AccessibleHub Presentation Script

Slide 1: Title Slide

Good morning. I'm Gabriel, and today I'm solving a fundamental problem plaguing mobile development: **How much does accessibility actually cost?**

This research delivers the first comprehensive quantitative framework for measuring mobile accessibility implementation costs. For the first time, project managers can budget accessibility with **empirical data** rather than fear-based estimates.

Key finding: React Native requires **45% less code** than Flutter for equivalent accessibility implementation. This transforms framework selection from subjective preference to evidence-based decision-making.

Slide 2: "First and not least: Mobile accessibility"

Before diving into our research framework, we must establish the **No** in today's digital landscape. Mobile accessibility is defined as enabling users to fully perceive, understand, navigate, and interact with digital content regardless of capabilities. But the mobile reality creates unprecedented implementation challenges that existing frameworks weren't designed to address.

- Consider the scale: **1.3 billion people worldwide live with disabilities** according to WHO 2023 data, while we have 6.8 billion smartphone users globally. This intersection creates massive potential impact for accessibility implementation - but mobile interfaces introduce barriers that didn't exist in web contexts.
- Unlike web accessibility, **mobile presents unique technical constraints**: touch-based interactions lack precision and require larger targets; limited screen real estate creates tension between content density and accessibility; screen reader gestures may conflict with application gestures requiring alternative methods; and iOS VoiceOver versus Android TalkBack behave fundamentally differently.

These aren't minor variations - they're architecturally different approaches requiring different implementation strategies. This complexity demands systematic, quantitative approaches rather than intuitive methods.

Slide 3: "First and not least: Mobile challenges"

Mobile accessibility presents systematically different challenges from web accessibility, requiring specialized frameworks and measurement approaches. Let me categorize these into two critical domains.

- **Touch interaction barriers** create immediate implementation complexity. Target sizes lack standardization across platforms and often fall below accessibility guidelines - we need empirical data to balance interface design with usability requirements. Complex gestures

exclude users with motor impairments, requiring alternative methods. One-handed operation affects significant portions of users, including those with temporary or permanent constraints.

- **Mobile context issues** introduce architectural considerations. Small screens fundamentally affect content hierarchy and information architecture, requiring careful accessibility feature prioritization within limited space.
    - Orientation changes disrupt navigation patterns and screen reader focus management, demanding robust responsive accessibility. Performance impact becomes critical - accessibility features must function effectively while maintaining battery efficiency.

These challenges demonstrate why mobile accessibility cannot simply adapt web approaches - it requires dedicated frameworks, specific implementation patterns, and crucially, quantitative metrics for measuring implementation effectiveness across platforms.

Slide 4: "Accessibility guidelines gap"

Here's where we encounter the fundamental problem that motivated this entire research. We have excellent theoretical foundations, but they're systematically failing to serve mobile developers in practice.

Let's examine our current standards landscape with precision. **WCAG 2.2**, released in October 2023, provides **four core principles - Perceivable, Operable, Understandable, and Robust** - the POUR framework - along with **three conformance levels: A, AA, and AAA**. But it remains fundamentally **web-focused** in its examples and implementation guidance, with success criteria written for web pages, not mobile screens.

MCAG from 2015 attempted mobile adaptation, but here's the critical issue - it was **based on WCAG 2.0**, missing crucial criteria from WCAG 2.1 and 2.2 that specifically address mobile interaction patterns like **touch target sizing, pointer gestures, and orientation handling**. **WCAG2Mobile**, introduced in January 2025 by the W3C Mobile Accessibility Task Force, offers the most recent mobile guidance, but crucially provides **only interpretations of existing criteria, not implementation frameworks**.

This creates three cascading problems that directly impact development practice. First, we have an **outdated foundation**. MCAG lacks the mobile-specific criteria from WCAG 2.1 and 2.2 that address specific mobile criteria (es. **2.5.5 Target Size Enhanced, 2.4.11 Focus Not Obscured, and 2.5.8 Target Size Minimum**). These aren't minor gaps - they're fundamental missing pieces for modern mobile accessibility.

