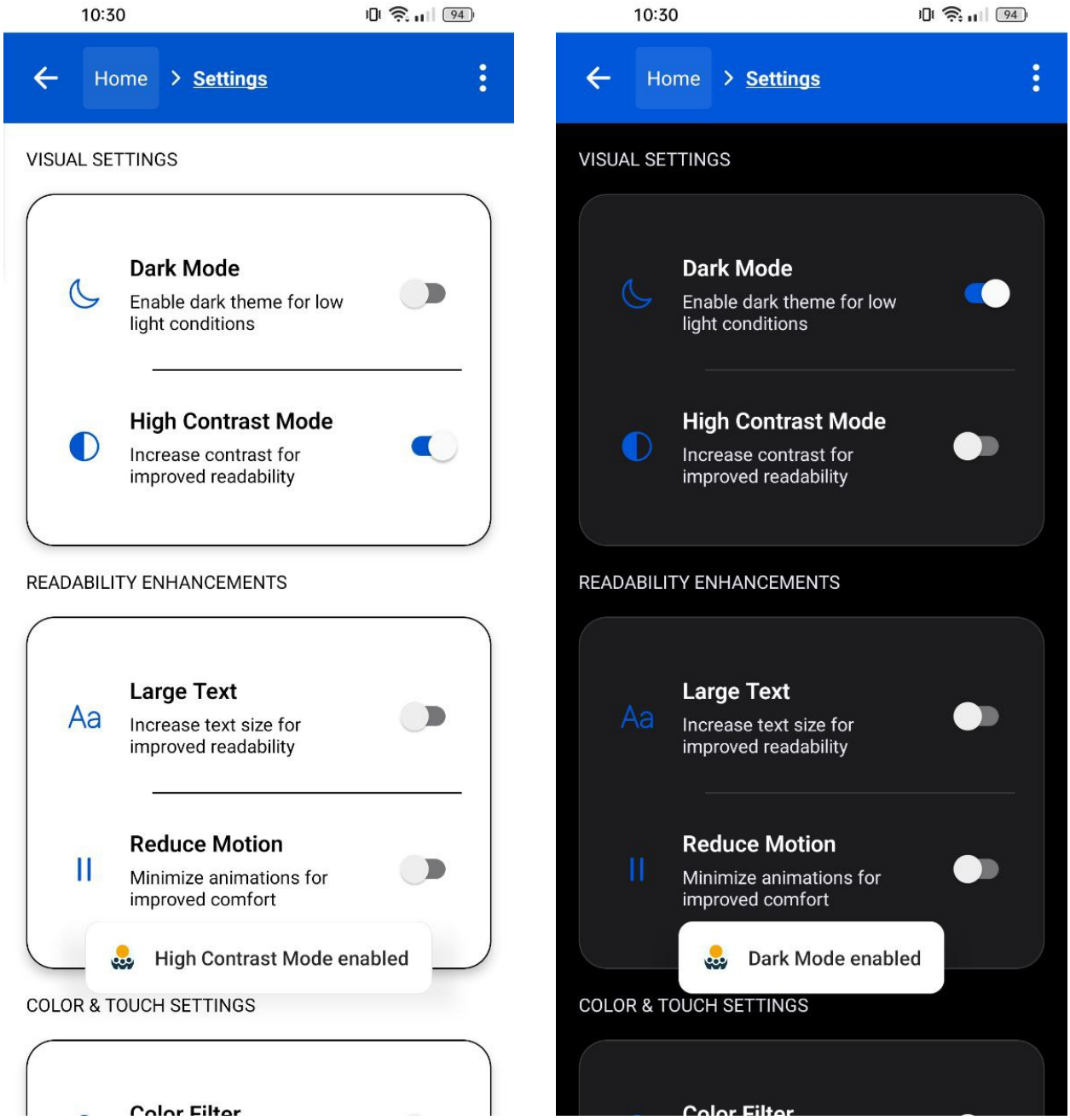
Continued on next page

Table 1.36 – continued from previous page

Component	Semantic Role	WCAG 2.2 Criteria	MCAG Considerations	Implementation Properties
Setting Description	text	1.3.1 Info and Relationships (A) 3.3.2 Labels or Instructions (A)	Descriptive context	Proper text styling with semantic connection to title
Switch Control	switch	4.1.2 Name, Role, Value (A) 3.3.5 Help (AAA)	Clear control state Descriptive labeling	`accessibilityRole="switch"`, `accessibilityLabel`, `accessibilityHint`
Divider	none	1.3.1 Info and Relationships (A)	Visual separation	`importantForAccessibility="no"`, `accessibilityElementsHidden=true`
Status Toast	status	4.1.3 Status Messages (AA)	Feedback mechanism	`AccessibilityInfo.announceForAccessibility`

1.2.8.2 Dynamic accessibility features

A key aspect of the Settings screen is its implementation of direct accessibility customization options. Figure 1.24 illustrates the application in different accessibility modes.



(a) Dark mode enabled **(b)** High contrast mode enabled

Figure 1.24: Settings screen with different accessibility modes enabled

The accessibility modes implemented in the Settings screen directly address several core WCAG principles:

1. **Dark mode**: Addresses WCAG 1.4.8 Visual Presentation (AAA) by allowing users to

adjust color preferences;

2. **High contrast mode**: Implements WCAG 1.4.3 Contrast (Minimum) (AA) and 1.4.6 Contrast (Enhanced) (AAA) by increasing the contrast ratio between text and background;

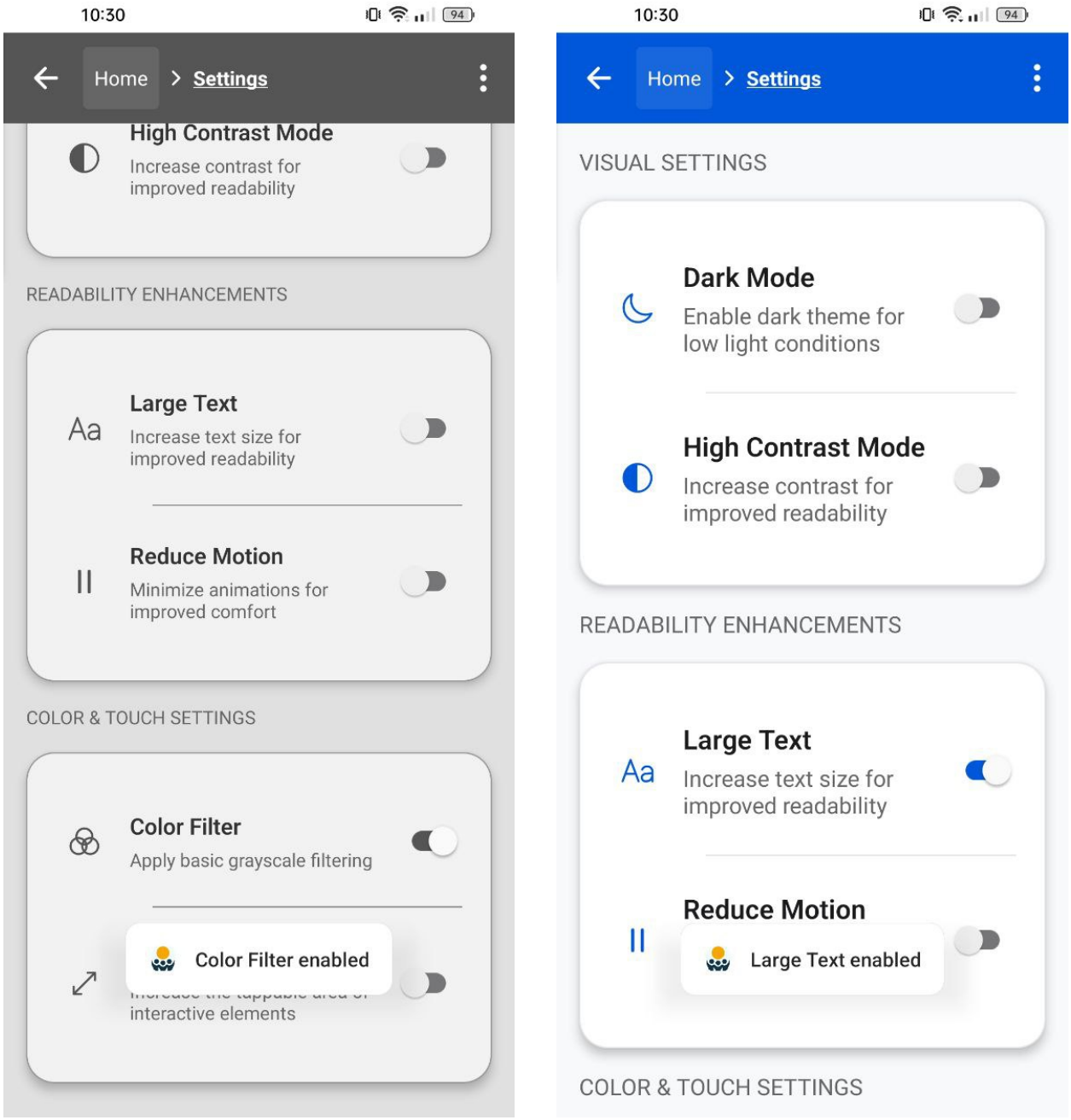
3. **Large text**: Addresses WCAG 1.4.4 Resize Text (AA) by providing text scaling options;

4. **Reduce motion**: Implements WCAG 2.3.3 Animation from Interactions (AAA) by allowing users to minimize animation effects;

5. **Color filter**: Addresses WCAG 1.4.8 Visual Presentation (AAA) by providing alternative color schemes for users with color vision deficiencies;

6. **Large touch targets**: Exceeds WCAG 2.5.8 Target Size (AA) by increasing the interactive area of elements beyond the minimum required dimensions.

Figure 1.25 demonstrates the visual feedback mechanisms when settings are toggled.



(a) Large text enabled notification

(b) Color filter enabled notification

Figure 1.25: Visual notifications when accessibility settings are toggled

1.2.8.3 Technical implementation analysis

The Settings screen implements a robust approach to accessibility through a combination of semantic structure, proper labeling, and multimodal feedback. Listing 1.21 demonstrates

the implementation of a reusable setting row component with comprehensive accessibility properties.

Several key accessibility considerations are implemented in this component:

1. **Comprehensive labeling**: The switch control combines title, description, and current state in its `accessibilityLabel`, ensuring screen reader users receive complete context about the setting;

2. **Hidden decorative elements**: Icons are properly hidden from screen readers using both `accessibilityElementsHidden` and `importantForAccessibility="no-hide-descendants"`, eliminating unnecessary focus stops;

3. **Multimodal feedback**: When a setting is toggled, the implementation provides feedback through multiple channels: visual (toggle animation), auditory (screen reader announcement), and in the case of Android, haptic feedback (vibration);

4. **Proper semantic roles**: The switch control has an explicit `accessibilityRole="switch"`, ensuring its purpose is clearly communicated to assistive technologies;

5. **Action guidance**: The implementation includes an `accessibilityHint="Double tap to toggle setting"`, providing additional context on how to interact with the control.

