React Native PoC: NFC, Camera & WebView Integration

A comprehensive technical specification for building a minimal proof-of-concept Android application demonstrating cross-platform capabilities with native hardware integration.

Executive Overview

Project Purpose

This proof-of-concept demonstrates React Native's capability to integrate critical native hardware features whilst maintaining a clean, maintainable codebase. The application serves as a technical validation for organisations evaluating React Native for mobile solutions requiring NFC tag reading, camera capture, and bidirectional communication with embedded web applications.

The PoC takes a pragmatic, fixed-scope approach with concrete technical choices that eliminate ambiguity and accelerate delivery. Rather than exploring multiple options, we've locked in battle-tested libraries and a clear three-screen architecture.

Key Capabilities

- Read NFC tags (NDEF text/URL payloads)
- Capture photos using device camera
- Embed React web application via WebView
- Bidirectional data bridge between native and web layers
- Clean, maintainable architecture suitable for scaling
- Android-first approach with optional iOS support

Timeline estimate: 2-3 weeks for Android implementation with basic testing and documentation.

Technical Stack: Core Decisions

React Native CLI

We're explicitly using **React Native CLI**, not Expo managed workflow. This decision is non-negotiable for the PoC because full NFC access requires native module configuration that Expo's managed workflow restricts.

The CLI approach gives us complete control over native Android code, gradle configuration, and module linking—essential for react-native-nfc-manager integration.

Android Primary Target

Android is the **mandatory platform** for this PoC. NFC support is mature and widely available across Android devices. Camera APIs are robust and well-documented.

Android development setup is simpler for teams without Mac hardware, making it the pragmatic choice for rapid PoC delivery.

iOS: Optional Enhancement

iOS support is categorised as "nice to have" if time permits. Whilst React Native supports iOS, NFC capabilities are more constrained—limited to background tag reading on iPhone 7+ with specific iOS versions.

For PoC validation, Android alone proves the technical feasibility adequately.

Navigation Architecture

The application uses **@react-navigation/native**, the de facto standard for React Native navigation. This library provides robust, production-ready navigation patterns with excellent TypeScript support and a large community.

For this PoC, we'll implement either a **bottom tab navigator** (recommended for quick access to all three screens) or a **simple stack navigator** if the client prefers a more linear flow. The bottom tab approach mirrors common mobile UX patterns and allows users to switch contexts freely.

Three-Screen Structure

- **NFC Screen:** Hardware interaction for reading NFC tags with session management
- **Camera Screen:** Live preview and photo capture with permission handling
- **WebView Screen:** Embedded React web application receiving data from native screens

Each screen is self-contained with clear responsibilities, making the codebase easy to understand and extend. Navigation state can optionally be persisted if the client requires it, though it's not critical for PoC validation.

Library Decisions: NFC Integration

Selected Package

react-native-nfc-manager

Also available as `@xiphoo/react-native-nfc-manager` (scoped variant).

This is the most mature and actively maintained NFC library for React Native, with over 1,400 GitHub stars and regular updates supporting recent React Native versions.

Implementation Scope

For this PoC, we'll implement a focused subset of NFC functionality:

- **Device capability detection:** Check if NFC hardware exists and is enabled
- **Session management:** Start and stop NFC reading sessions cleanly
- **NDEF payload reading:** Parse text and URL records from tags
- **Error handling:** Gracefully manage permission denials, disabled NFC, and read failures

Explicitly out of scope: Writing to NFC tags. Reading is sufficient to prove the integration pattern, and writing introduces additional complexity around tag formatting, write protection, and failure modes that aren't necessary for PoC validation.

The library supports both Android and iOS, though iOS has platform-specific limitations we'll document but won't block on for this PoC.

NFC Technical Details

01

Capability Check

On app launch or NFC screen mount, we query whether the device supports NFC hardware using `NfcManager.isSupported()`. If unsupported, we display a clear message rather than showing non-functional UI.

02

Permission & Enablement

Android requires NFC permissions in AndroidManifest.xml. We'll also check if NFC is currently enabled on the device and prompt users to enable it in system settings if disabled.

03

Session Lifecycle

When the user taps "Start NFC scan", we call `NfcManager.registerTagEvent()` to begin listening. This registers a callback that fires when an NFC tag enters proximity.

04

NDEF Parsing

On tag detection, we parse the NDEF message structure to extract text or URL records. The library provides helper methods to decode common NDEF record types.

05

Display & Bridge

The parsed content is displayed in the NFC screen UI and simultaneously posted to the WebView via the data bridge (detailed in later slides).

Library Decisions: Camera Integration

For camera functionality, we've evaluated two primary options and made a clear recommendation to avoid analysis paralysis during development.

Option A: react-native-vision-camera (Recommended)

This is our **fixed choice for the PoC** unless the client has specific constraints requiring Expo. Vision Camera is a modern, performant library created by Marc Rousavy with excellent TypeScript support and active maintenance.

Key advantages:

- Excellent documentation and examples
- High-performance camera preview with minimal lag
- Optional frame processor support (enables QR/barcode scanning)
- Works seamlessly with React Native CLI
- Strong community and regular updates

For this PoC, we'll use basic photo capture. Frame processors and video recording capabilities are available but out of scope.

Option B: expo-camera (Fallback)

If the client insists on using Expo in bare workflow, expo-camera is an acceptable fallback. It provides basic camera capture capabilities with simpler API surface.

Trade-offs:

- Simpler API, which can be beneficial for straightforward use cases
- Part of the Expo ecosystem, which some teams prefer
- Less flexibility for advanced features
- Requires Expo bare workflow setup, adding complexity

We don't recommend this path unless there are compelling organisational reasons to standardise on Expo tooling.

Camera Implementation Strategy

react-native-vision-camera on Android

This is the locked-in approach for the PoC. The implementation will be straightforward and focused on core functionality without unnecessary complexity.

Setup Requirements

- Add camera permissions to AndroidManifest.xml
- Request runtime permissions on first camera screen access
- Configure gradle dependencies per library documentation
- Handle permission denial gracefully with clear user messaging

Core Functionality

- Display live camera preview in Camera Screen
- Single "Capture" button to take photo
- Store last capture metadata in component state
- Generate simple filename (e.g., "image_2025-12-14.jpg")
- No gallery integration (out of scope for PoC)

The captured photo data will be simplified to a metadata object containing filename and timestamp, which is then passed to the WebView. We're not implementing full image gallery functionality or cloud upload—those are production concerns beyond PoC scope.

Library Decisions: WebView Integration

react-native-webview

The definitive library for embedding web content in React Native applications. This is a community-maintained package that was extracted from React Native core and is now the standard solution.

Capabilities we'll use:

- Load remote URLs or local HTML bundles
- Bidirectional communication via postMessage API
- JavaScript injection for bridge setup
- Progress indicators and error handling

Embedded React SPA

The WebView will host a simple React single-page application that receives data from the native layer. This can be either:

- **Remote:** Hosted at a URL like demo.mycompany.com/poc
- **Local:** Bundled with the app and loaded via `source={require('./web-app/index.html')}`

The local approach eliminates network dependencies and is recommended for PoC demos.

The web application will be intentionally simple—a React app with two data display sections and a message event listener. No complex state management or routing required.