Second, there's a massive **implementation void**. None of these standards provide practical frameworks for mobile developers. They tell you **what to achieve** - 'ensure content is perceivable' - but not **how to achieve it** with accessibilityLabel properties in React Native versus Semantics widgets in Flutter, or what the relative costs are for different approaches.

Third, **knowledge fragmentation** has become endemic. Resources are scattered across documentation sites, blog posts, and community forums, with unclear cost implications and no systematic comparison methodologies. The **result** is that mobile developers fundamentally lack

comprehensive accessibility implementation guidance, forcing them to piece together approaches without understanding trade-offs or implementation costs.

Slide 5: "Platform implementation gap"

Now we reach the technical complexity that makes systematic measurement so critical. The platform fragmentation in mobile accessibility is genuinely staggering and creates implementation challenges that simply don't exist in web development.

Consider the platform differences we're dealing with. **iOS** implements **VoiceOver** with its unique gesture system - three-finger swipes, rotor navigation, and specific announcement patterns. It includes **Voice Control** for users who can't perform touch gestures, and **Switch Control** for alternative navigation methods. **Android** provides **TalkBack** with completely different gesture patterns - two-finger scrolling, different focus management, and distinct announcement behaviors. It includes **Switch Access** and keyboard navigation that behaves fundamentally differently than iOS equivalents.

These aren't minor variations - they're architecturally different approaches to accessibility that require different implementation strategies.

Frameworks attempt to bridge this complexity through divergent architectural philosophies. **Flutter** takes the unified approach - it constructs a **single accessibility tree** that gets translated to platform-specific implementations through what they call a **platform adaptation layer**. This sounds elegant in theory, but in practice, it means accessibility issues can be obscured by abstraction layers, making debugging and optimization more complex.

React Native chooses the opposite philosophy - **platform-specific accessibility props** that map directly to native APIs through **bridge architecture**. This maintains platform-native behavior but requires developers to understand platform differences explicitly.

Here's where **Perinello and Gaggi's research applied into Budai's thesis** becomes crucial. Budai conducted systematic Flutter component accessibility testing and documented the fragmented knowledge landscape. His work demonstrated that while Flutter widgets show promise for accessibility, developers lack systematic guidance for comprehensive implementation.

But Budai's research revealed the critical gap that defines our research space: **no comprehensive learning resource bridges platforms with quantitative implementation cost analysis**. Developers are left making framework choices without understanding accessibility implications, implementation overhead, or cross-platform consistency trade-offs.

Slide 6: "AccessibleHub - Bridging the gap"

This brings us to how AccessibleHub directly addresses these systematic gaps through a dual methodology that serves both immediate practical needs and rigorous research requirements.

I structured the investigation around three research questions that operationalize the core challenges we've just discussed.

- **RQ1** asks whether React Native components are accessible by default - this isn't just academic curiosity, it's fundamental project planning information. If 38% of components work out of the box versus 32%, that dramatically changes development timelines and resource allocation.
- **RQ2** examines whether non-accessible components can be made accessible - this validates framework capabilities and identifies potential limitations or complexity barriers that might make certain accessibility features impractical or prohibitively expensive to implement.
- **RQ3** - and this is the fundamental question - quantifies implementation cost through **lines of code analysis**. How many lines of code must be added to components to make them accessible? A practical example coming from this research: **text language implementation** requires **7 lines in React Native versus 21 lines in Flutter** - that's a **200% overhead difference**.

But here's what makes this research methodology innovative: AccessibleHub isn't just a tool for generating research data. It's a complete React Native application tested on both Android and iOS that serves as an interactive accessibility manual for mobile developers. Every screen becomes a controlled case study that simultaneously solves real developer problems while generating measurable data.

