Research Square

Preprints are preliminary reports that have not undergone peer review.
They should not be considered conclusive, used to inform clinical practice,
or referenced by the media as validated information.

Migrating an Ecosystem: An Empirical Study of How React Native Libraries Adopted the New Architecture

Revanth Krishna Maddula

revanth802@gmail.com

Independent Researcher https://orcid.org/0009-0006-0596-6180

Research Article

Keywords: React Native; software ecosystems; library migration; mobile software engineering; mining software repositories

Posted Date: November 5th, 2026

DOI: https://doi.org/10.21203/rs.3.rs-11614118/v1

License: This work is licensed under a Creative Commons Attribution 4.0 International License.
Read Full License

Additional Declarations: The authors declare no competing interests.

Migrating an Ecosystem: An Empirical Study of How React Native Libraries Adopted the New Architecture

Revanth Krishna Maddula

Independent Researcher, San Jose, CA, USA

Abstract

Context: When a platform replaces its core architecture, every library built on it must migrate. React Native replaced its asynchronous bridge with a New Architecture in three steps: opt-in in version 0.68 (2022), default in 0.76 (2024), and mandatory in 0.82 (2025). **Objective:** We study how its library ecosystem responded: how far and how fast it migrated, by which routes, which libraries were left behind, and how much usage depends on them. **Method:** We analyze all 2,782 React Native Directory libraries matched to npm, combining full release histories, static scans of published packages, popularity data, and 70 monthly snapshots of declared compatibility. A blind second classification of 140 libraries agreed with ours (Cohen's $\kappa = 0.89$). **Results:** Of 1,444 native libraries, 62.0% have migrated and carry 95% of downloads; 35.9% still rely on interop layers. The monthly share of remaining libraries that migrated rose from 0.43% (optional) to 0.78% (default) and 1.55% (mandatory), although segmented regression could not attribute the increases to the milestones. Only an estimated 23% of libraries unmigrated at the default change migrated within two years. Unmigrated libraries are mostly old and dormant, fewer than one in ten documents its status, yet 115 still receive at least 10,000 weekly downloads. Of migrated libraries created in 2026, 62% use Expo Modules or Nitro rather than the official Codegen route. **Conclusions:** Milestones were followed by faster migration but did not reach the dormant tail; compatibility layers became long-term infrastructure, and the migration target fragmented.

Keywords: React Native, software ecosystems, library migration, mobile software engineering, mining software repositories, npm

Email address: revanth802@gmail.com (Revanth Krishna Maddula)

November 2, 2026

1. Introduction

Software platforms occasionally replace their foundations. Python moved from version 2 to version 3, Android replaced its Support Library with AndroidX, and web frameworks have repeatedly rewritten their rendering engines. Each transition is announced by the platform owner, but it is completed, or not, by thousands of independent library maintainers. Applications can adopt the new platform only when all of their dependencies have, so the pace of an ecosystem's slowest libraries sets the pace for everyone built on top of it [1, 2].

React Native, the cross-platform framework in which mobile applications are written in JavaScript and rendered with native user-interface components, is in the late stage of such a transition. Its original design passed every call between JavaScript and native code through an asynchronous, serialized "bridge". The New Architecture replaces the bridge with direct calls through the JavaScript Interface (JSI), a new native-module system (TurboModules), a new renderer (Fabric), and a code generator that derives typed native interfaces from JavaScript specifications [3]. The platform owner moved in three steps: the New Architecture became available as an opt-in in version 0.68 in March 2022 [4], became the default in version 0.76 in October 2024 [3], and became the only option in version 0.82 in October 2025, when the setting that switched it off began to be ignored [6]. To avoid breaking applications, the platform ships "interop layers" that let libraries written for the old architecture keep running under the new one; the team has said these layers will remain "for the foreseeable future" [6].

This staged design makes React Native a natural experiment in how an ecosystem responds to an architecture change, and its timeline separates three kinds of pressure: an invitation (opt-in), a new default, and a hard requirement. The ecosystem is also unusually observable. Libraries are distributed through npm, which records every release with its metadata, and the community-maintained React Native Directory catalogs the ecosystem's libraries along with maintainer-declared compatibility [7]. Yet, to our knowledge, no study has measured the migration. Practitioner accounts describe individual migrations, and the platform team has published headline figures, such as "more than 850 libraries" being compatible when the default changed [3], but the ecosystem-wide picture, including which libraries were

left behind and who depends on them, is unknown.

We address this gap with an empirical study of all 2,782 Directory libraries that we could match to an npm package, and the 1,444 among them that contain native code and therefore had to migrate. We ask four research questions:

- **RQ1 (Extent and routes):** How far has the ecosystem migrated, and by which technical routes?
- **RQ2 (Timing):** When did libraries migrate relative to the platform's milestones?
- **RQ3 (Stragglers):** What distinguishes libraries that have not migrated from those that have?
- **RQ4 (Exposure):** How much of the ecosystem's usage still depends on unmigrated libraries?

Our study makes three contributions. First, it provides the first ecosystem-wide measurement of a cross-platform mobile framework's architecture migration, combining release histories, static evidence from published packages, and maintainer declarations, with a validated classification. Second, it shows how the ecosystem responded to the platform's milestones: the monthly migration rate roughly doubled after the default change and doubled again after the old architecture was removed, although the increases are not statistically distinguishable from noise in a segmented regression, and most libraries that had not migrated by the default change had still not done so two years later, mostly because no one was maintaining them. Third, it identifies two developments that platform owners planning similar transitions should anticipate: compatibility layers that become a permanent home for a large tail of libraries, and a migration target that fragments as third-party frameworks offer alternative routes. We release our dataset and analysis scripts to support replication [22].

Section 2 explains the two architectures and the migration routes. Section 3 reviews related work. Section 4 describes the study design, and Section 5 reports the results. Section 6 discusses implications, Section 7 threats to validity, and Section 8 concludes.