App Structure

Screen 1: NFC Reader Interface

User Interface Elements

- **Primary Action Button**

  "Start NFC Scan" button prominently displayed, using platform-appropriate styling (Material Design for Android).

- **Status Display**

  Text area showing current state: "Ready to scan", "Scanning...", "Tag detected", or error messages.

- **Result Panel**

  Section displaying "Last scanned tag:" followed by the parsed content (text or URL from NDEF payload).

- **Capability Indicator**

  Small text at bottom showing NFC support status and whether NFC is currently enabled on device.

Interaction Flow

The user experience follows a clear, predictable pattern:

1. **User taps "Start NFC scan":** The app checks device NFC capability and permission status.
2. **Permission check:** If NFC permission isn't granted, request it. If NFC is disabled, show a modal prompting the user to enable it in system settings.
3. **Active scanning state:** Button changes to "Scanning..." with visual feedback (spinner or animation). The app is now listening for NFC tags.
4. **Tag detection:** When a tag enters proximity, the app reads the NDEF payload and immediately displays the content.
5. **Data bridging:** Simultaneously, the parsed tag data is packaged as JSON and posted to the WebView (covered in detail later).

Error states are handled explicitly with user-friendly messages rather than cryptic technical errors.

NFC Screen: Technical Implementation

Key Implementation Details

The NFC screen is implemented as a functional React component using hooks for state management. Here's the technical approach:

1

Initialisation

On component mount, we check `NfcManager.isSupported()` and store the result in state. This determines whether to show the scan button or a "Not supported" message.

2

Session Start

When the user taps the scan button, we call `NfcManager.registerTagEvent()` with a callback function. This callback receives the tag object when detected.

3

Data Parse

Inside the callback, we parse `tag.ndefMessage` to extract text or URI records. The library provides helper methods like `Ndef.text.decodePayload()`.

4

State Update

Parsed content is stored in component state and displayed. We also call a context method or navigation param to share it with the WebView screen.

Error handling wraps each step—if capability check fails, permission is denied, or tag reading throws an exception, we show appropriate error messages and log details for debugging.

Screen 2: Camera Capture Interface

User Interface Elements

- **Live camera preview:** Full-screen or large viewport showing real-time camera feed from device rear camera (default) with option to flip to front camera.
- **Capture button:** Large, easily tappable button overlaid on preview, typically positioned at bottom centre following platform conventions.
- **Last capture indicator:** Small thumbnail or text label showing "Last capture: [filename]" or "No captures yet" on initial load.
- **Permission prompt:** Modal or overlay requesting camera permission on first access, with clear explanation of why it's needed.

Interaction Flow

1. **First entry permission request:** When the user navigates to the Camera screen for the first time, we check camera permission status. If not granted, we show a permission request dialogue.
2. **Preview activation:** Once permission is granted, the live camera preview activates immediately, showing what the rear camera sees.
3. **User taps "Capture":** The app takes a photo using the current camera configuration (rear camera, auto-focus, auto-exposure).
4. **Processing:** The captured image is saved temporarily. For PoC purposes, we don't implement full gallery functionality—we just keep the last capture reference in state.
5. **Feedback:** The filename and timestamp are displayed in the "Last capture" indicator.
6. **Data bridge:** Capture metadata (filename, timestamp) is packaged as JSON and posted to the WebView.

Camera Screen: Technical Implementation

The Camera screen leverages react-native-vision-camera's declarative API with hooks-based configuration. Here's the technical approach:

01

Permission Management

We use the `useCameraPermission` hook provided by react-native-vision-camera. This returns permission status and a request function. On component mount, we check the status and request if needed.

02

Camera Device Selection

The `useCameraDevice('back')` hook retrieves the rear camera device. Vision Camera handles the complexity of device enumeration and selection internally.

03

Camera Component

We render the `<Camera>` component with the selected device, `isActive={true}`, and a ref for capturing photos. The component handles the preview rendering automatically.

04

Photo Capture

When the capture button is pressed, we call `cameraRef.current.takePhoto()`. This returns an object with the file path and metadata. We extract the path and construct our simple filename.

05

State & Bridge

Store the capture metadata in component state (using useState), update the UI to show the filename, and post the metadata to the WebView via the data bridge.

No gallery, no image processing, no cloud upload—the PoC scope is deliberately minimal to prove the integration pattern works.

Screen 3: WebView Display Interface

Purpose & Architecture

The WebView screen serves as the integration demonstration centrepiece. It embeds a React single-page application that receives data from the native NFC and Camera screens, proving that bidirectional communication works reliably.

React Native Side

The screen is a functional component rendering the `<WebView>` component from react-native-webview. Key configuration:

- **Source:** Either a remote URL (e.g., `source={{ uri: 'https://demo.mycompany.com/poc' }}`) or a local bundle (`source={require('./web-app/index.html')}`).
- **Ref:** We maintain a ref to the WebView instance to call `postMessage()` for sending data.
- **onMessage handler:** Receives messages from the web app if bidirectional communication is implemented.

Embedded Web App

The React SPA inside the WebView is intentionally simple:

- Two display sections: "NFC data from native" and "Camera data from native".
- Each section shows "No data yet" initially.
- JavaScript event listener for `window.addEventListener('message', handler)` that parses incoming JSON messages.
- When a message arrives, the appropriate section updates to display the received data.

The web app can be a Create React App build or a simple HTML file with embedded JavaScript—whatever is easiest to deploy.

WebView Screen: Technical Implementation

The WebView screen is the simplest from a React Native perspective but requires careful attention to the communication bridge setup.

Component Structure

A functional component using `useRef` to maintain a reference to the WebView instance. We also use Context or Redux to access global state containing the latest NFC tag data and camera capture metadata.

```
const webViewRef = useRef(null);
const { nfcData, cameraData } = useContext(AppContext);
```

Data Posting

Whenever `nfcData` or `cameraData` changes (detected via useEffect), we post the updated value to the WebView:

```
webViewRef.current?.postMessage(
  JSON.stringify({ type: 'NFC_TAG', payload: nfcData })
);
```

The WebView's JavaScript receives this via the message event listener.

Web App Loading

For local bundles, we package the web app build output with the React Native app and reference it with `source={require()}`. For remote URLs, we simply pass the URI. Local bundles are recommended for PoC demos to avoid network dependencies.

Error Handling

The `<WebView>` component supports `onError` and `onHttpError` props. We implement these to show user-friendly error messages if the web app fails to load.

Data Exchange Pattern

The postMessage Bridge: Architectural Overview

The communication pattern between React Native and the embedded WebView is built on the **postMessage API**, which is the standard, documented approach provided by react-native-webview. This is not a custom bridge—it's a well-tested, reliable mechanism used in production applications.

Why postMessage?

Standard & Documented

The postMessage API is the official communication method documented by react-native-webview. It's based on web standards (window.postMessage) adapted for the native-web boundary, making it familiar to web developers.

Type-Safe with JSON

By serialising data as JSON, we maintain type safety and can easily validate message structure on both sides. This is far superior to string concatenation or URL parameter hacks.

Bidirectional Capability

While the PoC focuses on native → web communication, the same mechanism supports web → native messages if needed. This makes it easy to extend the PoC later with commands from the web app to native (e.g., "trigger NFC scan again").