When developers use AccessibleHub to learn button accessibility implementation, they're accessing the results of systematic measurement showing exactly how many lines of code, what complexity factors, and which WCAG criteria are involved. This transforms the abstract question 'how much will accessibility cost?' into empirical data that project managers can use for sprint planning and resource estimation

---

Slide 7: "AccessibleHub - Overview"

Now let me show you how AccessibleHub through its comprehensive five-section architecture, where each component serves dual educational and research functions through systematic measurement – **in case talk to prof. and see if we want to add a video here.**

Accessible Components forms the practical foundation. Developers access over 20 UI implementation patterns organized into four progressive categories.

- **Buttons and Touchables** covers fundamental interaction accessibility - the building blocks every mobile app needs;
- **Forms** addresses complex input patterns that go beyond basic compliance into real-world usability;
- **Dialogs** manages focus behavior in modal contexts, one of the trickiest accessibility challenges in mobile development;
- **Media** implements comprehensive alternative content strategies beyond simple alt text.

Each category provides complete, copyable code examples that developers can integrate directly into their projects. But this isn't just documentation - every pattern generates precise Implementation Overhead measurements. When developers view button accessibility implementation, they're seeing the results of systematic analysis showing that basic button

accessibility requires exactly 12 lines in React Native with 13.3% overhead and Low complexity classification.

Best Practices directly attacks the WCAG abstraction problem we discussed. Instead of abstract guidance like 'ensure proper heading structure,' developers see the exact implementation: accessibilityRole="header" with specific WCAG 2.2 criteria mapping.

- **WCAG Guidelines** provides concrete implementation patterns;
- **Gesture Tutorials** demonstrates interaction equivalence;
- **Semantic Structure** establishes hierarchical organization;
- **Logical Navigation** manages focus behavior;
- **Screen Reader Support** provides adaptive guidance based on our empirical testing results.

Tools and Settings creates our comprehensive testing methodology documentation while providing practical customization options.

- **Tools** catalogues testing resources for real development workflows - not just links, but integrated guidance for how these tools fit into actual development processes;
- **Settings** demonstrates accessibility customization through live examples - dark mode, high contrast adjustment, reduced motion, text scaling - that developers can experience firsthand. This reinforces that accessibility is about personalization rather than one-size-fits-all solutions.

Framework Comparison operationalizes our complete research methodology into an interactive analysis tool. Users explore the architectural differences between React Native's property-based model and Flutter's widget-based approach, examine empirical implementation overhead measurements, and analyze Screen Reader Support Scores based on our systematic testing across both VoiceOver and TalkBack.

Instruction and Community addresses accessibility's collaborative nature, providing pathways for ongoing learning and community connection beyond individual technical implementation.

The research innovation emerges from comprehensive measurement: every screen analyzed as a controlled case study, over 20 components tested systematically with both major screen readers, and cross-platform validation ensuring implementation patterns work universally. This creates the first empirical framework where educational content directly reflects quantified research findings, bridging theory to practice through systematic measurement rather than anecdotal guidance.

Slide 8: "Systematic analysis approach"

Here's the methodological innovation that makes this research possible - what I call the systematic solution to the **WCAG transformation challenge** that forms the core of our accessibility implementation guidelines methodology.

The fundamental challenge is this: WCAG guidelines exist at a level of abstraction that makes direct implementation nearly impossible. Look at this transformation methodology - it shows exactly how I bridge that gap through **three systematic layers** that embeds accessibility into the development process rather than treating it as post-implementation testing.

- **Layer one: Theoretical foundation.** This encompasses the abstract principles and success criteria defined by **WCAG, MCAG**, and **WCAG2Mobile**. For example, WCAG's four core principles require that content be perceivable, operable, understandable, and robust. But these criteria serve only as our analytical benchmark - they don't tell developers how to code.
- **Layer two: Implementation patterns.** Here's where the innovation happens - I translate abstract requirements into concrete code structures within mobile development contexts.
    - This involves systematic use of React Native accessible properties like `accessibilityLabel` and `accessibilityRole`, but more importantly, it establishes **replicable patterns** that developers can apply consistently. Every pattern is tested, measured, and validated.
- **Layer three: Screen-based analysis.** This considers how end users actually interact with components through assistive technologies - proper focus management, screen reader behavior, real-world usability context. This isn't theoretical - it's empirical testing with **VoiceOver on iPhone 14 with iOS 16** and **TalkBack on Google Pixel 7 with Android 15**.