2. Background

2.1. The legacy architecture and the New Architecture

In React Native's legacy architecture, application logic runs in a JavaScript engine and the user interface is drawn by native platform components. The two sides communicate through the bridge: calls are serialized, queued and executed asynchronously, so a JavaScript function cannot call native code and receive a result synchronously. Library authors expose native functionality by writing a native module, registered with platform-specific macros and base classes (for example `RCT_EXPORT_MODULE` on iOS and `ReactContextBaseJavaModule` on Android), or a native view manager, and JavaScript code reaches them through the `NativeModules` object or `requireNativeComponent`.

The New Architecture removes the bridge. JSI lets JavaScript hold references to native objects and call them directly and synchronously. Native modules become TurboModules, which are loaded lazily, and native views are rendered by Fabric, a C++ renderer that supports concurrent rendering. Both are typically produced through Codegen: the library author writes a typed specification in TypeScript or Flow, declares it in the `codegenConfig` field of the package manifest, and the build generates the native interfaces that the implementation must satisfy [3].

2.2. Interop layers and the migration timeline

Because rewriting every library at once was impossible, the platform introduced interop layers that let legacy native modules and view managers run inside an application that uses the New Architecture. The layers do not support every legacy feature (for example, custom shadow nodes and concurrent rendering), and they add a translation step, but they allow an application to enable the New Architecture before all of its dependencies have migrated [3]. Table 1 summarizes the milestones we use in this study.

2.3. Routes to the New Architecture

A library can support the New Architecture natively by three main routes. The *Codegen* route is the platform's own: TurboModule and Fabric specifications processed by Codegen. The *Expo Modules* route uses the Expo Modules API, a third-party framework in which modules are written in Swift and Kotlin against Expo's abstractions; modules built this way "all support the New Architecture" and remain backward compatible with the old one [8].

Table 1: React Native New Architecture milestones used in this study.

Version	Date	Change for applications and libraries
0.68	30 Mar 2022	New Architecture available as an opt-in [4]
0.76	23 Oct 2024	New Architecture enabled by default; opt-out possible; interop layers on [3]
0.80	12 Jun 2025	Legacy architecture frozen: no new fixes or features, warnings for legacy APIs [5]
0.82	8 Oct 2025	New Architecture only: the opt-out setting is ignored; interop layers retained [6]

The *Nitro Modules* route uses Nitro, a third-party framework whose modules are "backed by Hybrid Objects and statically compiled JSI bindings" generated from TypeScript definitions [9]. A library can also bypass these frameworks and expose functions through hand-written JSI code while still registering itself as a legacy module. Libraries that do none of these continue to run, if at all, through the interop layers.

3. Related Work

Ecosystem-wide transitions. The closest analogue to our study is the migration from Python 2 to Python 3. Malloy and Power analyzed the use of Python 3 features across a large corpus of applications and found that, years after Python 3's release, features exclusive to Python 3 were rarely used and most code remained compatible with Python 2 [1]. In mobile software, Li et al. studied the replacement of Android's Support Library by AndroidX and found that about 13% of commercial and 17% of open-source apps still used the old library, while developers struggled most with API usage, dependencies and build configuration [2]. McDonnell et al. showed that Android's APIs evolved faster than client applications adopted them [10]. These studies concern transitions in which the old platform remained usable; React Native's transition is unusual in that the old architecture was eventually switched off while a compatibility layer kept old libraries running.

Dependency updates and technical lag. A large literature studies how slowly clients follow their dependencies. Kula et al. found that most systems keep outdated dependencies and that developers are often unaware of updates [11]; Salza et al. found that mobile apps rarely update third-party libraries [12]; Zerouali et al. measured the "technical lag" of npm packages

behind the latest releases of their dependencies [13]; and Decan et al. compared the dependency networks of seven packaging ecosystems [14]. Work on API evolution has examined how clients react to deprecation [15, 16]. Our study inverts the usual perspective: rather than asking whether clients follow their libraries, we ask whether libraries follow their platform.

Breaking changes and abandonment. Bogart et al. showed that ecosystems differ in how they negotiate breaking changes and who bears their cost [17]. Avelino et al. studied why open-source projects are abandoned and how some survive [18], and Miller et al. found that abandonment of widely used npm packages is common, exposes many dependents, and often goes unaddressed [19]. An architecture migration turns dormancy into incompatibility: a library that has not changed in years can become a blocker without any change of its own.

Cross-platform mobile development. Research on cross-platform frameworks has surveyed their concepts and challenges [20], but we are not aware of empirical work on React Native's New Architecture or on how its library ecosystem has responded to it.

4. Study Design

4.1. Population and data sources

Population. Our population is the React Native Directory [7], a community-curated catalog maintained under the React Native Community organization that lists libraries with their supported platforms and, optionally, maintainer-declared New Architecture compatibility. We used the catalog file in the Directory's public repository as of 28 October 2026, which lists 2,799 libraries by their source repository URL. We chose the Directory rather than all npm packages that mention React Native because it is the ecosystem's de facto index, it excludes the many abandoned experiments and forks on npm, and it is the reference the platform team points developers to when checking compatibility [3].

Matching to npm. For each library we resolved its npm package: from the Directory's explicit package name when present (886 libraries), otherwise from the repository name, its scoped variant, or the package folder of a monorepo. We accepted a match when the repository in the package's latest manifest pointed to the same GitHub repository, to a repository of the same name under a different owner (a transferred or renamed repository), or when

the manifest named no repository and the package name equaled the repository name. This matched 2,782 libraries (99.4%); we excluded the remaining 17.

Release histories. From the npm registry we downloaded each package's full metadata document, which records every published version with its publication time and complete manifest. From the manifests we extracted, per version, whether a `codegenConfig` field is present and whether the package depends (as a dependency or peer dependency) on `expo-modules-core` or `react-native-nitro-modules`. We treated versions with pre-release tags (for example `-beta` or `-rc`) as unreleased.