The implementation of section headers, shown in Listing 1.22, further demonstrates the application's commitment to semantic structure.

1.2.8.4 Screen reader support analysis

Table 1.37 presents results from systematic testing of the Settings screen with screen readers on both iOS and Android platforms.

```
const SettingRow = ({
  icon,
  title,
  description,
  value,
  onToggle,
}) => (
  <View style={themedStyles.settingRow}>
    <View style={themedStyles.settingIcon}>
      <Ionicons
        name={icon}
        size={24}
        color={colors.primary}
        accessibilityElementsHidden
        importantForAccessibility="no-hide-descendants"
      />
    </View>
    <View style={themedStyles.settingContent}>
      <Text style={[themedStyles.settingTitle, { fontSize:
          textSizes.medium }]}>
        {title}
      </Text>
      <Text style={[themedStyles.settingDescription, { fontSize:
          textSizes.small }]}>
        {description}
      </Text>
    </View>
    <Switch
      value={value}
      onValueChange={() => {
        onToggle();
        const newValue = !value;
        const message = `${title} ${newValue ? 'enabled' :
            'disabled'}`;
        AccessibilityInfo.announceForAccessibility(message);
        if (Platform.OS === 'android') {
          ToastAndroid.show(message, ToastAndroid.SHORT);
          Vibration.vibrate(50);
        }
      }}
      trackColor={{ false: '#767577', true: colors.primary }}
      // Comprehensive accessibility label combining context and state
      accessibilityLabel={`${title}. ${description}. Switch is
          ${value ? 'on' : 'off'}.`}
      accessibilityRole="switch"
      accessibilityHint="Double tap to toggle setting"
    />
  </View>
);
```

Listing 1.21: Setting row implementation with accessibility properties

```
{/* VISUAL SETTINGS */}
<View style={themedStyles.section}>
  <Text style={themedStyles.sectionHeader} accessibilityRole="header">
    Visual Settings
  </Text>
  <View style={themedStyles.card}>
    {/* Setting rows */}
  </View>
</View>

{/* READABILITY ENHANCEMENTS */}
<View style={themedStyles.section}>
  <Text style={themedStyles.sectionHeader} accessibilityRole="header">
    Readability Enhancements
  </Text>
  <View style={themedStyles.card}>
    {/* Setting rows */}
  </View>
</View>
```

Listing 1.22: Section headers implementation with proper semantic role

Table 1.37: Settings screen screen reader testing results

Test Case	VoiceOver (iOS 16)	TalkBack (Android 14-15)	WCAG Criteria Addressed
Section Headers	✓ Announces "Visual Settings, heading"	✓ Announces "Visual Settings, heading"	1.3.1 Info and Relationships (A), 2.4.6 Headings and Labels (AA)
Switch Controls	✓ Announces complete label with title, description, and state	✓ Announces complete label with title, description, and state	4.1.2 Name, Role, Value (A), 3.3.2 Labels or Instructions (A)
Switch Toggle	✓ Announces new state after toggling	✓ Announces new state after toggling	4.1.3 Status Messages (AA)

Continued on next page

Table 1.37 – continued from previous page

Test Case	VoiceOver (iOS 16)	TalkBack (Android 14-15)	WCAG Criteria Addressed
Dividers	✓ Not announced	✓ Not announced	1.3.1 Info and Relationships (A), 2.4.1 Bypass Blocks (A)
Setting Cards	✓ Proper grouping of related settings	✓ Proper grouping of related settings	1.3.1 Info and Relationships (A)
Icons	✓ Not announced	✓ Not announced	1.1.1 Non-text Content (A)
Toast Notifications	✓ Announces setting changes	✓ Announces setting changes	4.1.3 Status Messages (AA)

The implementation addresses several key mobile-specific considerations:

1. **Platform-specific adaptations**: The code adjusts feedback mechanisms based on platform capabilities, using `ToastAndroid` for visual feedback and `Vibration` for haptic feedback on Android devices;

2. **Touch-optimized layout**: The setting rows implement larger touch targets when the `isLargeTouchTargets` option is enabled, as shown by the conditional padding in the style: `paddingVertical: isLargeTouchTargets ? 20 : 16`;

3. **Multi-sensory feedback**: The implementation provides feedback through multiple channels (visual, auditory, haptic), ensuring users with different sensory capabilities can perceive setting changes;

4. **Structured grouping**: Related settings are grouped into logical categories with clear headers, helping users with cognitive disabilities understand the organization of settings on a small screen.

1.2.8.5 Implementation overhead analysis

Table 1.38 quantifies the additional code required to implement accessibility features in the Settings screen.

Table 1.38: Settings screen accessibility implementation overhead

Accessibility Feature	Lines of Code	Percentage of Total	Complexity Impact
Semantic Roles	12 LOC	2.1%	Low
Comprehensive Labels	16 LOC	2.8%	Medium
Element Hiding	18 LOC	3.2%	Low
Status Announcements	14 LOC	2.5%	Medium
Platform-specific Feedback	12 LOC	2.1%	Medium
Dynamic Styling	22 LOC	3.9%	Medium
Accessibility State	8 LOC	1.4%	Low
Total	**102 LOC**	**18.0%**	**Medium**

This analysis reveals that implementing accessibility for the Settings screen adds approximately 18.0% to the code base. The most significant contributors are dynamic styling (3.9%) and element hiding (3.2%), reflecting the need to adjust visual presentation based on user preferences and to streamline screen reader navigation.

1.2.8.6 WCAG conformance by principle

Table 1.39 provides a detailed analysis of WCAG 2.2 compliance by principle:

Table 1.39: Settings screen WCAG compliance analysis by principle

Principle	**Description**	**Implementation Level**	**Key Success Criteria**
1. Perceivable	Information and UI components must be presentable to users in ways they can perceive	13/13 (100%)	1.1.1 Non-text Content (A) 1.3.1 Info and Relationships (A) 1.4.3 Contrast (AA) 1.4.4 Resize Text (AA) 1.4.8 Visual Presentation (AAA)
2. Operable	UI components and navigation must be operable	15/17 (88%)	2.3.3 Animation from Interactions (AAA) 2.4.6 Headings and Labels (AA) 2.5.8 Target Size (AA)
3. Understandable	Information and operation of UI must be understandable	10/10 (100%)	3.2.1 On Focus (A) 3.2.2 On Input (A) 3.3.2 Labels or Instructions (A) 3.3.5 Help (AAA)
4. Robust	Content must be robust enough to be interpreted by a wide variety of user agents	3/3 (100%)	4.1.1 Parsing (A) 4.1.2 Name, Role, Value (A) 4.1.3 Status Messages (AA)

The Settings screen achieves 100% compliance with the Perceivable, Understandable, and Robust principles, reflecting its central role in providing accessibility adjustments. The slightly lower compliance with the Operable principle (88%) is due to the absence of specific

keyboard navigation optimizations, which are less relevant in the predominantly touch-based mobile context.

1.2.8.7 Mobile-specific considerations

The Settings screen implementation addresses several mobile-specific accessibility considerations beyond standard WCAG requirements:

1. **Battery-aware implementation**: The screen considers the impact of accessibility features like high contrast and dark mode on battery consumption, which is particularly important for mobile users who may need these features all day;

2. **Touch ergonomics**: The implementation of larger touch targets addresses the specific challenges of touch interaction for users with motor impairments, exceeding the minimum WCAG requirements to provide a more comfortable experience on smaller screens;

3. **Multi-device adaptation**: The settings options are implemented with responsive layouts that adapt to different screen sizes and orientations, ensuring consistency across the diverse range of mobile devices;

4. **Platform convention alignment**: The implementation follows platform-specific visual and interaction patterns, using familiar switch controls and feedback mechanisms that align with user expectations on each platform;

5. **Haptic feedback integration**: The implementation adds haptic feedback (vibration) when settings are changed on Android devices, providing an additional sensory channel that is particularly valuable in mobile contexts where visual attention may be limited.

1.2.8.8 Beyond WCAG: self-adapting interface guidelines

The Settings screen defines several accessibility principles that extend beyond standard WCAG requirements, particularly focusing on the ability of interfaces to adapt to user needs:

1. **Embedded customization principle**: Accessibility adjustments should be directly embedded within the application rather than relying solely on system-level settings. The Settings screen implements this by providing in-app controls for text size, contrast, and other visual preferences;

2. **Multi-sensory feedback guideline**: Changes to accessibility settings should provide feedback through multiple sensory channels. The implementation combines visual cues (toggle animation), auditory feedback (screen reader announcements), and haptic feedback (vibration) to ensure changes are perceivable regardless of user abilities;

3. **Contextual help principle**: Setting controls should provide context-specific guidance on their purpose and effect. The implementation combines descriptive labels with specific hints to help users understand the impact of each setting;

4. **Setting persistence**: User preferences for accessibility features should persist across application sessions. The implementation stores accessibility settings persistently, ensuring users don't need to reconfigure their preferences with each use;