No Additional Dependencies

This approach requires no additional libraries or custom native modules. Everything we need ships with react-native-webview.

Data Flow: Native to WebView

This is the primary data flow for the PoC: NFC and Camera screens send data into the WebView.

Step-by-Step Implementation

NFC Screen: Tag Read Event

When an NFC tag is successfully read, the NFC screen callback function receives the parsed tag data (e.g., a text string or URL). This data is stored in global state (Context or Redux).

State Management

We use React Context to provide `nfcData` and `cameraData` across the app. When the NFC screen updates `nfcData` via `setNfcData(tagText)`, all consumers of that context re-render.

WebView Screen Reaction

The WebView screen has a `useEffect` hook watching `nfcData`. When it changes, the effect fires and calls `webViewRef.current.postMessage(JSON.stringify({ type: 'NFC_TAG', payload: tagText }))`.

Web App Receives Message

Inside the WebView, the web app has a listener: `window.addEventListener('message', event => { const data = JSON.parse(event.data); ... })`. When the message arrives, the web app parses the JSON and updates its UI.

The same pattern applies for camera capture data—Camera screen updates `cameraData`, WebView screen posts it, web app receives and displays.

Message Structure: JSON Schema

For clarity and maintainability, we define a simple JSON message schema that both the React Native side and web app side understand.

NFC Tag Message

```
{
  "type": "NFC_TAG",
  "payload": "Hello from NFC tag",
  "timestamp": "2025-12-14T10:30:00Z"
}
```

- **type:** String identifier for message type, allows the web app to handle different message kinds.
- **payload:** The actual tag content (text or URL string).
- **timestamp:** ISO 8601 timestamp of when the tag was read (optional but useful for debugging).

Camera Capture Message

```
{
  "type": "CAMERA_CAPTURE",
  "payload": {
    "filename": "image_2025-12-14_10-30-15.jpg",
    "timestamp": "2025-12-14T10:30:15Z"
  }
}
```

- **type:** Identifies this as a camera capture event.
- **payload:** Object containing filename and timestamp. We're not sending the actual image data (too large and unnecessary for PoC).

This schema is intentionally simple. In a production app, you might add version numbers, error fields, or richer metadata. For the PoC, this is sufficient.

React Native Side: Sending Messages

Code Example: NFC Screen

```
// After reading NFC tag
const handleTagRead = (tag) => {
  const tagText = parseNdefMessage(tag);

  // Update global state
  setNfcData({
    content: tagText,
    timestamp: new Date().toISOString()
  });

  // Display in UI
  setLastScanned(tagText);
};
```

Code Example: WebView Screen

```
const webViewRef = useRef(null);
const { nfcData, cameraData } = useContext(AppContext);

useEffect(() => {
  if (nfcData && webViewRef.current) {
    webViewRef.current.postMessage(
      JSON.stringify({
        type: 'NFC_TAG',
        payload: nfcData.content,
        timestamp: nfcData.timestamp
      })
    );
  }
}, [nfcData]);
```

This pattern ensures that whenever global state updates, the WebView receives the new data automatically. The web app doesn't need to poll or request updates—data is pushed whenever it changes.

Web App Side: Receiving Messages

Event Listener Setup

The React SPA inside the WebView needs to listen for message events. This is standard web JavaScript, not React Native-specific code:

```
// In a useEffect hook or componentDidMount
useEffect(() => {
  const handleMessage = (event) => {
    try {
      const data = JSON.parse(event.data);

      switch (data.type) {
        case 'NFC_TAG':
          setNfcContent(data.payload);
          setNfcTimestamp(data.timestamp);
          break;
        case 'CAMERA_CAPTURE':
          setCameraFilename(data.payload.filename);
          setCameraTimestamp(data.payload.timestamp);
          break;
        default:
          console.log('Unknown message type:', data.type);
      }
    } catch (error) {
      console.error('Failed to parse message:', error);
    }
  };

  window.addEventListener('message', handleMessage);

  // Cleanup
  return () => window.removeEventListener('message', handleMessage);
}, []);
```

The web app maintains its own React state for `nfcContent` and `cameraFilename`, which drive the UI rendering. This approach keeps concerns cleanly separated—native handles hardware, web handles presentation.

Web App UI: Display Components

The embedded React SPA has a simple structure focused on clearly displaying received data from the native layer.

NFC Data Section

```
<div className="data-section">
  <h2>NFC Data from Native</h2>
  {nfcContent ? (
    <>
      <p className="content">{nfcContent}</p>
      <p className="timestamp">
        Received: {nfcTimestamp}
      </p>
    </>
  ) : (
    <p className="placeholder">
      No NFC data yet. Scan a tag in the
      NFC screen.
    </p>
  )}
</div>
```

Camera Data Section

```
<div className="data-section">
  <h2>Camera Data from Native</h2>
  {cameraFilename ? (
    <>
      <p className="content">
        Last captured: {cameraFilename}
      </p>
      <p className="timestamp">
        At: {cameraTimestamp}
      </p>
    </>
  ) : (
    <p className="placeholder">
      No captures yet. Take a photo in
      the Camera screen.
    </p>
  )}
</div>
```

Basic CSS styling makes the sections visually distinct. The PoC doesn't require sophisticated design—clarity and functionality are the priorities.

Optional: Bidirectional Communication

Whilst the PoC scope focuses on native → web data flow, the architecture supports web → native messages with minimal additional code. This is worth documenting as a "bonus feature" if time permits.

Use Case Example

The web app could have a button "Request NFC Scan" that sends a command to the native layer, which then triggers the NFC screen to start scanning.

1

Web App Sends

The web app calls `window.ReactNativeWebView.postMessage(JSON.stringify({ type: 'REQUEST_NFC_SCAN' }))`.

2

Native Receives

The WebView component has an `onMessage` prop that fires when the web app posts a message. The handler parses the JSON and executes the appropriate native action.

3

Action Execution

The native side calls the NFC manager to start scanning, or navigates to the NFC screen with a flag to auto-start scanning.

Implementing this is straightforward once the unidirectional flow is working. It's marked as **optional** but demonstrates the full potential of the bridge architecture.

Global State Management Approach

For this PoC, we need a simple way to share `nfcData` and `cameraData` between screens. The recommended approach is **React Context** rather than Redux or other state management libraries.

Why React Context?

Simplicity

Context API is built into React—no additional dependencies. For a PoC with minimal state, this is ideal.

Sufficient Performance

We're not updating state frequently (only on NFC reads and camera captures), so Context's performance characteristics are perfectly adequate.

Easy to Understand

Engineers unfamiliar with Redux or MobX can immediately understand Context. This reduces onboarding friction if the PoC transitions to production development.

Context Structure

```
const AppContext = createContext({
  nfcData: null,
  setNfcData: () => {},
  cameraData: null,
  setCameraData: () => {}
});
```

The provider wraps the navigation container, making state available to all screens.

Error Handling Strategy

A robust PoC demonstrates error handling, not just the happy path. Each screen must gracefully handle failure modes.

NFC Screen Errors

- **NFC not supported:** Show clear message "Your device doesn't support NFC" with explanation.
- **NFC disabled:** Prompt user to enable NFC in system settings with a deep link to the settings page.
- **Permission denied:** Explain why NFC permission is needed and offer to request again.
- **Read failure:** If tag reading throws an exception, show "Failed to read tag. Please try again" with error code.