Package contents. For the latest released version of each package, we downloaded the published archive from the registry and scanned it in memory, excluding example applications and bundled dependencies. We recorded the presence of native source files (Objective-C, Swift, Java, Kotlin, C/C++) and podspecs; configuration files of the Expo Modules and Nitro frameworks; and textual signals of each architecture: Codegen specifications (`TurboModuleRegistry.get` and `codegenNativeComponent`), legacy registration APIs (`RCT_EXPORT_MODULE`, `ReactContextBaseJavaModule`, view-manager base classes), legacy JavaScript access (`NativeModules`, `requireNativeComponent`), direct JSI code, and build-time architecture checks.

Popularity. We recorded each package's weekly and monthly downloads and its number of dependent packages as reported by the npm registry's search service on the snapshot date.

Declared compatibility over time. We cloned the Directory's repository and extracted the catalog file as it stood on the first day of each month from January 2021 to October 2026 (70 snapshots), counting libraries declared as supporting the New Architecture, declared unsupported, or declared "new-architecture only".

4.2. Classification

We first separate *native* libraries, whose published package contains native source files or a podspec (1,444), from *JavaScript-only* libraries (1,338), which run unchanged under either architecture and needed no migration. We then classify each native library by its route to the New Architecture, applying the rules in order:

1. **Expo Modules**: the package contains an Expo module configuration or depends on `expo-modules-core`.

2. **Nitro Modules**: the package depends on `react-native-nitro-modules`.
3. **Codegen**: the manifest declares `codegenConfig`, or the sources contain TurboModule or Fabric component specifications.
4. **Custom JSI on legacy registration**: none of the above, but the sources contain hand-written JSI code.
5. **Legacy bridge**: none of the above.

We call the first three *migrated*, the fourth *partial*, because these libraries bypass the bridge for calls but still register as legacy modules, and the fifth *not migrated*: such libraries run under the New Architecture only through the interop layers. Our classification therefore measures native adoption of the New Architecture, which is stricter than whether a library *works* in a New Architecture application.

Validation. To check the classifier, we drew a stratified random sample of ten native libraries from each of the six categories produced by an initial version of the rules (60 libraries), re-downloaded each package and inspected its file list, manifest and architecture-related code. All 30 libraries classified as migrated used the identified route (Codegen specifications declared in the manifest, Expo module definitions, or Nitro hybrid objects). All ten classified as legacy bridge used only legacy registration and access APIs; one was a code-generation tool that ships native template files rather than a runtime library. The inspection led to two refinements. A category of libraries containing only build-time architecture checks turned out, in all ten cases, to consist of legacy modules whose checks came from project templates, so we merged it into the legacy-bridge category. And the ten libraries with hand-written JSI all registered through legacy APIs, which motivated the separate *partial* category. We note that the legacy JavaScript-access signal misses calls with type arguments (such as `requireNativeComponent<T>(...)`); this affects only descriptive counts, not the classification, which depends on the presence of New Architecture signals.

To estimate agreement on a larger, independent sample, we drew a second stratified random sample of 140 libraries not used in the first check (30 from each route and 20 partial), shuffled it, and had it classified blind against a written codebook by a separate rater that inspected each published package without access to our labels. The rater was an AI agent (Claude, Anthropic) instructed to judge from package contents only; its labels therefore test the consistency of our rules against an independent reading of the code rather

than human judgment. The two classifications agreed on the route for 128 of 140 libraries (91.4%; Cohen's $\kappa = 0.89$) and on migrated versus not migrated for 137 (97.9%). The three migrated/unmigrated disagreements were all Nitro libraries that our rules had classified as partial because our dependency-based rule did not detect their use of Nitro, so our migration counts are, if anything, slightly conservative. The remaining disagreements concerned the route among migrated libraries (three libraries using Codegen alongside an Expo configuration plugin or a Nitro dependency, which our rule ordering assigned to Expo or Nitro), the boundary between partial and legacy (three libraries), and three packages that the rater judged to contain no runtime native code.

4.3. Dating migrations

For libraries on the Codegen and Nitro routes, we date the migration as the publication time of the first released version that declares `codegenConfig` or depends on Nitro, respectively. We distinguish *conversions*, in which an existing library migrated (its first migrated release came more than 31 days after the package was created), from libraries *born migrated*. We do not date Expo Modules migrations: many first-party Expo packages adopted `expo-modules-core` in 2021, years before the New Architecture became React Native's default, so the dependency date does not mark a New Architecture migration. Time-based analyses therefore exclude the 271 Expo Modules libraries.

4.4. Analysis

For RQ1 we report the distribution of routes, weighted both by library and by weekly downloads, and compare the code-level classification with the Directory's declarations. For RQ2 we compute monthly counts of conversions and born-migrated libraries, conversion rates within the intervals defined by the milestones, and a Kaplan–Meier estimate [21] of the time to migration for native libraries that existed when version 0.76 made the New Architecture the default and had not yet migrated; libraries that had not migrated by the snapshot date are right-censored. To test whether the milestones changed the migration rate beyond the underlying trend, we fit a segmented (interrupted time series) regression to the monthly number of conversions from April 2022 to October 2026, with the number of libraries still at risk of converting as an exposure offset, a linear time trend, and level and slope changes at 0.76 and 0.82; because the counts were overdispersed (Pearson dispersion 2.1),

Table 2: Routes to the New Architecture among the 1,444 native libraries (latest released version, 28 October 2026). Downloads: share of the native libraries' combined weekly downloads.

Status	Route	Libraries	%	Downloads
Migrated	Codegen (TurboModule/Fabric)	470	32.5	
	Expo Modules	271	18.8	
	Nitro Modules	154	10.7	
	Subtotal	*895*	*62.0*	*95.0%*
Partial	Custom JSI on legacy registration	30	2.1	0.1%
Not migrated	Legacy bridge (interop layers)	519	35.9	4.8%
	Total	1,444	100	

we used a negative binomial model, and we also fit a simpler model with level changes only. For RQ3 we compare migrated and unmigrated native libraries with Mann–Whitney U tests and fit a logistic regression for being unmigrated, with weekly downloads (log-transformed), whether the library published any release after 0.76, package age, whether it has more than one npm maintainer, and whether it ships both iOS and Android code as predictors. To learn what maintainers of unmigrated libraries say, we extracted every passage of their latest README that mentions the New Architecture or its components, compatibility or interop layers, deprecation, maintenance status or replacements, together with npm deprecation notices, and coded the statements by hand. For RQ4 we aggregate downloads and dependents of unmigrated libraries and examine the most downloaded ones.