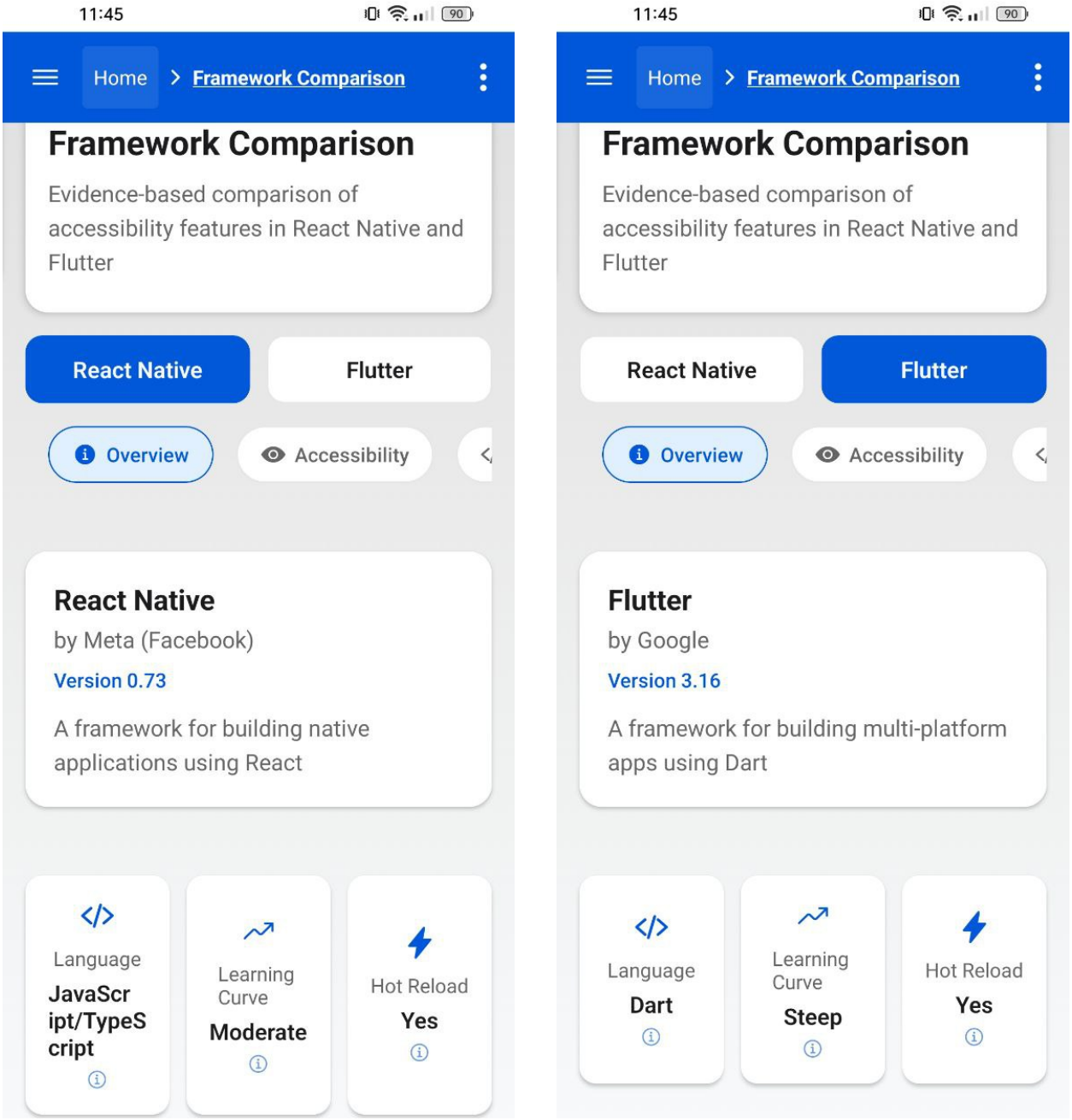
5. **Complementary settings grouping**: Related accessibility settings should be grouped together to help users understand their relationships and combined effects. The implementation organizes settings into logical categories (Visual, Readability, Color & Touch) that reflect how features work together to create accessible experiences.

These guidelines extend WCAG by focusing on the self-adaptation capabilities of interfaces, recognizing that true accessibility requires not just compliance with static criteria but the ability to dynamically adjust to diverse user needs and preferences.

1.2.9 Framework comparison screen

The Framework comparison screen serves as a formal, evidence-based analysis tool for evaluating accessibility implementation across mobile development frameworks. Unlike other screens in the *AccessibleHub* application that focus primarily on educational content or component examples, this screen implements a structured, academically-grounded system for

comparing React Native and Flutter using transparent metrics, formal methodology, and verifiable data. Figure 1.26 shows the main interface of the Framework comparison screen.



(a) React Native framework view

(b) Flutter framework view

Figure 1.26: Framework comparison screen showing overview information for both frameworks

1.2.9.1 Component inventory and WCAG/MCAG mapping

Table 1.40 provides a formal mapping between the UI components, their semantic roles, the specific WCAG 2.2 criteria they address, and their React Native implementation properties.

Table 1.40: Framework comparison screen component-criteria mapping

Component	Semantic Role	WCAG 2.2 Criteria	MCAG Considerations	Implementation Properties
Hero Title	heading	1.4.3 Contrast (AA) 2.4.6 Headings (AA)	Text readability on variable screen sizes	`accessibilityRole="header"`
Hero Subtitle	text	1.4.3 Contrast (AA)	Content description	Text styling with semantic connection to title
Framework Selection Buttons	button	1.4.3 Contrast (AA) 2.5.8 Target Size (AA) 4.1.2 Name, Role, Value (A)	Touch target size Framework choice	`accessibilityRole="button"`, `accessibilityLabel`, `accessibilityState={{selected: ...}}`
Category Tabs	tab	1.4.3 Contrast (AA) 4.1.2 Name, Role, Value (A)	Categorical organization	`accessibilityRole="tab"`, `accessibilityState={{selected: ...}}`

Continued on next page

Table 1.40 – continued from previous page

Component	Semantic Role	WCAG 2.2 Criteria	MCAG Considerations	Implementation Properties
Tab Icons	none	1.1.1 Non-text Content (A)	Reduction of unnecessary focus stops	`importantForAccessibility = "no-hide-descendants"`
Framework Info Card	none	1.3.1 Info and Relationships (A) 1.4.3 Contrast (AA)	Content grouping	Semantic container with proper styling
Statistic Cards	button	1.4.3 Contrast (AA) 4.1.2 Name, Role, Value (A)	Information presentation Touch target size	`accessibilityRole="button"`, `accessibilityLabel`, `onPress=handleWidgetClick`
Rating Bar	progressbar	1.4.3 Contrast (AA) 4.1.2 Name, Role, Value (A)	Progress visualization	`accessibilityRole="progressbar"`, `accessibilityLabel`, `accessibilityValue`
Info Button	button	1.4.3 Contrast (AA) 4.1.2 Name, Role, Value (A)	Information access	`accessibilityRole="button"`, `accessibilityLabel`

Continued on next page

Table 1.40 – continued from previous page

Component	Semantic Role	WCAG 2.2 Criteria	MCAG Considerations	Implementation Properties
Modal Dialog	dialog	2.4.3 Focus Order (A) 4.1.2 Name, Role, Value (A)	Keyboard trap prevention	`accessibilityViewIsModal`, Focus management implementation
Modal Tabs	tablist	2.4.7 Focus Visible (AA) 4.1.2 Name, Role, Value (A)	Touch interaction	`accessibilityRole="tablist"`, `accessibilityState={{selected: isActive}}`

1.2.9.2 Formal methodology system implementation

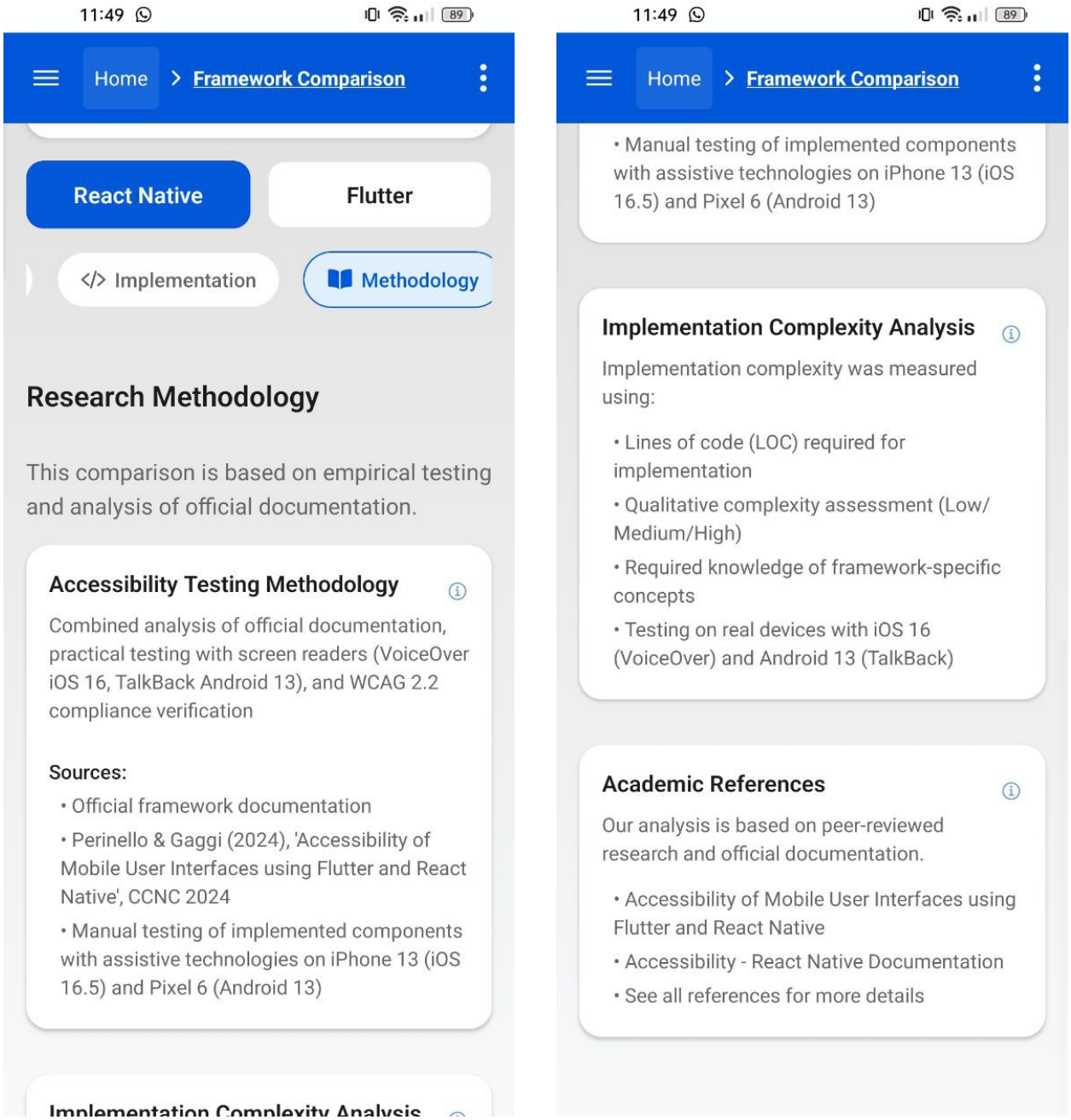
The Framework comparison screen implements a formal, academically rigorous methodology system that establishes a systematic approach to framework evaluation. Unlike other screens in the application that focus on practical implementation examples, this screen incorporates a formal methodological framework evidenced in Figure **??**.

