

A GOVERNANCE FRAMEWORK FOR NATIVE MODULE DEVELOPMENT IN ENTERPRISE REACT NATIVE APPLICATIONS

Sandeep Kumar Mauriya

Independent Researcher, London, United Kingdom

ABSTRACT

Enterprise organisations building React Native applications repeatedly face the same decision: whether a feature should be implemented as a native module in Kotlin or Swift or remain in the shared JavaScript layer. In practice this decision is rarely made systematically. It is shaped by individual familiarity, delivery pressure and precedent, and over several years it produces fragmented native layers that are difficult to maintain, audit or hand over. This study proposes a governance framework for making the native/shared decision consistently. The framework combines four criteria (performance sensitivity, security and compliance requirements, platform capability access, and long-term maintainability and team capability) into a 0–12 scoring model with three decision bands and an explicit security override. It is accompanied by a three-layer reference architecture and a lightweight governance process built around existing architecture reviews, a native module register, a periodic review cadence and deliberate deprecation. The framework was developed through a design-oriented, practitioner-informed methodology drawing on more than fifteen years of enterprise software delivery in regulated financial services, and it is demonstrated through three illustrative cases and a weighting sensitivity analysis. The analysis shows that clear-cut cases are stable under alternative weightings, while borderline cases are sensitive to organisational context, which is precisely where structured review adds most value. Limitations, including subjective scoring, possible misuse of the security override and the absence of empirical validation, are discussed explicitly.

Keywords:

React Native, Native Modules, Mobile Architecture, Architecture Governance, Cross-Platform Development, Architecture Decision Records

INTRODUCTION

React Native has earned its place as a serious option for enterprise mobile delivery. It is no longer the framework teams reach for only when budgets are tight; regulated organisations in banking, wealth management and payments run substantial parts of their customer-facing estates on it, often serving user bases of several hundred thousand. Cross-platform approaches of this kind promise a single codebase across iOS and Android, but research consistently notes that access to device-specific features and platform capabilities remains one of the central challenges of the approach (Biørn-Hansen et al., 2018). That maturity brings a question that is easy to defer early in a project and much harder to defer two years in: when should functionality be written as a native module in Kotlin or Swift, and when should it stay in the shared JavaScript layer?

In principle, the answer is simple: build native when native capability is needed and stay shared otherwise. In practice, the decision is made by whoever picks up the ticket, under whatever time pressure the sprint imposes, informed by whatever that individual built last. A developer with an iOS background reaches for a native module out of familiarity; another, more comfortable in JavaScript, avoids one past the point where it is sensible to do so. Neither is wrong, but neither applies a consistent standard, and a codebase built this way over several years becomes a patchwork that new joiners find difficult to reason about.

Problem statement. The native/shared boundary in React Native is an architectural decision with long-lived consequences, yet it is usually taken implicitly, at implementation time, and without durable rationale. Architecture research has long recognized that undocumented design decisions erode over time, a phenomenon described as architectural knowledge vaporization (Jansen & Bosch, 2005), and native modules are a particularly acute instance because they are compiled into the binary, duplicated across two platforms and frequently understood by only one or two engineers.

Research objectives. The study pursues five objectives: (i) to characterize the costs of drawing the native/shared boundary inconsistently; (ii) to identify why organisations struggle to make the decision systematically; (iii) to define decision criteria and a scoring model that can be applied quickly within existing review processes; (iv) to

propose a reference architecture and governance process that keep native code narrow, owned and reversible; and (v) to assess critically the limitations of the framework and of the study itself.

Significance and scope. The paper is aimed at engineering leads, mobile architects and platform owners in organisations with more than one React Native team, where inconsistency between teams has already caused friction through duplicated native modules, disagreements in architecture review or native code that few people understand. Smaller teams may find the full governance process heavier than they need, although the four decision criteria are useful at any scale. The paper is not a tutorial on writing native modules, nor a comparison of React Native with Flutter or native-first development; in most organisations with a production React Native estate that debate is effectively closed, settled as much by the existing codebase as by fresh technical comparison. It also does not catalogue native APIs, which change too quickly for a printed list to remain useful; it instead offers a way of reasoning about the decision that should remain valid as specific APIs evolve.

The remainder of the paper is organised as follows. The literature review covers cross-platform development, architecture decision-making, technical debt and mobile security standards. The next section consolidates the research gap and the reasons organisations struggle to decide systematically. The methodology section describes the design-oriented approach. Results and discussion present the decision criteria, scoring model, reference architecture, governance process, illustrative cases, sensitivity analysis and limitations, followed by future work and conclusions.

LITERATURE REVIEW

Cross-Platform Mobile Development and Native Access

Biørn-Hansen et al. (2018) survey cross-platform mobile development and propose a taxonomy of approaches, including hybrid, interpreted, cross-compiled and model-driven frameworks, identifying access to native device features and performance as recurring research challenges. Empirical work has since measured the performance overhead of several cross-platform frameworks against native implementations and found that the overhead is real but strongly context-dependent, varying with the framework and the type of device feature being accessed (Biørn-Hansen et al., 2020). Nagappan and Shihab (2016), reviewing future directions in mobile software engineering research, similarly highlight platform fragmentation and cross-platform development as persistent concerns. These studies explain why the native/shared boundary matters, but they compare frameworks rather than guide decisions within a framework that has already been chosen.

