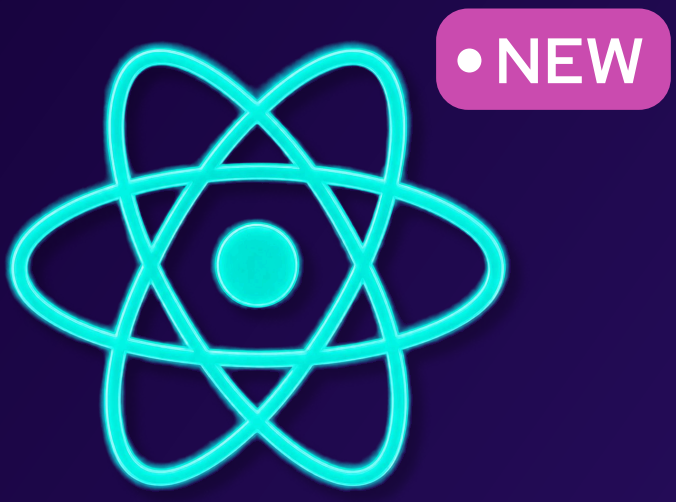
nine two three



React Native's New Architecture: Overview & Migration

BY OLEKSANDR KOBA,
MOBILE TEAM LEAD

TABLE OF CONTENTS

React Native's New Architecture is an updated framework's core to improve performance, type safety, and interoperability with native code. Although it offers significant performance improvements, implementing it involves migrating existing projects and weighing some trade-offs to match the new core architecture.

1. Overview

The RN's New Architecture is built using four main pillars:

- **JSI** - a C++ layer that lets JavaScript call native functions directly without JSON serialization, replacing the **Bridge**;
- **TurboModules** - native modules that load lazily on first use instead of all at startup, with type-safe interfaces defined in TypeScript.
- **Fabric** - the new UI renderer that shares the view tree between JS and native via C++, enabling synchronous layout and concurrent rendering.
- **Codegen** - a build-time tool that generates native boilerplate (Kotlin/ObjC/C++) from TypeScript specs, ensuring type safety between shared (JS) and native layers

Old architecture, which uses the **Bridge**, is much slower because it serializes every command between shared and native layers to JSON, even the UI rendering. To gain performance, it's being replaced with the JSI. The table below shows the RN versions that provide breaking changes in the New Architecture usage.

1.1. RN Releases

RN Version	Release Date	Changelog
0.76	Oct 23, 2024	New Architecture enabled by default; the Bridge is still available as a fallback.
0.82	Oct 6, 2025	Old architecture can no longer be disabled. (`newArchEnabled=false` is ignored)
0.84	Feb 9, 2026	Last version where bridge-only packages still work via the interop layer.
0.85	Apr 6, 2026	Bridge code fully removed from the codebase; bridge-only packages stop working.

According to the RN's support list, the 0.84.x and 0.85.x versions are currently active, while 0.83.x is at the end of the support cycle. Based on that, existing and new RN-based projects can continue using the 0.84.x version with older packages (that are running using Bridge), but it's required to prepare a migration plan to the 0.85.x version, use packages that support JSI, and handle New Architecture's known issues.

1.2. Migration Checklist

Please see the action items list, which highlights the main requirements of the codebase to match the New Architecture:

Action Item	Required Action
RN version should be higher than 0.73.x	Update the RN's version in `package.json`
All third-party packages should support JSI	Each third-party package should be checked manually for compatibility with the New Architecture. Not compatible ones should be replaced.
Hermes engine should be enabled (New Architecture doesn't work without it)	Add the `hermesEnabled=true` line to Android's `gradle.properties` file
New Architecture should be enabled	Android: add the `newArchEnabled=true` line to the `gradle.properties` file; iOS: add the `ENV['RCT_NEW_ARCH_ENABLED'] = '1'` line to the Podfile;

2. Project Configuration

The first step of the migration process is to configure the project to enable React Native's New Architecture features, including Hermes, Turbo Modules, and Fabric. This requires updating both iOS and Android project configurations.