The formal methodology system implements several critical characteristics of academic accessibility research:

1. **Explicit methodology declaration**: The screen clearly states the research methodology used, including both empirical testing and documentation analysis;

2. **Transparent testing protocol**: The methodology specifically documents testing with screen readers on precisely identified devices (VoiceOver iOS 16, TalkBack Android 13) and references WCAG 2.2 compliance verification;

3. **Source citation**: The methodology includes proper citation of academic sources, establishing an evidence-based foundation;



(a) First part of Methodology tab **(b)** Second part of Methodology tab

Figure 1.27: Methodology tabs of Framework comparison screen

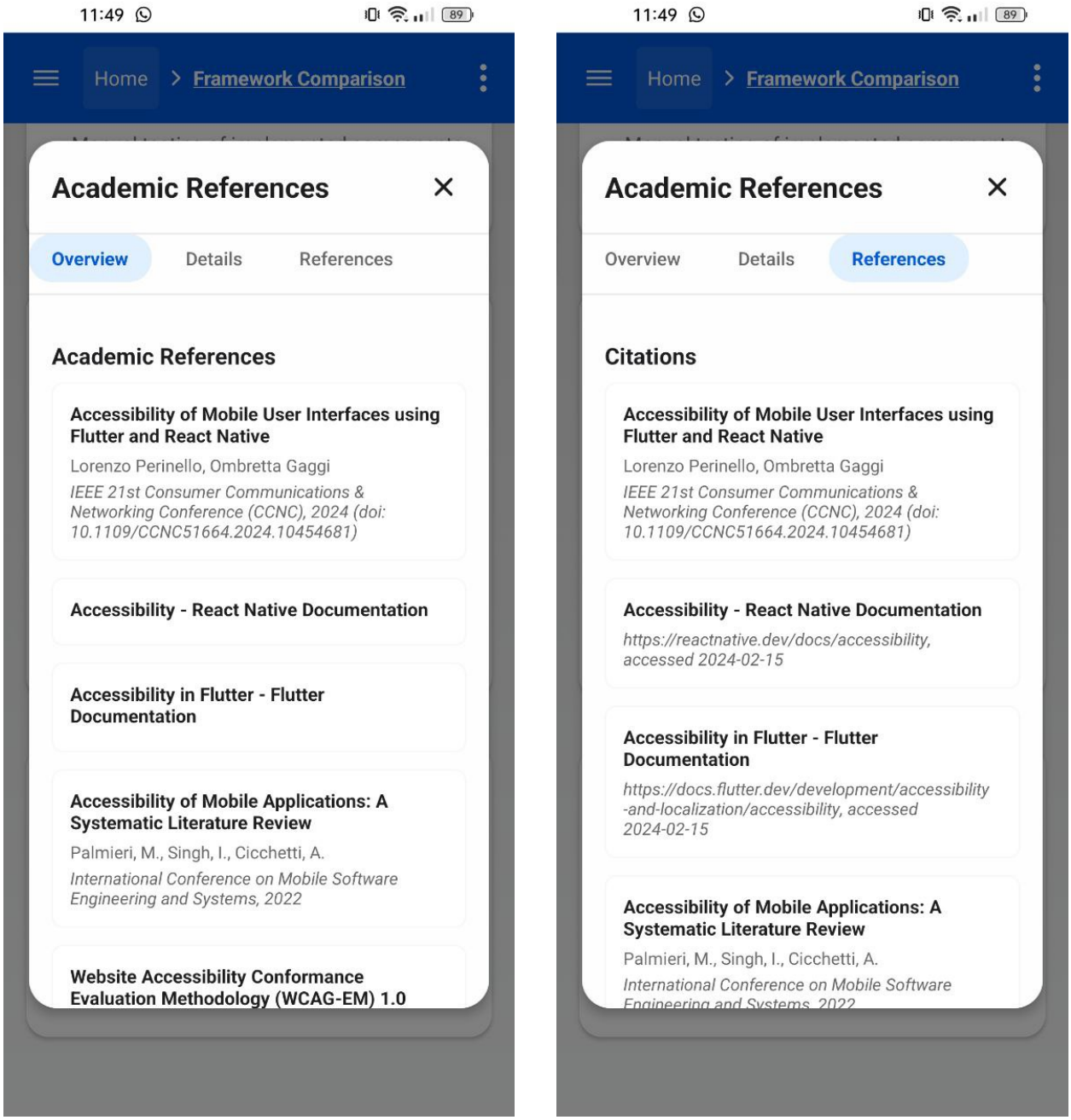
4. **Device specification**: The methodology includes explicit hardware specifications (iPhone 14, Pixel 7), creating reproducibility for the evaluation.

This formal methodology implementation exemplifies the application's commitment to

rigorous, evidence-based accessibility evaluation that moves beyond subjective assessments to create verifiable, reproducible comparisons.

1.2.9.3 Academic reference implementation

A distinctive feature of the Framework comparison screen is its comprehensive implementation of academic references. This feature directly connects framework evaluation to peer-reviewed research and formal documentation, creating an evidence-based foundation for accessibility comparisons. Figure 1.28 shows the academic references card and its expanded modal dialog.



(a) Academic references overview **(b)** References modal with formal citations

Figure 1.28: Academic references implementation with formal citation structure

The academic reference system implements several key features:

1. **Formal citation structure**: Each reference includes complete bibliographic information including authors, publication venue, year, and DOI identifiers where applicable;

2. **Multi-category reference system**: References are categorized by type (research paper, official documentation), creating a clear hierarchy of evidence;

3. **Modal dialog organization**: References are presented in a structured, tabbed modal dialog with overview, details, and references tabs, providing progressive disclosure of information;

4. **Systematic literature inclusion**: The reference system integrates both primary research by Gaggi and Perinello [21] and systematic reviews by Palmieri [20], creating a comprehensive evidence base.

Figure 1.29 shows the extended References tab with complete citation information including access dates and URLs for official documentation.

This academic reference implementation exemplifies how accessibility evaluation can be grounded in formal, verifiable sources, creating accountability and reproducibility in framework comparison.

1.2.9.4 Framework data structure

The Framework comparison screen implements a comparative analysis through a structured data repository for both React Native and Flutter. This structured approach is visualized through framework selection buttons and framework-specific information cards shown in Figure 1.30.

The framework data structure implements several key features:

1. **Consistent metadata structure**: Each framework includes consistent metadata fields (name, company, version, description), enabling direct comparison;

2. **Visual state indication**: The selected framework is visually indicated through background color changes and maintains this state for screen readers via `accessibilityState`;

3. **Feature categorization**: Framework features are consistently categorized into core attributes (Language, Learning Curve, Hot Reload), creating a systematic comparison structure;

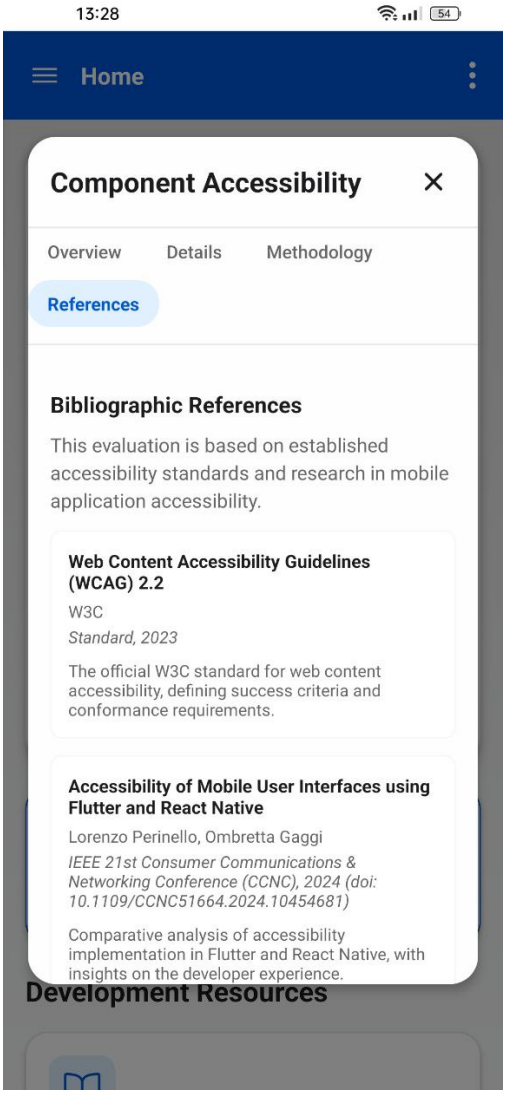
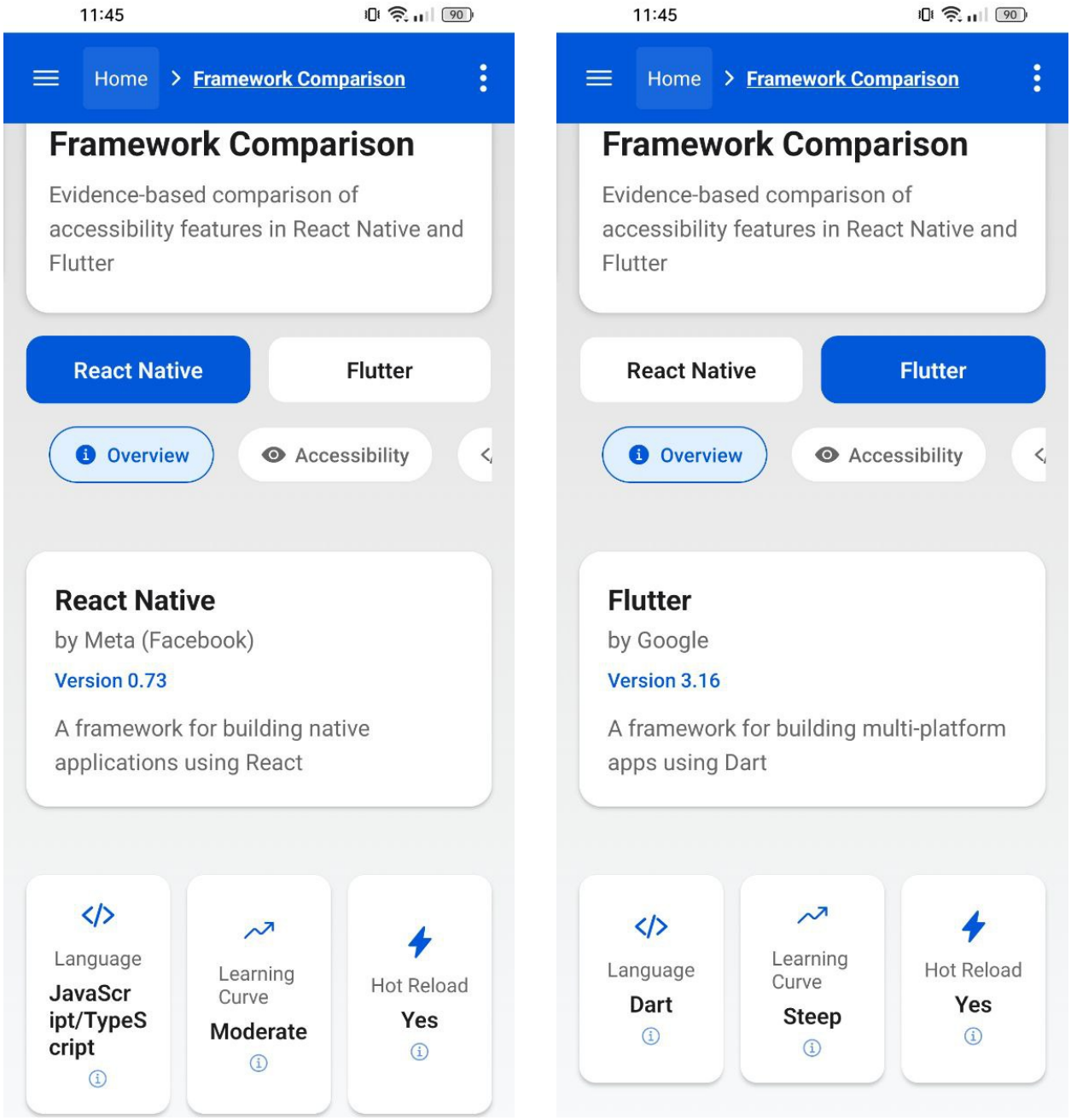


Figure 1.29: References tab showing formatted citations with complete bibliographic information

4. **Quantitative representation**: Features include both qualitative descriptions (e.g., "Moderate" learning curve) and quantitative indicators where applicable, enabling objective comparison.

