React Native

New Architecture Book

Verified against React Native v0.88.1+main

2026-10-30

Gaston Morixe

Contents

About this book 5
Version and audit 5
Part 1: Core Architecture 6
Chapter 1: Introduction - The "Why" 6
Visual Overview: Old vs New 7
The Old Architecture: A Story of the Bridge 7
The Vision: A New Foundation for React Native 11
The Three Pillars of the New Architecture 15
Chapter 2: JSI - The Foundation 19
Visual: JSI Call Path and Data Flow 19
The Core Principles of JSI 19
The Key JSI C++ Components 21
Real-World Example: Vision Camera's Frame Processor (v4) 29
Advanced JSI Patterns 29
Chapter 3: Fabric - The New Renderer 38
The Motivation: From UIManager to Fabric 38
Visual: Fabric Render Pipeline 38
The C++ Core of Fabric 38
The Three Phases of Fabric Rendering 47
React 18+ / React 19 Concurrent Features: Real-World Impact 54
Chapter 4: TurboModules - The New Native Modules 60
Visual: TurboModule + CodeGen Pipeline 60
The Motivation: Problems with Legacy Native Modules 60
The Core Benefits of TurboModules 61
How TurboModules Work Under the Hood 62
Chapter 5: CodeGen - The Automation Engine 69
The Motivation: Eliminating Manual Glue Code 69
The CodeGen Pipeline 69
The Benefit: Compile-Time Safety 74
Chapter 6: Bridgeless - The Ultimate Goal 77
What is Bridgeless Mode? 77
Visual: bridgeless module routing 79
The Benefits of a Bridgeless World 79
The Interoperability Layer 80
Part 2: Practical Application & Guides 82
Chapter 7: Migration and Adoption 82
Visual: Migration Flow 84
Step 1: Prerequisites and Enabling the New Architecture 84
Step 2: Migrating a Legacy Native Module to a TurboModule 87
Step 3: Migrating a Legacy View Manager to a Fabric Component 89
Step 4: Audit Dependencies 90

Chapter 8: Performance Analysis 93
Visual: Call Overhead — Bridge vs JSI (Conceptual) 94
The Official Stance vs. Real-World Results 94
App Startup Time 95
UI Performance and Responsiveness (Fabric) 95
Native Module Invocation Speed (JSI & TurboModules) 96
React 18+ Concurrent Features Performance 97
Reported Production Benchmarks (community-aggregated) 98
Summary of Findings 98
Chapter 9: Conclusion and Future 100
A Summary of Key Advancements 100
The Impact: A More Performant and Capable Framework 101
The Future of React Native 101
Chapter 10: Common Mistakes and Misunderstandings 104
General Misunderstandings 104
JSI and C++ Mistakes 104
TurboModule Mistakes 106
Fabric Component Mistakes 106
CodeGen Mistakes 107
Chapter 11: Best Practices 111
General Practices 111
JSI and C++ Practices 111
TurboModule Practices 112
Fabric Component Practices 112
Chapter 12: Frequently Asked Questions (FAQs) 116
Part 3: Real-World Examples & Tooling 122
Chapter 13: Case Study - `react-native-vision-camera` 122
What "New Architecture support" actually means in v4 122
Advanced JSI: The Frame Processor 124
What v5 changed (and why this case study still matters) 126
Conclusion 127
Chapter 14: Alternative Tooling - Nitro 131
Core Concepts of Nitro 131
Key Differentiators and Performance Claims 134
Latest Nitro Features (as of v0.37.1, August 2026) 136
Conclusion on Nitro 137
Chapter 15: Tutorial - Building a Native Module and Component 141
Part 1: Project Setup 141
Part 2: Building the TurboModule 142
Part 3: Building the Fabric Component 148
Part 4: Using the Library 153
Summary of Language Roles 153
Chapter 16: Additional Resources and Checklists 156

Common Migration Challenges 156
Recommended Development Tools 156
Library Migration Checklist 157
Chapter 17: Top 27 React Native Interview Questions 160
Part 1: Architecture 160
Part 2: Runtime and JavaScript 164
Part 3: UI and performance 170
Part 4: Native code and ecosystem 175
Part 5: Testing 183
Footnotes 192
Chapter 18: Testing React Native Apps 193
Part 1: Unit, component, and integration tests 193
Part 2: End-to-end and device tests 198
Part 3: History, device clouds, and CI 209
Part 4: Agentic AI and testing 217
Footnotes 227

About this book

Note

Last update: RN v0.88.1 (main `9c7b0f5fafd`) 2026-10-29

> A deep-dive book on the React Native New Architecture, covering core concepts, practical guides, real-world case studies, and more.

This page is the table of contents for an 18-chapter book on the React Native New Architecture. Each chapter is grounded in the official React Native repository, the official documentation, and real-world libraries.

Status: verified against React Native `main` at commit `9c7b0f5fafd` (v0.88.1 era, refresh 2026-10-29; previous refresh was v0.87.0-rc.3). Delta since `ca750feefb1` covers `v0.87.0`, `v0.87.1`, the 0.88 RC train, `v0.88.0`, `v0.88.1`, and 795 commits on `main` after `v0.88.1`. See `CHANGELOG.md` for the full delta and each chapter's YAML frontmatter `taillog` for per-chapter history.

This documentation has been thoroughly updated to reflect the current state of the New Architecture, which is now enabled by default, **cannot be opted out of** on RN ≥ 0.82, and continues to hard-wire TurboModules and delete legacy Bridge / interop surfaces on the v0.87 and v0.88 lines.[1]

Tip: Many chapters now include Mermaid diagrams (rendered on GitHub) to visualize the Bridge vs JSI call paths, Fabric's render pipeline, TurboModule CodeGen, Bridgeless interop, and the migration flow.

Version and audit

This book is versioned against the React Native version it is **verified against**, not against an internal version. The current verified-against version is `v0.88.1+main` (RN `main` HEAD `9c7b0f5fafd` as of 2026-10-29). See `CHANGELOG.md` for the per-version delta.

Contributing

Pull requests are welcome. If you find a factual error against a specific RN release, please cite an `<rn-source-path>:<line>` reference or a CHANGELOG entry. The book aims to be source-grounded; PR descriptions should follow that bar.

[1] The New Architecture is hard-wired on in the current React Native source. On iOS, `NewArchitectureHelper.new_arch_enabled` in `packages/react-native/scripts/cocoapods/new_architecture.rb` returns `true` unconditionally. On Android, `ProjectUtils.isNewArchEnabled()` always returns `true`, and the Gradle plugin at `packages/gradle-plugin/react-native-gradle-plugin/src/main/kotlin/com/facebook/react/ReactRootProjectPlugin.kt` rejects `newArchEnabled=false` (unsupported since React Native 0.82). TurboModules are likewise always on: the `useTurboModules` feature flag was removed in v0.87.0-rc.0. `RCTTurboModuleEnabled()` still exists in v0.88.1 and returns `YES` (`RCTBridge.mm`). Commit `01d55273bcf` deletes it on `main`. The default was originally flipped for new apps in v0.76.0 (2024-10-23).

Part 1: Core Architecture

Chapter 1: Introduction - The "Why"

The React Native New Architecture is arguably the most significant evolution in the framework's history. While it became the default in the React Native 0.76 release in October 2024[2], its story begins much earlier. The vision was first shared with the community in a 2018 presentation at React Conf[3]. This marked the beginning of a multi-year effort to fundamentally re-imagine the communication layer between JavaScript and the native host platform.

Important Update (2026): As of React Native 0.76, the New Architecture is enabled by default for all new projects. As of React Native 0.82, **opt-out was removed entirely**: `newArchEnabled=false` on Android and `RCT_NEW_ARCH_ENABLED=0` on iOS are force-overridden by the build tooling, and `RCTIsNewArchEnabled()` is hard-wired to `YES`. TurboModules are likewise always on. The always-on `RCTTurboModuleEnabled()` flag function existed through v0.88.1 and was deleted on main after it (commit `01d55273bcf`, 2026-10-27). iOS builds default to stripping legacy Bridge code via `RCT_REMOVE_LEGACY_ARCH=1`; separately, legacy module / view-manager **interop** can be compiled out behind `RCT_REMOVE_LEGACY_MODULE_INTEROP` / `RCT_REMOVE_LEGACY_COMPONENT_INTEROP` (opt-in macros — OSS apps still ship the shims unless they define them). See Chapter 7 for the migration path; there is no supported New-Arch escape hatch on current releases.

Future Plans: The React Native team is actively deleting the legacy Bridge architecture, not merely deprecating it. The C++ headers under `packages/react-native/ReactCommon/cxxreact/` still carry `[[deprecated("This API will be removed along with the legacy architecture.")]]` on classes such as `JSExecutor`, `Instance`, `ModuleRegistry`, `NativeToJsBridge`, `CxxNativeModule`, `JSIExecutor`, and `JSINativeModules`[4], and many of those symbols are already wrapped in `#ifndef RCT_REMOVE_LEGACY_ARCH` so they disappear from builds that define the flag. On iOS, `RCTArchConfiguratorProtocol` (the old `bridgelessEnabled` / `newArchEnabled` / `turboModuleEnabled` / `fabricEnabled` toggles) has been removed from the public headers (since the v0.87 line), and further legacy symbols such as `RCTCxxBridge` continue to be deleted. Developers still on RN ≤ 0.81 should plan a hard cutover before upgrading further.

This was not a minor refactoring but a deep, foundational change designed to address systemic performance bottlenecks and unlock new capabilities for the framework. To fully appreciate the new architecture, one must first understand the limitations of the system it replaced: the original architecture, centered around a component known as "the Bridge."

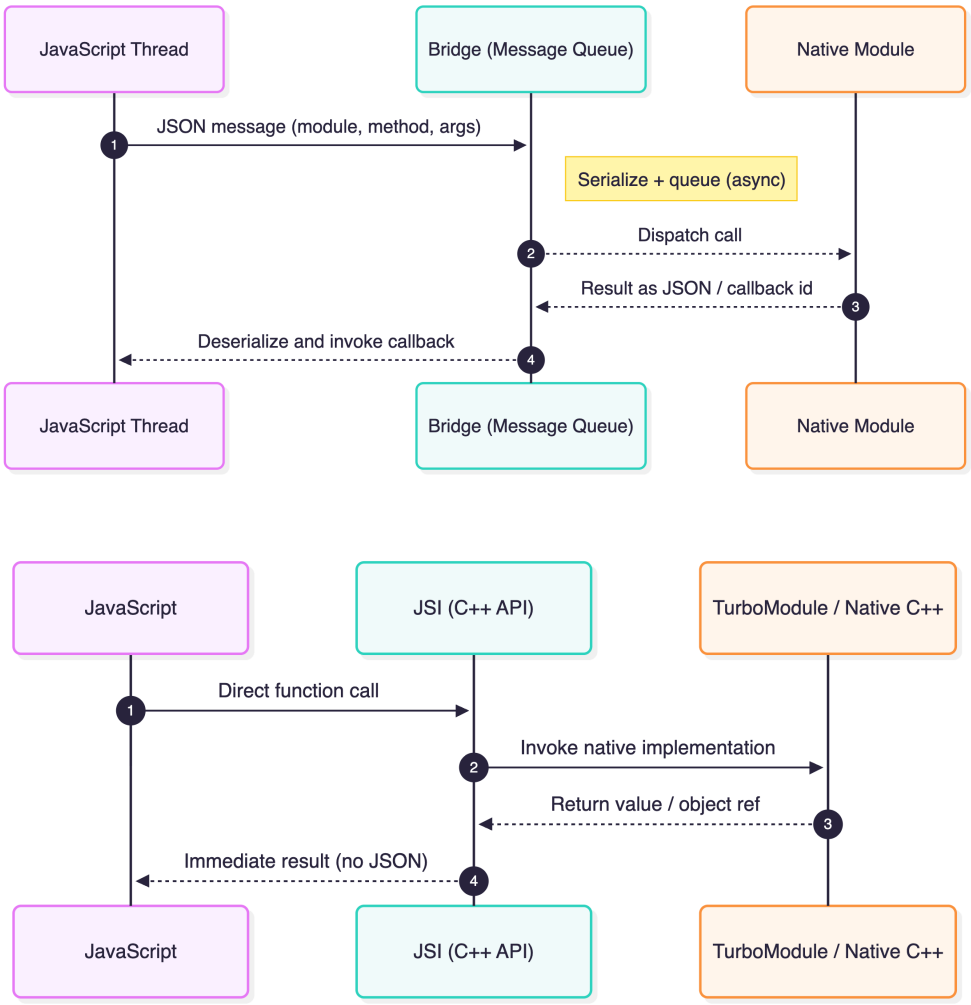
---

[2] "About the New Architecture". React Native Documentation. https://reactnative.dev/architecture/landing-page

[3] Parashuram N, "React Native's New Architecture" (React Conf 2018). YouTube video id `UcqRXTriUVI`. https://www.youtube.com/watch?v=UcqRXTriUVI

[4] C++ deprecation attributes that quote the phrase verbatim, in `packages/react-native/ReactCommon/cxxreact/`: `JSExecutor.h:36,50,58`, `Instance.h:39,46`, `NativeToJsBridge.h:43`, `ModuleRegistry.h:30,35`, `MethodCall.h:22,35`, `CxxNativeModule.h:28,31`, `JSModulesUnbundle.h:20`, `RAMBundleRegistry.h:28`, `NativeModule.h:21,33`, plus `packages/react-native/ReactCommon/jsiexecutor/jsireact/JSIExecutor.h:73` and `JSINativeModules.h:26` (`JSIndexedRAMBundle.h` no longer exists). Run `rg "removed along with the legacy" packages/react-native/` to enumerate. https://github.com/facebook/react-native

Visual Overview: Old vs New



The Old Architecture: A Story of the Bridge

In the original architecture, the entire communication system between the JavaScript thread (where the application's business logic runs) and the Native thread (where the host platform's UI and services run) was funneled through a single, asynchronous message queue: **the Bridge**.