Camera Screen Errors

- **Permission denied:** Show modal explaining camera permission requirement with "Go to Settings" button.
- **Camera unavailable:** Handle cases where camera hardware fails (rare but possible).
- **Capture failure:** If `takePhoto()` throws, show error message and log details.
- **Low storage:** Whilst unlikely for a single photo, handle storage errors gracefully.

WebView Errors

- **Load failure:** If web app fails to load (network error for remote URLs, missing file for local), show error page with retry button.
- **Message parse failure:** If JSON parsing fails, log error but don't crash—graceful degradation.
- **HTTP errors:** Handle 404, 500, etc. with user-friendly messages.

All errors should be logged to the console (and ideally to a logging service in production) with sufficient detail for debugging.

PoC Definition

PoC Goal Statement

Primary Objective

Deliver a functioning React Native Android application that demonstrates the technical feasibility of integrating NFC tag reading, camera photo capture, and bidirectional communication with an embedded React web application—all within a clean, maintainable codebase that could form the foundation for production development.

Success Criteria

NFC Integration

User can tap an NFC tag containing NDEF text or URL, and the content is read and displayed reliably within 2 seconds of tag proximity.

Camera Integration

User can capture a photo with a single tap, see confirmation, and the capture metadata is stored and displayed immediately.

WebView Integration

Embedded React web app receives data from both NFC and Camera screens via postMessage bridge, displaying updates in real-time without page refresh.

Code Quality

Codebase follows React Native best practises, includes basic error handling, and is documented sufficiently for another engineer to extend it.

Deliverables: What the Client Receives

At the end of the PoC development cycle, the client will receive a complete package demonstrating the solution.

Android APK

A signed, installable Android application package. This can be installed on any Android device (version 8.0+) without needing development tools. The APK will be provided via secure file transfer or download link.

Demo Video

A 3-5 minute video recording demonstrating all three screens in action. The video will show: NFC screen scanning a tag and displaying content, Camera screen capturing a photo, WebView screen receiving and displaying data from both sources. This video serves as documentation and can be shared with stakeholders who don't have Android devices available.

Source Code Repository

Complete React Native project source code in a Git repository (GitHub, GitLab, or Bitbucket). Includes all native Android configuration, JavaScript/TypeScript code, and the embedded web app source. Repository will have a comprehensive README with setup instructions.

Technical Documentation

Written documentation covering architecture decisions, library versions, setup instructions, and extension guidelines. This document enables the client's team to understand the codebase and continue development if the PoC proceeds to production.

PoC Demo Flow: Complete Walkthrough

Here's exactly what a demo session would look like when presenting the PoC to stakeholders.

1

App Launch & NFC Screen

Demo begins by launching the app on an Android phone. The NFC screen is the default landing screen (or first tab). The screen shows "NFC Ready" status and a "Start NFC Scan" button.

2

NFC Tag Reading

Presenter taps "Start NFC Scan", then holds an NFC tag (programmed with text like "Hello from PoC!") against the phone's NFC antenna. Within 1-2 seconds, the screen displays "Last scanned tag: Hello from PoC!" confirming the read was successful.

3

Camera Screen Navigation

Presenter navigates to the Camera screen using bottom tabs. The live camera preview activates immediately, showing the rear camera feed.

4

Photo Capture

Presenter points the camera at an object (e.g., a laptop, a coffee cup) and taps the "Capture" button. The screen briefly flashes, then displays "Last capture: image_2025-12-14_15-30-00.jpg" confirming the photo was taken.

5

WebView Data Display

Finally, presenter navigates to the WebView screen. The embedded web app displays two sections: "NFC data from native: Hello from PoC!" and "Camera data from native: Last captured image_2025-12-14_15-30-00.jpg at 15:30:00". This proves the data bridge is functioning.

PoC Scope: Explicitly Included

To avoid scope creep and ensure clear expectations, here's what IS included in the PoC scope.

NFC Tag Reading

- Device capability detection
- NFC session management
- NDEF text and URL payload parsing
- Error handling for common failure modes
- UI feedback for scan status

Camera Photo Capture

- Permission request and handling
- Live camera preview
- Single photo capture
- Capture confirmation UI
- Metadata extraction (filename, timestamp)

WebView Integration

- Embedded React SPA (remote or local)
- postMessage data bridge (native → web)
- Display sections for NFC and camera data
- Real-time UI updates on data receipt

General App Features

- Three-screen navigation (bottom tabs or stack)
- Basic error handling and user feedback
- Android-specific configuration and permissions
- Clean, readable code structure

PoC Scope: Explicitly Excluded

Equally important is clarity about what's NOT in scope. These items could be future enhancements if the PoC proceeds to production.

Out of Scope: NFC

- Writing to NFC tags
- Reading non-NDEF tag formats
- P2P (peer-to-peer) NFC communication
- HCE (Host Card Emulation)
- Tag authentication or encryption

Out of Scope: Camera

- Gallery or photo browsing UI
- Image editing or filters
- Video recording
- QR/barcode scanning (though technically possible with vision-camera)
- Cloud upload or server integration
- Image compression or optimisation

Out of Scope: WebView

- Complex web → native commands (beyond basic demo)
- File uploads from web to native
- Native plugin injection into web context
- Multiple WebViews or web-based routing

Out of Scope: General

- iOS implementation (optional nice-to-have only)
- Backend API integration
- User authentication
- Data persistence (SQLite, AsyncStorage)
- Analytics or crash reporting
- Production-grade UI design
- Automated testing
- App store deployment

Why This PoC Approach Works

Technical Validation

This PoC proves the critical technical unknowns that often block React Native adoption for hardware-dependent applications:

- **NFC feasibility:** Demonstrates that React Native can reliably access NFC hardware on Android, addressing common concerns about "does React Native support NFC?"
- **Camera quality:** Shows that react-native-vision-camera provides professional-grade camera capabilities without needing to write native code.
- **Web integration:** Proves that hybrid native-web architectures are viable, enabling teams to leverage existing web applications within React Native shells.

Decision-Making Foundation

After completing this PoC, the client will have concrete evidence to make critical technology decisions:

- Can we build our production app in React Native, or do we need fully native development?
- What's the performance like for hardware features?
- How complex is the development and maintenance burden?
- What's the developer experience like?

These questions can't be answered with research alone—running code is required.

Development Timeline & Effort Estimate

Realistic timeline for a single experienced React Native developer working full-time on the PoC.

Days 1-2: Project Setup

- Initialise React Native CLI project
- Configure navigation library
- Set up basic three-screen structure
- Install and link NFC, camera, and WebView libraries
- Configure Android permissions in manifest
- Create Context provider for global state

Days 3-5: Core Features

- Implement NFC screen with tag reading
- Build Camera screen with capture functionality
- Create simple React SPA for WebView
- Implement postMessage bridge
- Wire up data flow between screens
- Test end-to-end integration

Days 6-7: Polish & Testing

- Add error handling for all failure modes
- Improve UI feedback and loading states
- Test on multiple Android devices
- Fix bugs discovered during testing
- Record demo video

Days 8-10: Documentation & Delivery

- Write technical documentation
- Document architecture decisions
- Create README with setup instructions
- Build signed APK
- Prepare delivery package
- Client presentation/demo

Total estimate: 10-12 development days (2-2.5 weeks) for a polished, documented PoC. Add buffer for unknowns.