2.1. Android Configuration

Enable the New Architecture flags in `android/gradle.properties`:

```
# android/gradle.properties
hermesEnabled=true
newArchEnabled=true
TM_ENABLED=true
FABRIC_ENABLED=true
```

Then, update the React Native configuration in `android/app/build.gradle`:

```
// android/app/build.gradle
android {
    defaultConfig {
        buildConfigField(
            "boolean",
            "IS_NEW_ARCHITECTURE_ENABLED",
            isNewArchitectureEnabled().toString()
        )
    }
}
project.ext.react = [
    enableHermes: true,
    enableNewArchitecture: true
]
```

2.2. iOS Configuration

Update the `ios/Podfile` to enable Hermes and Fabric support:

```
# ios/Podfile
require_relative '../node_modules/react-native/scripts/
react_native_pods'
require_relative '../node_modules/@react-native-community/
cli-platform-ios/native_modules'

platform :ios, '13.0'

target 'YourApp' do
  config = use_native_modules!

  use_react_native!(
    :path => config[:reactNativePath],
    :hermes_enabled => true,
    :fabric_enabled => true,
  )

  post_install do |installer|
    react_native_post_install(installer)
  end
end
```

After updating the configuration, reinstall CocoaPods dependencies:

```
cd ios && pod deintegrate && pod install && cd ..
```

2.3. Codegen Configuration

If the project includes custom Turbo Modules or Fabric components, Codegen is used to generate the required native bindings and interfaces from the corresponding TypeScript specifications. Codegen runs automatically during the native build process and typically does not require any manual steps. It reads TypeScript specifications from third-party packages and generates the native bindings and interfaces required by the New Architecture.

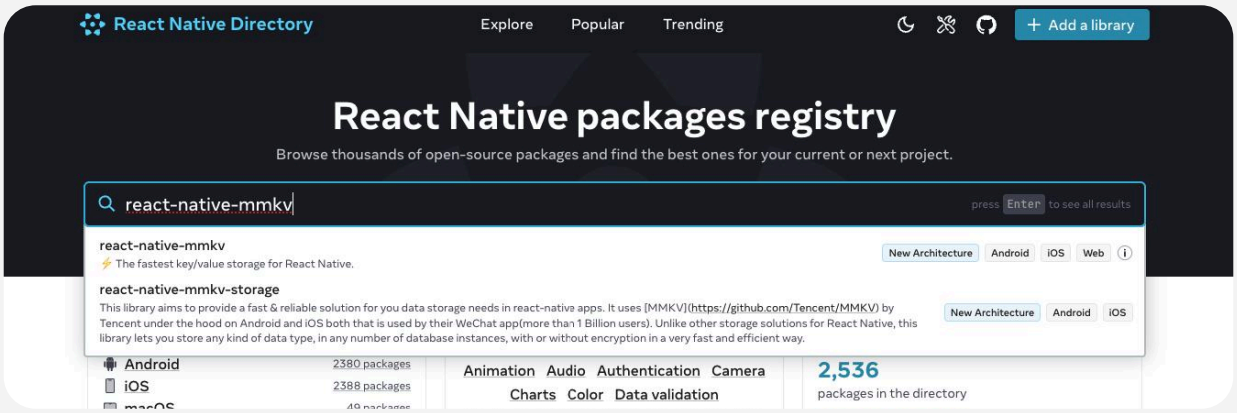
Manual regeneration is generally only required when developing or modifying custom Turbo Modules or Fabric components. If manual regeneration is ever required, the following scripts can be added to the `package.json`:

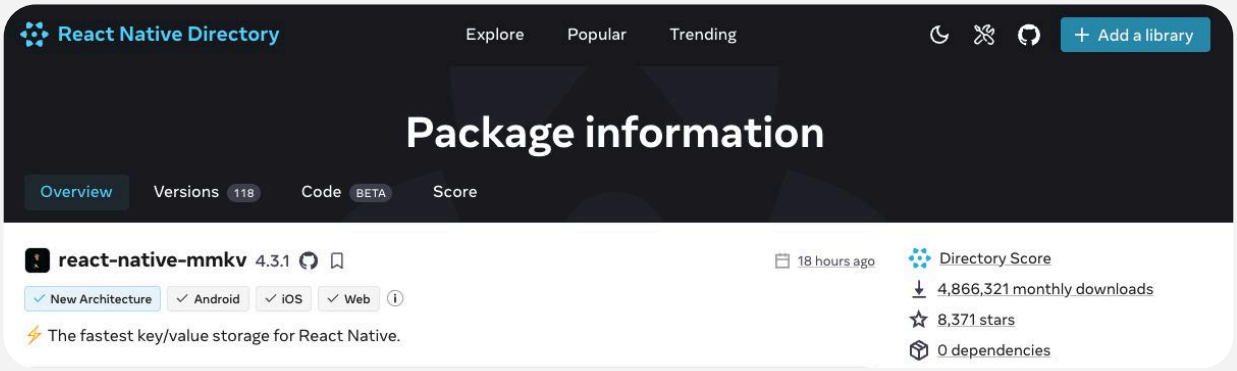
```
{
  "scripts": {
    "ios:codegen": "cd ios && RCT_NEW_ARCH_ENABLED=1 pod
install",
    "android:codegen": "cd android && ./gradlew
generateCodegenArtifactsFromSchema"
  }
}
```

3. Third-Party Package Management

As the New Architecture (starting from 0.85.x) doesn't support Bridge, every package must be compatible with JSI. To verify the package compatibility, the reactnative.directory website can be used. If the package is compatible, it'll show the checkmark near the "New Architecture" pill.

Please see the example below:





3.1. Popular third-party packages migration

As many projects still rely on outdated third-party packages, it is important to replace them with up-to-date and actively maintained alternatives.

Please see the list of the popular replacement & update candidates:

Package	Alternative	Comment
react-native-encrypted-storage	react-native-mmkv + AES encryption with dynamic key via react-native-keychain	`encrypted-storage` is deprecated since 2022; `react-native-mmkv` is the most popular storage option with built-in encryption.
react-native-i18n	react-native-localize + i18n-js	`react-native-i18n` has been unsupported since 2018.
react-native-push-notification	@notifee/react-native	Can also be used in combination with Firebase.
react-native-fs	@birdofpreyru/react-native-fs	Updated community fork of the `react-native-fs` with New Architecture support.
react-native-config	react-native-config v1.6.0+	Update to v1.6.0+ is required
react-navigation	react-navigation v7+	Update to v7+ is required
react-native-webview	react-native-webview v13.6.0+	Update to v13.6.0+ is required
react-native-maps	react-native-maps v1.11+	Update to v1.11+ is required

3.2. Summary

To summarize, the most widely used packages already provide versions compatible with the New Architecture, while some outdated but critical dependencies still require replacement with alternative solutions. Therefore, the third-party package update and migration strategy should be defined individually for each project.

4. Native Modules to Turbo Modules migration

All Native Modules should be migrated to Turbo Modules across the project. Native Modules are used to expose platform-specific functionality (e.g., camera, biometrics, Bluetooth, encrypted storage) to JavaScript through native Android and iOS code.

In the old architecture, communication between JavaScript and native code relied on the Bridge with asynchronous JSON serialization. Turbo Modules replace this approach with a type-safe, Codegen-based system that generates native bindings from a TypeScript specification and loads modules lazily via JSI, improving performance and startup time.