The React Native Architecture

React Native's documentation describes the mechanics of native modules and the newer architecture, in which the JavaScript Interface (JSI), TurboModules and the Fabric renderer reduce the serialisation overhead associated with the original asynchronous bridge (Meta Platforms, 2024a, 2024b). Apple and Google document the platform capabilities a module might need, including the Secure Enclave, biometric authentication and the Android Keystore (Apple Inc., 2024; Google LLC, 2024). None of these sources, however, addresses when a native module is justified; that judgement is left entirely to the implementing team.

Architecture Decisions and Knowledge Management

A substantial body of work treats software architecture as a set of design decisions rather than only as structure (Jansen & Bosch, 2005). Tyree and Akerman (2005) proposed decision templates that capture the issue, alternatives, rationale and implications of each decision, and Nygard (2011) popularised lightweight Architecture Decision Records (ADRs) that teams write alongside code. A systematic review of architecture knowledge management approaches found that capturing and sharing decision rationale is widely supported in principle but unevenly practised (Weinreich & Groher, 2016). Quality-attribute-driven design methods argue for separating concerns along criteria such as performance and security (Bass et al., 2021), and layering is a long-established way of isolating platform-specific concerns (Fowler, 2002). Yet none of this literature provides a decision rubric specific to the React Native native/shared boundary.

Multi-Criteria Decision-Making

Structured multi-criteria methods such as the Analytic Hierarchy Process derive weighted priorities from pairwise comparisons of criteria (Saaty, 2008). Such methods offer rigour but impose overhead that is difficult to justify for routine feature-level decisions in a sprint-based delivery model. The scoring model proposed here deliberately sits at the lightweight end of this spectrum: an additive, equally weighted rubric whose primary purpose is to structure discussion and produce a record, with weighting treated as an organisational adjustment rather than a calculated output.

Technical Debt and Knowledge Concentration

Technical debt describes design or implementation shortcuts that are expedient in the short term but make future change more costly (Avgeriou et al., 2016; Kruchten et al., 2012). Kruchten et al. (2019) emphasise that much technical debt is architectural and is incurred through decisions whose consequences surface only later. Unjustified or unowned native modules are a clear example: they double the code to be maintained, depend on platform release cycles and concentrate knowledge. Research on truck factors, the number of developers whose departure would stall a project, found that a large share of the popular open-source systems studied depended on very few key developers (Avelino et al., 2016), a pattern that native module ownership in enterprise teams tends to reproduce.

Mobile Security and Regulatory Context

Mobile security standards set expectations that directly affect the native/shared decision. The OWASP Mobile Application Security Verification Standard (MASVS) and Testing Guide define requirements for secure storage, cryptography and authentication (OWASP Foundation, 2024), and NIST guidance addresses both mobile application vetting and authentication assurance, including the conditions under which biometrics may be used (Grassi et al., 2017; Ogata et al., 2019). Payment-specific guidance from the PCI Security Standards Council constrains how mobile payment acceptance is implemented (PCI Security Standards Council, 2017), and UK financial services firms must manage technology risk within the operational resilience framework set by the Financial Conduct Authority (2021). Requirements of this kind often depend on platform security boundaries that a JavaScript layer cannot credibly replicate. Table 1 summarises the literature most relevant to this study.

Source	Focus and contribution	Gap addressed by this study
Biørn-Hansen et al. (2018, 2020)	Taxonomy of cross-platform approaches; measured performance overhead	Compares frameworks; no guidance within a chosen framework
Meta Platforms (2024a, 2024b)	Mechanics of native modules, JSI and the new architecture	Explains how to build modules, not when they are justified
Jansen & Bosch (2005); Tyree & Akerman (2005)	Architecture as decisions; decision templates	Generic; no criteria for the native/shared boundary
Nygard (2011); Weinreich & Groher (2016)	Lightweight ADRs; architecture knowledge management	Record format left to the author; quality varies
Saaty (2008)	Weighted multi-criteria decision-making (AHP)	Too heavy for routine feature-level decisions
Kruchten et al. (2012, 2019); Avgeriou et al. (2016)	Technical and architectural debt	Describes the cost; no preventive rubric for native code
Avelino et al. (2016)	Truck factor and knowledge concentration	Highlights ownership risk without a governance mechanism
OWASP (2024); NIST; PCI SSC; FCA	Security, payment and resilience requirements	Define obligations, not architectural placement

Table 1: Summary of key literature and its relevance to this study

RESEARCH GAP AND THE COST OF INCONSISTENCY**The Cost of Getting the Decision Wrong**

Too much native code. Teams that lean heavily on native modules duplicate logic across iOS and Android that could have lived once in JavaScript. Every duplicated piece of business logic is a place where the two platforms can drift apart; a validation rule updated on Android and forgotten on iOS is a common defect that is hard to catch because it appears only when the platforms are compared side by side. Over-reliance on native code also concentrates knowledge. In the author's experience, though anecdotal rather than data-backed, a Kotlin module written by a single engineer became a release blocker on more than one project as soon as that person was unavailable. Native modules also take longer to build, review and test, and usually need to be written twice.