The **basic workflow** demonstrates how this enables data-driven accessibility decisions: **Abstract WCAG principles** transform into **Implementation Patterns**, which generate **Quantified Metrics**, which populate our **Educational Platform**. Each stage builds on systematic measurement rather than subjective assessment.

But here's what makes this revolutionary: I first map UI elements to their semantic roles, then link each component to relevant WCAG and MCAG criteria with both minimum compliance and potential enhancements, and finally describe the technical solution with **quantified overhead analysis**. This structured approach doesn't just bridge abstract guidelines to code - it generates the empirical data that enables evidence-based framework comparison and project planning.

This approach scales from the **individual screen analysis in Section 3.4**, extends through **comprehensive screen documentation in technical appendix Chapter 1**, and establishes the **foundation for comparative framework analysis in Chapter 4**. The systematic nature ensures that when we compare React Native and Flutter, we're applying identical evaluation criteria rather than subjective assessment.

Slide 9: "Systematic analysis - Example (1)"

Now let me show you exactly how this three-layer systematic methodology works in practice through the **Components screen analysis** - demonstrating the accessibility implementation guidelines methodology applied to real interface components with measurable, replicable results.

What you're seeing here represents how the **three-layer approach works in practice**. The Components screen isn't just educational content - it's a **controlled case study** where every element undergoes identical systematic evaluation, generating the empirical data that supports our comparative framework analysis. Here's how the **three-layer methodology** works in practice:

- **Layer one** establishes our theoretical foundation - each interface component maps to specific WCAG 2.2 criteria with **WCAG2Mobile mobile-specific considerations**. The **Buttons & Touchables** section addresses **Success Criteria 2.5.8 Target Size, 4.1.2 Name**

Role Value, and 2.5.2 Pointer Cancellation with mobile interpretations for touch target optimization and one-handed operation.

- **Layer two** translates these requirements into **implementation patterns**. Essential components like buttons require basic accessibility properties with **Low complexity impact** - semantic roles, proper sizing, haptic feedback. Complex components like form controls require **state management, error handling, validation feedback** with **Medium complexity classification**. Advanced components like modal dialogs require **focus management, announcement sequences, navigation coordination** with **High complexity factors**.
- **Layer three** captures the **screen-based analysis through empirical testing**. Every component undergoes systematic screen reader testing across both platforms, following identical protocols for announcement quality, gesture support, role communication, and focus management.

This systematic analysis reveals patterns that would be invisible to anecdotal evaluation. **Accessibility complexity correlates with interaction patterns rather than visual design complexity**. A sophisticated visual interface with simple interactions requires minimal accessibility work. A basic interface with complex form validation requires substantial accessibility implementation effort.

Every component you see here represents **controlled measurement** generating empirical data about implementation costs, complexity factors, and cross-platform compatibility

---

Slide 10: "Systematic analysis - Example (2)"

This slide **reveals the detailed empirical foundation underlying our systematic methodology** - these are the actual measurement frameworks that enable rigorous comparative analysis and bridge our accessibility implementation guidelines to framework evaluation.