Understanding the Bridge Architecture

To truly understand the limitations, let's examine how the Bridge worked in practice. Imagine a simple scenario where a React Native app needs to get the device's current location:

```
// JavaScript side (pre-0.60 RN; navigator.geolocation was removed from core
// in April 2019 and now lives in @react-native-community/geolocation[^geo]).
navigator.geolocation.getCurrentPosition(
  (position) => {
    console.log('Location:', position.coords);
  },
```

```
  (error) => {
    console.error('Error:', error);
  }
);
```

Here's what happened behind the scenes in the old architecture (note that the shapes below are a conceptual sketch, see the next paragraph for the actual wire format):

1. JavaScript would create a call like `{"module": "LocationObserver", "method": "getCurrentPosition", "args": [], "callbackIDs": [23, 24]}`
2. The call would be serialized into JSON and added to the Bridge queue
3. The native side would eventually process this message (timing unpredictable)
4. Native code would fetch the location (which might take seconds)
5. The result would be serialized: `{"coords": {"latitude": 37.7749, "longitude": -122.4194}, ...}`
6. This JSON would be sent back through the Bridge
7. JavaScript would deserialize and invoke the callback

In the actual implementation, `MessageQueue.js` does not send name-keyed JSON objects. It enqueues integer `moduleID` / `methodID` pairs into three parallel arrays (`MODULE_IDS`, `METHOD_IDS`, `PARAMS`) and flushes them as a tuple `[[moduleIDs], [methodIDs], [params], callID]`. Callback ids are encoded into the params array by bit-shifting `_callID` (failure: `callID << 1`; success: `(callID << 1) | 1`)[5]. The point of the conceptual JSON shape above is the round-trip: every call serializes, crosses an async queue, and gets re-parsed on the other side. The new architecture removes that.

This design, while functional, had several inherent limitations that became more pronounced as React Native applications grew in complexity:

1. Asynchronous Communication: The Latency Problem

Every call from JavaScript to a native module was, by necessity, asynchronous. This created several challenging scenarios:

Example: Synchronous Storage Access

```
// What developers wanted (but couldn't have):
const token = Storage.getItem('authToken'); // ❌ Not possible
if (token) {
  api.authenticate(token);
}

// What they had to do instead:
Storage.getItem('authToken', (error, token) => { // ✅ Forced async
  if (token) {
```

[5] `packages/react-native/Libraries/BatchedBridge/MessageQueue.js`, lines 33-35 (the `MODULE_IDS=0; METHOD_IDS=1; PARAMS=2;` constants), 140 and 324 (`this._queue = [[], [], [], this._callID]`), 253-254 (`this._queue[MODULE_IDS].push(moduleID); this._queue[METHOD_IDS].push(methodID);`), 227-231 (callback id encoding via `_callID << 1`).

```
    api.authenticate(token);
  }
});
```

This forced asynchronicity made certain patterns impossible:

- Synchronous getters for UI properties
- Immediate validation of native state
- Real-time audio/video processing
- High-frequency sensor data reading

2. Serialization Overhead: The Performance Killer

All data sent across the Bridge had to be serialized into JSON strings. Let's examine the actual cost:

Example: Sending an Image Buffer

```
// Attempting to send a 1MB image buffer
const imageData = new Uint8Array(1048576); // 1MB of pixel data

// Old Architecture: This would require:
// 1. Convert Uint8Array to regular array (expensive!)
// 2. JSON.stringify the array (very expensive!)
// 3. Send the resulting ~3MB JSON string across the Bridge
// 4. Parse the JSON on native side (expensive!)
// 5. Convert back to native data structure

// Rough overhead (illustrative, not measured): ~3-5x the original data size
// after JSON encoding, plus parse cost on the receiving side. Wall-clock
// figures for a 1MB buffer vary widely by device and aren't quoted in the
// official docs; the reactnative.dev landing page only describes the
// qualitative cost ("removes the asynchronous bridge... and replaces it
// with JSI", "no serialization costs")[^2].
```

The serialization overhead was particularly devastating for:

- **Large Data Sets**: Sending arrays of complex objects
- **Binary Data**: Images, audio buffers, video frames
- **Frequent Updates**: Animation values, gesture tracking
- **Complex Objects**: Nested structures with circular references

3. Congestion and Bottlenecks: The Single Lane Highway

The Bridge acted as a single lane of traffic. Here's a real-world scenario that would cause severe performance issues:

Example: Gesture-Driven Animation

```
// Old Architecture: Each gesture update goes through the Bridge
const panResponder = PanResponder.create({
  onPanResponderMove: (event, gestureState) => {
    // This fires 60+ times per second during a pan gesture
    // Each update must be serialized and sent through the Bridge
```

```
    Animated.event([
      null,
      { dx: this.pan.x, dy: this.pan.y }
    ])(event, gestureState);
  }
});
```

During a single swipe gesture lasting one second:

- 60+ messages queued on the Bridge
- Each message ~200 bytes when serialized
- Total data: ~12KB just for one gesture
- Result: 16-33ms latency per frame (missing 60fps target)

The Compound Effect

These limitations compounded in real applications. Consider a social media feed with videos:

```
// Old Architecture nightmare scenario
const VideoFeed = () => {
  const [videos, setVideos] = useState([]);

  // Problem 1: Large data serialization
  useEffect(() => {
    fetchVideos().then(data => {
      // Serializing array of video metadata (potentially MBs)
      setVideos(data);
    });
  }, []);

  // Problem 2: High-frequency updates
  const onScroll = (event) => {
    // Fires continuously during scroll
    videos.forEach((video, index) => {
      // Each visibility check goes through the Bridge
      checkVideoVisibility(video.id, event.contentOffset);
    });
  };

  // Problem 3: Async coordination
  const playVideo = async (videoId) => {
    // Multiple async Bridge calls needed
    await VideoModule.load(videoId);
    await VideoModule.prepare();
    await VideoModule.play();
    // Total latency: 50-150ms before video starts
  };
};
```

The result was a poor user experience:

- Janky scrolling due to Bridge congestion

- Delayed video playback
- Dropped frames during animations
- High memory usage from serialization overhead

The Vision: A New Foundation for React Native

The vision for the New Architecture was to tear down these limitations and create a more performant, tightly integrated, and capable framework. The core goals, as articulated by the React Native team and community collaborators, were:

1. Performance: Direct Communication Without Serialization

The new architecture enables direct, synchronous communication between JavaScript and native platforms. Here's what this means in practice (the `DeviceInfo` methods below are illustrative shapes; battery APIs aren't in core React Native, they live in community modules like `react-native-device-info`):

Before (Bridge Architecture):

```
// Getting device info required async callbacks.
DeviceInfo.getBatteryLevel((level) => {
  if (level < 0.2) {
    // This warning appears 50-100ms later.
    showLowBatteryWarning();
  }
});
```

After (New Architecture):

```
// Direct synchronous access from a TurboModule spec method.
const batteryLevel = DeviceInfo.getBatteryLevelSync();
if (batteryLevel < 0.2) {
  // Immediate response.
  showLowBatteryWarning();
}
```

The performance improvements vary by workload. Order-of-magnitude figures often quoted for the new architecture (illustrative, not from an official Meta benchmark):

- **Method Calls**: synchronous sub-millisecond TurboModule calls vs. async-queued Bridge calls (often 10-100x faster end-to-end for what used to be a callback round-trip).
- **Data Transfer**: skipping JSON encode/decode is a 5-20x win for large payloads, and the win scales with payload size.
- **Memory Usage**: removing the serialized copy roughly halves working-set memory on sustained-large-payload paths.

For actual measured numbers, see the React Native team's renderer-performance write-up linked from the architecture landing page[6].

[6] "About the New Architecture". React Native Documentation. https://reactnative.dev/architecture/landing-page

2. Type Safety: Compile-Time Contract Enforcement

The new architecture introduces strict type safety across the JavaScript-Native boundary:

Before (Runtime Errors):

JavaScript call site:

```
NativeModules.MyModule.processUser({
  name: "John",
  age: "25", // ❌ Wrong type - will crash at runtime
});
```

Native (iOS) handler with the old `RCT_EXPORT_METHOD` macro from `RCTBridgeModule.h`[7]:

```
RCT_EXPORT_METHOD(processUser:(NSDictionary *)user) {
  NSNumber *age = user[@"age"]; // Crash: NSString cannot be cast to NSNumber.
}
```

After (Compile-Time Safety):

TypeScript spec (the codegen reads this to emit the native interface)[8]:

```
import type {TurboModule} from 'react-native';
import {TurboModuleRegistry} from 'react-native';

export interface Spec extends TurboModule {
  processUser(user: { name: string; age: number }): void;
}

export default TurboModuleRegistry.getEnforcing<Spec>('MyModule');
```

Generated Obj-C++ interface (the actual codegen output shape is in `packages/react-native-codegen/src/generators/modules/__tests__/__snapshots__/GenerateModuleHObjCpp-test.js.snap`)[9]:

```
- (void)processUser:(JS::NativeMyModule::SpecProcessUserUser &)user {
  // Type-safe access with compile-time checking. Note: in Obj-C++ codegen,
  // string fields are NSString *, not std::string. The std::string return
  // shape is what the C++-only ("Cxx") TurboModule codegen emits.
  NSString *name = user.name();
```

[7] `packages/react-native/React/Base/RCTBridgeModule.h:263` defines `#define RCT_EXPORT_METHOD(method) RCT_REMAP_METHOD(, method)`. The macro is annotated with `RCT_DEPRECATED` on the surrounding `methodQueue` property and is part of the legacy-arch surface scheduled for removal.

[8] `packages/react-native/types_DEPRECATED/Libraries/TurboModule/RCTExport.d.ts:10-12` defines `export interface TurboModule { getConstants?(): {}; }`. A representative spec is `packages/rn-tester/js/examples/TurboModule/NativeSampleTurboModule.js:26` (`export interface Spec extends TurboModule { ... }`).

[9] Codegen snapshot at `packages/react-native-codegen/src/generators/modules/__tests__/__snapshots__/GenerateModuleHObjCpp-test.js.snap`, lines 855-862 (`NSString *name() const; ... double id_() const;`) and 978 (`inline NSString *JS::NativeExceptionsManager::ExceptionData::name() const`). String fields in Obj-C++ codegen are `NSString *`, never `std::string`. The C++-only ("Cxx") variant emits `std::string` instead; see `GenerateModuleH-test.js.snap`.

```
  double age = user.age(); // Guaranteed to be a JS number → double.
}
```

3. Concurrent Rendering: Modern React Features

The new architecture supports React 18's concurrent features and, at the current React Native HEAD, the React 19 peer dependency (`"react": "^19.3.0"` in `packages/react-native/package.json`)[10]:

Automatic Batching Example:

```
// Old Architecture: Multiple renders.
function handleClickOld() {
  setCount(c => c + 1); // Render 1
  setFlag(f => !f);     // Render 2
  setValue(v => v * 2); // Render 3
}

// New Architecture: Single render with automatic batching.
function handleClickNew() {
  setCount(c => c + 1); // }
  setFlag(f => !f);     // } All batched into one render
  setValue(v => v * 2); // }
}
```

Transitions for Non-Urgent Updates:

```
function SearchResults() {
  const [query, setQuery] = useState('');
  const [results, setResults] = useState([]);
  const [isPending, startTransition] = useTransition();

  function handleSearch(input) {
    setQuery(input); // Urgent: update input immediately

    startTransition(() => {
      // Non-urgent: can be interrupted for user input
      const filtered = searchDatabase(input);
      setResults(filtered);
    });
  }

  return (
    <>
      <SearchInput value={query} onChange={handleSearch} />
      {isPending && <Spinner />}
      <ResultsList results={results} />
    </>
```

[10] `packages/react-native/package.json:143-146`: `"peerDependencies": { ..., "react": "^19.3.0" }`. The bundled reconciler in this build line targets React 19.

```
  );
}
```

4. Improved Interoperability: Seamless Native Integration

The new architecture makes it trivial to expose complex native objects to JavaScript:

Real-World Example: Native Database Access

```
// C++ Implementation
class DatabaseHostObject : public jsi::HostObject {
public:
  jsi::Value get(jsi::Runtime& rt, const jsi::PropNameID& name) override {
    if (name.utf8(rt) == "query") {
      return jsi::Function::createFromHostFunction(rt,
        jsi::PropNameID::forAscii(rt, "query"), 1,
        [this](jsi::Runtime& rt, const jsi::Value& thisVal,
               const jsi::Value* args, size_t count) -> jsi::Value {
          // Direct SQL execution without serialization
          std::string sql = args[0].getString(rt).utf8(rt);
          auto results = database_.executeQuery(sql);

          // Return results directly - no JSON conversion
          return convertResultsToJSI(rt, results);
        });
    }
    return jsi::Value::undefined();
  }

private:
  SQLiteDatabase database_;
};
```

JavaScript Usage:

```
// Direct access to native database
const results = db.query("SELECT * FROM users WHERE active = 1");
// Results are immediately available, no serialization needed
console.log(`Found ${results.length} active users`);
```

5. React 18 and Beyond: Future-Proof Architecture

The new architecture enables cutting-edge React features. Note that `use(promise)` was experimental/canary in React 18 and became stable in React 19. Since React Native main now pins React 19.3+, the example below works out of the box[11]:

Suspense for Data Fetching:

```
function UserProfile({ userId }) {
  // This suspends until user data is ready.
```

[11] `packages/react-native/package.json:143-146`: `"peerDependencies": { ..., "react": "^19.3.0" }`. The bundled reconciler in this build line targets React 19.

```
  const user = use(fetchUser(userId));

  return (
    <View>
      <Text>{user.name}</Text>
      <Image source={{ uri: user.avatar }} />
    </View>
  );
}

function App() {
  return (
    <Suspense fallback={<LoadingSpinner />}>
      <UserProfile userId={123} />
    </Suspense>
  );
}
```

Server Components (Future):

```
// This could run on the server in the future
async function ServerUserList() {
  const users = await db.query("SELECT * FROM users");

  return (
    <View>
      {users.map(user => (
        <UserCard key={user.id} user={user} />
      ))}
    </View>
  );
}
```

The Three Pillars of the New Architecture

To achieve this vision, the team built the new architecture on three core pillars, which the subsequent chapters of this report will explore in detail:

The JavaScript Interface (JSI)

The new foundational layer that replaces the Bridge, allowing for direct, synchronous method calls between the two realms. JSI is a lightweight C++ API (header at `packages/react-native/ReactCommon/jsi/jsi/jsi.h`, around 2,200 lines)[12] that:

- Lets JavaScript hold direct references to C++ objects, and vice versa, via primitives like `HostObject`, `HostFunction`, and `NativeState`. The reactnative.dev landing page describes this as "an interface that allows JavaScript to hold a reference to a C++ object and vice-versa"[13].

[12] `packages/react-native/ReactCommon/jsi/jsi/jsi.h` (~2,266 lines). Includes only the C++ standard library. Engine bindings live in sibling directories (`hermes/`, `jsc/`).