Technical Deep Dives

Android Permissions & Configuration

Proper permission configuration is critical for NFC and camera functionality. Here's exactly what needs to be added to `AndroidManifest.xml`.

NFC Permissions

```
<uses-permission android:name="android.permission.NFC" />
<uses-feature
  android:name="android.hardware.nfc"
  android:required="false" />
```

The `android:required="false"` attribute allows the app to be installed on devices without NFC, though the feature won't work. Set to "true" if NFC is mandatory.

Camera Permissions

```
<uses-permission android:name="android.permission.CAMERA" />
<uses-feature
  android:name="android.hardware.camera"
  android:required="false" />
<uses-feature
  android:name="android.hardware.camera.autofocus"
  android:required="false" />
```

Runtime permission requests are handled by react-native-vision-camera's built-in permission APIs.

Internet Permission (for remote WebView content)

```
<uses-permission android:name="android.permission.INTERNET" />
```

Required if the WebView loads content from a remote URL. Not needed for local bundle.

Library Versions & Dependencies

For reproducibility and to avoid dependency conflicts, we document exact versions used in the PoC. These are current as of December 2024.

Core Dependencies

```
{
  "dependencies": {
    "react": "18.2.0",
    "react-native": "0.73.0",
    "@react-navigation/native": "^6.1.9",
    "@react-navigation/bottom-tabs": "^6.5.11",
    "react-native-screens": "^3.29.0",
    "react-native-safe-area-context": "^4.8.2"
  }
}
```

Feature Dependencies

```
{
  "dependencies": {
    "react-native-nfc-manager": "^3.15.0",
    "react-native-vision-camera": "^3.6.0",
    "react-native-webview": "^13.6.0"
  }
}
```

Always check for the latest stable versions when starting development, but lock versions in `package.json` to ensure consistent builds across team members.

Gradle Configuration

React Native 0.73+ uses the New Architecture by default. For this PoC, we can disable it if needed for compatibility:

```
// android/gradle.properties
newArchEnabled=false
```

This avoids potential issues with native modules that haven't fully migrated to the New Architecture yet.

NFC Deep Dive: NDEF Message Structure

Understanding NDEF (NFC Data Exchange Format) is crucial for parsing tag data correctly.

What is NDEF?

NDEF is a standardised format for storing data on NFC tags. It consists of one or more NDEF Records, each containing:

- **Type Name Format (TNF):** Indicates how to interpret the type field (e.g., well-known type, MIME type, URI)
- **Type:** The type of the record (e.g., "T" for text, "U" for URI)
- **Payload:** The actual data (encoded according to the type)
- **ID:** Optional identifier for the record

Common Record Types

Text Record (TNF: Well-known, Type: "T")

Contains plain text with language encoding. Format: [encoding byte][language code][text]. Example: "\x02enHello World" represents English text "Hello World".

URI Record (TNF: Well-known, Type: "U")

Contains a URL. The first byte is a URI identifier code (e.g., 0x01 = "http://www.", 0x03 = "http://"). Example: "\x03example.com" represents "http://example.com".

The react-native-nfc-manager library provides helper functions like `Ndef.text.decodePayload()` and `Ndef.uri.decodePayload()` to handle these encodings.

NFC Implementation: Code Walkthrough

Here's a detailed code example showing how to implement NFC tag reading in the NFC screen component.

```
import React, { useState, useEffect } from 'react';
import NfcManager, { NfcTech, Ndef } from 'react-native-nfc-manager';

function NfcScreen() {
 const [isSupported, setIsSupported] = useState(false);
 const [lastScanned, setLastScanned] = useState(null);
 const [scanning, setScanning] = useState(false);

 useEffect(() => {
 async function checkNfc() {
 const supported = await NfcManager.isSupported();
 setIsSupported(supported);
 if (supported) {
 await NfcManager.start();
 }
 }
 checkNfc();

 return () => {
 NfcManager.cancelTechnologyRequest().catch(() => {});
 };
 }, []);

 async function startScan() {
 try {
 setScanning(true);
 await NfcManager.requestTechnology(NfcTech.Ndef);

 const tag = await NfcManager.getTag();
 if (tag.ndefMessage && tag.ndefMessage.length > 0) {
 const record = tag.ndefMessage[0];
 let text = '';

 if (record.tnf === 1) { // Well-known type
 if (record.type[0] === 84) { // "T" for text
 text = Ndef.text.decodePayload(record.payload);
 } else if (record.type[0] === 85) { // "U" for URI
 text = Ndef.uri.decodePayload(record.payload);
 }
 }

 setLastScanned(text);
 // Post to WebView here
```

Camera Deep Dive: Vision Camera Architecture

React Native Vision Camera uses a declarative API that feels natural to React developers whilst providing low-level camera control.

Core Concepts

01

Camera Devices

Physical cameras on the device (front, back, wide-angle, telephoto). The `useCameraDevice()` hook selects the appropriate device based on position ('front' or 'back') and criteria (wide-angle, ultra-wide, etc.).

02

Camera Component

The `<Camera>` component renders the live preview. It takes a `device` prop and an `isActive` boolean that controls whether the camera is running.

03

Camera Format

Optional configuration for resolution, frame rate, colour space, etc. For the PoC, we can use the default format which selects optimal settings automatically.

04

Photo Capture

The `useRef` hook maintains a reference to the Camera component. Calling `cameraRef.current.takePhoto()` returns a promise that resolves to an object with the photo's file path.

Camera Implementation: Code Walkthrough

Here's a detailed code example for the Camera screen component.

```
import React, { useState, useRef, useEffect } from 'react';
import { Camera, useCameraDevice, useCameraPermission } from 'react-native-vision-camera';

function CameraScreen() {
 const { hasPermission, requestPermission } = useCameraPermission();
 const device = useCameraDevice('back');
 const camera = useRef(null);
 const [lastCapture, setLastCapture] = useState(null);

 useEffect(() => {
 if (!hasPermission) {
 requestPermission();
 }
 }, [hasPermission]);

 async function capturePhoto() {
 if (!camera.current) return;

 try {
 const photo = await camera.current.takePhoto({
 qualityPrioritization: 'balanced',
 flash: 'off',
 });

 const filename = `image_${new Date().toISOString()}.jpg`;
 const timestamp = new Date().toISOString();

 setLastCapture({ filename, timestamp });

 // Post to WebView here
 // postToWebView({ type: 'CAMERA_CAPTURE', payload: { filename, timestamp } });

 } catch (error) {
 console.error('Capture error:', error);
 }
 }

 if (!hasPermission) {
 return <Text>No camera permission</Text>;
 }

 if (!device) {
 return <Text>Loading camera...</Text>;
```

WebView Deep Dive: The postMessage Bridge

The communication bridge between React Native and the embedded web app is the most conceptually interesting part of the PoC. Let's examine it in detail.

How postMessage Works

Native → Web Direction

React Native calls `webViewRef.current.postMessage(stringData)`. This triggers the web app's `window.addEventListener('message', handler)`.

The data must be a string (typically serialised JSON). The web app receives it in `event.data`.