5. Results

5.1. RQ1: How far has the ecosystem migrated, and by which routes?

Of the 2,782 libraries, 1,338 (48.1%) are JavaScript-only and needed no migration. Table 2 classifies the remaining 1,444 native libraries. Six in ten (895, 62.0%) have migrated, 30 (2.1%) are partial, and 519 (35.9%) have not migrated and run under the New Architecture only through the interop layers. The platform's own Codegen route is the most common (470 libraries), but the two third-party routes together account for nearly as many migrated libraries: Expo Modules (271) and Nitro Modules (154) make up 47.5% of all migrated libraries.

Table 3: Migration status of native libraries by weekly npm downloads.

Weekly downloads	Migrated	Not migrated	Partial	Total	% migrated
<100	310	209	20	539	57.5
100–999	149	98	3	250	59.6
1,000–9,999	114	97	2	213	53.5
10,000–99,999	141	74	3	218	64.7
≥100,000	181	41	2	224	80.8

Weighted by usage, the picture is very different. Migrated libraries receive 95.0% of the 423 million weekly downloads of native libraries; unmigrated ones receive 4.8%. Table 3 shows that migration is most complete among the most popular libraries: 80.8% of native libraries with at least 100,000 weekly downloads have migrated, against 53.5–64.7% in the lower popularity bands. The relationship is not monotonic below that level; libraries with fewer than 100 weekly downloads, many of them recent, migrate at a similar rate to those with thousands.

Declared versus implemented support. The Directory's compatibility declarations cover only half of the native libraries: 744 (51.5%) declare nothing. Of those undeclared libraries, 281 (37.8%) have in fact migrated, so the declarations understate adoption. Conversely, of the 687 native libraries declared as supporting the New Architecture (598) or as supporting only the New Architecture (89), 68 (9.9%) have not migrated: they are declared compatible because they work through the interop layers. A "supported" label therefore means "works in a New Architecture application", not "has adopted the New Architecture". None of the 89 "new-architecture only" libraries is unmigrated.

Migrated libraries often keep legacy code. Of the 895 migrated libraries, 435 (48.6%) still contain legacy registration APIs alongside their New Architecture implementation, typically so that the same release can serve applications on both architectures. Supporting both doubles part of the maintenance burden that the migration was meant to remove.

The routes are shifting. Figure 1 shows the routes of native libraries by the year their npm package was created. Among libraries created from 2015 to 2017, more than three in four have not migrated. Among libraries created in 2025 and 2026, fewer than one in ten use only the legacy bridge, but the platform's Codegen route is no longer the majority choice: of the

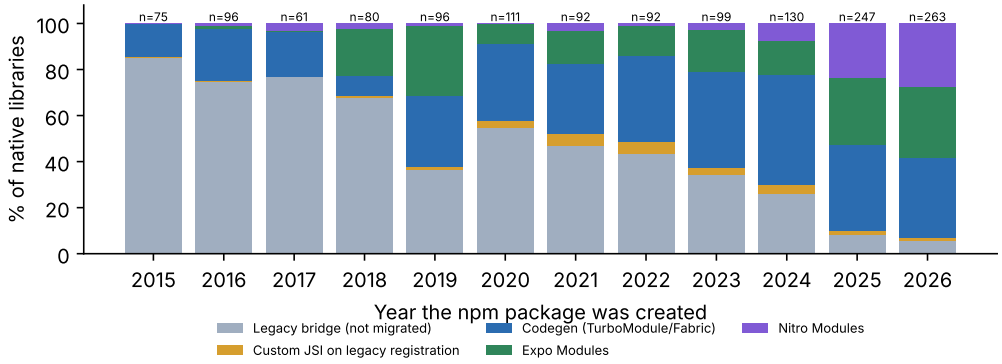


Figure 1: Route to the New Architecture by the year the library's npm package was created (native libraries, 2015–2026).

migrated libraries created in 2026, 62.4% use Expo Modules (80) or Nitro Modules (73) and 37.6% use Codegen (92). Nitro Modules, almost absent before 2024, account for 28% of native libraries created in 2026.

> **RQ1 summary.** 62.0% of native libraries have migrated, and they carry 95% of native-library downloads, but 519 libraries (35.9%) still depend on the interop layers. Directory declarations both understate migration and overstate it, because "supported" includes interop. New libraries increasingly migrate through third-party frameworks rather than the platform's own Codegen route.

5.2. RQ2: When did libraries migrate?

We could date 611 migrations on the Codegen and Nitro routes: 368 libraries were born migrated and 243 were conversions of existing libraries. Figure 2 shows them by month. Table 4 reports the conversion rate in each interval between milestones. Only 3 conversions preceded the opt-in release. While the New Architecture was optional (0.68 to 0.76, 31 months), 77 libraries converted, 2.5 per month. In the year in which it was the default but could still be switched off (0.76 to 0.82), the rate roughly doubled, to 4.9 per month. After 0.82 made it mandatory, the rate nearly doubled again, to 8.5 per month. New libraries born migrated rose over the same period and account for 66% of the 472 dated migrations since November 2024.