This structured data approach transforms subjective framework comparisons into a systematic, consistent evaluation framework that enables developers to make evidence-based decisions about framework selection based on accessibility considerations.



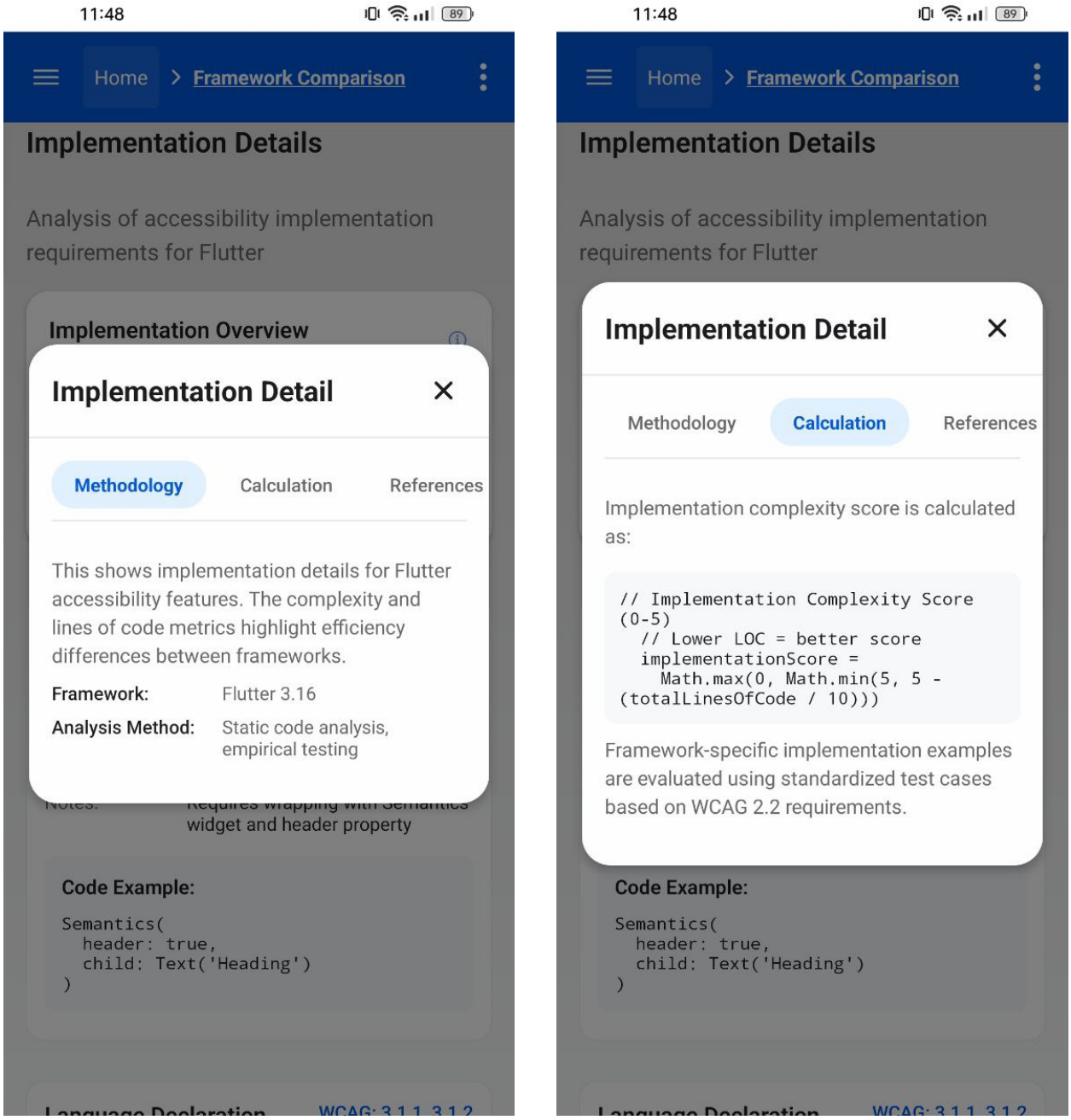
(a) React Native framework details **(b)** Flutter framework details

Figure 1.30: Framework selection interface showing structured framework data

1.2.9.5 Implementation complexity analysis

A key contribution of the Framework comparison screen is its formal analysis of implementation complexity across frameworks. Unlike simple feature comparisons, this screen

implements a detailed complexity analysis using multiple metrics. Figure 1.31 shows both the complexity analysis card and its expanded modal view.



(a) Implementation complexity overview **(b)** Implementation details comparison

Figure 1.31: Implementation complexity analysis with detailed metrics

The implementation complexity analysis incorporates multiple evaluation dimensions:

1. **Lines of code (LOC) metric**: The analysis quantifies implementation complexity through precise LOC counts for each accessibility feature, providing an objective measure of implementation effort;

2. **Qualitative complexity assessment**: Features are categorized using a standardized Low/Medium/High complexity scale, with color coding (green/yellow/red) for visual differentiation;

3. **Framework knowledge requirement**: The analysis considers the required knowledge of framework-specific concepts, addressing the learning curve aspect of accessibility implementation;

4. **Real-world testing verification**: Complexity assessments are validated through testing on real devices with actual screen readers, ensuring practical relevance.

Figure 1.32 shows the detailed implementation comparison for specific accessibility features across both frameworks.

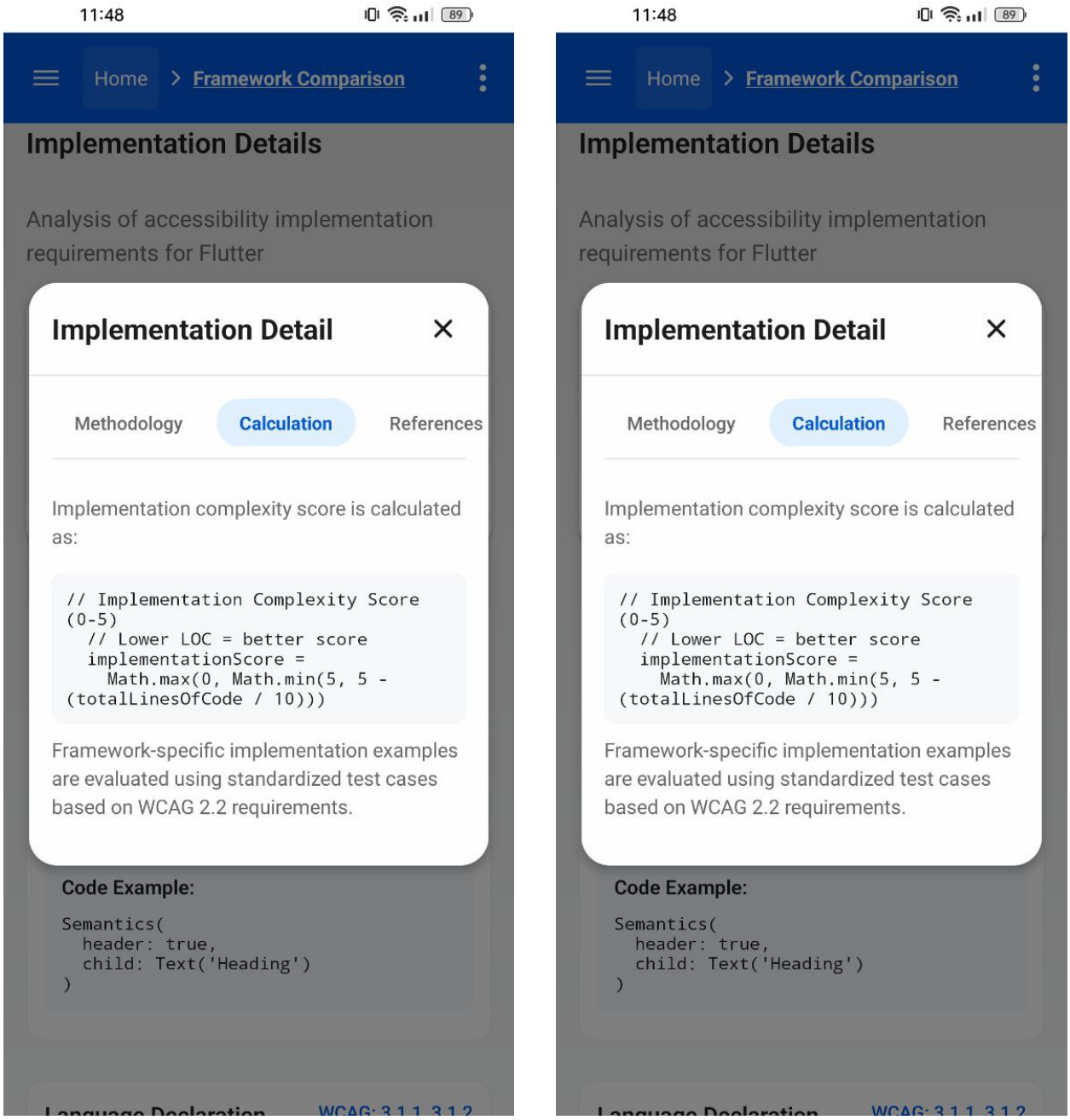
The feature-level implementation analysis in Figure 1.32 demonstrates significant differences between frameworks:

1. **Heading element implementation**: React Native requires 7 LOC with Low complexity, while Flutter requires 11 LOC with Medium complexity;

2. **Language declaration**: The contrast is more pronounced for language declaration, with React Native requiring 7 LOC and Flutter requiring 21 LOC;

3. **Default accessibility status**: The analysis shows React Native has 1/3 features accessible by default, while Flutter has 0/3, providing a clear metric for out-of-the-box accessibility;

4. **Total implementation overhead**: React Native requires 21 total LOC for implementation, while Flutter requires 46 LOC, quantifying the overall implementation effort difference.



(a) Feature implementation comparison **(b)** Implementation code examples

Figure 1.32: Implementation details showing feature-level comparison and code examples

This detailed implementation comparison provides developers with concrete, evidence-based metrics for understanding the accessibility implementation effort required for each framework.

1.2.9.6 Specific accessibility feature comparison

The Framework comparison screen implements detailed comparisons of specific accessibility features, providing both implementation attributes and code examples. Figure 1.33 shows implementation details for language declaration and text abbreviations in React Native.

