AccessibleHub Presentation Script

Slide 1: Title Slide

Good morning. I'm Gabriel, and today I'm presenting **AccessibleHub**, a learning toolkit that bridges the gap between guidelines and implementation.

The core problem is this: "How much will this accessibility actually cost in a project implementation?". Current accessibility resources tell us *what* to do, but they don't adequately address *how* to do it in practice, *when* to implement specific features, or *what the actual cost* of accessibility implementation looks like in real development scenarios.

This research shows how to make accessibility a **plannable development component** with precise empirical metrics.

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

Before we dive into the research, let me set the stage. **Mobile accessibility** ensures that all people—regardless of ability—can interact with the information or services provided in the mobile context. Simple definition, but the reality is much more complex.

- Here's the scale we're working with: **1.3 billion people worldwide live with disabilities**, and we have 6.8 billion smartphone users globally. That's a massive overlap that makes mobile accessibility incredibly important.
- But mobile creates **problems we never had with web development**. Think about it—we're dealing with small screens where every pixel matters, touch interfaces that aren't as precise as a mouse cursor, and users who might be holding their phone with one hand while walking down the street.
- The challenges get technical quickly. **Gesture conflicts** are a real problem—imagine VoiceOver's three-finger scroll interfering with your app's navigation. Orientation changes don't just mess up your layout; they completely change how screen readers navigate your content. And unlike web pages that load once, mobile apps need to manage focus and navigation state constantly.
- Then there's **performance**. Every accessibility feature you add uses battery and processing power. You're not just making your app inclusive—you're making trade-offs that affect every user's experience.

This is why we can't rely on gut feelings or simple guidelines anymore. We need **systematic measurement** to understand what accessibility actually costs and how different approaches compare. That's exactly what this research provides.

Slide 3: "Accessibility guidelines gap"

We have excellent theoretical foundations, but they're systematically failing mobile developers when it comes to practical implementation. Let me walk you through our current standards landscape and show you exactly where the gaps are.

- **WCAG 2.2**, released in October 2023, gives us solid principles - the **POUR framework**: **Perceivable, Operable, Understandable, and Robust** - with **three conformance levels**. But here's the issue: it remains fundamentally **web-focused**. The success criteria are written for web pages, not mobile screens with touch interfaces and gesture navigation.
- **MCAG** from 2015 attempted mobile adaptation, but it was **based on WCAG 2.0**. This means it's missing crucial mobile-specific criteria from WCAG 2.1 and 2.2 - things like **touch target sizing and orientation handling** that are absolutely critical for mobile accessibility. **WCAG2Mobile**, just released in January 2025, provides the most recent mobile guidance, but crucially offers **only interpretations, not implementation frameworks**.

This creates three cascading problems that directly impact every mobile development team.

- First, we have an **implementation void**. These standards tell you **what to achieve** - 'ensure content is perceivable' - but not **how to achieve it**. What does that mean for `accessibilityLabel` properties in React Native versus `Semantics` widgets in Flutter? What are the relative costs and trade-offs?
- Second, **knowledge fragmentation** has become endemic. Resources are scattered across documentation sites, blog posts, and community forums. Developers are forced to piece together approaches without understanding implementation costs or systematic comparison methodologies.
- Third, this creates **developer isolation**. Teams resort to ad-hoc accessibility solutions because there's no comprehensive mobile-native framework to guide systematic implementation.

The **result** is that mobile accessibility implementation lacks the empirical foundation that project managers need for planning and developers need for systematic execution. And that's exactly what motivated our research.

Slide 4: "Platform implementation gap"

So we've established that accessibility standards lack practical implementation guidance. But here's where things get even more complex - the technical reality of mobile platform fragmentation makes systematic measurement absolutely critical.

Let me show you exactly what developers face when implementing accessibility across platforms.

- **iOS** implements **VoiceOver** with its unique gesture system - three-finger swipes, rotor navigation, and specific announcement patterns. Add **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, plus **Switch Access** with fundamentally different keyboard navigation than iOS.

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

Now, frameworks attempt to bridge this complexity and in this research both Flutter and React Native were considered, which are the main ones within the cross-platform development.