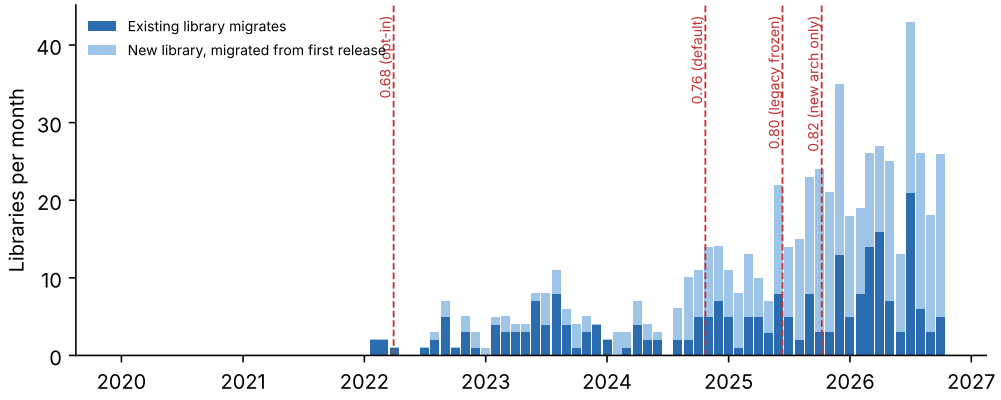


Figure 2: Monthly migrations on the Codegen and Nitro routes, split into conversions of existing libraries and new libraries migrated from their first release. Dashed lines mark the milestones in Table 1.

Table 4: Conversions of existing native libraries (Codegen and Nitro routes) per interval between milestones.

Interval	Months	Conversions	Per month
Before 0.68 (before 30 Mar 2022)	–	3	–
0.68 to 0.76 (optional)	30.8	77	2.5
0.76 to 0.82 (default, opt-out possible)	11.5	56	4.9
After 0.82 (mandatory; to 28 Oct 2026)	12.6	107	8.5

Raw counts partly reflect how many libraries remained to convert, so we also normalized them by the number of libraries at risk. The monthly hazard, the share of not-yet-migrated libraries that converted in a month, was 0.43% while the New Architecture was optional, 0.78% once it became the default, and 1.55% once it became mandatory. The segmented regression, however, does not establish that the milestones caused these increases. With a time trend and the at-risk offset, the level change after 0.76 had an incidence rate ratio of 1.53 (95% CI 0.61–3.82, $p = 0.36$) and the level change after 0.82 a ratio of 2.05 (0.72–5.83, $p = 0.18$); in the simpler model with level changes only, the ratios were 1.50 (0.69–3.26, $p = 0.31$) and 1.77 (0.95–3.31, $p = 0.07$). The direction is consistent across models, but with 239 conversions spread over 55 months, the data cannot distinguish a step change at each milestone from a gradual acceleration.

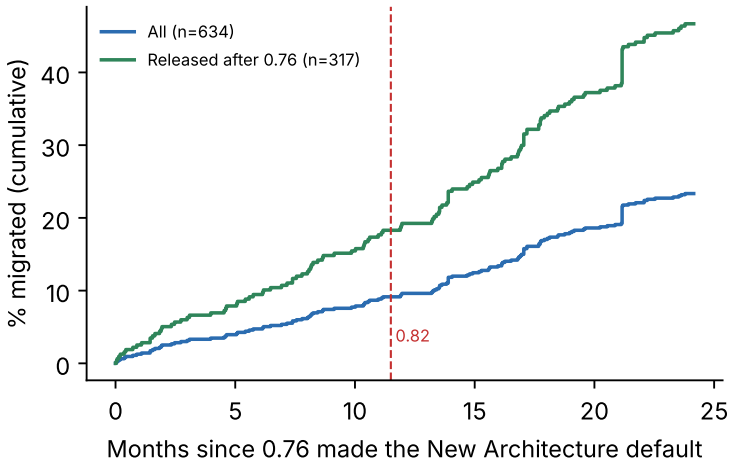


Figure 3: Kaplan–Meier estimate of the cumulative share of native libraries migrated after 0.76 made the New Architecture the default, for libraries that existed and had not migrated at that point (all, and those with at least one release after 0.76).

Rates alone hide how many libraries never moved. We therefore followed the 634 native libraries that existed on 23 October 2024, when 0.76 made the New Architecture the default, and had not yet migrated (excluding Expo Modules libraries, which we cannot date, and partial libraries). Figure 3 shows the Kaplan–Meier estimate of the share that had migrated over time. One year later, when 0.82 removed the opt-out, an estimated 9.6% had migrated; two years later, at our snapshot, 23.3% had. Among the 317 of these libraries that published at least one release after 0.76, and were therefore demonstrably maintained, the share was 46.7%: even among active libraries, half did not migrate within two years of the default change.

The Directory's declarations tell the same story from the maintainers' side. The first declaration of support appears in the snapshot of September 2022. When 0.76 shipped, 179 of the 1,529 listed libraries had declared support; on 1 October 2026, 912 of 2,722 had, and 74 declared that they support only the New Architecture, a category that first appears in September 2025, shortly before 0.82.

> **RQ2 summary.** Conversions became more frequent after each milestone, from 2.5 per month while migration was optional to 4.9 after the default change and 8.5 after the old architecture

Table 5: Characteristics of migrated and unmigrated native libraries (medians and percentages).

	Migrated	Not migrated
Libraries	895	519
Median weekly downloads	818	266
Median months since last release	1.7	38.0
Released after 0.76 (%)	96.3	39.3
Median package age (years)	1.7	7.0
More than one npm maintainer (%)	28.8	34.7
Flagged unmaintained in the Directory (%)	0.6	17.0
Latest version deprecated on npm (%)	1.2	4.0

was removed (a monthly hazard of 0.43%, 0.78% and 1.55%), but a segmented regression cannot attribute the increases to the milestones themselves. Even so, only an estimated 23% of the libraries that had not migrated by the default change did so in the following two years, and fewer than half of the actively maintained ones.

5.3. RQ3: What distinguishes libraries that have not migrated?

Table 5 compares migrated and unmigrated native libraries. Unmigrated libraries are older (median age 7.0 versus 1.7 years), far less recently released (median 38.0 versus 1.7 months since the last release) and less downloaded (median 266 versus 818 weekly downloads). Only 39.3% of them published any release after 0.76, against 96.3% of migrated libraries, and 17.0% are flagged as unmaintained in the Directory. The differences are significant for downloads, time since release and age (Mann–Whitney U, $p < 0.001$), with large effects for time since release and age (rank-biserial correlation 0.70 and 0.63) and a small one for downloads (-0.14).