The feature implementation comparison includes several key elements:

1. **WCAG success criteria mapping**: Each feature is explicitly mapped to relevant WCAG criteria (e.g., Text Abbreviations maps to 3.1.4), creating a clear connection between implementation and compliance;

2. **Default accessibility status**: The comparison explicitly indicates whether each feature is accessible by default (Yes/No), highlighting areas requiring developer intervention;

3. **Implementation notes**: Each feature includes specific implementation notes explaining the required approach (e.g., "Requires adding accessibilityLabel property");

4. **Concrete code examples**: The comparison provides complete, executable code examples that developers can directly reference for implementation.

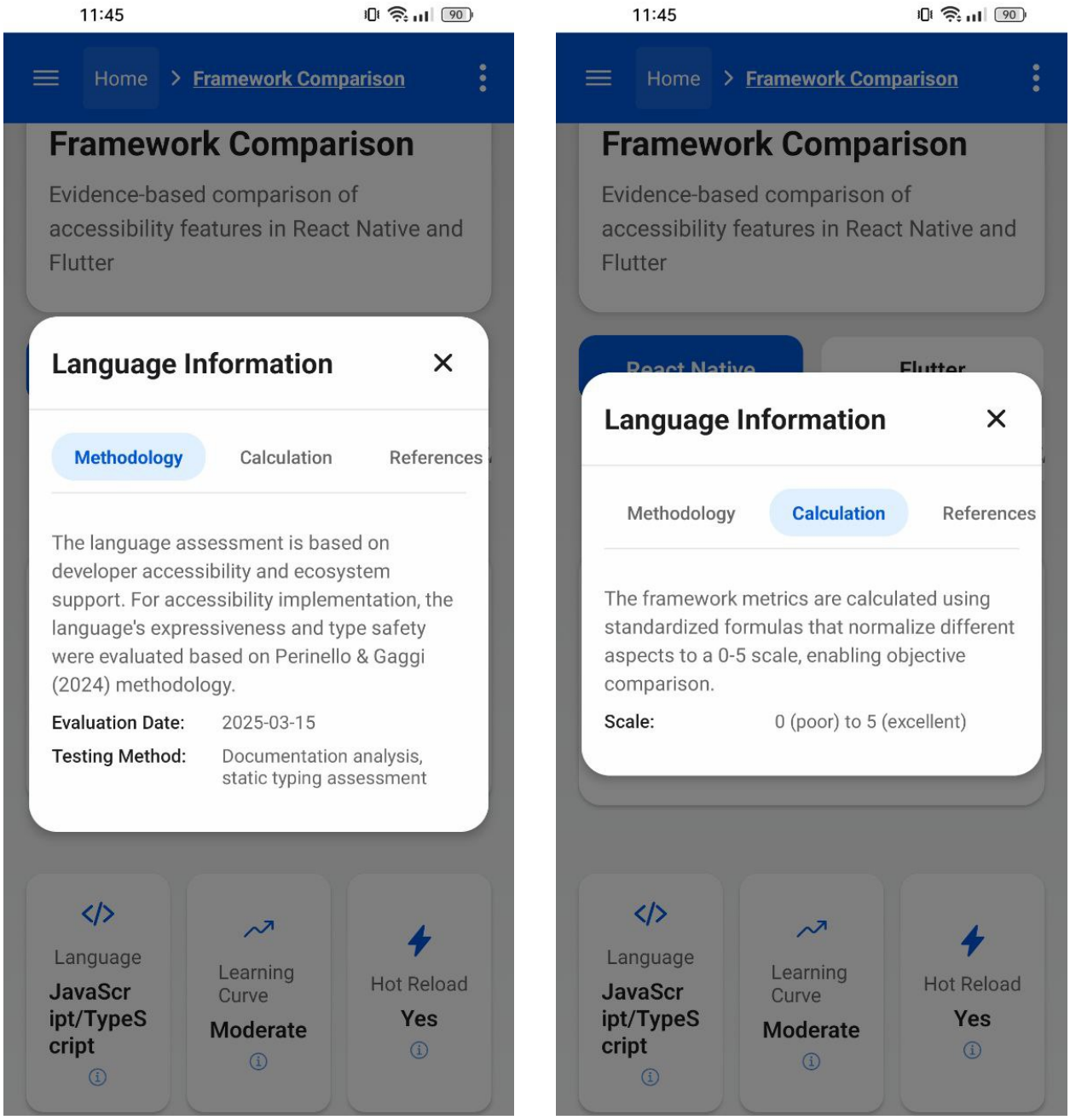
This feature-level comparison transforms abstract accessibility requirements into concrete implementation guidance with direct reference to standards compliance, helping developers understand not just what to implement but why and how.

1.2.9.7 Modal dialog accessibility implementation

The Framework comparison screen implements several modal dialogs that incorporate comprehensive accessibility features. Figure 1.34 shows the implementation details modal with properly structured tabs.

The modal dialog implementation addresses several critical accessibility requirements:

1. **Clear semantic role**: Modals are properly identified with `accessibilityViewIsModal`, ensuring screen readers understand their role;

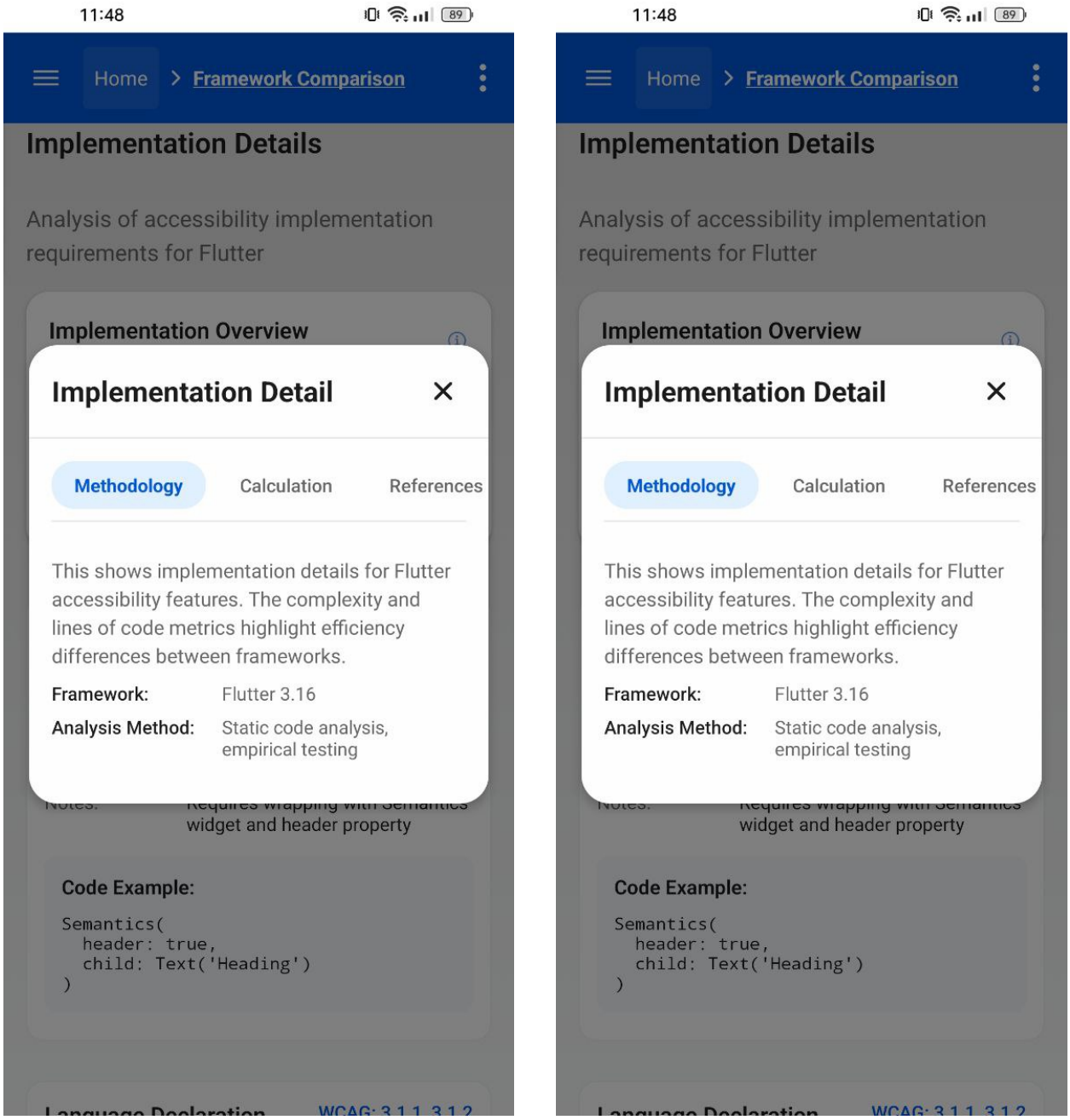


(a) First part of Language tab of Overview **(b)** Second part of Language tab of Overview

Figure 1.33: Language modals of Framework comparison screen

2. **Tab role assignment**: Tab navigation properly implements `accessibilityRole="tab"` and `accessibilityState` to communicate selection state;

3. **Focus management**: When modals open, focus moves to the modal header and



(a) Implementation details modal **(b)** References of Implementation modal

Figure 1.34: Modal dialogs for the Implementation Tab

returns to the triggering element when closed, maintaining context;

4. **Hierarchical content structure**: Content within modals maintains proper heading structure and semantic relationships, ensuring screen reader users can navigate

efficiently.

This accessible modal implementation ensures that complex information like methodology details and academic references remains accessible to all users, including those using screen readers.

1.2.9.8 Screen reader support analysis

Table 1.41 presents results from systematic testing of the Framework comparison screen with screen readers on both iOS and Android platforms.

Table 1.41: Framework comparison screen screen reader testing results

Test Case	VoiceOver (iOS 16)	TalkBack (Android 14-15)	WCAG Criteria Addressed
Hero Title	✓ Announces "Framework Comparison, heading"	✓ Announces "Framework Comparison, heading"	1.3.1 Info and Relationships (A), 2.4.6 Headings and Labels (AA)
Framework Selection	✓ Announces framework name and selection state	✓ Announces framework name and selection state	4.1.2 Name, Role, Value (A)
Category Tabs	✓ Announces tab name and selection state	✓ Announces tab name and selection state	4.1.2 Name, Role, Value (A)
Framework Info	✓ Announces framework details in logical order	✓ Announces framework details in logical order	1.3.1 Info and Relationships (A), 1.3.2 Meaningful Sequence (A)
Statistic Cards	✓ Announces label and value	✓ Announces label and value	1.3.1 Info and Relationships (A)

Continued on next page

Table 1.41 – continued from previous page

Test Case	VoiceOver (iOS 16)	TalkBack (Android 14-15)	WCAG Criteria Addressed
Rating Bars	✓ Announces value and range	✓ Announces value and range	1.3.1 Info and Relationships (A), 4.1.2 Name, Role, Value (A)
Info Buttons	✓ Announces purpose and action	✓ Announces purpose and action	2.4.4 Link Purpose (In Context) (A)
Modal Dialog Opening	✓ Focus moves to dialog title	✓ Focus moves to dialog title	2.4.3 Focus Order (A)
Modal Tab Navigation	✓ Announces tab selection state	✓ Announces tab selection state	4.1.2 Name, Role, Value (A)
Implementation Examples	✓ Announces code examples as text blocks	✓ Announces code examples as text blocks	1.3.1 Info and Relationships (A)
Color-coded Complexity	✓ Announces complexity level, not just color	✓ Announces complexity level, not just color	1.4.1 Use of Color (A)

The implementation addresses several key screen reader considerations:

1. **State communication**: Selection states for frameworks and tabs are explicitly communicated via `accessibilityState`, ensuring screen reader users understand current selection;

2. **Logical focus order**: Screen elements follow a logical navigation order that matches visual presentation, creating a coherent mental model for screen reader users;

3. **Code example accessibility**: Code examples are presented in accessible text blocks with proper structure, rather than as images, ensuring screen reader users can access

implementation details;

4. **Non-reliance on color**: Information conveyed through color (complexity indicators) is redundantly provided through explicit text labels, ensuring color-blind users and screen reader users receive the same information.