This brings us to the current research landscape. **Budai's 2024 thesis** conducted systematic Flutter component accessibility testing and documented this fragmented knowledge problem. His work demonstrated that while Flutter widgets show promise for accessibility implementation, developers fundamentally lack systematic guidance for comprehensive implementation across platforms.

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 making framework choices without understanding accessibility implications, implementation overhead, or cross-platform consistency trade-offs.

That's exactly the empirical foundation this research provides.

Slide 5: "AccessibleHub - Bridging the gap”

So we've identified this systematic gap - no comprehensive learning resource that bridges platforms with quantitative implementation cost analysis. That's exactly what **AccessibleHub** addresses through a dual methodology that serves both immediate practical needs and rigorous research requirements.

Now, before I dive into the research questions, let me be crystal clear about the target audience here. **AccessibleHub is specifically designed for developers, not end users**. I structured the investigation around three research questions that directly address the core challenges developers face when implementing mobile accessibility.

- **RQ1: Are React Native components accessible by default?** This establishes the baseline - what can developers expect to work out of the box without any additional implementation effort? If 38% of components are accessible by default versus 62%, that fundamentally changes how developers approach project planning and initial architecture decisions.
- **RQ2: Can non-accessible components be made accessible?** This validates framework capabilities from a developer perspective. Can every component be made accessible, or are there technical limitations that developers need to account for? This identifies potential complexity barriers that might make certain accessibility features impractical or prohibitively expensive to implement.
- **RQ3: What's the implementation cost in terms of code overhead?** This is the critical question that drives everything - developers need quantifiable metrics for planning. How many lines of code must be added? Here's a concrete example: **text language implementation** requires **7 lines in React Native versus 21 lines in Flutter** - that's a **200% overhead difference** that directly impacts sprint planning and resource allocation.

AccessibleHub serves as both the research instrument and the practical solution - we're bridging WCAG theory to executable code with quantified implementation costs.

Slide 6: "AccessibleHub - Overview"

Now let me show you exactly how **AccessibleHub transforms theory into practice (go into video and show how the video moves on and how the Screen Readers actually read content here – it’s hard and as you can see it’s not totally easy how to make it).**

Slide 7: "Systematic analysis approach"

This brings us to the methodological innovation that makes this research possible - the systematic solution to what I call the **WCAG transformation challenge**.

Here's the fundamental problem: **WCAG guidelines exist at a level of abstraction that makes direct mobile implementation hard to understand**. Developers get principles like "content must be perceivable" but no concrete guidance on what that means for `accessibilityLabel` properties or touch target implementation.

AccessibleHub solves this through a unified four-layer transformation methodology that bridges theory and practice within a single integrated platform.

- **Layer one - Theoretical Foundation**: We systematically map abstract WCAG principles to mobile-specific success criteria, creating the analytical benchmark. But crucially, this doesn't stop at guidelines - it feeds directly into practical implementation.
- **Layer two - Implementation Patterns**: Here's where the innovation happens. We translate each success criterion into concrete React Native code structures with **quantified formal metrics**. Every accessibility feature becomes a measurable implementation pattern with defined overhead costs.
- **Layer three - Screen-Based Analysis**: We validate these patterns through empirical testing with real assistive technologies - **VoiceOver on iPhone 14** and **TalkBack on Google Pixel 7**. This generates actual usability metrics, not theoretical compliance scores.
- **Layer four - Platform Integration**: All of this flows into **AccessibleHub as the first unified measurement platform** - developers get the theoretical foundation, implementation patterns, and empirical validation in one integrated educational tool.

The **basic workflow** demonstrates the key innovation: **Abstract WCAG principles** become **Implementation Patterns** that generate **Quantified Metrics**, all accessible through a **single educational platform**.

This isn't just bridging theory and practice - it's creating a systematic methodology where **every theoretical requirement has a quantified implementation cost and validated code pattern**. AccessibleHub serves as both the research instrument and the practical solution, enabling **data-driven accessibility decisions** for the first time in mobile development.