- **Table 3.5 demonstrates component-criteria mapping with WCAG2Mobile considerations** - the systematic operationalization of our layer one theoretical foundation.
  - Look at this ScrollView Container analysis: it maps to **WCAG 2.2 Success Criteria 2.1.1 Keyboard and 2.4.3 Focus Order**, but **WCAG2Mobile considerations** include **screen-based navigation patterns and touch-based scrolling alternatives**.
  - The implementation properties show the exact code patterns required: `accessibilityRole="scrollview"` and `accessibilityLabel="Accessibility Components Screen"`. This isn't theoretical - this represents **systematic mapping methodology** applied consistently across all components.
- **Table 3.6 shows empirical screen reader testing with WCAG2Mobile focus** - our **layer three screen-based analysis** operationalized through formal testing protocols.
  - The Screen Title test case demonstrates **systematic validation methodology**: **VoiceOver** and **TalkBack** behavior is documented with **WCAG2Mobile considerations**

noting that **Success Criteria 1.3.1 and 2.4.6 are interpreted for screens instead of web pages**.

    - This represents **controlled device testing with actual screen readers** following standardized evaluation protocols.

- **Table 3.7 provides systematic implementation analysis** - documenting the structured approach to measuring accessibility implementation requirements.

    - **Semantic Roles** and **Descriptive Labels** are analyzed through consistent measurement methodology, with **complexity impact classification** based on nesting depth, dependency requirements, and property count.

This is exactly how the systematic methodology scales throughout the research: every component, every screen, every accessibility feature undergoes this **identical analytical framework**. **Component inventory mapping to WCAG criteria**, **empirical screen reader testing across both platforms**, and **systematic implementation analysis** following standardized protocols.

The critical innovation is that this systematic approach enables **methodologically rigorous comparison between frameworksm**, transforming **subjective accessibility assessment into reproducible, empirical analysis** - establishing the methodological foundation for evidence-based mobile accessibility development guidance

---

Slide 11: "Accessibility implementation costs"

Now we reach what I consider the most practically transformative finding of this entire research - the first comprehensive quantitative assessment of accessibility implementation costs that fundamentally changes how we approach project planning and resource allocation.

These findings emerge from **Section 4.5 Component Implementation Patterns** in Chapter 4, where I conducted systematic analysis across equivalent components in both React Native and Flutter. The methodology involved **component equivalence mapping**, ensuring fair comparison based on component purpose rather than implementation details, followed by **WCAG criteria mapping** against the same WCAG 2.2 standards used throughout the research.

The key finding that accessibility implementation requires **12-23% additional code across component types** represents a revolution in predictability. For the first time, project managers can budget accessibility implementation with empirical data rather than fear-based estimates or complete uncertainty.

Let me walk you through what these numbers actually mean.

- **Media components at 12.7% overhead with Low complexity** represent our baseline - alternative text and captions are essentially metadata additions requiring no behavioral logic changes.

- **Buttons at 13.3%** introduce semantic role requirements that demand understanding component context and purpose across both VoiceOver and TalkBack with consistent behavior.

- **Dialogs reach 16.2% with Medium complexity**, where accessibility implementation becomes more sophisticated. Focus management becomes critical - modal opening requires proper focus transfer, clear purpose announcement, and logical navigation patterns back to underlying content.
- **Forms hit 21.5% overhead**, representing the most complex accessibility implementation in typical mobile applications. The primary contributors - state management and error handling - require substantial additional logic for validation feedback that works with assistive technology.
- **Advanced components cap at 22.7% with High complexity** - custom gesture handlers and innovative navigation patterns that don't map to standard accessibility patterns, requiring building accessibility behavior from first principles.

But here's the critical insight: even the most complex components stay under 25% overhead. This isn't the 2x or 3x cost increase that developers fear. The **correlation between interaction complexity and implementation cost** reflects accessibility's fundamental nature - it's about making interactions comprehensible, which naturally correlates with interaction sophistication rather than visual design complexity.

Slide 12: "Formal evaluation metrics"

The methodological foundation that enables all our systematic comparison rests on five formal metrics that operationalize accessibility assessment through rigorous mathematical frameworks and empirical validation protocols. This represents the first evidence-based methodology for quantifying mobile accessibility implementation across frameworks.