The logistic regression (Table 6; $n = 1{,}414$, McFadden pseudo-$R^2 = 0.42$) confirms that activity is the strongest factor. Holding the other predictors constant, a library that published any release after 0.76 had 0.09 times the odds of being unmigrated. Each additional year of age increased the odds by 43%, and each tenfold increase in weekly downloads reduced them by 35%. Libraries shipping both iOS and Android code were less likely to be unmigrated. Having several npm maintainers was associated with slightly

Table 6: Logistic regression for being unmigrated (native libraries, migrated vs. not migrated). OR: odds ratio with 95% confidence interval.

Predictor	OR	95% CI	p
Weekly downloads ($\log_{10}$)	0.65	0.58–0.73	<0.001
Released after 0.76	0.09	0.06–0.13	<0.001
Package age (years)	1.43	1.35–1.52	<0.001
More than one npm maintainer	1.59	1.11–2.29	0.011
Both iOS and Android native code	0.59	0.39–0.90	0.014

higher odds of being unmigrated, which we attribute to older, organization-owned packages whose maintainer lists accumulated over time rather than to a causal effect.

Two distinct groups of stragglers emerge. The larger group, 315 libraries (60.7% of unmigrated), is dormant: they have published nothing since 0.76, and 311 have published nothing in over two years. For them, the migration did not stall; it never started. The second group, 204 libraries, is active, releasing after 0.76 yet staying on the legacy bridge. Fourteen of them require React Native 0.76 or later as a peer dependency, and nine require 0.80 or later: they target only applications that run the New Architecture by default, yet continue to rely on the interop layers to do so.

What maintainers say. Table 7 summarizes what the READMEs and npm notices of the 519 unmigrated libraries say. Most say nothing: only 51 (9.8%) make any statement about the New Architecture, compatibility, deprecation or their own maintenance status. Among those that do, abandonment is the most common message: 21 are deprecated on npm and 8 more state in their README that they are unmaintained, archived or looking for maintainers; 11 point users to a replacement, three of which are themselves built on Expo Modules or Nitro. Eighteen libraries claim to support or be compatible with the New Architecture while being implemented on the legacy bridge. Six of these say explicitly that they work through the interop layers or a "compatibility mode", for example a payment-terminal SDK whose README states that its native module "runs through React Native's interoperability layer and is not a TurboModule implementation"; one advertises "Turbo Modules" support although its published code registers only legacy modules. Three libraries are migrating in a pre-release or next major version that our analysis of the latest stable release does not capture, and four state

Table 7: Statements about the New Architecture and maintenance in the READMEs and npm deprecation notices of the 519 unmigrated libraries (categories overlap).

Statement	Libraries
Any relevant statement	51
Deprecated on npm	21
README: unmaintained, archived or looking for maintainers	8
Names a replacement library	11
Claims New Architecture support or compatibility	18
of which explicitly via interop layers or "compatibility mode"	6
Native migration in a pre-release or next major version	3
States incompatibility or old-architecture-only support	4
No relevant statement	468

that they do not work, or work only partially, under the New Architecture.

> **RQ3 summary.** Unmigrated libraries are mostly old and dormant: 61% have released nothing since the New Architecture became the default. Release activity is by far the strongest predictor of migration. A smaller group of active libraries treats the interop layers as a permanent platform rather than a temporary bridge, and some advertise New Architecture support that their code delivers only through those layers. Fewer than one in ten unmigrated libraries says anything about its status, so users cannot tell an abandoned library from one that works through interop without inspecting its code.

5.4. RQ4: How much usage depends on unmigrated libraries?

Although unmigrated libraries receive only 4.8% of native-library downloads, that share amounts to 20.4 million downloads a week. 115 unmigrated libraries receive at least 10,000 weekly downloads and 41 receive at least 100,000; 42 of the 115 have not released since 0.76. Together, unmigrated libraries are listed as dependencies by 6,053 other npm packages. Table 8 lists the most downloaded ones. They include widely used user-interface primitives (a masked view with 5.6 million weekly downloads and a linear-gradient view with 0.6 million), file-system and device-information modules, payments, notifications, real-time communication, video, analytics and testing tools. Some are dormant, such as the file-system module, last released

Table 8: The most downloaded unmigrated native libraries (latest released version on 28 October 2026). "Declared" is the Directory's compatibility declaration.

Package (version)	Weekly downloads	Last release	Releases since 0.76	Declared
@react-native-masked-view/masked-view (0.3.2)	5,618,884	Nov 2024	1	–
react-native-device-info (15.0.2)	1,499,006	Feb 2026	11	Supported
react-native-purchases (11.0.0)	1,014,719	Oct 2026	137	Supported
react-native-linear-gradient (2.8.3)	641,732	Sep 2023	0	–
@notifee/react-native (9.1.8)	440,450	Dec 2024	5	Supported
@livekit/react-native-webrtc (150.1.1)	416,473	Oct 2026	18	Supported
react-native-video (6.19.3)	412,986	Sep 2026	23	Supported
@livekit/react-native (3.1.0)	411,167	Oct 2026	35	Supported
react-native-fs (2.20.0)	398,442	May 2022	0	–
posthog-react-native-session-replay (1.6.0)	377,372	May 2026	33	Supported

in May 2022. Others are actively maintained by companies and declared compatible, but implemented on the legacy bridge.

> **RQ4 summary.** A small share of downloads hides a large absolute exposure: 20 million weekly downloads, 115 libraries above 10,000 weekly downloads, and 6,053 dependent packages rely on libraries that have not migrated. These dependencies tie applications to the interop layers, so the platform cannot remove the layers without breaking a substantial part of the ecosystem.