Too little native code. Teams that avoid native code on principle either miss platform capability or reimplement it poorly in JavaScript. Biometric authentication is the clearest example: it can be approximated with a library that wraps a weaker capability, but the security guarantees differ from a proper integration with the platform keystore or Secure Enclave (Apple Inc., 2024; Google LLC, 2024). In a regulated context that gap is not cosmetic; it is precisely what mobile security testing standards are designed to detect (OWASP Foundation, 2024). There is also

a perceived-quality cost: users notice stuttering scrolls, dropped animation frames and slow rendering of large charts, and some of these problems genuinely require native rendering.

Inconsistency itself. Most organisations suffer less from too much or too little native code than from inconsistency: several teams solving broadly the same integration problem in different ways, none clearly wrong, none compatible with one another, and no shared vocabulary for discussing the trade-offs. This is the gap the framework addresses.

Why Organisations Struggle to Decide Systematically

Four patterns recur across organisations of different sizes and sectors. They are drawn from repeated observation rather than a formal survey and should be read as common patterns rather than universal ones. **Precedent substitutes for principle:** once a native module exists for one purpose, it becomes the template for the next, whether or not the justification transfers. **The decision is made too late:** by the time a JavaScript implementation is found to be slow or incapable, the native rewrite competes with a deadline never designed to accommodate it. **No one owns the boundary:** architecture review tends to focus on service boundaries and integrations rather than on whether a feature belongs in the shared or native layer, so the decision defaults to whoever is closest to the code. **Rationale is not written down:** eighteen months later, nobody can say whether a module was built natively for a good reason or because it was fashionable at the time, which is the knowledge vaporisation problem described by Jansen and Bosch (2005).

Consolidated Research Gap

Taken together, the literature and practice point to a specific gap. Cross-platform research explains why native access matters but stops at framework comparison; architecture decision research offers general recording formats but no criteria for this boundary; and security standards define obligations without saying where in the architecture they should be met. What is missing is a lightweight, repeatable rubric for the React Native native/shared decision that produces a decision record as a by-product and is embedded in a governance process that keeps native code owned and reviewable. Table 2 sets out the gaps and the corresponding framework response.

Gap	Evidence and root cause	Framework response
G1: No decision criteria for the boundary	Documentation covers mechanics, not justification	Four explicit criteria applied to every candidate module
G2: Inconsistent decision records	ADR quality depends on the individual author	Scoring exercise produces a uniform record
G3: Late decisions	Native need discovered under deadline pressure	Scoring embedded in pre-build design review
G4: Unowned native code	Knowledge concentration and truck-factor risk	Named owner per module in a module register
G5: Modules outlive their justification	No review once a module ships	Standing review cadence and deliberate deprecation
G6: Security used as a blanket justification	Override invoked without challenge	Security-only approvals re-examined at each review

Table 2: Research gap matrix and the corresponding framework response

METHODOLOGY

Research Design

The study follows a qualitative, design-oriented approach in which the primary output is an artefact, a decision and governance framework, rather than a statistical test of a hypothesis (Hevner et al., 2004; Wieringa, 2014). The design proceeded through five iterative activities shown in Figure 1: problem identification from practice and literature; definition of objectives and gaps; design of the criteria, scoring model, reference architecture and governance process; demonstration through illustrative cases; and evaluation through sensitivity analysis and critical assessment of limitations.

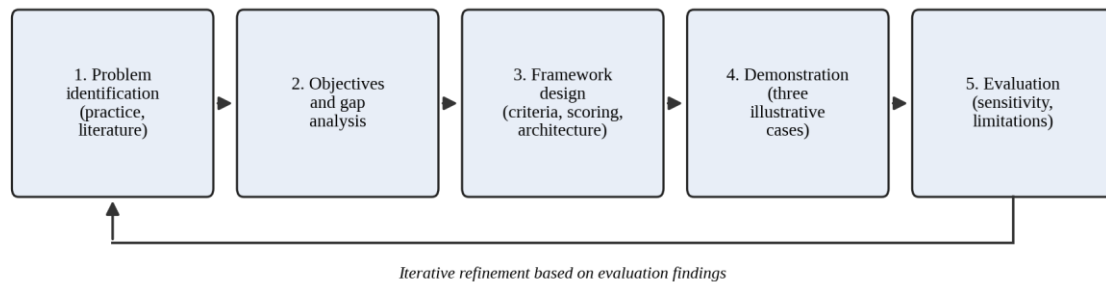


Figure 1: Research design adopted in this study

Data Sources

Two categories of evidence were used. The first is published literature and standards: cross-platform mobile research, architecture decision and knowledge management research, technical debt literature, platform documentation from Meta, Apple and Google, and security and regulatory guidance from OWASP, NIST, the PCI Security Standards Council and the Financial Conduct Authority. The second is structured practitioner reflection on more than fifteen years of software delivery across banking, wealth management, payments and enterprise client engagements, including leading React Native development for an application serving more than 500,000 active users, building native modules in Kotlin and Swift for biometric authentication, push notifications and payment SDKs, and setting coding standards and architecture guidelines for multi-team mobile estates.