4.1. Migration Example

Native battery module example - module registers at startup via Bridge, communicates through async JSON serialization. No compile-time type safety: wrong method names or argument types crash at runtime.

```
// JS — no spec, no type safety, runtime crash if wrong
  NativeModules.BatteryModule.getBatteryLevel();

// Android — extends ReactContextBaseJavaModule, uses @ReactMethod
  class BatteryModule(ctx: ReactApplicationContext)
      : ReactContextBaseJavaModule(ctx) {
      @ReactMethod
      fun getBatteryLevel(promise: Promise) {
          val bm = ctx.getSystemService(Context.BATTERY_SERVICE) as BatteryManager
          promise.resolve(
bm.getIntProperty(BatteryManager.BATTERY_PROPERTY_CAPACITY)
   )
      }
  }

// iOS — conforms to RCTBridgeModule, uses RCT_EXPORT_METHOD
  @interface BatteryModule : NSObject <RCTBridgeModule>
  @end
  RCT_EXPORT_METHOD(getBatteryLevel:(RCTPromiseResolveBlock)resolve
                    reject:(RCTPromiseRejectBlock)reject) {
      UIDevice.current.isBatteryMonitoringEnabled = YES;
      resolve(@(UIDevice.current.batteryLevel * 100));
  }
```

To transform it to the TurboModule, it’s required to define a TypeScript spec (an interface that declares the API).

Codegen generates these interfaces for both Android & iOS platforms, and the developer should provide the corresponding implementations, which will cause compile errors if they're implemented incorrectly. Please see the Turbo Module example:

```
// Spec – Codegen generates native stubs for both platforms from this
  export interface Spec extends TurboModule {
    getBatteryLevel(): Promise<number>;
    isCharging(): boolean;
  }

// Android – implements NativeBatteryModuleSpec (generated by Codegen)
  class BatteryModule(ctx: ReactApplicationContext): NativeBatteryModuleSpec(ctx)
      override fun getBatteryLevel(promise: Promise) {
          val bm = ctx.getSystemService(Context.BATTERY_SERVICE) as BatteryManage
          promise.resolve(
           bm.getIntProperty(BatteryManager.BATTERY_PROPERTY_CAPACITY)
          )
      }

      override fun isCharging(): Boolean {
          val bm = ctx.getSystemService(Context.BATTERY_SERVICE) as BatteryManage
          return bm.isCharging
      }
  }

// iOS – implements NativeBatteryModuleSpec (generated by Codegen)
  @interface BatteryModule : NSObject <NativeBatteryModuleSpec>
  @end
  - (void)getBatteryLevel:(RCTPromiseResolveBlock)resolve
                   reject:(RCTPromiseRejectBlock)reject {
      UIDevice.current.isBatteryMonitoringEnabled = YES;
      resolve(@(UIDevice.current.batteryLevel * 100));
  }

  - (NSNumber *)isCharging {
      UIDevice.current.isBatteryMonitoringEnabled = YES;
      return @(UIDevice.current.batteryState == UIDeviceBatteryStateCharging);
  }
```

Also, please see the Turbo Module usage example (with sync and async calls):

```
// JS – type-checked at compile time, lazy-loaded at runtime
  const Battery =
TurboModuleRegistry.getEnforcing<Spec>('BatteryModule');

  Battery.getBatteryLevel().then(level => console.log(level)); //
87
  Battery.isCharging(); // true – sync call, impossible with old Bridge
```

4.2. Summary

Turbo Modules provide an effective and type-safe way to receive the platform-specific information. In most cases, functionality that requires additional native modules is already covered by community packages. So, if all third-party dependencies support the New Architecture and the project doesn't include custom native modules, this step can be skipped. Otherwise, every Native Module should be migrated to the Turbo Module.

5. Legacy Renderer to Fabric Migration

All custom native UI components should be migrated to support Fabric, React Native's new rendering system. Custom views are used to implement platform-specific UI and use the native rendering capabilities in React Native application UI.
The Legacy Renderer relies on asynchronous communication between shared and native UI layers through the Bridge, which can introduce rendering delays and limit synchronization. Fabric replaces this with a modern rendering system based on JSI and synchronous layout updates, enabling improved rendering performance, better interoperability with native components, and more responsive UI updates.