Timing: Messages are queued if sent before the web app is fully loaded. Once the web app's message listener is registered, queued messages are delivered.

Web → Native Direction

The web app calls `window.ReactNativeWebView.postMessage(stringData)`. This triggers the WebView component's `onMessage` prop callback.

The callback receives an event object with `event.nativeEvent.data` containing the string sent from the web.

Security: Only strings can be sent. Objects must be serialised. This prevents code injection attacks.

Implementation Details

The `window.ReactNativeWebView` object is injected by the react-native-webview library. It's not a standard web API—it only exists in the WebView context.

WebView Implementation: Native Side Code

Here's the React Native WebView screen implementation with complete postMessage integration.

```
import React, { useRef, useEffect, useContext } from 'react';
import { WebView } from 'react-native-webview';
import { AppContext } from './AppContext';

function WebViewScreen() {
 const webViewRef = useRef(null);
 const { nfcData, cameraData } = useContext(AppContext);

 // Send NFC data when it changes
 useEffect(() => {
 if (nfcData && webViewRef.current) {
 const message = JSON.stringify({
 type: 'NFC_TAG',
 payload: nfcData.content,
 timestamp: nfcData.timestamp
 });
 webViewRef.current.postMessage(message);
 }
 }, [nfcData]);

 // Send camera data when it changes
 useEffect(() => {
 if (cameraData && webViewRef.current) {
 const message = JSON.stringify({
 type: 'CAMERA_CAPTURE',
 payload: {
 filename: cameraData.filename,
 timestamp: cameraData.timestamp
 }
 });
 webViewRef.current.postMessage(message);
 }
 }, [cameraData]);

 // Handle messages from web app (optional)
 function handleMessage(event) {
 try {
 const data = JSON.parse(event.nativeEvent.data);
 console.log('Received from web:', data);
 // Handle web → native commands here
 } catch (error) {
 console.error('Failed to parse web message:', error);
 }
```

WebView Implementation: Web App Side Code

The React SPA inside the WebView needs to listen for messages and update its UI accordingly.

```
import React, { useState, useEffect } from 'react';

function WebApp() {
 const [nfcData, setNfcData] = useState(null);
 const [cameraData, setCameraData] = useState(null);

 useEffect(() => {
 function handleMessage(event) {
 try {
 const data = JSON.parse(event.data);

 switch (data.type) {
 case 'NFC_TAG':
 setNfcData({
 content: data.payload,
 timestamp: data.timestamp
 });
 break;

 case 'CAMERA_CAPTURE':
 setCameraData({
 filename: data.payload.filename,
 timestamp: data.payload.timestamp
 });
 break;

 default:
 console.log('Unknown message type:', data.type);
 }
 } catch (error) {
 console.error('Failed to parse message:', error);
 }
 }

 // Listen for messages from React Native
 window.addEventListener('message', handleMessage);

 // Cleanup
 return () => window.removeEventListener('message', handleMessage);
 }, []);

 // Optional: Send message to React Native
 function sendToNative(data) {
```

Testing Strategy for the PoC

Whilst this is a PoC and not production code, basic testing ensures the demo is reliable and builds confidence in the approach.

Manual Testing

- **Happy path:** Test complete flow NFC → Camera → WebView on a physical Android device
- **Error paths:** Test permission denials, disabled NFC, camera failures
- **Edge cases:** Empty NFC tags, multiple rapid captures, WebView reload
- **Multiple devices:** Test on at least 2 different Android devices (different manufacturers/OS versions)

Automated Testing (Optional)

For a PoC, automated tests are optional but can be valuable:

- **Unit tests:** Test data parsing functions (NDEF parsing, JSON serialisation)
- **Component tests:** Test individual screen components with mocked dependencies
- **Integration tests:** Test data flow through Context

Use Jest and React Native Testing Library if implementing automated tests.

Testing Checklist

- ✓ NFC tag read on multiple tag types (text, URL)
- ✓ Camera capture in different lighting conditions
- ✓ WebView displays data correctly
- ✓ Navigation between screens works smoothly
- ✓ App handles permission denials gracefully
- ✓ Error messages are clear and actionable
- ✓ App doesn't crash on background/foreground

Common Pitfalls & How to Avoid Them

Based on experience with similar PoCs, here are common issues that can derail development and how to prevent them.

Pitfall: NFC Library Linking Issues

React Native 0.60+ uses autolinking, but sometimes native modules don't link correctly. **Solution:** After installing react-native-nfc-manager, run `cd android && ./gradlew clean` then rebuild. Check that the package is listed in `settings.gradle`.

Pitfall: Camera Permission Requests Not Appearing

If the permission dialogue doesn't show, it's usually because the permission is already denied permanently. **Solution:** Uninstall the app completely and reinstall, or manually reset app permissions in Android settings. Always test permission flow on a fresh install.

Pitfall: WebView postMessage Not Working

If messages aren't reaching the web app, the listener might not be registered yet. **Solution:** Ensure the web app's message listener is set up in a useEffect or componentDidMount that runs immediately. You can also add `onLoadEnd` prop to WebView to only send messages after the page loads.

Pitfall: NFC Session Not Cancelling

If you start an NFC session but don't cancel it properly, subsequent scans may fail. **Solution:** Always call `NfcManager.cancelTechnologyRequest()` in a finally block or cleanup function.

Performance Considerations

Even for a PoC, basic performance awareness ensures the app feels responsive and professional.

Camera Preview Performance

react-native-vision-camera uses native camera APIs with hardware acceleration, so the preview should be smooth at 30+ FPS. If you see lag, check that you're not rendering heavy components over the camera preview. The Camera component should take up most of the screen with minimal overlays.