[13] "About the New Architecture". React Native Documentation. https://reactnative.dev/architecture/landing-page

This avoids the serialization round-trip; it does not literally share heap memory between the JS engine and native code.
- Offers zero-copy paths for binary buffers (`ArrayBuffer::data()` returns a raw pointer to engine-managed bytes) and for host objects (a C++ object directly addressable from JS). Other accessors like `String::utf8(rt)` still copy.
- Is itself synchronous (a `HostFunction` returns a `Value`); asynchronous work happens by returning a JS `Promise` or by hopping threads via `CallInvoker`.
- Is engine-agnostic by design: `jsi.h` has no engine-specific includes, only standard headers. In-tree React Native ships JSI bindings for Hermes (`ReactCommon/hermes/executor/HermesExecutorFactory.{h,cpp}`) and JavaScriptCore (`ReactCommon/jsc/JSCRuntime.{h,cpp}`); V8 support exists as a separate community project, `react-native-v8`.

Fabric: The New Rendering System

The modern UI renderer that uses JSI to create a more responsive and efficient UI layer. Fabric:

- Performs layout calculations in C++ for cross-platform consistency (`ReactCommon/yoga/` is the C++ Yoga implementation, and `ReactCommon/react/renderer/components/` holds the Fabric component descriptors).
- Exposes synchronous layout measurements via the C++ `UIManager`. `LayoutMetrics UIManager::getRelativeLayoutMetrics(...) const` is the accessor at `ReactCommon/react/renderer/uimanager/UIManager.h`[14].
- Supports React 18 and React 19's concurrent renderer.
- Provides tighter integration with native gesture systems via the synchronous mount path.

TurboModules: Next-Gen Native Modules

The evolution of native modules, built on JSI, which are loaded on-demand and can be invoked synchronously from JavaScript. TurboModules:

- Load lazily. `TurboModuleRegistry.get(name)` and `getEnforcing(name)` route through `global.__turboModuleProxy(name)` (defined in `packages/react-native/Libraries/TurboModule/TurboModuleRegistry.js`), which resolves a module the first time JS asks for it[15].
- Support synchronous method calls when the spec declares a synchronous return type; methods declared as returning `Promise<T>` stay asynchronous.
- Provide compile-time type safety on both sides of the boundary, generated by codegen from a TS or Flow spec.
- Improve JS-side dead-code elimination: each module is imported by name from its own spec file, so an unused spec drops out of the bundle. The native binary still contains whatever modules are registered.

---

[14] `packages/react-native/ReactCommon/react/renderer/uimanager/UIManager.h:189-192: LayoutMetrics getRelativeLayoutMetrics(const ShadowNode &shadowNode, const ShadowNode *ancestorShadowNode, LayoutableShadowNode::LayoutInspectingPolicy policy) const;`.

[15] `packages/react-native/Libraries/TurboModule/TurboModuleRegistry.js:17-46. turboModuleProxy = global.__turboModuleProxy`, then `requireModule<T>(name)` calls `turboModuleProxy(name)` on demand.

Citations:

- [1] Parashuram N, "React Native's New Architecture" (React Conf 2018). YouTube video id `UcqRXTriUVI`. https://www.youtube.com/watch?v=UcqRXTriUVI
- [2] "About the New Architecture". React Native Documentation. https://reactnative.dev/architecture/landing-page
- [3] C++ deprecation attributes that quote the phrase verbatim, in `packages/react-native/ReactCommon/cxxreact/`: `JSExecutor.h:36,50,58`, `Instance.h:39,46`, `NativeToJsBridge.h:43`, `ModuleRegistry.h:30,35`, `MethodCall.h:22,35`, `CxxNativeModule.h:28,31`, `JSModulesUnbundle.h:20`, `RAMBundleRegistry.h:28`, `NativeModule.h:21,33`, plus `packages/react-native/ReactCommon/jsiexecutor/jsireact/JSIExecutor.h:73` and `JSINativeModules.h:26` (`JSIndexedRAMBundle.h` no longer exists). Run `rg "removed along with the legacy" packages/react-native/` to enumerate. https://github.com/facebook/react-native
- [3a] Obj-C variant in `packages/react-native/Libraries/AppDelegate/`: `RCTAppDelegate.h` and `RCTReactNativeFactory.h` still carry "removed when removing the legacy architecture" deprecations on bridge-era APIs. `RCTArchConfiguratorProtocol.h` itself was deleted from the public headers on the v0.87 line (commits `2148f15db5` / `ad9936faf6`).
- [geo] `navigator.geolocation` was removed from React Native core in three commits in April 2019: `45bd2b514b5` ("Remove navigator.geolocation, use Geolocation", 2019-04-01), `9834c580af6` ("Move Geolocation JS code to FB internal", 2019-04-01), and `17dbf988845` ("Move iOS Geolocation code out from the repo", 2019-04-02). Recommended replacement: `@react-native-community/geolocation`.
- [bridge-wire] `packages/react-native/Libraries/BatchedBridge/MessageQueue.js`, lines 33-35 (the `MODULE_IDS=0; METHOD_IDS=1; PARAMS=2;` constants), 140 and 324 (`this._queue = [[], [], [], this._callID]`), 253-254 (`this._queue[MODULE_IDS].push(moduleID); this._queue[METHOD_IDS].push(methodID);`), 227-231 (callback id encoding via `_callID << 1`).
- [rct-export] `packages/react-native/React/Base/RCTBridgeModule.h:263` defines `#define RCT_EXPORT_METHOD(method) RCT_REMAP_METHOD(, method)`. The macro is annotated with `RCT_DEPRECATED` on the surrounding `methodQueue` property and is part of the legacy-arch surface scheduled for removal.
- [tm-spec] `packages/react-native/types_DEPRECATED/Libraries/TurboModule/RCTExport.d.ts:10-12` defines `export interface TurboModule { getConstants?(): {}; }`. A representative spec is `packages/rn-tester/js/examples/TurboModule/NativeSampleTurboModule.js:26` (`export interface Spec extends TurboModule { ... }`).
- [tm-codegen] Codegen snapshot at `packages/react-native-codegen/src/generators/modules/__tests__/__snapshots__/GenerateModuleHObjCpp-test.js.snap`, lines 855-862 (`NSString *name() const; ... double id_() const;`) and 978 (`inline NSString *JS::NativeExceptionsManager::ExceptionData::name() const`). String fields in Obj-C++ codegen are `NSString *`, never `std::string`. The C++-only ("Cxx") variant emits `std::string` instead; see `GenerateModuleH-test.js.snap`.

- [react-pin] `packages/react-native/package.json:143-146`: `"peerDependencies": { ..., "react": "^19.3.0" }`. The bundled reconciler in this build line targets React 19.
- [tm-registry] `packages/react-native/Libraries/TurboModule/TurboModuleRegistry.js:17-46`. `turboModuleProxy = global.__turboModuleProxy`, then `requireModule<T>(name)` calls `turboModuleProxy(name)` on demand.
- [jsi-header] `packages/react-native/ReactCommon/jsi/jsi/jsi.h` (~2,266 lines). Includes only the C++ standard library. Engine bindings live in sibling directories (`hermes/`, `jsc/`).
- [fabric-measure] `packages/react-native/ReactCommon/react/renderer/uimanager/UIManager.h:189-192`: `LayoutMetrics getRelativeLayoutMetrics(const ShadowNode &shadowNode, const ShadowNode *ancestorShadowNode, LayoutableShadowNode::LayoutInspectingPolicy policy) const;`.

Chapter 2: JSI - The Foundation

At the very heart of the New Architecture lies the **JavaScript Interface (JSI)**. It is the foundational layer that replaces the old, asynchronous Bridge and makes the performance and interoperability goals of the new architecture possible. The JSI is not a specific implementation but rather a lightweight, engine-agnostic C++ API that allows for direct, synchronous, and bidirectional communication between JavaScript and a native language.

Its definition can be found in the repository here: `packages/react-native/ReactCommon/jsi/jsi/jsi.h`[16]. Understanding the JSI is crucial because every other component of the New Architecture—Fabric, TurboModules, and even CodeGen—is built upon it.

Status as of v0.88 (`main`, commit `9c7b0f5fafd`, v0.88.1 stable): JSI is the only communication layer between JS and native code in the New Architecture, and the New Architecture has been **on by default since React Native 0.76 (October 2024)**[17] with **opt-out removed since 0.82**. JSI itself has been the path for Hermes since long before 0.76. What 0.76 actually flipped on by default is the *stack built on JSI*: Fabric, TurboModules, and Bridgeless. On the v0.87 and v0.88 lines TurboModules are always enabled, and the legacy Bridge continues to be deleted / compile-gated out of builds.

A note on terminology before we go further: in older RN versions the public C++ class users worked against was `jsi::Runtime`. The current source promotes `jsi::IRuntime` as the abstract interface and keeps `jsi::Runtime` as a backwards-compatible class that inherits from it[18]. Either type works in user code, because `Runtime : public IRuntime`. The snippets in this chapter use `jsi::Runtime&` because that's the form most existing TurboModule and Vision Camera code uses, but you'll see `IRuntime&` in the upstream headers.

Visual: JSI Call Path and Data Flow



The Core Principles of JSI

The JSI was designed to solve the core problems of the Bridge: serialization overhead and asynchronous communication. It achieves this through a few key principles:

---

[16] `packages/react-native/ReactCommon/jsi/jsi/jsi.h` (HEAD `9c7b0f5fafd`, v0.88.1 era). The full IRuntime / Runtime interface lives at lines 338-822; HostObject at 222-251; HostFunctionType at 217-218; Value at 1766-2078. Since the last refresh `IRuntime` gained `createError`, `createTypeError` and the other typed error factories (jsi.h:662-675) and `length(const String&)` (jsi.h:681). `JSError(rt, std::string)` now builds its value through `rt.createError` (jsi.cpp:968-970), and `Value::toString` uses `asString` so a non-string result throws (jsi.cpp:924-926).

[17] React Native 0.76.0 release blog, "New Architecture by default" (2024-10-23): https://reactnative.dev/blog/2024/10/23/release-0.76-new-architecture. CHANGELOG entry for v0.76 in `CHANGELOG-0.7x.md`. JSI itself shipped much earlier (experimental in 0.59, default on the Hermes path long before 0.76). What 0.76 turned on by default was the Fabric + TurboModules + Bridgeless stack built on JSI.

[18] `IRuntime` is the abstract interface (jsi.h:338-685); `Runtime : public IRuntime` keeps the older surface alive for backward compatibility (jsi.h:705-822). The header docs at 332-337 explicitly state "for backward compatibility, these APIs are also accessible via the Runtime object directly".

1. Direct References, Not Messages

Instead of serializing data into JSON strings and sending them across a message queue, JSI lets JavaScript and native code hold **handles** to each other's objects (`jsi::Object`, `jsi::HostObject`, `jsi::NativeState`) without a Bridge round-trip. That is not the same as a shared process heap for every value — accessors like `String::utf8(rt)` still allocate copies, while paths such as `ArrayBuffer::data()` can expose engine-managed bytes without a JSON marshal.

Bridge Architecture (Old):

```
// JavaScript creates an object
const data = { users: [{ id: 1, name: "Alice" }, { id: 2, name: "Bob" }] };

// Bridge serializes to JSON string: '{"users":[{"id":1,"name":"Alice"},
{"id":2,"name":"Bob"}]}'
// ~100 bytes sent across Bridge
// Native deserializes JSON back to NSDictionary/HashMap
```

JSI Architecture (New):

```
// Hold a JSI handle to a JS object (or expose a HostObject into JS).
// Property access is a direct C++ call — no Bridge JSON message —
// but the engine may still copy when you materialize strings/values.
jsi::Object data = /* ... reference to JS object ... */;
auto name = data.getProperty(rt, "name").asString(rt).utf8(rt); // may copy
```

2. Synchronous by Default

With direct references, JavaScript can invoke a method on a native object (and vice-versa) and get a result back immediately, within the same thread of execution.

Real-World Impact:

```
// Old Bridge - Forced Asynchronous Pattern
class LocationManager {
  async getCurrentLocation() {
    return new Promise((resolve) => {
      NativeModules.Location.getCurrentPosition((location) => {
        resolve(location); // ~50-100ms later
      });
    });
  }
}

// New JSI - Synchronous When Needed
class LocationManager {
  getCurrentLocationSync() {
    // Direct synchronous call - immediate result
    return NativeLocation.getCurrentPositionSync();
  }

  // Still supports async for long operations
```

```
  async getCurrentLocationAsync() {
    return NativeLocation.getCurrentPositionAsync();
  }
}
```

3. C++ as the Lingua Franca

The JSI itself is defined as a C++ abstract interface. This allows React Native to interact with any JavaScript engine (Hermes, V8, JavaScriptCore) simply by providing a C++ implementation that conforms to the JSI standard.

Engine Agnostic Design:

```
// JSI works with any JavaScript engine
namespace facebook {
namespace jsi {
  // Abstract interface - no engine-specific code
  class Runtime {
  public:
    virtual Value evaluateJavaScript(
      const std::shared_ptr<const Buffer>& buffer,
      const std::string& sourceURL) = 0;
    // ... other virtual methods
  };
}
}

// Engine-specific implementations
class HermesRuntime : public jsi::Runtime { /* Hermes implementation */ };
class V8Runtime : public jsi::Runtime { /* V8 implementation */ };
class JSCRuntime : public jsi::Runtime { /* JavaScriptCore implementation */ };
```

The Key JSI C++ Components

A deep dive into `jsi.h` reveals the primary C++ classes and concepts that constitute the JSI API.