Framework Construction

The criteria were derived by abstracting the reasons that recurred whenever native modules were proposed, challenged or later regretted in practice, and cross-checking them against the quality attributes emphasised in the architecture literature (Bass et al., 2021) and the obligations set by security standards. Four criteria survived this process because each corresponds to a distinct and recurring justification for, or cost of, native code. A 0–3 ordinal scale was chosen for each criterion to keep scoring quick and to avoid false precision, producing a total of 0–12 that is divided into three decision bands. The reference architecture and governance process were designed to attach to activities that mature engineering organisations already perform, so that adoption adds minimal process overhead.

Demonstration and Evaluation

The framework is demonstrated through three illustrative cases drawn, in generalised form, from regulated financial services work: a clear native case, a clear shared case and a borderline case. Scores are shown in full so the arithmetic can be checked. Because no controlled before-and-after data were available, the framework is evaluated analytically rather than empirically: through a weighting sensitivity analysis that tests whether case outcomes are stable under alternative organisational priorities, and through an explicit assessment of the limitations of both the framework and the study, consistent with guidance that case-based software engineering research should state the limits of its validity explicitly (Runeson & Höst, 2009).

Case Selection

The three cases were selected purposively to span the decision space rather than to be statistically representative: one case expected to fall clearly in the native band, one clearly in the shared band and one near the boundary between them. This mirrors the logic of selecting cases that test the boundaries of an artefact rather than its typical use (Runeson & Höst, 2009). Each case was re-scored retrospectively against the final rubric, and the recorded decision was compared with the decision actually taken, so that any disagreement between the framework and practice would be visible.

Ethical Considerations

The study was conducted independently of any client engagement. Illustrative cases are generalised from patterns observed across several organisations, and details that could identify a particular client's architecture, codebase or internal processes have been altered or omitted.

RESULTS AND DISCUSSION

Four Criteria for the Native/Shared Decision

The first result is a set of four decision criteria. None is decisive on its own, because a feature may score highly on one and low on the others, which is why they are combined into a single scoring model rather than treated as independent vetoes.

Performance sensitivity. Some interactions are frame-rate critical: continuous gestures, complex animations, large lists with rich item rendering, or work on the UI thread where a dropped frame is immediately visible. Even with JSI reducing serialisation overhead (Meta Platforms, 2024a), crossing between JavaScript and native code still matters for a narrow set of interactions. The useful question is not whether something could run faster natively, since almost everything could, but whether users will notice the difference and whether it matters for the product. A donut chart rendered once on an asset allocation screen has a very different profile from a live price ticker updating several times a second.

Security and compliance requirements. This is usually the strongest and least arguable justification. Biometric authentication, Secure Enclave or keystore access, certificate pinning below the JavaScript layer and payment SDK integrations mandated by a card scheme or regulator are all cases where the platform's own security boundary does work a JavaScript library cannot safely replicate (Apple Inc., 2024; Google LLC, 2024; PCI Security Standards Council, 2017). In banking and wealth management this criterion often overrides the others, because the cost of error is measured in regulatory exposure rather than engineering hours (Financial Conduct Authority, 2021).

Platform capability access. Some functionality is not exposed to JavaScript, or is exposed unreliably across OS versions: deep push notification handling with custom actions, background task scheduling, certain camera and file system operations, and partner SDKs shipped only as native libraries. The test is largely mechanical: has a reasonable search of React Native core and the community ecosystem failed to find a maintained library that does this reliably? If so, a native module is often unavoidable, and the discussion shifts from whether to build one to how narrowly to scope it.

Long-term maintainability and team capability. This criterion is most often skipped and causes the most pain two or three years later. A native module commits the team to two additional codebases that must be updated as OS versions change, APIs are deprecated and store review requirements tighten. A performance or security case for native code can still be the wrong decision if the organisation has no realistic path to maintaining it, which is how unowned modules become architectural debt (Kruchten et al., 2019).

The Scoring Model

Each criterion is scored from 0 to 3 according to how strongly it points towards native implementation, and the scores are summed (Table 3). The rubric is deliberately simple: applied to a feature that has already been discussed, it adds only a few minutes, while genuinely contested cases take longer, which is the purpose of the borderline band.

Criterion	0 — Native not indicated	3 — Native strongly indicated
Performance sensitivity	Static or infrequent rendering; no continuous interaction	Continuous gesture handling or animation on the critical UI thread
Security / compliance	No sensitive data or platform security boundary involved	Requires keystore, Secure Enclave or a regulator-mandated SDK
Platform capability access	Fully supported by a maintained JavaScript or community library	No reliable JavaScript-level access to the required capability
Maintainability / team capability	No Kotlin/Swift ownership exists or is planned	Dedicated native ownership already exists on the team

Table 3: Scoring rubric for the native/shared decision

Totals are read in three bands. A score of **0–3** indicates building in the shared JavaScript layer, where a native module is unlikely to justify its maintenance commitment. A score of **4–7** is borderline and warrants a short architecture review discussion rather than a unilateral decision; this is where documenting the rationale matters most. A score of **8–12** indicates that a native module is very likely the right decision. The numbers are not meant to be precise; their purpose is to ensure that the same four questions are asked every time and to leave a record of why the decision was made, which is often more valuable eighteen months later than the decision itself.

The security override. A strong security or compliance requirement is treated as close to a standalone justification for native code, even when the other criteria score low. In the author's experience across banking and wealth management platforms, no case has been observed in which a genuine platform security requirement was reasonably met by staying in the JavaScript layer. Teams may therefore skip the arithmetic when security alone is a clear 3, but this override is also the part of the framework most exposed to misuse, as discussed below. Figure 2 summarises the resulting decision flow.