6. Discussion

6.1. Milestones and the limits of pressure

The three milestones applied increasing pressure, and conversions became more frequent after each of them, with the highest rate after the old architecture could no longer be enabled. Our segmented regression cannot attribute these increases to the milestones, so we read them as consistent with, not proof of, an effect of pressure. For two and a half years the New Architecture was available, documented and promoted, and roughly 80 existing libraries adopted it. Making it the default did not oblige library maintainers to do anything, because the interop layers kept their libraries working; it mainly shifted the burden to application developers, who now had to opt out if a dependency failed. Removing the opt-out in 0.82 changed the calculation for applications rather than for libraries, yet it was followed by the fastest conversion rate, possibly because application developers, no longer able to

postpone, began to report problems and send fixes upstream. What the data show more clearly is the limit of pressure: whatever their effect on active libraries, none of the milestones reached the dormant majority of the unmigrated tail, whose maintainers were no longer there to respond. For platform owners, this suggests that a default without a deadline mainly reveals which libraries break, and that announcing the end of the old path early, with a firm date, gives active maintainers time to act, while the dormant tail needs a different remedy, such as replacement, adoption by new maintainers, or a long-lived compatibility layer.

6.2. Compatibility layers become destinations

The interop layers were designed as a bridge across the transition. Our results suggest that, for a large part of the ecosystem, they have become the destination. Two years after the default change, more than three quarters of the libraries that had not migrated still had not, most of them dormant packages whose maintainers are unlikely to return. A smaller but more telling group is active and even requires recent React Native versions, yet remains on the legacy bridge. Because unmigrated libraries still serve 20 million downloads a week, removing the interop layers would break a substantial share of applications. Platform owners should therefore plan compatibility layers as long-lived infrastructure, with tests and documented limits, and should measure who depends on them before announcing their removal. Our classification offers one way to do so, and it can be repeated as the ecosystem changes.

6.3. The migration target fragmented

The platform defined one migration route, Codegen, but the ecosystem took three. Expo Modules and Nitro Modules together already account for almost half of all migrated libraries and for most migrated libraries created in 2026. Third-party frameworks offered library authors a simpler or faster path, and in doing so absorbed much of the migration effort the platform asked for. This lowered the cost of migration for many authors, but it also means that the ecosystem now depends on two additional frameworks with their own release cycles and maintainers. A future change to React Native's native interfaces will reach many libraries only through these frameworks, so their maintainers have become, in effect, part of the platform. Platform owners planning similar transitions should expect such intermediaries to emerge and should involve them early.

6.4. Implications for application developers and library maintainers

For application developers, a compatibility label is not the same as a migration. A library declared "supported" may still run through the interop layers, and half of the native libraries declare nothing at all. Teams auditing their dependencies can check for the code-level signals we used: a `codegenConfig` field in the library's manifest, or a dependency on Expo Modules or Nitro. Dormant libraries with high download counts, such as several in Table 8, are the most likely to become blockers if the interop layers change, and they are candidates for replacement or for a maintained fork. For library maintainers, release activity was the strongest predictor of migration, which suggests that the barrier is less the technical difficulty of migrating than the absence of anyone to do it. Organizations that depend on such libraries can lower that barrier by funding or contributing the migration.

6.5. Implications for research

Most studies of ecosystem change ask whether clients follow their libraries. An architecture migration reverses the direction: libraries must follow their platform, and applications are held back by libraries they do not control. This inversion deserves more attention, together with the role of intermediary frameworks that, as here, absorb a platform transition on behalf of many libraries. Open questions include how the interop layers perform compared with native migration, how often application teams replace or fork unmigrated dependencies, and whether earlier announcement of a removal date shortens transitions.

7. Threats to Validity

Construct validity. We measure native adoption of the New Architecture from code-level signals in the latest published package. A library can work correctly in a New Architecture application without migrating, through the interop layers, so our "not migrated" category is not a measure of breakage. Conversely, a migrated library may still have defects under the New Architecture. Our classifier relies on textual signals; the manual check of 60 stratified libraries found no misclassification of migrated libraries and led to two refinements, and a blind second classification of 140 libraries agreed on 91.4% of routes ($\kappa = 0.89$) and 97.9% of migrated/unmigrated decisions. That second rater was an AI agent rather than a human expert, so it tests consistency with an independent reading of the code but not human

judgment. Our analysis of maintainers' statements is limited to READMEs and npm notices; issue trackers and discussions, which we did not analyze, may record reasons that maintainers never put in their documentation. We dated migrations by the first release declaring Codegen or depending on Nitro, which may follow a library's actual development work by weeks, and we could not date migrations through Expo Modules.

Internal validity. The increase in conversion rates after each milestone is consistent with an effect of the milestones but does not establish one: the ecosystem also grew, tooling and documentation improved, and Expo's platform made the New Architecture its default for its own users during the same period. We report rates and a segmented regression whose milestone effects are not statistically significant; we make no causal claim. The regression identifies associations; for example, the association between several maintainers and non-migration likely reflects package age and ownership history rather than any effect of maintainers.

External validity. The React Native Directory is curated and overrepresents libraries that someone chose to list; the wider npm ecosystem contains many more React Native packages, most of them likely less maintained, so our migration rates are probably an upper bound for React Native packages in general. Results for React Native may not transfer to other platforms, although the staged transition we studied resembles those of other ecosystems.

Reliability. Popularity figures come from the npm registry's search service and reflect a single snapshot; download counts include automated installs. All data sources are public, and we release the collection and analysis scripts with the derived dataset so that the study can be repeated and extended.

8. Conclusion

React Native's four-year move to the New Architecture shows how a platform transition plays out across an ecosystem of independent libraries. Sixty-two percent of native libraries have migrated, and they serve almost all downloads, but a long tail of 519 libraries, mostly old and dormant, still depends on compatibility layers. Conversions became more frequent after each milestone, most of all after the old architecture could no longer be enabled, although we cannot attribute the increases to the milestones; and most libraries that had not moved by the default change did not move in the following two years,

largely because no one was maintaining them. Meanwhile, the migration target itself fragmented, as new libraries chose third-party frameworks over the platform's own route. For platform owners, the results argue for early, firm deadlines for active maintainers, for planning replacements for the dormant tail, for treating compatibility layers as long-lived infrastructure, and for engaging the intermediaries that will carry much of the migration.