Slide 8: “1 - Component-criteria mapping"

Now let me show you exactly how this systematic methodology works in practice through the **Components screen analysis** - demonstrating the first layer of our systematic approach where we map interface elements to specific WCAG criteria.

What you're seeing here is **systematic component-criteria mapping** in action. The Components screen organizes accessibility complexity progressively: **Buttons & Touchables labeled "Essential"**, **Form Controls labeled "Complex"**, and **Media Content labeled "Advanced"**. These aren't arbitrary classifications - they represent measured implementation complexity based on systematic analysis.

Table 3.5 demonstrates the precise mapping methodology. Look at the **ScrollView Container**: it maps to **WCAG 2.1.1 Keyboard** and **2.4.3 Focus Order**, but the **WCAG2Mobile considerations** specify **screen-based navigation patterns and touch-based scrolling alternatives** - mobile-specific interpretations that don't exist in web contexts.

- The **Implementation Properties** column shows exactly what developers need: `accessibilityRole="scrollview"` and `accessibilityLabel="Accessibility Components Screen"`. This is the bridge from abstract WCAG criteria to concrete React Native code.
- Similarly, the **Hero Title** maps to **WCAG 1.4.3 Contrast** and **2.4.6 Headings**, with **WCAG2Mobile considerations** emphasizing **screen title differentiation and proper semantic structure in screen context** - again, mobile-specific interpretations.
- This systematic mapping approach ensures every component has **explicit WCAG criteria connections**, **mobile-specific considerations**, and **concrete implementation properties**. It's the foundation that enables our empirical measurement methodology - transforming abstract accessibility requirements into measurable, comparable implementation patterns.

This is how we make concrete the theoretical foundation into practical development guidance.

Slide 9: "2 – Empirical screen reader testing"

This brings us to the **empirical validation layer** - where we systematically test how these mapped components actually perform with real assistive technologies across both platforms.

The **code snippet** shows the exact React Native implementation being tested - a TouchableOpacity with `accessibilityRole="button"` and a comprehensive `accessibilityLabel` that includes component description, purpose, and complexity classification. This isn't theoretical code - this is the actual implementation that undergoes systematic screen reader testing.

Table 3.6 demonstrates our empirical testing methodology with systematic evaluation protocols applied consistently across both platforms. This represents **controlled device testing with actual screen readers** - **VoiceOver on iOS** and **TalkBack on Android** - following standardized evaluation procedures.

- Look at the **Screen Title test case**: both VoiceOver and TalkBack announce "Accessibility Components, heading", but the **WCAG2Mobile considerations** specify that **Success**

Criteria 1.3.1 and 2.4.6 are interpreted for screens instead of web pages. This mobile-specific interpretation affects how heading structures work in mobile contexts versus web contexts.

- The **Component Card test case** shows both screen readers announcing the complete label with component description and complexity. The **WCAG2Mobile considerations** note that **Success Criteria 4.1.2 for mobile context requires comprehensive labels for navigation decisions** - users need complete information before committing to navigation on small screens.
- This systematic testing methodology employs **standardized evaluation criteria** across four dimensions: **Announcement Quality**, **Gesture Support**, **Role Communication**, and **Focus Management** - each rated on a 1-5 Likert scale to generate empirical scores rather than subjective assessments.

The critical innovation is that every accessibility feature undergoes **identical testing protocols** on both platforms, generating comparable empirical data that reveals platform-specific behavior patterns and enables systematic framework evaluation.

Slide 10: "3 – Implementation overhead analysis"

This brings us to the **quantitative measurement layer** - where systematic evaluation transforms into concrete data that project managers can use for planning and resource allocation.

Table 3.7 provides systematic implementation overhead analysis - documenting exactly how many lines of code each accessibility feature requires and its complexity impact. This represents **controlled measurement methodology** applied to every accessibility implementation decision.