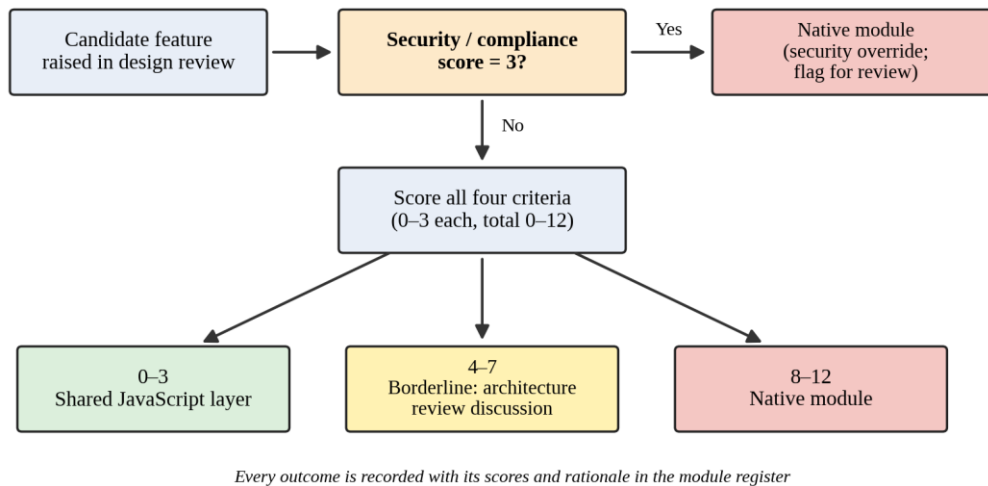


Figure 2: Decision flow combining the security override and the scoring bands

Reference Architecture

A scoring model determines what to build; it does not determine how native code should sit alongside the shared layer so that the decision remains reversible, auditable and understandable to newcomers. The reference architecture in Figure 3 and Table 4 separates three layers, consistent with established layering practice (Fowler, 2002).

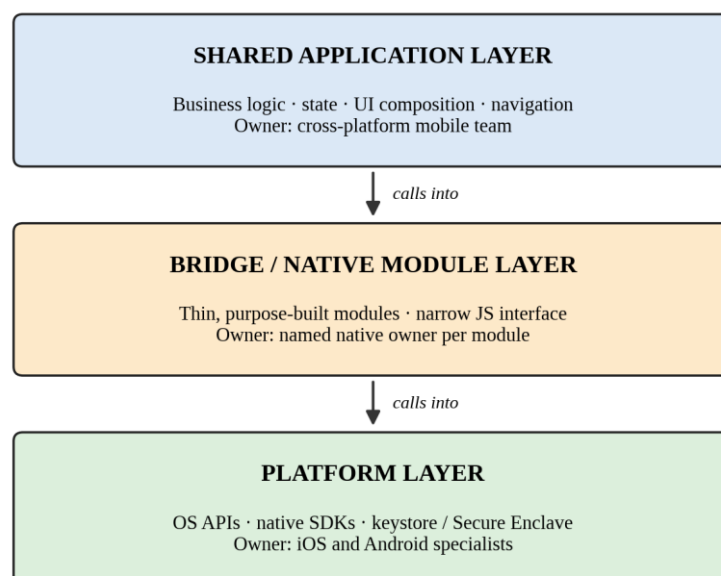


Figure 3: Three-layer reference architecture for native modules

Layer	Contents	Ownership
Shared application layer	Business logic, state management, UI composition and navigation: everything that does not require platform-specific capability	Cross-platform mobile team, shared ownership
Bridge / native module layer	Thin, purpose-built native modules exposing a narrow, documented JavaScript interface limited to the capability that justified the module	Named native owner per module, reviewed at architecture guild
Platform layer	OS APIs, third-party native SDKs and platform security primitives such as the keystore and Secure Enclave	Platform specialists for iOS and Android respectively

Table 4: Layers, contents and ownership in the reference architecture

Four principles govern the architecture. **Keep the native surface area narrow:** a module built for biometric approval should expose approve and reject operations, not become a general container for anything security-related. **Give every module a named owner and a written justification:** a single paragraph recording which criteria justified the module, written when it is built, prevents later confusion. **Version the bridge contract deliberately:** because native modules ship in the binary and cannot be hot-fixed like JavaScript, the JavaScript-to-native interface should be treated as a versioned contract. **Prefer community-maintained libraries where the gap is small:** accepting a slightly imperfect fit is often better than taking on the long-term cost of a bespoke module, a judgement the maintainability criterion is designed to support.

Governance in Practice

A framework that lives only in a document is not governance. The process is therefore attached to activities that already occur. **Where the decision is made:** the scoring model becomes one section of the existing design review template, with four scores, a total and two or three sentences of justification, the kind of record ADRs were designed to capture (Nygard, 2011). **A lightweight module register:** a simple register, for which a spreadsheet is sufficient, records each module's name, justifying criteria, named owner and last review date, answering the recurring question of why a module exists. **A standing review cadence:** reviewing the register once or twice a year at an architecture guild catches modules that have outlived their justification because React Native core or a community library has caught up, with particular attention to modules approved on the security override alone. **Deprecation as part of governance:** the same process should make it straightforward to retire modules, since a process that only ever adds modules is not governing anything.