- **Implementation Overhead (IMO)** provides direct code cost measurement through systematic lines-of-code analysis for functionally equivalent implementations.
    - This metric transforms the abstract question "how much will accessibility cost?" into concrete data: form components require **21.5% additional code** with medium complexity classification, enabling accurate planning and resource allocation with empirical rather than estimated data.
- **Screen Reader Support Score (SRSS)** implements systematic Likert scaling through comprehensive empirical testing using **VoiceOver on iPhone 14 with iOS 16** and **TalkBack on Google Pixel 7 with Android 15**, evaluating five criteria categories across multiple dimensions.
    - **Announcement Quality (1-5)**: Completeness, accuracy, and clarity of screen reader announcements
    - **Gesture Support (1-5)**: Support for screen reader-specific gestures and interaction patterns
    - **Role Communication (1-5)**: Correct announcement of component roles and states
    - **Focus Management (1-5)**: Logical focus order and appropriate focus handling

- **WCAG Compliance Ratio (WCR)** measures conformance through systematic evaluation against applicable WCAG 2.2 success criteria using the calculation: **WCR = (Number of satisfied criteria / Total applicable criteria) × 100%**.
    - Results aggregate by WCAG principle - **Perceivable, Operable, Understandable, Robust** - enabling identification of specific framework strengths and limitations rather than just overall compliance scores.
- **Complexity Impact Factor (CIF)** quantifies implementation difficulty through formal calculation incorporating **weighted factors for nesting depth, dependency requirements, and property count**.
    - This moves beyond simple line-counting to capture the **cognitive load and structural complexity** that impact long-term maintainability and developer experience, using a mapping system from qualitative assessments (Low/Medium/High) to numeric scores.
- **Development Time Estimate (DTE)** translates technical metrics into business planning data through controlled time measurement with complexity adjustments, providing realistic estimates for project planning that account for both implementation effort and learning curve factors.

These metrics collectively **transform accessibility from an unknown cost variable** into a measurable, plannable development component with predictable patterns and complexity profiles.

Slide 13: "Framework comparison"

This brings us to the **definitive empirical comparison** that answers our core research questions through systematic measurement across multiple evaluation dimensions. The architectural differences between frameworks create measurably different developer experiences and implementation costs.

Research Questions Addressed:

- **RQ1 (Default accessibility support)**: To what extent are components accessible by default without developer intervention?
- **RQ2 (Implementation feasibility)**: What is the technical feasibility of enhancing non-accessible components to meet accessibility standards?
- **RQ3 (Development overhead)**: What is the quantifiable development overhead required to implement accessibility features?

The **fundamental architectural distinction** drives all our findings. React Native implements a **property-based model** where accessibility features integrate directly into existing components through properties like `accessibilityLabel` and `accessibilityRole`. Flutter uses a **widget-based approach** requiring explicit `Semantics` wrappers that create additional structural layers in the component hierarchy.

Now let's examine what systematic measurement reveals about these architectural differences:

- **Implementation Overhead** shows React Native as baseline, with **Flutter requiring 119% more code** - this isn't marginal difference, it's substantial impact on development velocity.

    - When a React Native accessibility implementation takes 15 lines, the equivalent Flutter implementation averages 33 lines.
    - This difference compounds across large applications with hundreds of components requiring accessibility implementation.
- **Screen Reader Support** demonstrates React Native achieving **4.2/5 versus Flutter's 3.8/5** through systematic testing across both platforms. More importantly, React Native shows superior cross-platform consistency with **better platform variance**, indicating more predictable behavior between iOS and Android implementations.
- **Default Accessibility** reveals React Native providing **38% baseline accessibility versus Flutter's 32%**. While both require substantial developer intervention, React Native provides more functionality out-of-box, reducing initial implementation burden.
- **WCAG Compliance** shows React Native achieving **92% compliance on the Perceivable principle compared to Flutter's 85%**, demonstrating better alignment with accessibility standards across the most critical accessibility foundation.