5.1. Migration Example

Please see the Old Architecture (Legacy Renderer) example below. Custom native views are registered using requireNativeComponent and rendered through the asynchronous Bridge. Layout calculations are performed on the JavaScript thread and then sent to the native layer asynchronously. This approach does not support synchronous measurements or concurrent rendering. The example below demonstrates a simple badge component that displays a number with a colored background.

```
// JS — manual registration, no type safety on props
  import { requireNativeComponent } from 'react-native';
  const BadgeView = requireNativeComponent('BadgeView');

  <BadgeView count={5} color="#ff0000" />

// Android — extends SimpleViewManager, props via @ReactProp
  class BadgeViewManager : SimpleViewManager<BadgeView>() {
      override fun getName() = "BadgeView"

      @ReactProp(name = "count")
      fun setCount(view: BadgeView, count: Int) {
          view.text = count.toString()
      }

      @ReactProp(name = "color")
      fun setColor(view: BadgeView, color: String) {
          view.background.setTint(Color.parseColor(color))
      }
  }

// iOS — extends RCTViewManager, props via RCT_EXPORT_VIEW_PROPERTY
  @interface BadgeViewManager : RCTViewManager
  @end

  RCT_EXPORT_MODULE()
  RCT_EXPORT_VIEW_PROPERTY(count, NSInteger)
  RCT_EXPORT_VIEW_PROPERTY(color, NSString)

  - (UIView *)view {
      return [[BadgeView alloc] init];
  }
```

To migrate the component to Fabric, it is required to define a TypeScript spec that describes the component's props and API. Codegen then generates type-safe native interfaces and stubs for both Android and iOS,

while developers provide the corresponding native implementations. This approach ensures compile-time validation and reduces integration errors. The example below demonstrates the same badge view migrated to Fabric.

```
// JS - typed props, Codegen generates native registrations
  interface NativeProps extends ViewProps {
    count?: Int32;
    color?: ColorValue;
  }

  export default codegenNativeComponent<NativeProps>('BadgeView');

  <BadgeView count={5} color="#00CCCA" />

// Android - implements generated interface
  class BadgeViewManager : SimpleViewManager<BadgeView>(),
      BadgeViewManagerInterface<BadgeView> {

      override fun getName() = "BadgeView"

      override fun setCount(view: BadgeView, count: Int) {
          view.text = count.toString()
      }

      override fun setColor(view: BadgeView, color: Int) {
          color?.let { view.background.setTint(it) }
      }
  }
// iOS - extends RCTViewComponentView, props via shared C++ shadow tree
 @interface BadgeView : RCTViewComponentView
 @end
 - (void)updateProps:(const Props::Shared &)props
             oldProps:(const Props::Shared &)oldProps {
      const auto &newProps = static_cast<const BadgeViewProps &>(*props);
      const auto &old = static_cast<const BadgeViewProps &>(*oldProps);

      if (newProps.count != old.count) {
          _label.text = [NSString stringWithFormat:@"%d", newProps.count];
      }
      if (newProps.color != old.color) {
          _badge.backgroundColor = RCTUIColorFromSharedColor(newProps.color);
      }
      [super updateProps:props oldProps:oldProps];
  }
```

5.2. Summary

To summarize, Fabric provides a modern and efficient rendering system for React Native UI components. Most standard React Native components are automatically migrated as part of the New Architecture adoption, while custom native UI components and views require the migration to support Fabric-specific rendering APIs and Codegen integration. If the project doesn't include custom native views, this step can be skipped. Otherwise, all custom native UI components should be migrated to Fabric to ensure compatibility with the New Architecture.

6. Known Issues Handling

The following issues are commonly reported by developers and identified during the adoption of React Native's New Architecture. Most of these issues primarily affect physical Android devices, while Android emulators and iOS devices may continue to work correctly.

6.1 Touchable Components do not respond on physical Android devices

Pressable, TouchableOpacity, and TouchableWithoutFeedback from react-native may stop responding to touch events on physical Android devices with the New Architecture enabled.

GestureHandlerRootView intercepts touch events at the native level, preventing non-registered components from receiving them. This issue has been reported across multiple libraries and projects. This issue has been reported across multiple libraries and projects, including React Native #44643, React Navigation #12039, React Native Screens #2219, and Expo #31505.