Illustrative Cases

The following cases are drawn, in generalised form, from regulated financial services work; specific client and system details have been omitted or altered. Scores are summarised in Table 5.

Case A: biometric payment approval. A wealth management application needed customers to approve payment requests from a push notification using device biometrics. Performance sensitivity scored 0, since this is an infrequent, deliberate action. Security and compliance scored 3, because the approval token had to be signed using secure hardware. Platform capability access scored 3, since reliable biometric-backed signing is not safely exposed in the shared layer. Maintainability scored 2: the team had Kotlin and Swift capability but no named owner yet. The total of 8 falls in the native band, and the security override applies independently. The result was a narrowly scoped native module for signing and approving a transaction token, with a named owner assigned on each platform at build time.

Case B: cross-platform design token and theming system. A token system of primitives, semantic roles and component-level tokens for light and dark themes was built to unify a product line that had become inconsistent across acquisitions. There is no security boundary, no capability gap and negligible performance sensitivity, and no native ownership was needed. All four criteria scored 0, and the work was correctly built entirely in the shared layer. The case is a useful counter-example because a team without a framework might have reached for native theming APIs out of habit, the precedent-over-principle pattern described earlier.

Case C: chart rendering for portfolio data. A summary screen rendering asset allocation as a donut chart, updated on load and occasionally on interaction, is genuinely borderline. Performance sensitivity scored 2, as shared-layer chart libraries perform adequately for typical data volumes but can show jank with very large datasets. Security scored 0; platform capability access scored 1, because mature libraries exist with occasional gaps at scale; maintainability scored 1, as a bespoke module would need updating with every design change and no ownership was planned. The total of 4 sits at the low end of the borderline band. The decision favoured the

shared layer, with a design review note that native rendering should be revisited only if a specific performance complaint arises. Landing in the borderline band is itself informative, because it flags the decision for revisiting if assumptions change.

Case	Perf.	Security	Capability	Maint.	Total / band
A: Biometric payment approval	0	3	3	2	8 — Native (override applies)
B: Design token and theming system	0	0	0	0	0 — Shared
C: Portfolio chart rendering	2	0	1	1	4 — Borderline

Table 5: Scores for the three illustrative cases

Weighting Sensitivity Analysis

Because organisations differ in regulatory context and risk appetite, the equal weighting of the baseline model is unlikely to suit every setting. To test how robust the case outcomes are, the three cases were re-scored under two illustrative weighting profiles, with totals rescaled to the 0–12 range so that the same bands apply. The regulated profile weights security at 1.5 and the other criteria at 1.0, reflecting banking and wealth management priorities. The consumer profile weights security at 0.5 and platform capability at 1.5, reflecting an application in which a compromised session is an inconvenience rather than a regulatory event. Results are shown in Table 6.

Case	Baseline (equal)	Regulated profile	Consumer profile
A: Biometric payment approval	8.0 — Native	8.4 — Native	8.0 — Native
B: Design token and theming system	0.0 — Shared	0.0 — Shared	0.0 — Shared
C: Portfolio chart rendering	4.0 — Borderline	3.6 — Shared	4.5 — Borderline

Table 6: Case outcomes under alternative weighting profiles (rescaled to 0–12)

Two observations follow. First, the clear-cut cases are stable: Case A remains native and Case B remains shared under every profile, so the framework's conclusions for these cases do not depend on the weighting chosen. Second, the borderline case is sensitive: under the regulated profile, Case C moves into the shared band, while under the consumer profile it remains borderline. This is the intended behaviour. Borderline outcomes are precisely those where organisational context should shape the decision, and the sensitivity confirms that such cases belong in a review conversation rather than being settled by arithmetic alone. The analysis is illustrative rather than empirical, but it shows that organisations can adjust weights without destabilising the decisions the framework is most confident about.

Common Pitfalls

Four pitfalls recur in adoption. **Building native because it is familiar:** an engineer with a strong native background will gravitate towards native solutions, and the framework's job is to make the justification explicit rather than assumed. **Treating the framework as a one-off gate:** requirements change and libraries mature, so a module justified in 2024 may not be justified in 2026. **Letting modules grow beyond their original scope:** it is often easier to add one more method to an existing bridge than to justify a new one, which erodes the narrow-surface principle. **No handover plan:** naming an owner is useful only if ownership is actively transferred when that person changes role or leaves, rather than quietly lapsing.

Limitations of the Framework

Scoring is not fully objective. The 0–3 scale reduces judgement to a number, which is useful for structuring discussion but should not be mistaken for precision. Two engineers can reasonably disagree whether an interaction is a 1 or a 2, and the model offers no tie-breaking mechanism beyond discussion; the borderline band mitigates this by routing disagreement into review.

The security override can be misused. An engineer who wants to build native for reasons of familiarity has an incentive to inflate the security score to invoke the override and avoid scrutiny. The mitigation is procedural rather than mathematical: the standing review should specifically re-examine modules approved on security grounds alone.