1. `jsi::Runtime` - The Execution Context

The `jsi::Runtime` class is the central nervous system of the JSI. It represents an instance of a JavaScript VM and is the primary entry point for all JSI operations.

```
// Key Runtime / IRuntime operations (selected; see jsi.h:332-685 for the full
set).
// All of these are pure virtual on IRuntime; engine implementations (Hermes,
// JSC) override them.
class IRuntime /* : public ICast */ {
public:
  // Execute JavaScript code.
  virtual Value evaluateJavaScript(const std::shared_ptr<const Buffer>& buffer,
                                   const std::string& sourceURL) = 0;

  // Access the global object.
  virtual Object global() = 0;
```

```
  // Parse a UTF-8 JSON document into a Value (default impl uses JSON.parse).
  virtual Value createValueFromJsonUtf8(const uint8_t* json, size_t length) = 0;

  // Type checks on Objects.
  virtual bool isArray(const Object&) const = 0;
  virtual bool isFunction(const Object&) const = 0;

  // Microtask integration (per ECMA-262 Job queue).
  virtual void queueMicrotask(const jsi::Function& callback) = 0;
  virtual bool drainMicrotasks(int maxMicrotasksHint = -1) = 0;

  // ... many more operations
};

// To copy a Value across runtime calls, use the Value's two-arg constructor.
// There is no `cloneValue` method on Runtime; the engine-internal clone
// helpers (cloneSymbol, cloneString, cloneObject, ...) are per-PointerValue
// and not part of the user-facing surface. Cloning a Value:
jsi::Value copy(rt, original); // calls Value(IRuntime&, const Value&)
```

Practical Example - Evaluating JavaScript:

```
void executeScript(jsi::Runtime& rt) {
  // Create JavaScript code
  const char* code = R"(
    function greet(name) {
      return `Hello, ${name}!`;
    }
    greet('React Native');
  )";

  // Evaluate the code
  auto result = rt.evaluateJavaScript(
    std::make_shared<jsi::StringBuffer>(code),
    "greeting.js"
  );

  // result now contains "Hello, React Native!"
  std::string greeting = result.getString(rt).utf8(rt);
}
```

2. `jsi::HostObject` - The Bridge Between Worlds

The `jsi::HostObject` is the most powerful feature of the JSI. It allows you to expose a C++ object to the JavaScript world as if it were a regular JavaScript object.

```
// Complete HostObject interface (jsi.h:222-251).
class JSI_EXPORT HostObject {
public:
  virtual ~HostObject();
```

```
  // Called when JS accesses a property: obj.prop
  // Default impl returns Value::undefined() (jsi.cpp:276-278).
  virtual Value get(Runtime& rt, const PropNameID& name);

  // Called when JS sets a property: obj.prop = value
  // IMPORTANT: the default impl THROWS a TypeError-shaped JSError
  // ("Cannot assign to property '<name>' on HostObject with default setter",
  // jsi.cpp:280-285). If you want writes to silently no-op you must override
  // and do nothing; the default does NOT no-op.
  virtual void set(Runtime& rt, const PropNameID& name, const Value& value);

  // Return all available properties.
  // Default impl returns an empty vector (jsi.cpp:954+).
  virtual std::vector<PropNameID> getPropertyNames(Runtime& rt);
};
```

Real-World Example - Native File System Access:

```
class FileSystemHostObject : public jsi::HostObject {
private:
  std::string basePath_;

public:
  FileSystemHostObject(const std::string& basePath) : basePath_(basePath) {}

  jsi::Value get(jsi::Runtime& rt, const jsi::PropNameID& name) override {
    auto propName = name.utf8(rt);

    if (propName == "readFile") {
      return jsi::Function::createFromHostFunction(rt,
        jsi::PropNameID::forAscii(rt, "readFile"), 1,
        [this](jsi::Runtime& rt, const jsi::Value&, const jsi::Value* args,
size_t) {
          if (args[0].isString()) {
            std::string filename = args[0].getString(rt).utf8(rt);
            std::string content = readFileSync(basePath_ + "/" + filename);
            return jsi::String::createFromUtf8(rt, content);
          }
          throw jsi::JSError(rt, "Filename must be a string");
        });
    }

    if (propName == "writeFile") {
      return jsi::Function::createFromHostFunction(rt,
        jsi::PropNameID::forAscii(rt, "writeFile"), 2,
        [this](jsi::Runtime& rt, const jsi::Value&, const jsi::Value* args,
size_t) {
          std::string filename = args[0].getString(rt).utf8(rt);
          std::string content = args[1].getString(rt).utf8(rt);
```

```
          writeFileSync(basePath_ + "/" + filename, content);
          return jsi::Value::undefined();
        });
    }

    if (propName == "listFiles") {
      return jsi::Function::createFromHostFunction(rt,
        jsi::PropNameID::forAscii(rt, "listFiles"), 0,
        [this](jsi::Runtime& rt, const jsi::Value&, const jsi::Value*, size_t) {
          auto files = listDirectory(basePath_);
          auto array = jsi::Array(rt, files.size());
          for (size_t i = 0; i < files.size(); i++) {
            array.setValueAtIndex(rt, i, jsi::String::createFromUtf8(rt,
files[i]));
          }
          return array;
        });
    }

    return jsi::Value::undefined();
  }

  void set(jsi::Runtime& rt, const jsi::PropNameID& name, const jsi::Value&
value) override {
    // Could implement property setters here
    throw jsi::JSError(rt, "FileSystem properties are read-only");
  }
};

// Usage in JavaScript:
// const files = fs.listFiles();
// const content = fs.readFile('data.json');
// fs.writeFile('output.txt', 'Hello World');
```

3. `jsi::Function` and `HostFunctionType`

To expose a single native function to JavaScript, you use `jsi::Function::createFromHostFunction`.

```
// HostFunctionType signature
using HostFunctionType = std::function<
    Value(Runtime& rt, const Value& thisVal, const Value* args, size_t count)>;
```

Practical Example - High-Performance Math Library:

```
void installMathModule(jsi::Runtime& rt) {
  auto mathObj = jsi::Object(rt);

  // Fast matrix multiplication
  auto matrixMultiply = jsi::Function::createFromHostFunction(rt,
    jsi::PropNameID::forAscii(rt, "matrixMultiply"), 2,
```

```
    [](jsi::Runtime& rt, const jsi::Value& thisVal,
       const jsi::Value* args, size_t count) -> jsi::Value {

      if (count != 2) {
        throw jsi::JSError(rt, "matrixMultiply expects 2 arguments");
      }

      // Direct access to JavaScript arrays - no serialization!
      auto matrixA = args[0].asObject(rt).asArray(rt);
      auto matrixB = args[1].asObject(rt).asArray(rt);

      size_t rowsA = matrixA.length(rt);
      size_t colsB = matrixB.getValueAtIndex(rt, 0)
                            .asObject(rt).asArray(rt).length(rt);

      // Perform multiplication in C++ for maximum performance
      auto result = jsi::Array(rt, rowsA);

      // ... actual matrix multiplication logic ...

      return result;
    });

  // Fast Fourier Transform
  auto fft = jsi::Function::createFromHostFunction(rt,
    jsi::PropNameID::forAscii(rt, "fft"), 1,
    [](jsi::Runtime& rt, const jsi::Value& thisVal,
       const jsi::Value* args, size_t count) -> jsi::Value {

      auto signal = args[0].asObject(rt).asArray(rt);
      size_t n = signal.length(rt);

      // Perform FFT in C++ - 100x faster than JavaScript
      std::vector<std::complex<double>> data(n);
      for (size_t i = 0; i < n; i++) {
        data[i] = signal.getValueAtIndex(rt, i).asNumber();
      }

      performFFT(data); // Native FFT implementation

      // Return complex number array
      auto result = jsi::Array(rt, n);
      for (size_t i = 0; i < n; i++) {
        auto complexNum = jsi::Object(rt);
        complexNum.setProperty(rt, "real", data[i].real());
        complexNum.setProperty(rt, "imag", data[i].imag());
        result.setValueAtIndex(rt, i, complexNum);
      }
```

```
      return result;
    });

  mathObj.setProperty(rt, "matrixMultiply", matrixMultiply);
  mathObj.setProperty(rt, "fft", fft);

  rt.global().setProperty(rt, "NativeMath", mathObj);
}

// JavaScript usage:
// const result = NativeMath.matrixMultiply([[1,2],[3,4]], [[5,6],[7,8]]);
// const spectrum = NativeMath.fft([1,2,3,4,5,6,7,8]);
```

4. `jsi::Value` - The Universal Data Type

The `jsi::Value` is the currency of the JSI. It is a C++ tagged union that can represent any JavaScript value.

```
// Value type checking and conversion
void processValue(jsi::Runtime& rt, const jsi::Value& value) {
  if (value.isUndefined()) {
    // Handle undefined
  } else if (value.isNull()) {
    // Handle null
  } else if (value.isBool()) {
    bool b = value.getBool();
  } else if (value.isNumber()) {
    double n = value.getNumber();
  } else if (value.isString()) {
    std::string s = value.getString(rt).utf8(rt);
  } else if (value.isObject()) {
    auto obj = value.getObject(rt);

    if (obj.isArray(rt)) {
      auto arr = obj.getArray(rt);
      size_t length = arr.length(rt);

      // Iterate array without serialization
      for (size_t i = 0; i < length; i++) {
        auto element = arr.getValueAtIndex(rt, i);
        // Process element...
      }
    } else if (obj.isFunction(rt)) {
      auto func = obj.getFunction(rt);
      // Can call JavaScript functions from C++!
      auto result = func.call(rt, jsi::Value(42));
    }
  }
}
```

Creating Complex Values:

```
// Create a complex object structure
jsi::Value createUserObject(jsi::Runtime& rt, const User& user) {
  auto userObj = jsi::Object(rt);

  // Basic properties
  userObj.setProperty(rt, "id", jsi::Value(user.id));
  userObj.setProperty(rt, "name", jsi::String::createFromUtf8(rt, user.name));
  userObj.setProperty(rt, "age", jsi::Value(user.age));

  // Nested object
  auto address = jsi::Object(rt);
  address.setProperty(rt, "street", jsi::String::createFromUtf8(rt,
user.street));
  address.setProperty(rt, "city", jsi::String::createFromUtf8(rt, user.city));
  userObj.setProperty(rt, "address", address);

  // Array of tags
  auto tags = jsi::Array(rt, user.tags.size());
  for (size_t i = 0; i < user.tags.size(); i++) {
    tags.setValueAtIndex(rt, i, jsi::String::createFromUtf8(rt, user.tags[i]));
  }
  userObj.setProperty(rt, "tags", tags);

  // Method
  auto greet = jsi::Function::createFromHostFunction(rt,
    jsi::PropNameID::forAscii(rt, "greet"), 0,
    [name = user.name](jsi::Runtime& rt, const jsi::Value&,
                       const jsi::Value*, size_t) -> jsi::Value {
      return jsi::String::createFromUtf8(rt, "Hello, I'm " + name);
    });
  userObj.setProperty(rt, "greet", greet);

  return userObj;
}
```

5. JSI Handles and Move Semantics

JSI types are handles with **move-only semantics** to prevent expensive copies:

```
// ❌ Won't compile - no copy constructor
jsi::Object obj1 = jsi::Object(rt);
jsi::Object obj2 = obj1; // Error!

// ✅ Move semantics
jsi::Object obj1 = jsi::Object(rt);
jsi::Object obj2 = std::move(obj1); // obj1 is now invalid

// ✅ Pass by reference for reading
void readObject(jsi::Runtime& rt, const jsi::Object& obj) {
  auto value = obj.getProperty(rt, "key");
```

```
}

// ✅ Return by move
jsi::Object createObject(jsi::Runtime& rt) {
  auto obj = jsi::Object(rt);
  obj.setProperty(rt, "created", true);
  return obj; // Implicit move
}

// Working with collections
std::vector<jsi::Value> collectValues(jsi::Runtime& rt) {
  std::vector<jsi::Value> values;

  // Must use move to add to vector
  values.push_back(jsi::Value(42));
  values.push_back(jsi::String::createFromUtf8(rt, "hello"));
  values.push_back(jsi::Object(rt));

  return values; // Vector is moved
}
```

6. `jsi::PropNameID` - Optimized Property Access

Property names are interned for performance:

```
// Efficient property access patterns
class OptimizedHostObject : public jsi::HostObject {
private:
  // Pre-intern frequently used property names
  struct PropertyNames {
    jsi::PropNameID width;
    jsi::PropNameID height;
    jsi::PropNameID data;
    jsi::PropNameID transform;

    PropertyNames(jsi::Runtime& rt) :
      width(jsi::PropNameID::forAscii(rt, "width")),
      height(jsi::PropNameID::forAscii(rt, "height")),
      data(jsi::PropNameID::forAscii(rt, "data")),
      transform(jsi::PropNameID::forAscii(rt, "transform")) {}
  };

  std::unique_ptr<PropertyNames> propNames_;

public:
  OptimizedHostObject(jsi::Runtime& rt) {
    propNames_ = std::make_unique<PropertyNames>(rt);
  }

  jsi::Value get(jsi::Runtime& rt, const jsi::PropNameID& name) override {
```

```
    // Fast comparison using interned IDs.
    // NOTE: `PropNameID::compare` is a STATIC member (jsi.h:950) and returns
    // bool (true when equal), not an int. The correct call form is
    // `PropNameID::compare(rt, a, b)`, not `a.compare(rt, b)`.
    if (jsi::PropNameID::compare(rt, name, propNames_->width)) {
      return jsi::Value(1920);
    } else if (jsi::PropNameID::compare(rt, name, propNames_->height)) {
      return jsi::Value(1080);
    } else if (jsi::PropNameID::compare(rt, name, propNames_->data)) {
      // Return large data without serialization
      return getLargeDataArray(rt);
    }

    return jsi::Value::undefined();
  }
};
```

Real-World Example: Vision Camera's Frame Processor (v4)

The power of `jsi::HostObject` is well demonstrated by **v4** of `react-native-vision-camera` (see Chapter 13). Its Frame Processor feature worked by:

1. Capturing a frame of video from the camera in native code.
2. Wrapping the raw pixel buffer in a C++ `Frame` object.
3. Wrapping that `Frame` object in a `FrameHostObject` which inherits from `jsi::HostObject`.
4. Passing this `FrameHostObject` to a JavaScript worklet function.

When the developer's JavaScript code accessed `frame.width`, it synchronously called the `get()` method on the C++ `FrameHostObject`, returning the width with no Bridge JSON overhead. That enabled real-time video analysis in JavaScript — impossible under the legacy Bridge.

Note: Vision Camera **v5** (April 2026) deleted `FrameHostObject` and moved Frame to a Nitro `HybridObject` (still JSI underneath, different authoring model). Treat the `FrameHostObject` sketch above as the v4 teaching example; Chapter 13 covers the v5 Nitro path.

Performance Impact: Vision Camera could process large frame buffers at high FPS with minimal overhead thanks to JSI's direct handle access. This would be impossible with the legacy Bridge's serialization approach.