The Decision Framework transforms these technical metrics into practical guidance:

- **Implementation Overhead** impacts development speed - teams choosing React Native can implement comprehensive accessibility features with **45% less code on average**
- **Screen Reader Support** affects user experience quality - React Native provides more reliable assistive technology integration
- **Architecture differences** impact code complexity and long-term maintainability patterns

Learning Curve comparison reveals React Native's **moderate versus Flutter's steep** accessibility learning curve, particularly affecting team onboarding and knowledge transfer when accessibility expertise must scale across development teams.

Evidence-Based Framework Selection:

- **Choose React Native** when: Rapid development cycles, teams with web accessibility experience, tight deadlines requiring implementation efficiency
- **Choose Flutter** when: Complex custom components requirements, long-term maintenance teams, applications requiring granular accessibility behavior control that justifies higher implementation overhead

---

Slide 14: "Framework comparison results (1)"

These empirical findings represent **months of systematic measurement** across equivalent implementations. What emerges isn't just data - it's a **clear pattern of architectural trade-offs** that fundamentally impact both developer experience and user outcomes.

- **Text Language implementation** reveals the **most dramatic architectural difference**. React Native provides this **by default through accessibilityLanguage** - zero additional code, immediate functionality.
    - Flutter requires **manual AttributedString construction** with **200% overhead**. This isn't about developer convenience; it's about **fundamental design philosophy**.

- **Think about what this means at scale**: in a multilingual application with hundreds of text elements, React Native handles language declaration **automatically**, while Flutter demands **explicit implementation for every component**. The cognitive load difference compounds exponentially.
- **Headings require manual implementation** in both frameworks - **neither treats semantic structure as first-class** - but Flutter shows **57% more overhead**.
    - React Native achieves semantic headings through **accessibilityRole="header"** - simple, direct, platform-native. Flutter requires **Semantics wrapper widgets** with explicit declarations that **complicate component hierarchy**.
- **Screen Reader Support** consistently favors React Native: **4.3 versus 4.0 for headings**, **4.0 versus 3.8 for forms**. These differences might seem minor, but they represent **measurably better user experiences** across the most common interaction patterns.
- **Form Fields demonstrate 53% overhead** with particularly revealing patterns. React Native's **form accessibility integrates seamlessly** with existing input components - developers work within familiar patterns. Flutter requires **additional wrapper widgets** that can **complicate state management** and validation workflows.
    - **Here's the critical insight**: Flutter's wrapper approach creates **architectural friction**. Every accessibility enhancement adds structural complexity that impacts **maintainability, debugging, and team knowledge transfer**.
- **Custom Gestures** show the **smallest gap at 27%** but reveal something crucial: **both frameworks struggle equally** with complex interactions. Screen reader scores - **RN: 3.8, FL: 3.2** - are our **lowest across all categories**. This indicates **universal accessibility limitations** requiring framework-level improvements, not just better implementation patterns.

Overall metrics provide definitive guidance: React Native's **38% default accessibility** versus Flutter's **32%**, requiring **119% less code overall**. The **screen reader consistency - 4.2 versus 3.8** - reflects **more reliable assistive technology integration**.

These patterns compound: in applications with comprehensive accessibility requirements, architectural differences create **measurable impact on both development velocity and user experience quality**.

Slide 15: "Framework comparison results (2)"

This consolidated analysis **transforms technical findings into strategic guidance** - moving beyond metrics to **actionable framework selection criteria** based on systematic evidence rather than subjective preferences.

- **Default Accessibility** shows React Native's **6% advantage** - modest numerically, but **significant practically**.
    - The difference between **38% and 32%** represents **measurably less initial implementation work**. However, both frameworks demonstrate that **accessibility-by-**

default remains largely mythical - substantial developer intervention is **mandatory regardless of choice**.