The governance process has its own maintenance cost. The register and review cadence are lightweight compared with unmanaged native sprawl, but someone must own updating the register, chairing the review and

chasing deprecated entries. If that ownership lapses, the register becomes stale and untrustworthy, so the governance process itself needs a named owner.

The framework assumes an existing architecture review function. Teams without a design review process, common in smaller organisations and early-stage products, must build one first, and this paper offers no guidance on doing so. In that situation the four criteria remain useful as an individual checklist, but the governance layer will not transfer without something to attach to. **The additive model ignores interactions between criteria.** A high performance score combined with no native ownership may be riskier than the sum suggests, and the linear rubric cannot represent such interactions.

Limitations of the Study

The study itself has limitations that bound the strength of its conclusions, summarised in Table 7.

Limitation	Implication	Mitigation / further work
No empirical evaluation	Effect on defect rates, delivery time or maintenance cost is not measured	Claims framed as practitioner guidance; multi-team field study proposed
Anecdotal evidence base	Observed patterns may not generalise	Patterns stated as common rather than universal; grounded in published literature
Single-practitioner perspective	Criteria may reflect one practitioner's experience	Expert panel and inter-rater reliability testing proposed
Generalised illustrative cases	Cases illustrate but cannot verify the framework	Full scores shown so reasoning can be checked
Sector focus on regulated finance	Weighting may not suit other domains	Sensitivity analysis shows how weights can be adapted
Illustrative weighting profiles	Weights are assumed, not elicited	Future elicitation of weights using structured methods such as AHP

Table 7: Limitations of the study and mitigation

In particular, the absence of quantitative data means the framework's benefits, such as fewer duplicated modules, clearer ownership and faster decisions, are argued from experience and literature rather than observed under controlled conditions. Platform documentation and regulatory guidance also change faster than any static reference can track, so the specific examples cited should be treated as a starting point for verification rather than a substitute for it.

FUTURE WORK

Several directions follow from these limitations. The most important is a field evaluation in which multiple React Native teams adopt the framework and track measures such as the number of native modules created and retired, time spent on native maintenance, cross-platform defect rates and decision lead time, compared with a baseline period. A second is validation of the criteria and scoring scale through an expert panel, with inter-rater reliability testing to determine how consistently practitioners score the same features. A third is structured elicitation of sector-specific weights, for example using the Analytic Hierarchy Process (Saaty, 2008), to replace the illustrative profiles used here.

Further work could also extend the framework as React Native continues to evolve. As the new architecture matures and more capabilities move into the core framework and community libraries, the platform capability criterion should shift, and the review cadence provides a mechanism for capturing that shift. Lightweight tooling that generates the decision record and register entry directly from design review templates, and that flags modules whose justification may have lapsed, would further reduce the overhead of governance.

CONCLUSION

The native-versus-shared question in React Native is not answered once and forgotten. Platform capabilities expand, community libraries mature and organisational priorities shift, all of which move the boundary over time. What organisations can control is not the position of that boundary but the consistency with which it is drawn.

This paper has proposed a framework for achieving that consistency: four criteria, a simple scoring model with an explicit security override, a narrow three-layer reference architecture and a governance process light enough to be followed. The illustrative cases and sensitivity analysis show that the framework yields stable conclusions

for clear-cut decisions while surfacing borderline ones for review. Its limitations are equally clear: scoring is subjective, the security override can be misused, the governance process carries its own cost, and the framework has not yet been evaluated empirically. It is therefore offered as a starting point rather than a finished standard, and organisations should adjust its weighting to their own regulatory context and risk appetite.

Teams do not need the full governance process to benefit. The fastest start is to apply the four criteria to the next feature in an upcoming design review and compare the result with what the team would have decided anyway; the register and review cadence are worth adding once the scoring habit has proved useful. What should not change is the underlying discipline: ask the same four questions every time, write the answer down and revisit it periodically.

NOMENCLATURE

Abbreviation	Meaning
ADR	Architecture Decision Record; a short document capturing a single architectural decision and its rationale
AHP	Analytic Hierarchy Process; a weighted multi-criteria decision-making method
FCA	Financial Conduct Authority, the UK financial services regulator
JSI	JavaScript Interface; React Native's mechanism for direct calls between JavaScript and native code
MASVS / MASTG	OWASP Mobile Application Security Verification Standard / Mobile Application Security Testing Guide
OS	Operating system (iOS or Android)
PCI SSC	Payment Card Industry Security Standards Council
SDK	Software Development Kit
TurboModules / Fabric	Native module system and rendering system of React Native's new architecture
UI	User interface

Table 8: Abbreviations used in this paper

CONFLICT OF INTEREST

The author declares no conflict of interest.

FUNDING

This research received no specific grant from any funding agency in the public, commercial or not-for-profit sectors.

DATA AVAILABILITY STATEMENT

No new datasets were generated in this study. The illustrative cases are generalised from professional practice and contain no client-identifiable data.