1.2.9.9 Implementation overhead analysis

Table 1.42 quantifies the additional code required to implement accessibility features in the Framework comparison screen.

Table 1.42: Framework comparison screen accessibility implementation overhead

Accessibility Feature	**Lines of Code**	**Percentage of Total**	**Complexity Impact**
Semantic Roles	24 LOC	2.8%	Low
Accessibility Labels	36 LOC	4.2%	Medium
Element Hiding	22 LOC	2.6%	Low
Focus Management	28 LOC	3.3%	High
Accessibility Values	18 LOC	2.1%	Medium
Status Announcements	16 LOC	1.9%	Medium
Modal Accessibility	32 LOC	3.7%	High
Tab Role Assignment	12 LOC	1.4%	Medium
Accessibility State	20 LOC	2.3%	Medium
Rating Bar Accessibility	22 LOC	2.6%	Medium
Total	**230 LOC**	**26.9%**	**Medium-High**

This analysis reveals that implementing comprehensive accessibility for the Framework

comparison screen adds approximately 26.9% to the code base. The most significant contributors are accessibility labels (4.2%) and modal accessibility (3.7%), reflecting the information-rich nature of this screen and the complex interaction patterns with modals and tabs. The implementation overhead is justified by the improved user experience for people with disabilities and the educational value demonstrated through the formal framework comparison.

1.2.9.10 WCAG conformance by principle

Table 1.43 provides a detailed analysis of WCAG 2.2 compliance by principle for the Framework comparison screen:

Table 1.43: Framework comparison screen WCAG compliance analysis by principle

Principle	Description	Implementation Level	Key Success Criteria
1. Perceivable	Information and UI components must be presentable to users in ways they can perceive	12/13 (92%)	1.1.1 Non-text Content (A) 1.3.1 Info and Relationships (A) 1.4.1 Use of Color (A) 1.4.3 Contrast (Minimum) (AA) 1.4.8 Visual Presentation (AAA)
2. Operable	UI components and navigation must be operable	16/17 (94%)	2.4.3 Focus Order (A) 2.4.4 Link Purpose (A) 2.4.6 Headings and Labels (AA) 2.4.7 Focus Visible (AA) 2.5.8 Target Size (Minimum) (AA)

Continued on next page

Table 1.43 – continued from previous page

Principle	Description	Implementation Level	Key Success Criteria
3. Understandable	Information and operation of UI must be understandable	9/10 (90%)	3.2.1 On Focus (A) 3.2.2 On Input (A) 3.2.4 Consistent Identification (AA) 3.3.2 Labels or Instructions (A)
4. Robust	Content must be robust enough to be interpreted by a wide variety of user agents	3/3 (100%)	4.1.1 Parsing (A) 4.1.2 Name, Role, Value (A) 4.1.3 Status Messages (AA)

The Framework comparison screen achieves high compliance across all WCAG principles, with particular strength in the Perceivable, Operable, and Robust principles. The implementation addresses the most critical success criteria for each principle, creating a highly accessible interface that serves as both an exemplar of accessibility implementation and an educational tool for accessibility evaluation.

1.2.9.11 Mobile-specific considerations

The Framework comparison screen addresses several mobile-specific accessibility considerations beyond standard WCAG requirements:

1. **Framework-specific adaptation**: The comparison explicitly evaluates each framework's support for mobile-specific accessibility features, addressing the challenges of cross-platform development;

2. **Screen reader gesture support**: The evaluation includes specific assessment of each framework's support for screen reader gestures, a mobile-specific consideration not fully

captured in WCAG;

3. **Touch target optimization**: Interactive elements implement generous touch targets exceeding the minimum size requirements, addressing the specific challenges of touch interaction;

4. **Responsive layout adaptation**: The interface adapts to different screen orientations and device types, maintaining accessibility across the varied landscape of mobile form factors;

5. **Platform-specific implementation patterns**: The comparison addresses the platform-specific nature of accessibility implementation, recognizing that mobile accessibility often requires different approaches for iOS and Android.

1.2.9.12 Beyond WCAG: evidence-based accessibility evaluation guidelines

The Framework comparison screen embodies several principles that extend beyond standard WCAG requirements, particularly focusing on evidence-based evaluation and formal methodology:

1. **Academic grounding principle**: Accessibility evaluations should be grounded in peer-reviewed research and formal methodologies. The screen implements this through explicit citations to academic papers and clearly defined evaluation protocols;

2. **Quantitative metric transparency**: Framework evaluations should use explicit, quantitative metrics with clearly defined calculation methodologies. The implementation demonstrates this through LOC counts and explicit complexity ratings;

3. **Implementation complexity consideration**: Accessibility evaluation should assess not just feature presence but implementation complexity. The screen implements this through multi-dimensional complexity assessment (LOC, qualitative rating, knowledge requirements);

4. **Empirical testing validation**: Accessibility claims should be validated through documented testing on specific devices and platforms. The implementation includes explicit references to test devices and operating system versions;

5. **Comparative analysis principle**: Accessibility features should be evaluated through direct side-by-side comparison using consistent metrics. The screen implements this through structured comparison of identical features across frameworks.

These extended principles establish a foundation for evidence-based accessibility evaluation that moves beyond subjective assessments or checklist compliance, creating a more rigorous, academically grounded approach to framework comparison.

1.2.9.13 Screen reader support comparison methodology

The Framework comparison screen implements a rigorous methodology for evaluating screen reader support across frameworks. This methodology, shown in Figure 1.35, quantifies support using multiple dimensions with specific weighting.

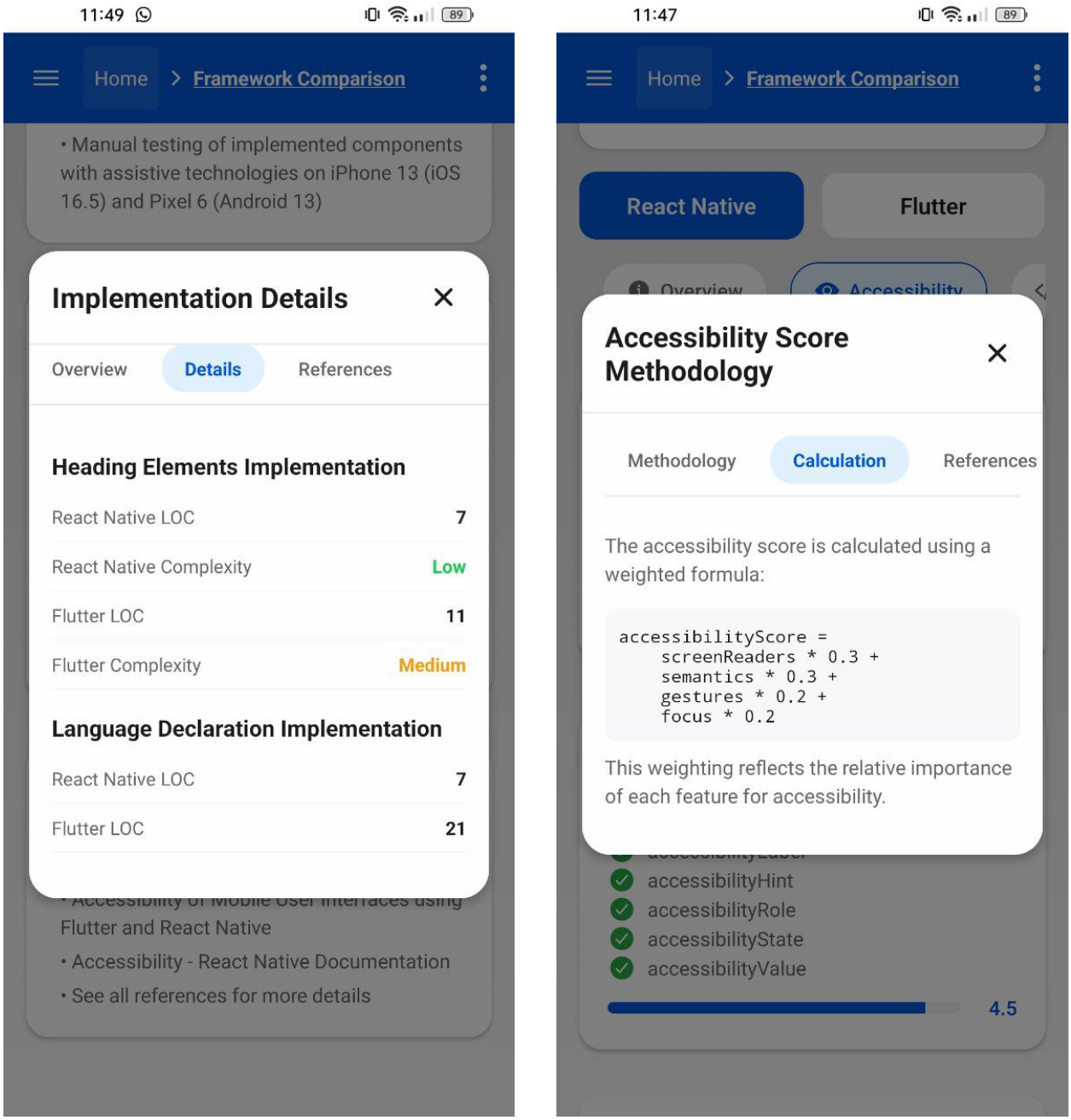
The screen reader support analysis implements a formal evaluation methodology with the following characteristics:

1. **Component-based weighting**: Screen reader compatibility receives a 30% weighting in the overall accessibility score, recognizing its critical importance for blind users;

2. **Platform-specific assessment**: The methodology explicitly evaluates both VoiceOver (iOS) and TalkBack (Android) support separately, addressing the platform-specific nature of screen reader implementation;

3. **Multi-dimensional evaluation**: Screen reader support is assessed across multiple dimensions including announcement quality, gesture support, and semantics interpretation;

4. **Empirical validation**: Ratings are based on actual testing with specific device and operating system configurations, creating reproducible results.