- Notice the **granular breakdown**: **Semantic Roles** require just **15 lines of code** representing **2.6% overhead with Low complexity** - essentially metadata additions that don't require behavioral logic changes. **Descriptive Labels** need **28 lines at 4.9% with Medium complexity** - requiring contextual understanding and proper screen reader optimization.
- But look at the **complexity escalation**: **Breadcrumb Implementation** reaches **42 lines at 7.4% with High complexity**, and **Drawer Accessibility** requires **35 lines at 6.2% with High complexity**. These aren't arbitrary classifications - they reflect **systematic evaluation** based on nesting depth, dependency requirements, and platform-specific implementation needs.
- The **critical finding** is the **total implementation overhead of 206 lines representing 36.3% of the Components screen codebase**. This isn't a theoretical estimate - it's **empirical measurement** showing exactly how accessibility implementation scales across different feature types.
- **AAA-specific Enhancements** add **20 lines at 3.5%** - demonstrating that **beyond-compliance accessibility** has **measurable, predictable costs** rather than unknown implementation complexity.

The **Medium-High complexity classification** reflects the balanced approach between comprehensive accessibility implementation and maintainable code structure. This systematic

measurement methodology enables **predictable cost estimation** across different component types and complexity levels.

For the first time, developers and project managers have **empirical data** showing that accessibility implementation follows **systematic patterns** rather than unpredictable complexity, enabling evidence-based planning and resource allocation.

Slide 11: "Component implementation patterns"

Now we reach the most practically transformative finding of this research - **systematic correlation that enables predictable accessibility cost estimation across frameworks**.

This analysis emerges from **Chapter 4's systematic component comparison** where I measured equivalent accessibility implementations across React Native and Flutter. But what you're seeing here goes beyond framework comparison - it reveals **fundamental patterns** that apply regardless of your technology choice – **note: tell React Native is used as baseline here.**

- **Media components at 12.7% overhead** represent our baseline - alternative text and captions are essentially content description additions. The **complexity driver** is straightforward: describing content doesn't require complex behavioral changes.
- **Buttons jump to 13.3% overhead**, and here's where it gets interesting. The **complexity driver** becomes **semantic roles and labels** - you're not just describing content, you're defining how components behave with assistive technologies across both VoiceOver and TalkBack.
- **Dialogs hit 16.2% overhead** with **navigation control** as the complexity driver. Now you're managing focus transfer, modal behavior, and ensuring users can navigate back to the underlying content systematically.
- **Forms reach 21.5% overhead** - the highest practical complexity for most applications. The **complexity driver** is **state management** - validation feedback, error handling, and form submission that works seamlessly with screen readers requires substantial additional logic.
- **Advanced components cap at 22.7%** with **custom interactions** driving complexity. These are things like custom sliders, loading animations, progress bars or gesture handlers that don't map to standard accessibility patterns.

But here's the crucial insight: **even the most complex components stay under 25% overhead**. This isn't the 2x or 3x cost increase developers fear. The **pattern validation** shows these percentages are **consistent** across component types - this creates a **foundation for framework-agnostic accessibility cost prediction**.

For the first time, project managers can budget accessibility implementation with the same precision they apply to any other technical requirement.

Slide 12: "Formal evaluation metrics"

This brings us to the methodological foundation that makes everything we've discussed possible – presenting, as far as my knowledge goes, the first **evidence-based methodology for quantifying mobile accessibility implementation across frameworks**.

What makes this research innovative isn't just the findings, but the **five formal metrics** that enable systematic comparison instead of subjective assessment.

- **Implementation Overhead (IMO)** gives us direct code cost measurement. Instead of guessing "how much will accessibility cost?", we can now say forms require **21.5% additional code** with medium complexity. This transforms abstract questions into concrete planning data.
- **Screen Reader Support Score (SRSS)** uses systematic Likert scaling with **VoiceOver on iPhone 14** and **TalkBack on Google Pixel 7**. We test four dimensions: **announcement quality, gesture support, role communication, and focus management** - each rated 1-5. This gives us empirical scores rather than "it works" or "it doesn't."
- **WCAG Compliance Ratio (WCR)** tracks standards adherence using **satisfied criteria divided by total applicable criteria times 100**. But here's the key - results aggregate by WCAG principle: **Perceivable, Operable, Understandable, Robust** - so we can identify specific framework strengths rather than just overall compliance.
- **Complexity Impact Factor (CIF)** goes beyond simple line counting. It incorporates **nesting depth, dependency requirements, and property count** to capture the cognitive load that impacts maintainability. A component might be 15 lines but structurally complex, or 30 lines but straightforward.
- **Development Time Estimate (DTE)** translates technical metrics into business planning. It accounts for both implementation effort and learning curve factors, giving project managers realistic estimates with complexity adjustments.