NFC Reading Latency

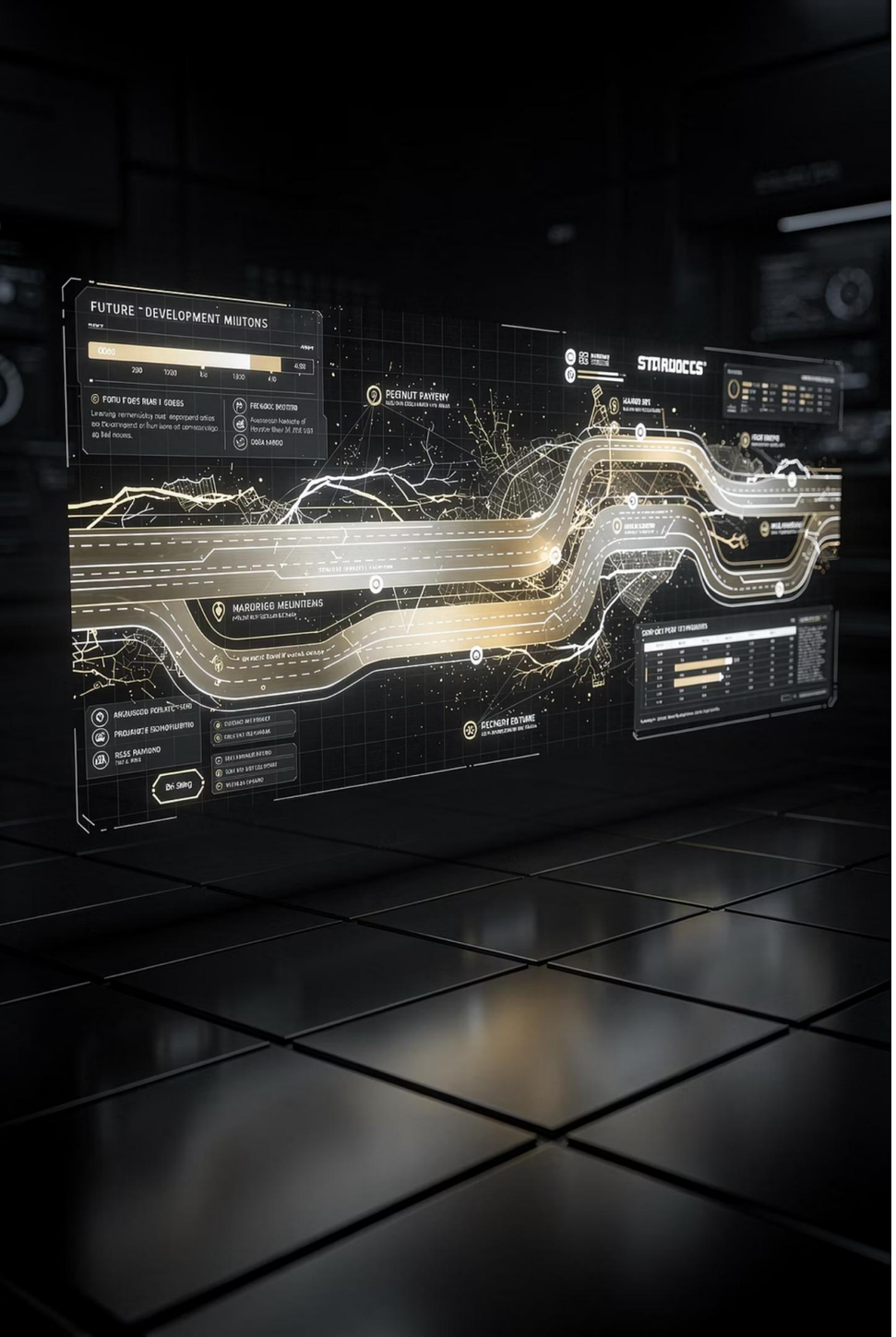
NFC tag reading typically takes 100-500ms depending on tag type and data size. This is hardware-limited and can't be optimised much. Provide immediate visual feedback (spinner or "Reading..." text) so users know the app is working.

WebView Loading Time

If loading a remote URL, the WebView may take 1-3 seconds to load depending on network speed. Show a loading indicator using the WebView's `onLoadProgress` prop. For the PoC, consider using a local bundle to eliminate network latency entirely.

Context Re-renders

React Context causes all consumers to re-render when context value changes. For this PoC with only 3 screens, this isn't a problem. If scaling to more screens, consider using multiple contexts or a state management library like Zustand for better performance.



Extending the PoC

If the PoC Proceeds to Production

Assuming the PoC successfully validates the technical approach, here's what would need to change for production development.

Architecture Enhancements

State Management

Replace React Context with a more scalable solution like Redux, Zustand, or Jotai. These libraries offer better performance for frequent updates, time-travel debugging, and clearer separation of concerns.

Navigation Improvements

Add deep linking support, persist navigation state, implement proper back button handling, and add transition animations for a polished feel.

Error Tracking

Integrate Sentry or Bugsnag for crash reporting and error tracking. Add proper logging infrastructure to debug production issues.

Data Persistence

Add local storage using AsyncStorage or SQLite to cache NFC reads and camera captures. Implement offline-first patterns so the app works without connectivity.

Backend Integration

If the production app needs server communication, add API clients, authentication, and data sync mechanisms.

Testing Infrastructure

Implement comprehensive unit, integration, and E2E tests. Set up CI/CD pipelines for automated testing and deployment.

Feature Extensions: NFC Enhancements

Beyond basic tag reading, there are many NFC features that could be added for production.

NFC Tag Writing

Allow users to write data to NFC tags. This requires understanding tag types (Topaz, MIFARE, etc.), memory structure, and write protection. The react-native-nfc-manager library supports writing, but it's more complex than reading.

P2P Communication

Enable phone-to-phone data transfer using NFC. This is useful for sharing contact information, photos, or app data. Android Beam (deprecated) can be replaced with custom P2P implementations.

HCE (Host Card Emulation)

Allow the phone to emulate an NFC card, enabling payment or access control use cases. This requires native Android development and is significantly more complex than tag reading.

Feature Extensions: Camera Enhancements

The camera functionality can be significantly enhanced for production applications.

QR/Barcode Scanning

react-native-vision-camera supports frame processors that can detect and decode QR codes and barcodes in real-time. This is useful for product scanning, ticketing, or inventory management applications. Requires installing the vision-camera-code-scanner plugin.

Photo Gallery & Management

Implement a full gallery UI for browsing captured photos, deleting unwanted ones, and selecting multiple photos. Integration with the device's photo library requires additional permissions and the @react-native-camera-roll/camera-roll library.

Video Recording

Vision Camera supports video recording with configurable quality, frame rate, and codec settings. This adds complexity around file size management, storage permissions, and video playback UI.

Image Processing & Filters

Add real-time filters, image editing capabilities, or computer vision features. This could use frame processors for effects or post-processing libraries like react-native-image-filter-kit for editing captured photos.

Feature Extensions: WebView Enhancements

The WebView integration can evolve into a sophisticated hybrid architecture for production applications.

Rich Command Set

Expand beyond simple data display to support complex web → native commands: "Open camera", "Start NFC scan", "Request location", "Open external link in browser", etc. This requires a well-defined command schema and routing logic on the native side.

File Upload Support

Allow the web app to upload files (photos, documents) to native storage or cloud services. This requires implementing custom file picker UI and bridging file data between web and native contexts.

Offline Support

Cache the web app bundle locally so it works offline. Service workers and progressive web app techniques can be combined with native caching to create a robust offline experience.

Multiple WebViews

Some production apps use multiple WebViews for different sections (e.g., one for the main app, one for help documentation). This requires careful memory management and state isolation between WebView instances.

iOS Support: Considerations & Constraints

If the client wants iOS support, here's what changes and what constraints apply.

NFC on iOS

Hardware support: iPhone 7 and later with iOS 11+ support NFC reading. iPhone XS and later with iOS 13+ support background tag reading.

Limitations:

- iOS NFC is more restricted than Android—no always-on background reading
- User must actively trigger a scan (can't passively detect tags)
- NDEF reading works, but writing and HCE are limited
- CoreNFC API is more complex than Android's NFC API

The react-native-nfc-manager library abstracts these differences, but iOS behaviour will feel different to users.

Camera on iOS

react-native-vision-camera has excellent iOS support with feature parity to Android. Camera functionality will work the same on both platforms with minimal code changes.