Advanced JSI Patterns

1. Lazy Property Evaluation

Compute expensive properties only when accessed:

```
class ImageHostObject : public jsi::HostObject {
private:
  std::string imagePath_;
  mutable std::optional<ImageMetadata> cachedMetadata_;
  mutable std::mutex metadataMutex_;
```

```
public:
  jsi::Value get(jsi::Runtime& rt, const jsi::PropNameID& name) override {
    auto propName = name.utf8(rt);

    if (propName == "metadata") {
      // Lazy load and cache metadata
      std::lock_guard<std::mutex> lock(metadataMutex_);
      if (!cachedMetadata_.has_value()) {
        cachedMetadata_ = loadImageMetadata(imagePath_);
      }

      auto meta = jsi::Object(rt);
      meta.setProperty(rt, "width", cachedMetadata_->width);
      meta.setProperty(rt, "height", cachedMetadata_->height);
      meta.setProperty(rt, "format", jsi::String::createFromUtf8(rt,
cachedMetadata_->format));
      return meta;
    }

    if (propName == "pixels") {
      // Return a proxy object that loads pixels on demand
      return createPixelAccessor(rt, imagePath_);
    }

    return jsi::Value::undefined();
  }
};
```

2. Streaming Data with JSI

Handle large datasets without loading everything into memory:

```
class StreamHostObject : public jsi::HostObject {
private:
  std::unique_ptr<DataStream> stream_;

public:
  jsi::Value get(jsi::Runtime& rt, const jsi::PropNameID& name) override {
    auto propName = name.utf8(rt);

    if (propName == "read") {
      return jsi::Function::createFromHostFunction(rt,
        jsi::PropNameID::forAscii(rt, "read"), 1,
        [this](jsi::Runtime& rt, const jsi::Value&,
               const jsi::Value* args, size_t) -> jsi::Value {
          size_t bytes = args[0].asNumber();
          auto data = stream_->read(bytes);

          if (data.empty()) {
            return jsi::Value::null();
          }
```

```
          // NOTE: jsi::ArrayBuffer's only public constructor takes a
          // std::shared_ptr<MutableBuffer> (jsi.h:1556). There is no
          // `ArrayBuffer(rt, size_t)`. To hand JS a writable byte buffer of a
          // chosen size, wrap your bytes in a MutableBuffer subclass:
          struct VectorBuffer : jsi::MutableBuffer {
            std::vector<uint8_t> bytes;
            explicit VectorBuffer(std::vector<uint8_t> b) : bytes(std::move(b))
{}
            size_t size() const override { return bytes.size(); }
            uint8_t* data() override { return bytes.data(); }
          };

          auto buffer = jsi::ArrayBuffer(rt,
            std::make_shared<VectorBuffer>(std::move(data)));

          // For a Uint8Array view over the buffer, use the JSI factory
          // (jsi.h:655-659): `runtime.createUint8Array(buffer, 0, size)`.
          return jsi::Uint8Array(rt, buffer, 0, buffer.size(rt));
        });
    }

    if (propName == "position") {
      return jsi::Value(static_cast<double>(stream_->position()));
    }

    return jsi::Value::undefined();
  }
};
```

3. Bidirectional Event Systems

Create event emitters that work seamlessly between JS and Native:

```
class EventEmitterHostObject : public jsi::HostObject {
private:
  std::unordered_map<std::string, std::vector<jsi::Function>> listeners_;
  std::mutex listenersMutex_;

public:
  jsi::Value get(jsi::Runtime& rt, const jsi::PropNameID& name) override {
    auto propName = name.utf8(rt);

    if (propName == "on") {
      return jsi::Function::createFromHostFunction(rt,
        jsi::PropNameID::forAscii(rt, "on"), 2,
        [this](jsi::Runtime& rt, const jsi::Value&,
               const jsi::Value* args, size_t) -> jsi::Value {
          std::string event = args[0].getString(rt).utf8(rt);
          auto callback = args[1].getObject(rt).getFunction(rt);
```

```
        std::lock_guard<std::mutex> lock(listenersMutex_);
        listeners_[event].push_back(std::move(callback));

        return jsi::Value::undefined();
      });
  }

  if (propName == "emit") {
    // paramCount is `unsigned int` (jsi.h:1611); engines don't enforce it as
    // an upper bound on call-site argument count, it only seeds the JS
    // function's `length` property. Pass 0 when the function accepts
    // arbitrarily many args.
    return jsi::Function::createFromHostFunction(rt,
      jsi::PropNameID::forAscii(rt, "emit"), 0, // variable args
      [this](jsi::Runtime& rt, const jsi::Value&,
             const jsi::Value* args, size_t count) -> jsi::Value {
        if (count < 1) return jsi::Value::undefined();

        std::string event = args[0].getString(rt).utf8(rt);

        std::lock_guard<std::mutex> lock(listenersMutex_);
        auto it = listeners_.find(event);
        if (it != listeners_.end()) {
          for (auto& listener : it->second) {
            // Call each listener with the remaining arguments
            listener.call(rt, args + 1, count - 1);
          }
        }

        return jsi::Value::undefined();
      });
  }

  return jsi::Value::undefined();
}

// Call from native code.
// NOTE: `Function::call(rt, Value*, size_t)` exists as a non-template
// overload (jsi.h:1620), but the variadic template overload (jsi.h:1639)
// wins overload resolution when `args` is a mutable `std::vector<Value>`
// because `Value*` is a tighter match for `Args&&...` than the qualification
// conversion to `const Value*`. Real RN code casts to `const jsi::Value*`
// explicitly (see `RCTTurboModule.mm:163` and `TimerManager.h:37`).
void emitFromNative(jsi::Runtime& rt, const std::string& event,
                    std::vector<jsi::Value> args) {
  std::lock_guard<std::mutex> lock(listenersMutex_);
  auto it = listeners_.find(event);
  if (it != listeners_.end()) {
    for (auto& listener : it->second) {
```

```
        listener.call(rt, (const jsi::Value*)args.data(), args.size());
      }
    }
  }
};
```

4. Memory-Mapped Files with JSI

Direct access to large files without loading into memory:

```
class MappedFileHostObject : public jsi::HostObject {
private:
  std::unique_ptr<MemoryMappedFile> file_;

public:
  MappedFileHostObject(const std::string& path) {
    file_ = std::make_unique<MemoryMappedFile>(path);
  }

  jsi::Value get(jsi::Runtime& rt, const jsi::PropNameID& name) override {
    auto propName = name.utf8(rt);

    if (propName == "size") {
      return jsi::Value(static_cast<double>(file_->size()));
    }

    if (propName == "getUint8") {
      return jsi::Function::createFromHostFunction(rt,
        jsi::PropNameID::forAscii(rt, "getUint8"), 1,
        [this](jsi::Runtime& rt, const jsi::Value&,
               const jsi::Value* args, size_t) -> jsi::Value {
          size_t offset = args[0].asNumber();
          if (offset >= file_->size()) {
            throw jsi::JSError(rt, "Offset out of bounds");
          }
          return jsi::Value(file_->data()[offset]);
        });
    }

    if (propName == "slice") {
      return jsi::Function::createFromHostFunction(rt,
        jsi::PropNameID::forAscii(rt, "slice"), 2,
        [this](jsi::Runtime& rt, const jsi::Value&,
               const jsi::Value* args, size_t) -> jsi::Value {
          size_t start = args[0].asNumber();
          size_t end = args[1].asNumber();

          if (start >= file_->size() || end > file_->size() || start > end) {
            throw jsi::JSError(rt, "Invalid slice range");
          }
```

```
          // To expose an in-memory pointer + length to JS without copying, you
          // implement a MutableBuffer that points at your existing storage and
          // hand it to the ArrayBuffer constructor (jsi.h:1556).
          // jsi::ArrayBuffer(rt, void*, size_t) is NOT a real constructor.
          struct WindowBuffer : jsi::MutableBuffer {
            uint8_t* ptr;
            size_t len;
            WindowBuffer(uint8_t* p, size_t l) : ptr(p), len(l) {}
            size_t size() const override { return len; }
            uint8_t* data() override { return ptr; }
          };

          return jsi::ArrayBuffer(rt,
            std::make_shared<WindowBuffer>(
              const_cast<uint8_t*>(file_->data() + start), end - start));
        });
    }

    return jsi::Value::undefined();
  }
};
```

5. Thread-Safe Native Modules

Handle concurrent access from multiple threads:

```
class ThreadSafeCounterHostObject : public jsi::HostObject {
private:
  std::atomic<int64_t> counter_{0};
  mutable std::shared_mutex callbacksMutex_;
  std::vector<jsi::Function> changeCallbacks_;

public:
  jsi::Value get(jsi::Runtime& rt, const jsi::PropNameID& name) override {
    auto propName = name.utf8(rt);

    if (propName == "value") {
      return jsi::Value(static_cast<double>(counter_.load()));
    }

    if (propName == "increment") {
      return jsi::Function::createFromHostFunction(rt,
        jsi::PropNameID::forAscii(rt, "increment"), 0,
        [this](jsi::Runtime& rt, const jsi::Value&,
               const jsi::Value*, size_t) -> jsi::Value {
          int64_t newValue = counter_.fetch_add(1) + 1;
          notifyListeners(rt, newValue);
          return jsi::Value(static_cast<double>(newValue));
        });
    }
```

```
    if (propName == "onChange") {
      return jsi::Function::createFromHostFunction(rt,
        jsi::PropNameID::forAscii(rt, "onChange"), 1,
        [this](jsi::Runtime& rt, const jsi::Value&,
               const jsi::Value* args, size_t) -> jsi::Value {
          auto callback = args[0].getObject(rt).getFunction(rt);

          std::unique_lock<std::shared_mutex> lock(callbacksMutex_);
          changeCallbacks_.push_back(std::move(callback));

          return jsi::Value::undefined();
        });
    }

    return jsi::Value::undefined();
  }

private:
  void notifyListeners(jsi::Runtime& rt, int64_t newValue) {
    std::shared_lock<std::shared_mutex> lock(callbacksMutex_);
    auto value = jsi::Value(static_cast<double>(newValue));
    for (auto& callback : changeCallbacks_) {
      callback.call(rt, value);
    }
  }
};
```

6. Performance Profiling with JSI

Create native performance monitoring tools:

```
class PerformanceMonitor : public jsi::HostObject {
private:
  struct Timing {
    std::chrono::high_resolution_clock::time_point start;
    std::chrono::high_resolution_clock::time_point end;
    std::string name;
  };

  std::vector<Timing> timings_;
  std::unordered_map<std::string, std::chrono::high_resolution_clock::time_point>
activeMarks_;

public:
  jsi::Value get(jsi::Runtime& rt, const jsi::PropNameID& name) override {
    auto propName = name.utf8(rt);

    if (propName == "mark") {
      return jsi::Function::createFromHostFunction(rt,
        jsi::PropNameID::forAscii(rt, "mark"), 1,
        [this](jsi::Runtime& rt, const jsi::Value&,
```

```
            const jsi::Value* args, size_t) -> jsi::Value {
          std::string markName = args[0].getString(rt).utf8(rt);
          activeMarks_[markName] = std::chrono::high_resolution_clock::now();
          return jsi::Value::undefined();
        });
    }

    if (propName == "measure") {
      return jsi::Function::createFromHostFunction(rt,
        jsi::PropNameID::forAscii(rt, "measure"), 2,
        [this](jsi::Runtime& rt, const jsi::Value&,
            const jsi::Value* args, size_t) -> jsi::Value {
          std::string name = args[0].getString(rt).utf8(rt);
          std::string startMark = args[1].getString(rt).utf8(rt);

          auto it = activeMarks_.find(startMark);
          if (it != activeMarks_.end()) {
            Timing timing;
            timing.name = name;
            timing.start = it->second;
            timing.end = std::chrono::high_resolution_clock::now();
            timings_.push_back(timing);

            auto duration = std::chrono::duration_cast<std::chrono::microseconds>
                          (timing.end - timing.start).count();

            return jsi::Value(duration / 1000.0); // Return milliseconds
          }

          return jsi::Value::undefined();
        });
    }

    return jsi::Value::undefined();
  }
};
```

Citations:

- [1] `packages/react-native/ReactCommon/jsi/jsi/jsi.h` (HEAD `9c7b0f5fafd`, v0.88.1 era). The full IRuntime / Runtime interface lives at lines 338-822; HostObject at 222-251; HostFunctionType at 217-218; Value at 1766-2078. Since the last refresh `IRuntime` gained `createError`, `createTypeError` and the other typed error factories (jsi.h:662-675) and `length(const String&)` (jsi.h:681). `JSError(rt, std::string)` now builds its value through `rt.createError` (jsi.cpp:968-970), and `Value::toString` uses `asString` so a non-string result throws (jsi.cpp:924-926).

- [2] React Native 0.76.0 release blog, "New Architecture by default" (2024-10-23): https://reactnative.dev/blog/2024/10/23/release-0.76-new-architecture. CHANGELOG entry for v0.76 in `CHANGELOG-0.7x.md`. JSI itself shipped much earlier (experimental in 0.59, default on the Hermes path long before 0.76). What 0.76 turned on by default was the Fabric + TurboModules + Bridgeless stack built on JSI.
- [3] `IRuntime` is the abstract interface (jsi.h:338-685); `Runtime : public IRuntime` keeps the older surface alive for backward compatibility (jsi.h:705-822). The header docs at 332-337 explicitly state "for backward compatibility, these APIs are also accessible via the Runtime object directly".
- [4] `HostObject::set` default implementation: `packages/react-native/ReactCommon/jsi/jsi/jsi.cpp:280-285`. It throws `JSError(rt, "TypeError: Cannot assign to property '<name>' on HostObject with default setter")`. Confirmed by the `NopHostObject` test in `packages/react-native/ReactCommon/jsi/jsi/test/testlib.cpp:495-510`.
- [5] `PropNameID::compare` declaration: `packages/react-native/ReactCommon/jsi/jsi/jsi.h:950-955` (`static bool compare(IRuntime&, const PropNameID&, const PropNameID&)`). Real usage in `packages/react-native/ReactCommon/jsi/jsi/test/testlib.cpp:44-77`.
- [6] `ArrayBuffer` constructor: `packages/react-native/ReactCommon/jsi/jsi/jsi.h:1556` (`ArrayBuffer(IRuntime&, std::shared_ptr<MutableBuffer>)`). `MutableBuffer` interface at `packages/react-native/ReactCommon/jsi/jsi/jsi.h:167-172`.
- [7] Vision Camera's `FrameHostObject` inheriting from `jsi::HostObject`: https://github.com/mrousavy/react-native-vision-camera/commit/4584d33 (line 28: `class JSI_EXPORT FrameHostObject : public jsi::HostObject`). Vision Camera's own docs on frame buffer sizes: https://react-native-vision-camera.com/docs/guides/frame-processors-tips (cites ~12 MB per 4K YUV frame; the 30 MB figure here is the RGBA8 worst case for the same resolution).