Observed Behavior:

- Touchable components render correctly but do not respond to taps
- No related console errors
- The issue only occurs on physical Android devices

Solution:

Use Pressable from react-native-gesture-handler instead of the default React Native implementation:

```
// Before
import { Pressable } from 'react-native';

// After
import { Pressable } from 'react-native-gesture-handler';
```

6.2. TextInput does not receive focus on physical Android devices

TextInput from react-native may fail to receive focus on physical Android devices because it is not integrated into the gesture handler system. Tapping the input does not open the keyboard or trigger focus events. Related reports include React Native #44643 and React Native Paper #4517.

Observed Behavior:

- Input fields cannot be focused
- The keyboard does not appear
- The issue only affects physical Android devices

Solution:

Use TextInput from react-native-gesture-handler:

```
import { TextInput as GHTextInput } from 'react-native-gesture-handler';

<Input
   InputComponent={GHTextInput}
   // ...other props
/>
```

For non-editable inputs used as dropdown selectors:

```
<View pointerEvents={props.editable === false ? 'none' :
'auto'}>
   <Input
       InputComponent={GHTextInput}
       editable={false}
   />
</View>
```

6.3. BottomSheetFlatList is not scrollable inside BottomSheetView

Scrollable lists inside @gorhom/bottom-sheet may stop scrolling when wrapped with BottomSheetView under Fabric on Android devices. This is caused by gesture coordination conflicts between the container and the scrollable component. This issue is tracked in gorhom/react-native-bottom-sheet #2539.

Observed Behavior:

- The list renders correctly, but cannot be scrolled
- The issue only affects bottom sheets with scrollable content

Solution:

Replace BottomSheetView with a regular View when using fixed snapPoints:

```
// Before
<BottomSheetView style={styles.container}>
   <BottomSheetFlatList data={items}
renderItem={renderItem} />
</BottomSheetView>

// After
<View style={styles.container}>
   <BottomSheetFlatList data={items}
renderItem={renderItem} />
</View>
```

6.4. Keyboard overlaps inputs inside Bottom Sheets

After the removal of the deprecated shouldHandleKeyboardEvents API, bottom sheets may no longer adjust correctly when the keyboard appears, causing inputs to become hidden behind the keyboard. Related discussions include gorhom/react-native-bottom-sheet #2517 and gorhom/react-native-bottom-sheet #1981.

Observed Behavior:

The keyboard overlaps input fields inside the bottom sheets

The bottom sheet does not adjust automatically

Solution:

Use BottomSheetTextInput from @gorhom/bottom-sheet:

```
import { BottomSheetTextInput } from '@gorhom/bottom-
sheet';

<InputWError
   InputComponent={BottomSheetTextInput}
   label="Name"
   // ...other props
/>
```

6.5. forwardRef type mismatch with React 19

React 19 introduced changes to the ForwardedRef type definition, causing type incompatibilities in some third-party libraries that still rely on legacy class component typings. This issue affects TypeScript only and has no runtime impact, so it can be ignored. This issue is tracked in react-native-elements #3445. Additionally, forwardRef itself is now considered legacy and is planned for future removal according to the official React documentation: React forwardRef Documentation

Observed Behavior:

TypeScript errors when passing refs to certain third-party components

The application still builds and runs correctly

Solution:

Suppress the TypeScript error using @ts-expect-error:

```
<Input
   // @ts-expect-error React 19 ForwardedRef incompatible
with RNEUI class component ref
   ref={ref}
/>
```

7. Summary

Migration to React Native's New Architecture involves:

- enabling Hermes, Turbo Modules, Fabric, and Codegen support;
- updating Android and iOS project configurations;
- validating third-party package compatibility;
- migrating custom Native Modules and Fabric components when required;
- performing comprehensive QA testing on physical devices.

The New Architecture provides several benefits, including improved rendering performance, faster startup time, synchronous UI updates, concurrent rendering support, better type safety, and more efficient communication between JavaScript and native layers.
At the same time, migration requires careful compatibility validation, especially on physical devices, where ecosystem-level issues related to gestures, touch handling, scrolling, and keyboard interactions are commonly reported.

General Inquiries:
contact@ninetwothree.co

PR&Collaborations:
marketing@ninetwothree.co