Permissions: iOS uses a different permission model. You'll need to add usage descriptions to Info.plist:

```
<key>NSCameraUsageDescription</key>
<string>We need camera access to
capture photos</string>
```

WebView on iOS

react-native-webview works identically on iOS. The postMessage bridge functions the same way. iOS uses WKWebView (the modern WebKit engine) which is fast and standards-compliant.

iOS Development: Additional Considerations

Development Environment

iOS development requires macOS and Xcode. If the development team doesn't have Mac hardware, this is a significant blocker. Options:

- **Use Mac hardware:** Purchase MacBook or iMac for iOS development
- **Use cloud services:** MacStadium, AWS Mac instances, or similar cloud Mac solutions
- **Use CI/CD for builds:** Develop on Android, use cloud CI/CD (like Bitrise or CircleCI with Mac executors) for iOS builds

CocoaPods & Dependencies

iOS uses CocoaPods for native dependency management. After installing npm packages, you must run `cd ios && pod install` to install native iOS dependencies. This is an extra step not needed on Android.

Apple Developer Account

Testing on physical iOS devices requires an Apple Developer account ($99/year). Simulator testing works without an account but doesn't support NFC (simulator can't access hardware).

Timeline Impact

Adding iOS support to the PoC would add approximately **3-5 additional development days**, mainly for iOS-specific configuration, testing on iOS devices, and handling platform-specific quirks.

Security Considerations

Even for a PoC, basic security awareness prevents vulnerabilities from being baked into the architecture.

1

WebView JavaScript Injection

Never inject untrusted JavaScript into the WebView using `injectedJavaScript` prop. If you need to run code in the web context, bundle it with the web app or load it from a trusted domain. Injecting user-controlled strings can lead to XSS attacks.

2

postMessage Data Validation

Always validate and sanitise data received via postMessage before using it. Check that JSON parsing succeeds, verify message type and structure, and validate payload contents. Malicious web apps could send crafted messages to exploit native code.

3

NFC Tag Data Sanitisation

NFC tags can contain malicious URLs or executable code. If displaying URLs from tags, don't automatically open them in a browser—show them to users and let them choose to open. Validate URL schemes (allow http/https, block javascript:, file:, etc.).

4

File Storage Permissions

If capturing photos, ensure they're stored in the app's private directory initially. Don't save to public storage without user consent. Respect user privacy by implementing proper data retention policies.

Documentation Deliverables

Comprehensive documentation ensures the PoC can be understood and extended by other engineers.

README.md Structure

01

Project Overview

Brief description of the PoC purpose, what it demonstrates, and key technologies used. Should be understandable to non-technical stakeholders.

02

Prerequisites

Required software versions: Node.js, React Native CLI, Android Studio, Java JDK. Include download links and version numbers used during development.

03

Installation Steps

Step-by-step instructions: clone repo, run `npm install`, Android-specific setup (gradle sync, emulator config), running the app with `npx react-native run-android`.

04

Project Structure

Explain directory layout: where to find screens, context providers, navigation config, native Android code, and the embedded web app.

05

Key Concepts

Document the postMessage bridge, NFC reading flow, camera capture pattern, and state management approach with code snippets.

06

Troubleshooting

Common issues and solutions: linking problems, permission errors, NFC not working, WebView blank screen, etc.

Cost-Benefit Analysis: React Native vs. Native

After completing the PoC, the client will want to evaluate React Native against fully native development. Here's the framework for that decision.

React Native Advantages

- **Code sharing:** 70-90% of code shared between iOS and Android (depending on how much native code is needed)
- **Development speed:** Faster iteration with hot reload, single codebase to maintain
- **Developer availability:** Larger pool of JavaScript/React developers compared to native specialists
- **Web integration:** Natural fit for apps that need embedded web content or want to share logic with web applications
- **Community & ecosystem:** Large library ecosystem, active community, extensive documentation

Native Development Advantages

- **Performance:** Slightly better performance for computationally intensive tasks or complex animations
- **Platform features:** Immediate access to new iOS/Android APIs without waiting for React Native support
- **Deep platform integration:** Easier to implement platform-specific features that don't fit the React Native model
- **No bridge overhead:** No JavaScript-native bridge means no serialisation cost for data passing
- **Mature tooling:** Xcode and Android Studio have decades of refinement

For this PoC's use case (NFC + Camera + WebView), React Native is a strong fit because the performance-critical parts (camera preview, NFC hardware) are handled by native modules, whilst the UI and business logic benefit from React's developer experience.

Budget & Resource Planning

If proceeding from PoC to production, here's a realistic resource plan.

Team Composition

2-3

React Native Developers

Responsible for feature development, UI implementation, state management, and business logic. Mid to senior level experience required.

1

Native Android/iOS Specialist

Part-time role for complex native module development, debugging platform-specific issues, and performance optimisation.

1

Backend Developer

If the production app needs server integration, one backend developer for API development, authentication, and data sync.

1

QA Engineer

Manual and automated testing, device compatibility testing, bug reporting and verification. Essential for production quality.

Timeline Estimate: PoC to MVP

Assuming the PoC is successful and the client wants to build a minimal viable product:

- **Phase 1 (Weeks 1-4):** Refactor PoC code for production quality, implement state management, add comprehensive error handling
- **Phase 2 (Weeks 5-8):** Build out additional features, implement backend integration, add offline support
- **Phase 3 (Weeks 9-12):** UI polish, comprehensive testing on multiple devices, bug fixes, performance optimisation
- **Phase 4 (Weeks 13-16):** iOS implementation (if required), app store preparation, beta testing, launch

Total: 3-4 months from PoC to production MVP with the team above.

Next Steps After PoC Completion

Once the PoC is delivered and demoed, here's the recommended decision-making process.

Stakeholder Review

Present the PoC to all stakeholders: engineering leadership, product management, and any end users if possible. Gather feedback on functionality, performance, and user experience. Document any concerns or requests.

Technical Assessment

Engineering team evaluates the codebase: Is the architecture sound? Are there any technical red flags? Can this be scaled to production requirements? What would need to change?

Go/No-Go Decision

Based on PoC results, decide whether to proceed with React Native for production development, or explore alternative approaches (fully native, Flutter, etc.). This decision should consider: technical feasibility (proven by PoC), developer availability, timeline constraints, budget, and strategic fit.

Production Planning

If proceeding, create detailed production development plan: feature roadmap, team hiring/allocation, architecture refinements, backend design, and timeline with milestones.

Summary: Key Takeaways

Technical Feasibility: Proven

This PoC demonstrates that React Native can reliably access NFC hardware, camera, and integrate with embedded web applications on Android. The postMessage bridge provides a clean, maintainable pattern for native-web communication.

Fixed Scope: Essential

By locking in specific libraries (react-native-nfc-manager, react-native-vision-camera, react-native-webview) and focusing on core functionality, we eliminate analysis paralysis and ensure rapid delivery. The PoC proves the concept without building unnecessary features.

Android-First: Pragmatic

Targeting Android as the primary platform for the PoC accelerates development and proves feasibility on the more open, NFC-mature platform. iOS support can be added later if needed, with most code shared between platforms.

Production-Ready Foundation

Whilst this is a PoC, the architecture patterns—Context for state, postMessage for bridging, clean screen separation—provide a solid foundation for production development. Extending the PoC to MVP is straightforward.

Final Recommendation

React Native is well-suited for this use case. The PoC de-risks the hardware integration concerns and demonstrates that the developer experience is good. If the team has React expertise and the timeline/budget favour cross-platform development, proceeding with React Native for production is recommended. The PoC has successfully validated the technical approach.