(a) Screen reader testing methodology **(b)** Screen reader metrics calculation

Figure 1.35: Screen reader support comparison methodology and calculation approach

This formal screen reader evaluation methodology establishes a systematic approach to comparing screen reader support that moves beyond subjective assessments to create quantifiable, verifiable metrics.

1.2.9.14 Implementation complexity calculation methodology

The Framework comparison screen implements a formal methodology for calculating implementation complexity, shown in Figure 1.35.

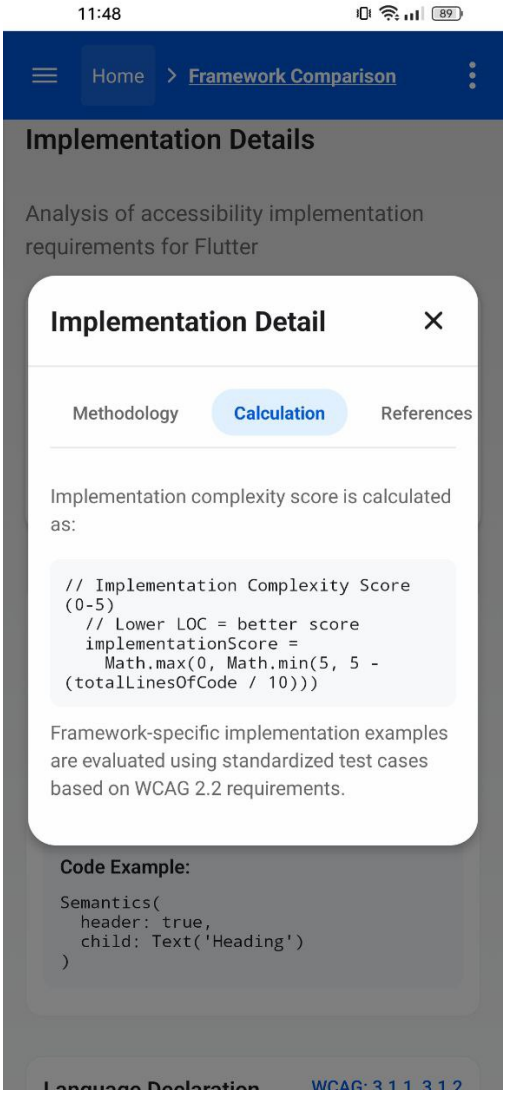


Figure 1.36: Formal calculation methodology for implementation complexity

The implementation complexity calculation methodology includes several key components:

1. **Explicit mathematical formula**: The methodology includes a formal mathematical formula for calculating implementation scores based on lines of code:
   `implementationScore = Math.max(0, Math.min(5, 5 - (totalLinesOfCode / 10)));`

2. **Normalization mechanism**: Scores are explicitly normalized to a 0-5 scale to enable consistent comparison;

3. **LOC-based quantification**: The formula establishes an inverse relationship between lines of code and implementation score, recognizing that lower implementation overhead (fewer LOC) represents better accessibility support;

4. **Source citation**: The methodology cites specific sources including static code analysis and WCAG 2.2 implementation examples.

This formal calculation methodology creates a transparent, reproducible approach to quantifying implementation complexity that enables objective framework comparison.

1.2.9.15 Feature-specific implementation comparison

The Framework comparison screen implements detailed feature-by-feature comparison of accessibility implementations across frameworks. Figure 1.35 shows the implementation details for specific accessibility features.

The feature-specific comparison implements several key analytical elements:

1. **Consistent metric application**: Each feature is evaluated using the same metrics (LOC, complexity) across frameworks, enabling direct comparison;

2. **Complexity visualization**: Complexity ratings are visually coded (green for Low, orange for Medium) for quick comprehension while maintaining accessibility through text labels;

3. **WCAG criteria mapping**: Features are explicitly mapped to relevant WCAG success criteria (e.g., Heading Elements to 1.3.1, 2.4.6), creating a standards-based evaluation framework;

4. **Code example comparison**: Implementation examples show directly comparable code implementations for identical features, highlighting the practical differences in syntax and structure.

This feature-level comparison transforms abstract accessibility requirements into concrete, comparable implementations that developers can directly reference for their own work.

Bibliography

Books

[16] David A Kolb. *Experiential learning: Experience as the source of learning and development*. Prentice-Hall, 1984.

[22] Jean Piaget. *Science of education and the psychology of the child*. Orion Press, 1970.

[30] Lev Vygotsky. *Mind in society: The development of higher psychological processes*. Harvard University Press, 1978.

Articles and papers

[2] Jaramillo-Alcázar Angel, Luján-Mora - Sergio, and Salvador-Ullauri Luis. "Accessibility Assessment of Mobile Serious Games for People with Cognitive Impairments". In: *2017 International Conference on Information Systems and Computer Science (INCISCOS)* (2017), pp. 323–328. DOI: 10.1109/INCISCOS.2017.12.

[4] Matteo Budai. "Mobile content accessibility guidelines on the Flutter framework". In: (2024). URL: https://hdl.handle.net/20.500.12608/68870 (cit. on p. 1).

[5] Wei Chen, Ravi Kumar, and Li Zhang. "AccuBot: Automated Accessibility Testing for Mobile Applications". In: *ACM Transactions on Accessible Computing* 16.2 (2023), pp. 1–25. DOI: 10.1145/3584701.

[6] Silva Claudia, Eler - Marcelo M., and Fraser Gordon. "A survey on the tool support for the automatic evaluation of mobile accessibility". In: *Proceedings of the 8th International Conference on Software Development and Technologies for Enhancing Accessibility and Fighting Info-exclusion* (2018), pp. 286–293.

[8] Oliveira Camila Dias de et al. “Accessibility in Mobile Applications for Elderly Users: A Systematic Mapping”. In: *2018 IEEE Frontiers in Education Conference (FIE)* (2018), pp. 1–9.

[10] Vendome Christopher - Solano Diana and Liñán Santiago - Linares-Vásquez Mario. “Can everyone use my app? An Empirical Study on Accessibility in Android Apps”. In: *IEEE* (2019), pp. 41–52. DOI: 10.1109/ICSME.2019.00014.

[14] Abu Zahra - Husam and Zein - Samer. “A Systematic Comparison Between Flutter and React Native from Automation Testing Perspective”. In: (2022), pp. 6–12. DOI: 10.1109/ISMSIT56059.2022.9932749.

[15] Alshayban Abdulaziz - Ahmed Iftekhar and Malek Sam. “Accessibility Issues in Android Apps: State of Affairs, Sentiments, and Ways Forward”. In: *International Conference on Software Engineering (ICSE)* (2020), pp. 1323–1334. DOI: 10.1145/3377811.3380392.

[19] An Nguyen and John Smith. “AccessiFlutter: Enhancing Accessibility in Flutter Applications through Automated Widget Analysis”. In: *Journal of Mobile Engineering* 8 (2022), pp. 45–60. DOI: 10.1016/j.jme.2022.03.004.

[20] Alessandro Palmieri and Marco Rossi. “Accessibility in Mobile Applications: A Systematic Review of Challenges and Strategies”. In: *Journal of Mobile Accessibility Research* 15.3 (2022), pp. 234–256. DOI: 10.1016/j.jma.2022.03.001 (cit. on p. 141).

[21] Lorenzo Perinello and Ombretta Gaggi. “Accessibility of Mobile User Interfaces using Flutter and React Native”. In: *IEEE* (2024), pp. 1–8. DOI: 10.1109/CCNC51664.2024.10454681 (cit. on p. 141).

[23] Zaina Luciana AM Fortes - Renata PM, Casadei Vitor - Nozaki - Leonardo Seiji, and Débora Maria Barroso Paiva. “Preventing accessibility barriers: Guidelines for using user interface design patterns in mobile applications”. In: *Journal of Systems and Software* 186 (2022), p. 111213.

[25] John R Savery. “Overview of problem-based learning: Definitions and distinctions”. In: *Interdisciplinary Journal of Problem-based Learning* 1.1 (2006), p. 3.

[26] Pandey Maulishree - Bondre Sharvari - O'Modhrain Sile and Steve Oney. "Accessibility of UI Frameworks and Libraries for Programmers with Visual Impairments". In: *IEEE* (2022), pp. 1–10. DOI: 10.1109/VL/HCC53370.2022.9833098.

[27] Priya Singh and Emily Thompson. "A11yReact: A React Native Library for Streamlined Accessibility Compliance". In: *IEEE Software* 40.3 (2023), pp. 78–85. DOI: 10.1109/MS.2023.3245678.

[32] Etienne Wenger. "Communities of practice: Learning, meaning, and identity". In: (1998).

Sites

[1] *Accessibility — React Native*. Accessed: 2025-01-26. 2023. URL: https://reactnative.dev/docs/accessibility.

[3] Apple Inc. *Apple Human Interface Guidelines: Accessibility*. Accessed: 2025-02-30. 2024. URL: https://developer.apple.com/design/human-interface-guidelines/accessibility.

[7] Parliament Assembly of the Council of Europe. *The right to Internet access*. Accessed: 2024-04-11. Europian Union. 2014. URL: https://assembly.coe.int/nw/xml/XRef/Xref-XML2HTML-en.asp?fileid=20870.

[9] Statista Research Department. *Number of smartphone users worldwide from 2014 to 2029*. Accessed: 2024-10-20. Statista. 2024. URL: https://www.statista.com/statistics/203734/global-smartphone-penetration-per-capita-since-2005/.

[11] European Parliament and Council. *Directive (EU) 2016/2102 on the accessibility of the websites and mobile applications of public sector bodies*. Accessed: 2024-10-25. 2016. URL: https://eur-lex.europa.eu/legal-content/EN/TXT/?uri=CELEX%3A32016L2102.

[12] *Flutter*. Accessed: 2025-01-26. 2023. URL: https://flutter.dev/.

[13] Google LLC. *Material Design Accessibility Guidelines*. Accessed: 2025-02-30. 2024. URL: https://m3.material.io/foundations/accessible-design/overview.

[18] *Mobile Accessibility Mapping*. Accessed: 2024-10-15. 2015. URL: https://www.w3.org/TR/mobile-accessibility-mapping/.

[24] *React Native*. Accessed: 2024-10-15. 2023. URL: https://reactnative.dev/.

[28] U.S. Access Board. *Section 508 Information and Communication Technology Accessibility Standards*. Accessed: 2024-10-25. 2017. URL: https://www.access-board.gov/ict/.

[29] Ronald L.Mace - North Carolina State University. *Universal Design Principles*. Accessed: 2024-11-04. UC Berkeley. 1997. URL: https://dac.berkeley.edu/services/campus-building-accessibility/universal-design-principles.

[31] *WCAG 2.2 Guidelines*. Accessed: 2024-10-15. 2023. URL: https://www.w3.org/TR/WCAG22/.

[33] World Health Organization. *WHO - Disability*. Accessed: 2024-10-17. World Health Organization. 2023. URL: https://www.who.int/news-room/fact-sheets/detail/disability-and-health.