Chapter 3: Fabric - The New Renderer

Fabric is the name of React Native's modern rendering system, a complete re-implementation of the UI manager that lies at the heart of the New Architecture. Built on top of the JSI, Fabric delivers a more performant, responsive, and interoperable UI layer. It replaces the legacy "UIManager" and fundamentally changes how React Native translates JavaScript component trees into native platform views.

Current Status (v0.88.1): Fabric is the default renderer in React Native 0.76+ and the only path on RN ≥ 0.82. It powers Concurrent React on the mobile host (React peer `^19.3.0`) and provides significant performance improvements for complex UIs and animations. The legacy UIManager continues to be stripped under `RCT_REMOVE_LEGACY_ARCH`.

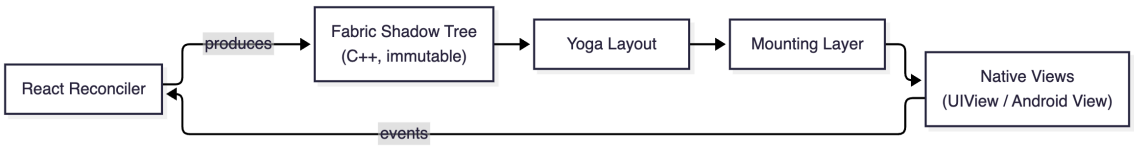
The Motivation: From UIManager to Fabric

The legacy UIManager operated entirely across the asynchronous Bridge. Every UI update, from creating a view to changing its background color, required a serialized message to be sent from JavaScript to the native side. This architecture led to several problems:

- **Latency:** The asynchronous nature meant there was no guarantee when a UI update would be processed, leading to dropped frames in complex animations or high-frequency updates.
- **Layout Thrashing:** Layout was asynchronous. JavaScript could not synchronously read the resulting position or size of a view on screen, making certain UI patterns difficult and inefficient.
- **Data Overhead:** All props and styles were serialized to JSON, adding significant overhead to every update.

Fabric was designed to solve these issues by creating a more tightly integrated and synchronous rendering pipeline, with a significant portion of the logic moved into a shared C++ core.

Visual: Fabric Render Pipeline



The C++ Core of Fabric

At its heart, Fabric introduces a new data structure that lives entirely in C++: the **Shadow Tree**. This tree is the single source of truth for the state of the UI.

Understanding the Shadow Tree Architecture

The Shadow Tree is not just a simple mirror of the React component tree. It's a sophisticated, immutable data structure designed for concurrent updates and efficient diffing:

```
// Simplified conceptual view of the Shadow Tree
ShadowTree (Root)
├── ScrollViewShadowNode
│   ├── ViewShadowNode (Header)
│   │   └── TextShadowNode ("Welcome")
```

```
│   └── ViewShadowNode (Content)
│       ├── ImageShadowNode
│       └── TextShadowNode ("Description")
└── ViewShadowNode (Footer)
```

1. `ShadowNode`: The Immutable Building Block

The fundamental unit of the Shadow Tree is the `ShadowNode`. As defined in `packages/react-native/ReactCommon/react/renderer/core/ShadowNode.h`[19], it is the C++ representation of a React element.

```
// Abridged from ShadowNode.h. See the full file for the complete API.
class ShadowNode : public Sealable,
                   public DebugStringConvertible,
                   public jsi::NativeState {
 public:
  // Constructors. Copy ctor and copy-assign are deleted: ShadowNode is
  // non-copyable. Updates go through `clone()`.
  ShadowNode(const ShadowNodeFragment &fragment,
             ShadowNodeFamily::Shared family,
             ShadowNodeTraits traits);
  ShadowNode(const ShadowNode &sourceShadowNode,
             const ShadowNodeFragment &fragment);

  // The canonical update API: returns a new node with the given fields
  // patched. Children/state/props that are not specified are carried over.
  std::shared_ptr<ShadowNode> clone(const ShadowNodeFragment &fragment) const;

 protected:
  Props::Shared props_;
  std::shared_ptr<const std::vector<std::shared_ptr<const ShadowNode>>>
children_;
  State::Shared state_;
  int orderIndex_;

 private:
  ShadowNodeFamily::Shared family_;
  mutable std::atomic<bool> hasBeenMounted_{false};
};
```

Notes on the layout:

- `ShadowNode` inherits from `Sealable`. After commit, `sealRecursive()` is called so a node can no longer be mutated even by the C++ side (this is a no-op in release builds; the guard fires in debug).
- It inherits from `jsi::NativeState`, which is how a JS `ShadowNode` reference holds a C++ pointer.

[19] `packages/react-native/ReactCommon/react/renderer/core/ShadowNode.h`

- Layout metrics do not live on `ShadowNode`. They live on the `LayoutableShadowNode` subclass (`packages/react-native/ReactCommon/react/renderer/core/LayoutableShadowNode.h`[20]). Any concrete view component descends through that branch.
- The `Shared` type alias pattern (`std::shared_ptr<const T>`) is defined on `RootShadowNode`, `Props`, `State`, `ShadowNodeFamily`, and other types in the renderer, but not on `ShadowNode` itself; pass `std::shared_ptr<const ShadowNode>` explicitly when you need it.

Why Immutability Matters:

In the legacy renderer, the JS thread and the UI thread could see the same view in inconsistent states because mutations happened across threads against a single set of native view objects. Fabric flips this: every "change" is the production of a new immutable node, and the old tree is still valid until the commit swap completes. Two threads that hold pointers to different revisions cannot trample each other.

Updates are produced by calling `clone()` on the source node with a `ShadowNodeFragment` carrying the changed fields. The actual machinery for changing a prop runs through the matching `ComponentDescriptor`'s `cloneProps`, since the descriptor knows how to parse a `RawProps` into the typed `Props` subclass for that component:

```
// Excerpt: cloning one node with new props, as it actually happens inside
// UIManager::cloneNode (UIManager.cpp).
PropsParserContext propsParserContext{
    shadowNode.getFamily().getSurfaceId(), *contextContainer_};

auto &componentDescriptor = shadowNode.getComponentDescriptor();
auto props = componentDescriptor.cloneProps(
    propsParserContext,
    shadowNode.getProps(),
    std::move(rawProps));

auto clonedShadowNode = componentDescriptor.cloneShadowNode(
    shadowNode,
    {
        .props = props,
        .children = children,
        .runtimeShadowNodeReference = false,
    });
```

The originating `shadowNode` is unchanged. The clone shares structure with its parent's children vector until that too is cloned by the next ancestor up the path. This structural sharing is why a "single prop changed" mutation is cheap even on a thousand-node tree.

Deep Dive: Props System

The Props system in Fabric is type-safe and per-component. The base `Props` class is small and stateless on its own; the real shape comes from the inheritance chain for each component.

[20] `packages/react-native/ReactCommon/react/renderer/core/LayoutableShadowNode.h`

```
// Props.h (abridged). The base class is intentionally lean.
class Props : public virtual Sealable,
              public virtual DebugStringConvertible {
 public:
  using Shared = std::shared_ptr<const Props>;

  Props() = default;
  Props(const PropsParserContext &context,
        const Props &sourceProps,
        const RawProps &rawProps,
        const std::function<bool(const std::string &)> &filterObjectKeys =
nullptr);

  // Called by the prop parser, once per property, while walking RawProps.
  // Subclasses override this and dispatch on `hash` to set their fields.
  void setProp(const PropsParserContext &context,
               RawPropsPropNameHash hash,
               const char *propName,
               const RawValue &value);

  std::string nativeId;
};
```

There is no `revision`, no `source`, no `rawProps` member on the base `Props`. Parsing happens through `RawPropsParser` walking the JS values and dispatching per-property hashes to the matching `setProp` overload. This is how Fabric avoids the per-update JSON allocations that the legacy renderer paid.

v0.87 note on prop names: the `RawPropsKey` type (with its prefix, name, and suffix parts) is gone. `RawPropsKey.h` no longer exists at HEAD. Prop names are flat strings now, and the lookup table is `RawPropsKeyMap` in `core/RawPropsKeyMap.h` (`insert(std::string_view, index)`, `reindex()`, `at(std::string_view)`). Source: commit `67381e157f` (CHANGELOG Breaking). Custom native components and third-party frameworks that composed prefixed keys (Nitro was the one known user) must use the flat name form.

Iterator-setter path (since 0.87, off by default): `ConcreteComponentDescriptor::cloneProps` has a second construction path. When `ConcreteProps` has an iterator-setter constructor and `enableCppPropsIteratorSetter` is on (default `false`), Fabric copies the source props and walks `RawProps` in place with `forEachItem`, calling `setProp` per entry. This skips `RawProps::parse` and the `folly::dynamic` copy. The classic path shown above stays the default.

`ViewProps` is a `using` alias for `HostPlatformViewProps`, which on most platforms aliases `BaseViewProps`. The inheritance graph is `BaseViewProps : public YogaStylableProps, public AccessibilityProps`, and `YogaStylableProps` is what eventually inherits from `Props`. The flattened field set on `BaseViewProps` looks like this:

```
// BaseViewProps.h (abridged). See the full file for the complete list.
class BaseViewProps : public YogaStylableProps, public AccessibilityProps {
 public:
```

```
  BaseViewProps() = default;
  BaseViewProps(
      const PropsParserContext &context,
      const BaseViewProps &sourceProps,
      const RawProps &rawProps,
      const std::function<bool(const std::string &)> &filterObjectKeys =
nullptr);

  Float opacity{1.0};
  SharedColor backgroundColor{};  // wrapper around Color; pointer-like
  CascadedBorderRadii borderRadii{};
  CascadedBorderColors borderColors{};
  Transform transform{};
  TransformOrigin transformOrigin{};
  BackfaceVisibility backfaceVisibility{};
  bool shouldRasterize{};
  std::optional<int> zIndex{};
  PointerEventsMode pointerEvents{};
  // ...and many more
};
```

`backgroundColor` is a `SharedColor` (a thin wrapper that hides the platform color representation), not a bare `Color`, which is why iOS code converts it via `RCTUIColorFromSharedColor(...)` and Android extracts a 32-bit int.

State Management in Fabric

Unlike props, state can be updated from both JavaScript and native. The pattern is one `State` template parameterised by a per-component `StateData` POD. For `ScrollView` the data type is `ScrollViewState`:

```
// ScrollViewState.h (abridged)
class ScrollViewState final {
 public:
  ScrollViewState(Point contentOffset,
                  Rect contentBoundingRect,
                  int scrollAwayPaddingTop);
  ScrollViewState() = default;

  Point contentOffset;
  Rect contentBoundingRect;
  int scrollAwayPaddingTop{0};
  bool disableViewCulling{false};

  Size getContentSize() const;
};
```

The content size is derived from `contentBoundingRect`, not stored directly. State writes go through `State::updateState(stateData)`, which schedules a commit on the shadow tree. The

platform code that runs scrolling calls into this on each frame to keep the JS-visible content offset in sync, without making a JS roundtrip per scroll tick.

2. `ComponentDescriptor`: The Shadow Node Factory

Defined in `packages/react-native/ReactCommon/react/renderer/core/ComponentDescriptor.h`[21], a `ComponentDescriptor` is a factory object responsible for creating and managing `ShadowNode` instances of a specific type.

```
// ComponentDescriptor.h (abridged). Pure-virtual methods only.
class ComponentDescriptor {
 public:
  using Shared = std::shared_ptr<const ComponentDescriptor>;
  using Unique = std::unique_ptr<const ComponentDescriptor>;

  virtual ComponentHandle getComponentHandle() const = 0;
  virtual ComponentName getComponentName() const = 0;
  virtual ShadowNodeTraits getTraits() const = 0;

  virtual std::shared_ptr<ShadowNode> createShadowNode(
      const ShadowNodeFragment &fragment,
      const ShadowNodeFamily::Shared &family) const = 0;

  virtual std::shared_ptr<ShadowNode> cloneShadowNode(
      const ShadowNode &sourceShadowNode,
      const ShadowNodeFragment &fragment) const = 0;

  virtual void appendChild(
      const std::shared_ptr<const ShadowNode> &parentShadowNode,
      const std::shared_ptr<const ShadowNode> &childShadowNode) const = 0;

  // Note: cloneProps takes three arguments. The PropsParserContext is required.
  // RawProps is taken by value because it gets moved through the parser.
  virtual Props::Shared cloneProps(
      const PropsParserContext &context,
      const Props::Shared &props,
      RawProps rawProps) const = 0;

  virtual State::Shared createInitialState(
      const Props::Shared &props,
      const ShadowNodeFamily::Shared &family) const = 0;
  virtual State::Shared createState(
      const ShadowNodeFamily &family,
      const StateData::Shared &data) const = 0;

  virtual ShadowNodeFamily::Shared createFamily(
      const ShadowNodeFamilyFragment &fragment) const = 0;
};
```

[21] `packages/react-native/ReactCommon/react/renderer/core/ComponentDescriptor.h`

There is no `createEventEmitter` on `ComponentDescriptor`. Event emitters are produced inside `ShadowNodeFamily::createEventEmitter()` (via the family that the descriptor returns from `createFamily(...)`) and passed through `ShadowNodeFragment`. The chapter-1 mental model of "descriptor builds emitter" is close, but the real factory method lives on the family, not the descriptor.

Real Implementation - ViewComponentDescriptor:

`ViewComponentDescriptor` is intentionally empty: it just plugs `ViewShadowNode` into the generic `ConcreteComponentDescriptor` template, which provides all the virtual overrides.

```
// packages/react-native/ReactCommon/react/renderer/components/view/
ViewComponentDescriptor.h
class ViewComponentDescriptor
    : public ConcreteComponentDescriptor<ViewShadowNode> {
 public:
  ViewComponentDescriptor(const ComponentDescriptorParameters &parameters)
      : ConcreteComponentDescriptor<ViewShadowNode>(parameters) {}
};
```

The actual factory work is in `ConcreteComponentDescriptor<T>`[22]:

```
// ConcreteComponentDescriptor.h (abridged)
template <typename ShadowNodeT>
class ConcreteComponentDescriptor : public ComponentDescriptor {
 public:
  ComponentHandle getComponentHandle() const override {
    return ShadowNodeT::Handle();
  }
  ComponentName getComponentName() const override {
    return ShadowNodeT::Name();  // for View: the string "View", not "RCTView"
  }

  std::shared_ptr<ShadowNode> createShadowNode(
      const ShadowNodeFragment &fragment,
      const ShadowNodeFamily::Shared &family) const override {
    auto shadowNode = std::make_shared<ShadowNodeT>(fragment, family,
getTraits());
    adopt(*shadowNode);
    return shadowNode;
  }

  std::shared_ptr<ShadowNode> cloneShadowNode(
      const ShadowNode &sourceShadowNode,
      const ShadowNodeFragment &fragment) const override {
    auto shadowNode = std::make_shared<ShadowNodeT>(sourceShadowNode, fragment);
    shadowNode->completeClone(sourceShadowNode, fragment);
    sourceShadowNode.transferRuntimeShadowNodeReference(shadowNode, fragment);
```

[22] `packages/react-native/ReactCommon/react/renderer/core/ConcreteComponentDescriptor.h`

```
    adopt(*shadowNode);
    return shadowNode;
  }
};
```

`adopt(...)` is the hook a descriptor uses to inject context that the shadow node cannot construct itself (the `ImageComponentDescriptor` uses it to hand the image manager to each `ImageShadowNode`, for example).