Data Availability

The dataset (one row per library with all derived variables), the monthly Directory snapshots, the validation samples and labels with the codebook, the coded README statements, the monthly series used in the segmented regression, and the collection and analysis scripts are openly available on Zenodo: `https://doi.org/10.5281/zenodo.23749468`. All raw data come from public sources: the React Native Directory repository and the npm registry.

Declaration of Competing Interest

The author declares no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper. No organization whose libraries are discussed funded, reviewed or approved this work.

CRediT Authorship Contribution Statement

Revanth Krishna Maddula: Conceptualization, Methodology, Software, Data curation, Formal analysis, Investigation, Validation, Visualization, Writing – original draft, Writing – review & editing.

Declaration of Generative AI and AI-Assisted Technologies in the Manuscript Preparation Process

During the preparation of this work the author used Claude (Anthropic) in order to write data-collection and analysis scripts and to draft and edit text. After using this tool, the author reviewed and edited the content as needed and takes full responsibility for the content of the published article.

References

[1] B. A. Malloy, J. F. Power, An empirical analysis of the transition from Python 2 to Python 3, Empirical Software Engineering 24 (2) (2019) 751–778. doi:10.1007/s10664-018-9637-2.

[2] J. Li, K. Huang, S. Wang, Y. Liu, Towards the adoption and adaptation of the AndroidX library: An empirical study, in: Proceedings of the 23rd IEEE International Conference on Software Quality, Reliability, and Security (QRS), 2023.

[3] React Native Team, The New Architecture is here, React Native blog, 23 October 2024. `https://reactnative.dev/blog/2024/10/23/the-new-architecture-is-here`.

[4] React Native Team, Announcing React Native 0.68, React Native blog, 30 March 2022. `https://reactnative.dev/blog/2022/03/30/version-068`.

[5] React Native Team, React Native 0.80, React Native blog, 12 June 2025. `https://reactnative.dev/blog/2025/06/12/react-native-0.80`.

[6] React Native Team, React Native 0.82 – A new era, React Native blog, 8 October 2025. `https://reactnative.dev/blog/2025/10/08/react-native-0.82`.

[7] React Native Community, React Native Directory, `https://reactnative.directory`; source repository `https://github.com/react-native-community/directory` (accessed 28 October 2026).

[8] Expo, Expo Modules API: Overview, Expo documentation. `https://docs.expo.dev/modules/overview/` (accessed 28 October 2026).

[9] Margelo, Nitro Modules, Nitro documentation. `https://nitro.margelo.com/docs/nitro-modules` (accessed 28 October 2026).

[10] T. McDonnell, B. Ray, M. Kim, An empirical study of API stability and adoption in the Android ecosystem, in: Proceedings of the IEEE International Conference on Software Maintenance (ICSM), 2013, pp. 70–79.

[11] R. G. Kula, D. M. German, A. Ouni, T. Ishio, K. Inoue, Do developers update their library dependencies? An empirical study on the impact of security advisories on library migration, Empirical Software Engineering 23 (1) (2018) 384–417.

[12] P. Salza, F. Palomba, D. Di Nucci, C. D'Uva, A. De Lucia, F. Ferrucci, Do developers update third-party libraries in mobile apps?, in: Proceedings of the 26th International Conference on Program Comprehension (ICPC), 2018, pp. 255–265. doi:10.1145/3196321.3196341.

[13] A. Zerouali, E. Constantinou, T. Mens, G. Robles, J. González-Barahona, An empirical analysis of technical lag in npm package dependencies, in: New Opportunities for Software Reuse (ICSR 2018), Lecture Notes in Computer Science, vol. 10826, Springer, 2018, pp. 95–110.

[14] A. Decan, T. Mens, P. Grosjean, An empirical comparison of dependency network evolution in seven software packaging ecosystems, Empirical Software Engineering 24 (1) (2019) 381–416.

[15] A. Hora, R. Robbes, N. Anquetil, A. Etien, S. Ducasse, M. T. Valente, How do developers react to API evolution? The Pharo ecosystem case, in: Proceedings of the IEEE International Conference on Software Maintenance and Evolution (ICSME), 2015, pp. 251–260.

[16] A. A. Sawant, R. Robbes, A. Bacchelli, On the reaction to deprecation of clients of 4+1 popular Java APIs and the JDK, Empirical Software Engineering 23 (4) (2018) 2158–2197.

[17] C. Bogart, C. Kästner, J. Herbsleb, F. Thung, When and how to make breaking changes: Policies and practices in 18 open source software ecosystems, ACM Transactions on Software Engineering and Methodology 30 (4) (2021) Article 42.

[18] G. Avelino, E. Constantinou, M. T. Valente, A. Serebrenik, On the abandonment and survival of open source projects: An empirical investigation, in: Proceedings of the ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), 2019.

[19] C. Miller, M. Jahanshahi, A. Mockus, B. Vasilescu, C. Kästner, Understanding the response to open-source dependency abandonment in the

npm ecosystem, in: Proceedings of the 47th IEEE/ACM International Conference on Software Engineering (ICSE), 2025.

[20] A. Biørn-Hansen, T.-M. Grønli, G. Ghinea, A survey and taxonomy of core concepts and research challenges in cross-platform mobile development, ACM Computing Surveys 51 (5) (2018) Article 108.

[21] E. L. Kaplan, P. Meier, Nonparametric estimation from incomplete observations, Journal of the American Statistical Association 53 (282) (1958) 457–481.

[22] R. K. Maddula, Replication package: Migrating an Ecosystem – React Native New Architecture migration study, Zenodo, 2026. doi:10.5281/zenodo.23749468.