- **Implementation Overhead** provides our **most striking finding**: Flutter requiring **119% more code** for equivalent accessibility.
    - **Concrete impact**: when React Native accessibility requires **20 lines**, Flutter averages **44 lines**. This isn't marginal - it's **substantial development velocity impact** that compounds across applications with **hundreds of accessibility-requiring components**.
- **Screen Reader Support** demonstrates React Native's **superior assistive technology integration** at **4.2/5** versus **3.8/5**. Beyond numeric difference, this reflects **more consistent cross-platform behavior**, particularly benefiting **Android TalkBack users** where Flutter shows **variable performance patterns**.
- **WCAG Compliance** reveals React Native achieving **92% AA compliance** versus Flutter's **85%**. This **seven-percentage-point difference** represents **measurable accessibility features** that function reliably in React Native but require **workarounds or additional complexity** in Flutter.

Strategic Decision Framework based on evidence!

Choose React Native when:

- **Rapid development cycles** demand implementation efficiency
- Teams have **existing web accessibility experience** - patterns transfer directly
- Projects face **constrained timelines** where accessibility implementation speed impacts feature delivery
- **Cross-platform consistency** is prioritized over granular control

Choose Flutter when:

- **Complex custom components** require granular accessibility behavior control
- **Long-term maintenance teams** can invest in accessibility expertise development
- **Architectural clarity** justifies higher implementation overhead
- Applications demand **precise semantic control** over assistive technology interaction

Critical considerations for decision-making:

- **Team expertise** dramatically impacts implementation success regardless of framework choice
- **Project timeline** pressure correlates with accessibility implementation quality - choose tools that **reduce barriers to doing the right thing**
- **Maintenance burden** compounds over time - consider long-term complexity management

This represents **the first evidence-based framework selection guidance** for accessibility-focused mobile development, **replacing subjective assessment with systematic empirical analysis**.

Slide 16: "Research impact and conclusions"

This research establishes **four fundamental contributions** that advance both academic understanding and industry practice, while providing **definitive empirical answers** that transform how we approach mobile accessibility implementation.

Key Contributions represent genuine advancement:

- **Extended research framework** transitions from **qualitative Flutter-specific analysis** to **comprehensive quantitative cross-framework evaluation**. We've established **systematic comparative methodology** that future researchers can build upon rather than developing ad-hoc evaluation approaches.
- **First quantitative framework** for mobile accessibility cost assessment transforms **project planning from estimation to empirical data**. Our formal metrics - **IMO, SRSS, WCR, CIF, DTE** - enable **resource allocation with measurable complexity factors** rather than unknown cost variables.
- **45% implementation overhead reduction** with React Native represents **systematic measurement across multiple component categories**, providing **concrete guidance for framework selection** when accessibility implementation efficiency is prioritized.

Systematic methodology bridging WCAG theory to mobile practice creates **reusable analytical frameworks** for ongoing evaluation as frameworks evolve and new platforms emerge.

Research Answers provide **definitive empirical responses**:

- **RQ1**: Neither framework is accessible by default - **38% versus 32%** demonstrates that **deliberate developer intervention is mandatory** regardless of framework choice.
- **RQ2**: Both frameworks **can achieve 100% WCAG compliance** with proper implementation, validating technical feasibility while revealing **different implementation approaches and complexity profiles**.
- **RQ3**: React Native requires **45% less code** for equivalent accessibility implementations, providing **empirical foundation for development planning** and framework selection decisions.

Broader Impact extends beyond academic contribution:

- **Industry transformation**: The era of framework selection based on **subjective assessment** or treating accessibility as an **unknowable project risk** has ended. We now possess **systematic analytical tools** for evidence-based decision-making.
- **Practical application**: Development teams can **budget accessibility implementation with empirical data** rather than fear-based estimates or complete uncertainty.

The ultimate measure of this research isn't academic citations - it's **whether it results in more accessible mobile applications** reaching users with disabilities. This research provides the **empirical foundation** for making accessibility implementation **predictable, plannable, and achievable** in real-world development contexts.