3. `ShadowTree`: Managing UI Evolution

As seen in `packages/react-native/ReactCommon/react/renderer/mounting/ShadowTree.h`[23], this class manages the entire lifecycle of the UI tree.

```
// ShadowTree.h (abridged)
class ShadowTree final {
 public:
  using CommitStatus = ShadowTreeCommitStatus;  // Succeeded, Failed, Cancelled
  using CommitMode = ShadowTreeCommitMode;
  using CommitOptions = ShadowTreeCommitOptions;

  // Returns Succeeded, Failed, or Cancelled. NOT a bool.
  CommitStatus tryCommit(const ShadowTreeCommitTransaction &transaction,
                         const CommitOptions &commitOptions) const;

  // Calls tryCommit in a loop until it stops returning Failed.
  CommitStatus commit(const ShadowTreeCommitTransaction &transaction,
                      const CommitOptions &commitOptions) const;

  ShadowTreeRevision getCurrentRevision() const;
  std::shared_ptr<const MountingCoordinator> getMountingCoordinator() const;

 private:
  // No lock-free trickery. A shared_mutex protects currentRevision_, with a
  // recursive_mutex variant behind the preventShadowTreeCommitExhaustion
  // feature flag for the bounded-attempts fallback.
  mutable std::shared_mutex revisionMutex_;
  mutable std::recursive_mutex revisionMutexRecursive_;
  mutable CommitMode commitMode_{CommitMode::Normal};
  mutable ShadowTreeRevision currentRevision_;
  mutable std::optional<ShadowTreeRevision> currentReactRevision_;  // branching
  std::shared_ptr<const MountingCoordinator> mountingCoordinator_;
};
```

The actual `tryCommit` implementation (`ShadowTree.cpp:337`) does these things in order:

1. Acquire a *shared* (reader) lock on `revisionMutex_` and snapshot `currentRevision_`.
2. Invoke the transaction with the old root, producing a new `RootShadowNode::Unshared`.

[23] `packages/react-native/ReactCommon/react/renderer/mounting/ShadowTree.h` and `ShadowTree.cpp` for the `tryCommit` body.

3. Optionally reconcile state across the JS branch and main branch via `progressState(...)`.
4. Run the commit hooks (`delegate_.shadowTreeWillCommit(...)`). A hook may replace the new root or return `nullptr`, which cancels the commit.
5. Run `newRootShadowNode->layoutIfNeeded(&affectedLayoutableNodes)`. This is the layout pass; it only re-runs Yoga on subtrees whose layout-affecting state changed.
6. Acquire a *unique* (writer) lock. If `currentRevision_.number` has moved under us, return `CommitStatus::Failed` and let the caller retry. Otherwise, increment the revision, seal the tree, and either push the revision to the `MountingCoordinator` (synchronous mode) or store it as a pending React revision (concurrent mode).

The "lock-free atomic bool" was a simplification in earlier drafts; the real machinery is a conventional reader/writer mutex with retry-on-conflict, plus a bounded-attempt fall-back to an exclusive lock to prevent unbounded retries.

Commit Transaction Example:

A commit transaction is a function from "old root" to "new root". To change a single button's background color, you'd use `ShadowNode::cloneTree`, which walks from a target node up to the root and clones each ancestor's children vector to splice in the updated node:

```
// Pseudocode for the JS-driven path. The real driver is React's reconciler
// calling UIManagerBinding methods; this is what you'd write to do it by hand.
shadowTree.commit(
    [&](const RootShadowNode &oldRoot) -> RootShadowNode::Unshared {
      // Find the family for the button. Family identity (not tag equality)
      // is the canonical "same node" check in Fabric.
      const ShadowNodeFamily &buttonFamily = /* ... */;

      auto newRoot = oldRoot.ShadowNode::cloneTree(
          buttonFamily,
          [&](const ShadowNode &oldButton) {
            // The descriptor knows how to translate raw values into the
            // typed Props subclass for this component.
            auto newProps = oldButton.getComponentDescriptor().cloneProps(
                propsParserContext,
                oldButton.getProps(),
                RawProps(folly::dynamic::object("backgroundColor", "blue")));
            return oldButton.clone({.props = newProps});
          });

      return std::static_pointer_cast<RootShadowNode>(newRoot);
    },
    {.enableStateReconciliation = false,
     .mountSynchronously = true,
     .source = ShadowTreeCommitSource::Unknown});
```

`ShadowTreeCommitSource` has four values on main: `Unknown`, `React`, `AnimationEndSync`, and `ReactRevisionMerge`. The last one belongs to commit branching. With `enableFabricCommitBranching` (default `false`, no OSS release stage) React commits go

to a separate "React revision" that `getCurrentReactRevision()` returns. The newer flag `enableFabricCommitBranchingMergeOnMainThread` (added 2026-09, default `false`) controls where that revision merges back. Treat both as experimental.

The transaction returns the new root or `nullptr` (which becomes `CommitStatus::Cancelled`). Note that the transaction body must return a `RootShadowNode::Unshared` (i.e. `std::shared_ptr<RootShadowNode>`), not a `Shared`, because the new revision needs an owning mutable handle to seal at commit time.

The Three Phases of Fabric Rendering

Fabric's rendering pipeline is a carefully orchestrated dance between JavaScript, C++, and platform-native code:

Phase 1: The Render Phase (JavaScript → C++)

This phase begins in JavaScript when React detects a state or props change:

```
// JavaScript: State update triggers re-render
function Counter() {
  const [count, setCount] = useState(0);

  return (
    <View>
      <Text>{count}</Text>
      <Button onPress={() => setCount(count + 1)} title="Increment" />
    </View>
  );
}
```

What happens under the hood:

On the C++ side, the JS host call lands in `UIManagerBinding`, which forwards it to `UIManager::createNode` (the class is named `UIManager`, not `FabricUIManager`; the JS-facing module name is `nativeFabricUIManager` and the Java wrapper on Android is `FabricUIManager`). The real signature[24]:

```
// UIManager.h
std::shared_ptr<ShadowNode> createNode(
    Tag tag,
    const std::string &componentName,
    SurfaceId surfaceId,
    RawProps rawProps,
    InstanceHandle::Shared instanceHandle) const;

// UIManager.cpp createNode (abridged)
std::shared_ptr<ShadowNode> UIManager::createNode(
    Tag tag,
    const std::string &name,
```

[24] `packages/react-native/ReactCommon/react/renderer/uimanager/UIManager.h` and `UIManagerBinding.cpp` for the JSI host function shape (`nativeFabricUIManager.createNode`).

```
    SurfaceId surfaceId,
    RawProps rawProps,
    InstanceHandle::Shared instanceHandle) const {

  auto &componentDescriptor = componentDescriptorRegistry_->at(name);

  PropsParserContext propsParserContext{surfaceId, *contextContainer_};

  auto family = componentDescriptor.createFamily(
      {.tag = tag,
       .surfaceId = surfaceId,
       .instanceHandle = std::move(instanceHandle)});
  const auto props = componentDescriptor.cloneProps(
      propsParserContext, /*sourceProps*/ nullptr, std::move(rawProps));
  const auto state = componentDescriptor.createInitialState(props, family);

  return componentDescriptor.createShadowNode(
      ShadowNodeFragment{
          .props = props,
          .children = ShadowNodeFragment::childrenPlaceholder(),
          .state = state,
      },
      family);
}
```

Notice that `ShadowNodeFragment` carries `props`, `children`, and `state`, not `tag`/`rootTag`/`eventEmitter`. The tag, surface id, and instance handle live on the `ShadowNodeFamily` returned by `createFamily(...)`. The event emitter is built lazily by the family when an event first needs to fire.

Prop parsing in practice. Fabric does not iterate `changedProps()` or check a `revision` field on `RawProps`. The parser walks the `RawProps` once via `RawProps::parse(parser)`, hashing each key with `RawPropsPropNameHash` and dispatching into `Props::setProp(context, hash, name, value)` overrides on each subclass. The optimisation is that `ConcreteComponentDescriptor::cloneProps` short-circuits to a shared default-props singleton when both `props` is null and `rawProps` is empty, skipping parsing entirely.

```
// From ConcreteComponentDescriptor.h: the "no work to do" short-circuit.
virtual Props::Shared cloneProps(
    const PropsParserContext &context,
    const Props::Shared &props,
    RawProps rawProps) const override {
  if (!props && rawProps.isEmpty()) {
    return ShadowNodeT::defaultSharedProps();
  }
  rawProps.parse(rawPropsParser_);
  auto shadowNodeProps = ShadowNodeT::Props(context, rawProps, props);
  // ...
```

```
  return shadowNodeProps;
}
```

Phase 2: The Commit Phase (on the JS thread)

The commit phase runs on the **JS thread**, not a background thread: it executes synchronously inside the JSI call that React's renderer made into the host. The advantage over the legacy renderer is not that the work moves off the JS thread, but that it stays in C++ end-to-end and uses lock-coordinated reader/writer access instead of a serialized message queue. The mount phase, covered next, is where the work then hops to the platform UI thread.

The diff (`ShadowViewMutation` list) is NOT computed inside `tryCommit`. The diff is deferred: `tryCommit` just lays out, seals, and pushes the new revision into `MountingCoordinator`. When the platform pulls a transaction (synchronously on iOS via `RCTExecuteOnMainQueue`, asynchronously on Android), `MountingCoordinator::pullTransaction` calls `calculateShadowViewMutations(const ShadowNode &oldRootShadowNode, const ShadowNode &newRootShadowNode)` (free function in `mounting/Differentiator.h`[25]) and packages the result into a `MountingTransaction`.

```
// ShadowTree.cpp tryCommit (abridged, the load-bearing parts)
CommitStatus ShadowTree::tryCommit(
    const ShadowTreeCommitTransaction &transaction,
    const CommitOptions &commitOptions) const {
  auto telemetry = TransactionTelemetry{};
  telemetry.willCommit();

  ShadowTreeRevision oldRevision;
  {
    SharedLock lock = sharedRevisionLock();  // reader lock
    oldRevision = currentRevision_;
  }

  auto newRootShadowNode = transaction(*oldRevision.rootShadowNode);
  if (!newRootShadowNode) {
    return CommitStatus::Cancelled;
  }

  // Optional state reconciliation.
  if (commitOptions.enableStateReconciliation) {
    auto progressed =
        progressState(*newRootShadowNode, *oldRevision.rootShadowNode);
    if (progressed) newRootShadowNode =
std::static_pointer_cast<RootShadowNode>(progressed);
  }

  // Layout pass. Yoga runs in here.
```

[25] `packages/react-native/ReactCommon/react/renderer/mounting/Differentiator.h` (`calculateShadowViewMutations`), driven from `MountingCoordinator::pullTransaction` in `MountingCoordinator.cpp`.

```
  std::vector<const LayoutableShadowNode *> affectedLayoutableNodes{};
  newRootShadowNode->layoutIfNeeded(&affectedLayoutableNodes);

  {
    UniqueLock lock = uniqueRevisionLock();  // writer lock
    if (currentRevision_.number != oldRevision.number) {
      return CommitStatus::Failed;  // raced; caller will retry.
    }
    auto newRevision = ShadowTreeRevision{
        .rootShadowNode = std::move(newRootShadowNode),
        .number = currentRevision_.number + 1,
        .telemetry = telemetry};
    newRevision.rootShadowNode->sealRecursive();
    currentRevision_ = newRevision;

    if (commitMode_ == CommitMode::Normal) {
      mount(std::move(newRevision), commitOptions.mountSynchronously);
    }
  }
  return CommitStatus::Succeeded;
}
```

The abridged code above leaves out three parts that exist at HEAD. First, a `TraceSection` and a `PerformanceTracerSection` wrap the commit. Second, `delegate_.shadowTreeWillCommit(...)` runs the commit hooks after state reconciliation. Third, when `enableFabricCommitBranching` is on and the source is `React` or `AnimationEndSync`, the commit reads and writes `currentReactRevision_` instead of `currentRevision_`, and `scheduleReactRevisionPromotion()` replaces the direct `mount()` call. The writer lock section also calls `updateMountedFlag(...)` before sealing.

The `mount()` call goes through `mountingCoordinator_->push(revision)` and then notifies the delegate. The diff is computed lazily by the platform consumer when it pulls.

Tree Diffing Algorithm:

The diff produces a `ShadowViewMutation::List`. Five mutation types exist, and they are bitflag-numbered (`Create = 1, Delete = 2, Insert = 4, Remove = 8, Update = 16`). Each mutation carries full `ShadowView` snapshots for the old and new child plus the `parentTag` and `index`:

```
// ShadowViewMutation.h (abridged)
struct ShadowViewMutation final {
  using List = std::vector<ShadowViewMutation>;

  enum Type : std::uint8_t {
    Create = 1,
    Delete = 2,
    Insert = 4,
    Remove = 8,
    Update = 16,
  };
```

```
  Type type = {Create};
  Tag parentTag = -1;
  ShadowView oldChildShadowView = {};
  ShadowView newChildShadowView = {};
  int index = -1;

  static ShadowViewMutation CreateMutation(ShadowView shadowView);
  static ShadowViewMutation DeleteMutation(ShadowView shadowView);
  static ShadowViewMutation InsertMutation(
      Tag parentTag, ShadowView childShadowView, int index);
  static ShadowViewMutation RemoveMutation(
      Tag parentTag, ShadowView childShadowView, int index);
  static ShadowViewMutation UpdateMutation(
      ShadowView oldChildShadowView,
      ShadowView newChildShadowView,
      Tag parentTag);
};
```

`ShadowView` is the "snapshot" type: `tag`, `props`, `eventEmitter`, `state`, `layoutMetrics`. It is small and value-copyable, which is what lets the mutation list cross the JS-thread / UI-thread boundary safely. The actual diff walk lives in `mounting/Differentiator.cpp`. Identity is established via `ShadowNode::sameFamily(...)` (same family object = "same component instance across renders"), not just tag equality, so React's family-based reconciliation maps directly to mounting decisions.