REFERENCES

- 1) Apple Inc. (2024). *Apple platform security guide*. <https://support.apple.com/guide/security/welcome/web>
- 2) Avelino, G., Passos, L., Hora, A., & Valente, M. T. (2016). A novel approach for estimating truck factors. In *Proceedings of the 24th IEEE International Conference on Program Comprehension (ICPC)* (pp. 1–10). IEEE. <https://doi.org/10.1109/ICPC.2016.7503718>
- 3) Avgeriou, P., Kruchten, P., Ozkaya, I., & Seaman, C. (2016). Managing technical debt in software engineering (Dagstuhl Seminar 16162). *Dagstuhl Reports*, 6(4), 110–138. <https://doi.org/10.4230/DagRep.6.4.110>
- 4) Bass, L., Clements, P., & Kazman, R. (2021). *Software architecture in practice* (4th ed.). Addison-Wesley.

- 5) Biørn-Hansen, A., Grønli, T.-M., & Ghinea, G. (2018). A survey and taxonomy of core concepts and research challenges in cross-platform mobile development. *ACM Computing Surveys*, 51(5), Article 108. <https://doi.org/10.1145/3241739>
- 6) Biørn-Hansen, A., Rieger, C., Grønli, T.-M., Majchrzak, T. A., & Ghinea, G. (2020). An empirical investigation of performance overhead in cross-platform mobile development frameworks. *Empirical Software Engineering*, 25, 2997–3040. <https://doi.org/10.1007/s10664-020-09827-6>
- 7) Financial Conduct Authority. (2021). *Building operational resilience* (Policy Statement PS21/3). <https://www.fca.org.uk/publications/policy-statements/ps21-3-building-operational-resilience>
- 8) Fowler, M. (2002). *Patterns of enterprise application architecture*. Addison-Wesley.
- 9) Google LLC. (2024). *Android keystore system*. Android Developers. <https://developer.android.com/privacy-and-security/keystore>
- 10) Grassi, P. A., Fenton, J. L., Newton, E. M., Perlner, R. A., Regenscheid, A. R., Burr, W. E., Richer, J. P., Lefkowitz, N. B., Danker, J. M., Choong, Y.-Y., Greene, K. K., & Theofanos, M. F. (2017). *Digital identity guidelines: Authentication and lifecycle management* (NIST Special Publication 800-63B). National Institute of Standards and Technology. <https://doi.org/10.6028/NIST.SP.800-63b>
- 11) Hevner, A. R., March, S. T., Park, J., & Ram, S. (2004). Design science in information systems research. *MIS Quarterly*, 28(1), 75–105. <https://doi.org/10.2307/25148625>
- 12) Jansen, A., & Bosch, J. (2005). Software architecture as a set of architectural design decisions. In *Proceedings of the 5th Working IEEE/IFIP Conference on Software Architecture (WICSA)* (pp. 109–120). IEEE. <https://doi.org/10.1109/WICSA.2005.61>
- 13) Kruchten, P., Nord, R. L., & Ozkaya, I. (2012). Technical debt: From metaphor to theory and practice. *IEEE Software*, 29(6), 18–21. <https://doi.org/10.1109/MS.2012.167>
- 14) Kruchten, P., Nord, R., & Ozkaya, I. (2019). *Managing technical debt: Reducing friction in software development*. Addison-Wesley.
- 15) Meta Platforms. (2024a). *About the new architecture*. React Native. <https://reactnative.dev/architecture/landing-page>
- 16) Meta Platforms. (2024b). *Native modules: Introduction*. React Native. <https://reactnative.dev/docs/native-modules-intro>
- 17) Nagappan, M., & Shihab, E. (2016). Future trends in software engineering research for mobile apps. In *Proceedings of the 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER)* (pp. 21–32). IEEE. <https://doi.org/10.1109/SANER.2016.88>
- 18) Nygard, M. (2011, November 15). *Documenting architecture decisions*. Cognitect. <https://cognitect.com/blog/2011/11/15/documenting-architecture-decisions>
- 19) Ogata, M., Franklin, J., Voas, J., Sritapan, V., & Quirolgico, S. (2019). *Vetting the security of mobile applications* (NIST Special Publication 800-163, Rev. 1). National Institute of Standards and Technology. <https://doi.org/10.6028/NIST.SP.800-163r1>
- 20) OWASP Foundation. (2024). *OWASP mobile application security verification standard (MASVS) and mobile application security testing guide (MASTG)*. <https://mas.owasp.org/>
- 21) PCI Security Standards Council. (2017). *Mobile payment acceptance security guidelines for merchants* (Version 2.0). <https://www.pcisecuritystandards.org/>
- 22) Runeson, P., & Höst, M. (2009). Guidelines for conducting and reporting case study research in software engineering. *Empirical Software Engineering*, 14(2), 131–164. <https://doi.org/10.1007/s10664-008-9102-8>
- 23) Saaty, T. L. (2008). Decision making with the analytic hierarchy process. *International Journal of Services Sciences*, 1(1), 83–98. <https://doi.org/10.1504/IJSSCI.2008.017590>
- 24) Tyree, J., & Akerman, A. (2005). Architecture decisions: Demystifying architecture. *IEEE Software*, 22(2), 19–27. <https://doi.org/10.1109/MS.2005.27>
- 25) Weinreich, R., & Groher, I. (2016). Software architecture knowledge management approaches and their support for knowledge management activities: A systematic literature review. *Information and Software Technology*, 80, 265–286. <https://doi.org/10.1016/j.infsof.2016.09.007>
- 26) Wieringa, R. J. (2014). *Design science methodology for information systems and software engineering*. Springer. <https://doi.org/10.1007/978-3-662-43839-8>