These metrics collectively **transform accessibility from an unknown cost variable into a measurable, plannable development component**. For the first time, teams can approach accessibility implementation with the same predictability they have for any other technical requirement.

Slide 13: "Framework comparison"

Now we get to the **fundamental architectural differences** that drive all the implementation overhead patterns we've been discussing. These aren't just coding style preferences - they're **core design philosophies** that directly impact how developers implement accessibility.

- **React Native follows a property-based model** where accessibility features integrate directly into component definitions. Look at this code - `accessibilityRole="header"` and `accessibilityLabel="Save document"` are **properties applied directly to the component**. The accessibility information lives alongside the component logic in a flat, integrated structure.

- **Flutter takes a widget-based approach** with explicit Semantics adapters. Notice how the same functionality requires **wrapping components in Semantics widgets** - `Semantics(header: true, child: Text("Settings"))` and `Semantics(button: true, label: "Save document", child: GestureDetector(...))`. This creates **additional nesting levels** and **structural complexity**.
- Here's what this means in practice. The React Native approach lets you **add accessibility properties** like you'd add any other component attribute - it's **property integration**. The Flutter approach requires **semantic wrappers** around existing components, creating **additional layers of abstraction**.

This architectural difference directly explains the **implementation overhead patterns** we measured earlier. When Flutter requires **119% more code** on average, it's not because the framework is poorly designed - it's because **explicit semantic wrappers** naturally require more lines of code than **integrated properties**.

But there's a deeper implication here. React Native's **property integration** means accessibility feels like a **natural part of component definition**. Flutter's **semantic wrappers** make accessibility feel like an **additional concern** that wraps around your main component logic.

Both approaches work, but they create **fundamentally different developer experiences** that directly impact how teams approach accessibility implementation. This architectural difference shapes everything from initial learning curves to long-term maintainability patterns.

Slide 14: "Framework comparison results - 1"

These empirical findings represent systematic measurement across equivalent implementations, and what emerges is a **clear pattern of architectural trade-offs** that directly impact both developer experience and user outcomes.

- **Text Language Declaration** reveals the most significant difference - React Native provides this **transparently through accessibilityLanguage** with zero additional overhead, while Flutter requires **200% more code** through explicit AttributedString construction.
    - In multilingual applications, this difference compounds dramatically. React Native enables automatic internationalization, while Flutter requires conscious semantic decisions for every text element.
- **Headings require manual implementation** in both frameworks - neither treats semantic structure as first-class - but Flutter shows **57% more overhead**. React Native achieves this through simple `accessibilityRole="header"`, while Flutter requires Semantics wrapper widgets that complicate component hierarchy.
- **Form Fields show 53% overhead** with particularly revealing patterns. React Native's form accessibility integrates directly with existing input components - developers work within familiar patterns. Flutter requires additional wrapper widgets that can complicate state management and validation workflows.

- **Custom Gestures show the smallest gap at 27%**, but here's what's crucial: both frameworks struggle equally with complex interactions. The screen reader scores - **React Native 3.8, Flutter 3.2** - are our **lowest across all categories**. This indicates universal accessibility limitations requiring framework-level improvements.
- **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 across platforms.

Critical observations: The **119% code overhead** in Flutter translates to **measurably longer development cycles** for accessibility-focused projects. For teams with **limited accessibility expertise**, React Native's integrated approach reduces the **learning curve and implementation errors**. However, Flutter's explicit semantic model provides **better debugging capabilities** when accessibility issues arise. The **custom gesture scores** reveal that **both frameworks need significant improvement** in complex interaction accessibility - this shouldn't drive framework selection.

These patterns reveal that architectural differences create measurable impacts on both development velocity and user experience quality that compound across applications with comprehensive accessibility requirements.

Slide 15: "Framework comparison results - 2"

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

- **Code Integration** shows the **fundamental architectural difference**: React Native's direct properties versus Flutter's wrapper layers resulting in **119% more code**. This isn't just about typing more lines - it's about **cognitive load and development velocity** that compounds across hundreds of components.
- **Developer Experience** reveals a **critical trade-off**: React Native requires **component modification** - familiar to web developers - while Flutter demands **semantic layer addition** - a steeper learning curve. Our analysis shows **40% faster implementation times** for web-experienced developers using React Native.
- **Default Accessibility** demonstrates that **both frameworks require intervention** - 38% versus 32% - but React Native's advantage means **measurably less initial work**. Neither framework delivers "accessibility by default," making this a **realistic baseline** rather than an aspiration.
- **Screen Reader Support** shows React Native's **better consistency** at 4.2 versus 3.8. This reflects **more reliable assistive technology integration**, particularly benefiting Android TalkBack users where Flutter shows **variable performance patterns**.
- **WCAG Compliance** reveals React Native's **8.2% advantage** at 92% versus 85% AA compliance. This represents **measurable accessibility features** that work reliably in React Native but require **workarounds or additional complexity** in Flutter.

Strategic Decision Framework based on evidence:

- **Choose React Native** for rapid development timelines, teams with web accessibility experience, and cross-platform consistency priority
- **Choose Flutter** for complex custom components, dedicated long-term maintenance teams, and granular accessibility control.

No framework is accessible by default and they both require intervention – not as they declare on their website (Flutter case)

The **critical insight**: team expertise and project timeline dramatically impact implementation success regardless of framework choice. Choose tools that **reduce barriers to doing the right thing** rather than adding complexity to accessibility implementation.

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 **fundamental contributions** that advance both academic understanding and industry practice, transforming how we approach mobile accessibility implementation from guesswork to systematic analysis.

- **Key contributions** represent genuine advancements in the field. The **systematic evaluation methodology** built on evidence-based formal metrics creates the first rigorous framework for mobile accessibility assessment. We've moved from anecdotal evaluation to **quantifiable measurement** that other researchers can build upon and extend.
- The **first quantitative framework for cross-platform accessibility assessments** changes everything for project planning. Our formal metrics - IMO, SRSS, WCR, CIF, DTE - enable teams to **budget accessibility implementation with empirical data** rather than fear-based estimates or treating it as an unknowable project risk.
- Most importantly, this **systematic methodology bridging WCAG theory to mobile practice** creates reusable analytical frameworks that remain relevant as frameworks evolve and new platforms emerge. We've solved the fundamental problem of translating abstract accessibility guidelines into concrete implementation guidance.
- **Research answers** provide **definitive empirical responses** to our core questions. **RQ1** establishes that **no framework is accessible by default** - intervention is required for both, ending the myth of "accessibility by default." **RQ2** confirms both frameworks **achieve high WCAG compliance with proper implementation**, validating technical feasibility while revealing different complexity profiles. **RQ3** quantifies that **React Native requires 45% less code** for equivalent accessibility, providing concrete guidance for framework selection.

The broader impact extends beyond academic contribution. Development teams can now approach accessibility implementation with **the same predictability they have for any other technical requirement**. The era of framework selection based on subjective assessment has ended - we now possess systematic analytical tools for evidence-based decision-making.

The ultimate measure of this research isn't academic recognition - 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.

Side notes from presentation trials:

- Don't cover the slide number yourself presenting (rookie mistake, quoting Scquizzy-man here)

- "As far as my knowledge goes" and "cross platform frameworks"

- Focus the fact that my research is both practical and theoretical; you did not only counted the lines of code, but did much more, both on metrics formally and the added code itself (compared to Budai thesis)

- Go faster but precise on first slides; on video focus how SRs actually read the content presented

Comments:

- Other than the video, with some habits, we stay within 20 minutes (17/18 minutes), so not so bad. Have to practice many times and become faster and smoother overtime, confident and stuff.