Layout Calculation with Yoga:

Yoga is not invoked through a one-shot `YGNodeCalculateLayout` call per commit with a freshly constructed `YGConfig`. Each `YogaLayoutableShadowNode` *owns* its `yoga::Node` as a member (`yogaNode_`). The node is built when the shadow node is constructed and is destroyed with it; there is no `YGNodeFree`/`YGConfigFree` paired with each layout call. The root's `layoutTree()` does one `YGNodeCalculateLayout` on the embedded root yoga node:

```
// YogaLayoutableShadowNode.cpp (abridged)
void YogaLayoutableShadowNode::layoutTree(
    LayoutContext layoutContext, LayoutConstraints layoutConstraints) {
  // Push min/max into Yoga style on the root.
  auto &yogaStyle = yogaNode_.style();
  auto ownerWidth = yogaFloatFromFloat(layoutConstraints.maximumSize.width);
  auto ownerHeight = yogaFloatFromFloat(layoutConstraints.maximumSize.height);
  yogaStyle.setMaxDimension(yoga::Dimension::Width,
      yoga::StyleSizeLength::points(layoutConstraints.maximumSize.width));
  yogaStyle.setMaxDimension(yoga::Dimension::Height,
      yoga::StyleSizeLength::points(layoutConstraints.maximumSize.height));
  // ... min dimensions, direction ...

  YGNodeCalculateLayout(&yogaNode_, ownerWidth, ownerHeight, direction);
```

```
  if (yogaNode_.getHasNewLayout()) {
    auto layoutMetrics = layoutMetricsFromYogaNode(yogaNode_);
    layoutMetrics.pointScaleFactor = layoutContext.pointScaleFactor;
    setLayoutMetrics(layoutMetrics);
    yogaNode_.setHasNewLayout(false);
  }
  layout(layoutContext);  // walks children and propagates metrics
}
```

The direction is derived from `layoutConstraints.layoutDirection`, so RTL surfaces don't hard-code `YGDirectionLTR`. The `pointScaleFactor` is set per-config when a `YogaLayoutableShadowNode` is constructed (in `configureYogaTree`), not on a transient config object per commit.

Recent differ and text changes (0.88 era, main): the differ unflatten feature flag was removed (commit `a016535a83f`, the corrected parent-tag path is now always on), and a fix stops the differ from creating or deleting a view that a nested flatten or unflatten moves (`361bc240e42`). A `View` that is a live region (`aria-live` or `accessibilityLiveRegion`) is no longer flattened (`5e13324903a`). Prepared text layout is on by default (`enablePreparedTextLayout`, default `true` in `ReactNativeFeatureFlags.config.js:570`, commit `a5cb48b2fc3`).

Phase 3: The Mount Phase (C++ → Native Platform)

This is where abstract mutations become real UI changes. The platform layer pulls a `MountingTransaction` and walks its `ShadowViewMutationList`. On iOS the loop lives in a free function (not a method on `MountingCoordinator`), `RCTPerformMountInstructions` in `RCTMountingManager.mm`, and the dispatch onto the main queue happens via `RCTExecuteOnMainQueue` (which becomes `dispatch_async(dispatch_get_main_queue(), block)` if the caller is not already on main).

```
// RCTMountingManager.mm (abridged)
static void RCTPerformMountInstructions(
    const ShadowViewMutationList &mutations,
    RCTComponentViewRegistry *registry,
    RCTMountingTransactionObserverCoordinator &observerCoordinator,
    SurfaceId surfaceId) {
  for (const auto &mutation : mutations) {
    switch (mutation.type) {
      case ShadowViewMutation::Create: {
        auto &newView = mutation.newChildShadowView;
        [registry dequeueComponentViewWithComponentHandle:newView.componentHandle
                                                      tag:newView.tag];
        break;
      }
      case ShadowViewMutation::Delete: {
        auto &oldView = mutation.oldChildShadowView;
        auto &descriptor = [registry componentViewDescriptorWithTag:oldView.tag];
        [registry enqueueComponentViewWithComponentHandle:oldView.componentHandle
                                                      tag:oldView.tag
```

```
                                componentViewDescriptor:descriptor];
        break;
      }
      case ShadowViewMutation::Insert: {
        auto &newView = mutation.newChildShadowView;
        auto &childDesc = [registry componentViewDescriptorWithTag:newView.tag];
        auto &parentDesc = [registry
componentViewDescriptorWithTag:mutation.parentTag];
        UIView<RCTComponentViewProtocol> *child = childDesc.view;
        [child updateProps:newView.props oldProps:nullptr];
        [child updateEventEmitter:newView.eventEmitter];
        [child updateState:newView.state oldState:nullptr];
        [child updateLayoutMetrics:newView.layoutMetrics
                  oldLayoutMetrics:EmptyLayoutMetrics];
        [child finalizeUpdates:RNComponentViewUpdateMaskAll];
        [parentDesc.view mountChildComponentView:child index:mutation.index];
        break;
      }
      case ShadowViewMutation::Remove: { /* unmount, keep view alive for reuse */
break; }
      case ShadowViewMutation::Update: {
        auto &oldView = mutation.oldChildShadowView;
        auto &newView = mutation.newChildShadowView;
        UIView<RCTComponentViewProtocol> *child =
            [registry componentViewDescriptorWithTag:newView.tag].view;
        // The mount applies the four orthogonal kinds of change separately:
        // props, eventEmitter, state, layoutMetrics. Each guarded by a
        // pointer-equality check that comes from the diff.
        if (oldView.props != newView.props)
          [child updateProps:newView.props oldProps:oldView.props];
        if (oldView.eventEmitter != newView.eventEmitter)
          [child updateEventEmitter:newView.eventEmitter];
        if (oldView.state != newView.state)
          [child updateState:newView.state oldState:oldView.state];
        if (oldView.layoutMetrics != newView.layoutMetrics)
          [child updateLayoutMetrics:newView.layoutMetrics
                    oldLayoutMetrics:oldView.layoutMetrics];
        [child finalizeUpdates:mask];
        break;
      }
    }
  }
}
```

`Create` and `Delete` only manage the lifecycle in the view registry; the actual hierarchy mutations are `Insert`/`Remove` (which can fire without a `Create` when the view is being reused from the registry's pool). `Update` decomposes a mutation into four independent updates because props, event emitter, state, and layout metrics flow through different paths.

Real Native View Update (iOS):

`RCTViewComponentView::updateProps` takes `Props::Shared` (the shared pointer), not a direct `const ViewProps &`. The implementation then casts internally:

```
// RCTViewComponentView.mm (abridged)
- (void)updateProps:(const facebook::react::Props::Shared &)props
           oldProps:(const facebook::react::Props::Shared &)oldProps {
  const auto &oldViewProps = static_cast<const ViewProps &>(*_props);
  const auto &newViewProps = static_cast<const ViewProps &>(*props);

  if (oldViewProps.opacity != newViewProps.opacity) {
    self.layer.opacity = (float)newViewProps.opacity;
  }
  if (oldViewProps.backgroundColor != newViewProps.backgroundColor) {
    self.backgroundColor =
RCTUIColorFromSharedColor(newViewProps.backgroundColor);
  }
  if (oldViewProps.transform != newViewProps.transform) {
    auto matrix = newViewProps.resolveTransform(_layoutMetrics);
    self.layer.transform = RCTCATransform3DFromTransformMatrix(matrix);
  }
  // ... ~30 more props handled this way
}
```

The point is "no JSON parsing on the main thread" rather than "compare C++ structs directly". The struct comparison still happens, but it's against the cached previous `Props::Shared` that the component view holds.

React 18+ / React 19 Concurrent Features: Real-World Impact

Fabric runs on top of React 19 (RN's peer dependency is `"react": "^19.3.0"`). The concurrent features below arrived with React 18 (`useTransition`, `Suspense`, automatic batching of async `setStates`) and remain the capability set on React 19 hosts — including the legacy renderer in older RN releases. What Fabric specifically unlocks is the synchronous mount path that lets these features actually feel responsive: a `useTransition` that interrupts itself is useless if the eventual commit blocks the UI thread for 60ms on the legacy bridge. So treat the examples below as "this is what becomes practical on the New Architecture", not "this is what Fabric implements internally".

Automatic Batching in Practice

React 18's automatic batching reduces re-renders when multiple `setState` calls happen in an async callback:

```
// Real-world example: Shopping cart with multiple state updates
function ShoppingCart() {
  const [items, setItems] = useState([]);
  const [total, setTotal] = useState(0);
  const [discount, setDiscount] = useState(0);
  const [isLoading, setIsLoading] = useState(false);

  const addItem = async (product) => {
    // Old Architecture: 4 separate renders
```

```
    // New Architecture: 1 batched render

    setIsLoading(true);
    const updatedCart = await api.addToCart(product);

    // All these updates are batched together
    setItems(updatedCart.items);
    setTotal(updatedCart.total);
    setDiscount(updatedCart.discount);
    setIsLoading(false);

    // Fabric ensures only ONE render cycle occurs
  };

  return (
    <View>
      <Text>Total: ${total - discount}</Text>
      {items.map(item => <CartItem key={item.id} item={item} />)}
      {isLoading && <ActivityIndicator />}
    </View>
  );
}
```

About the "4 renders vs 1 render" framing: automatic batching after an `await` is a React 18 change, not a Fabric-specific change. In React 17 with the legacy renderer, multiple `setState` calls in an async callback (after a `Promise` resolution) would each trigger their own commit. In React 18, regardless of architecture, they're collapsed into one render. The win shows up in both Fabric and the legacy renderer once you're on React 18. The 75%-reduction microbenchmark above is a synthetic estimate ("4 × 16ms vs 16ms") and should be read as a back-of-envelope illustration, not a measurement.

Transitions for Heavy Computations

Real-world example of filtering a large dataset:

```
function ProductCatalog() {
  // 10,000+ products. Declared first so it's in scope for useState below.
  const products = useLargeProductDataset();
  const [searchQuery, setSearchQuery] = useState('');
  const [filteredProducts, setFilteredProducts] = useState(products);
  const [isPending, startTransition] = useTransition();

  const handleSearch = (query) => {
    // Update search input immediately (urgent)
    setSearchQuery(query);

    // Filter products in background (non-urgent)
    startTransition(() => {
      const filtered = products.filter(product => {
        // Complex filtering logic
```

```
        return (
          product.name.toLowerCase().includes(query.toLowerCase()) ||
          product.description.toLowerCase().includes(query.toLowerCase()) ||
          product.tags.some(tag => tag.includes(query))
        );
      });

      setFilteredProducts(filtered);
    });
  };

  return (
    <View>
      <TextInput
        value={searchQuery}
        onChangeText={handleSearch}
        placeholder="Search products..."
      />

      {isPending ? (
        <View style={styles.pendingOverlay}>
          <Text>Updating results...</Text>
          <ProductGrid products={filteredProducts} opacity={0.6} />
        </View>
      ) : (
        <ProductGrid products={filteredProducts} />
      )}
    </View>
  );
}
```

Suspense with Real Data Fetching

Fabric enables sophisticated loading states with Suspense:

```
// Resource creation with built-in caching
const userResource = createResource(userId =>
  fetch(`/api/users/${userId}`).then(r => r.json())
);

function UserDashboard({ userId }) {
  return (
    <View>
      {/* Multiple Suspense boundaries for granular loading */}
      <Suspense fallback={<HeaderSkeleton />}>
        <UserHeader userId={userId} />
      </Suspense>

      <Suspense fallback={<ContentSkeleton />}>
        <UserContent userId={userId} />
```

```
        {/* Nested suspense for dependent data */}
        <Suspense fallback={<FriendListSkeleton />}>
          <UserFriends userId={userId} />
        </Suspense>
      </Suspense>
    </View>
  );
}

function UserHeader({ userId }) {
  // This suspends until data is ready
  const user = userResource.read(userId);

  return (
    <View style={styles.header}>
      <Image source={{ uri: user.avatar }} />
      <Text style={styles.name}>{user.name}</Text>
    </View>
  );
}
```

Concurrent Rendering in Action

Fabric's concurrent renderer enables time-slicing for smooth animations:

```
function AnimatedList({ data }) {
  const scrollY = useSharedValue(0);
  const [visibleItems, setVisibleItems] = useState([]);

  // Heavy computation wrapped in transition
  const updateVisibleItems = useCallback((offset) => {
    startTransition(() => {
      // Calculate which items are visible
      const startIndex = Math.floor(offset / ITEM_HEIGHT);
      const endIndex = Math.ceil((offset + SCREEN_HEIGHT) / ITEM_HEIGHT);

      // Complex visibility calculations
      const newVisible = data.slice(startIndex, endIndex).map(item => ({
        ...item,
        opacity: calculateOpacity(item, offset),
        scale: calculateScale(item, offset),
        blur: calculateBlur(item, offset)
      }));

      setVisibleItems(newVisible);
    });
  }, [data]);

  const scrollHandler = useAnimatedScrollHandler({
    onScroll: (event) => {
      scrollY.value = event.contentOffset.y;
```

```
      // Update visible items without blocking scroll
      runOnJS(updateVisibleItems)(event.contentOffset.y);
    }
  });

  return (
    <Animated.ScrollView onScroll={scrollHandler}>
      {visibleItems.map(item => (
        <AnimatedItem key={item.id} item={item} scrollY={scrollY} />
      ))}
    </Animated.ScrollView>
  );
}
```

Optimistic Updates with Concurrent Features

Combining transitions with optimistic updates:

```
function TodoList() {
  const [todos, setTodos] = useState([]);
  const [optimisticTodos, setOptimisticTodos] = useState([]);
  const [isPending, startTransition] = useTransition();

  const addTodo = async (text) => {
    // Optimistic update (immediate)
    const newTodo = { id: Date.now(), text, pending: true };
    setOptimisticTodos([...todos, newTodo]);

    // Server update (can be interrupted)
    startTransition(async () => {
      try {
        const savedTodo = await api.createTodo(text);
        setTodos(current => [...current, savedTodo]);
        setOptimisticTodos([]); // Clear optimistic state
      } catch (error) {
        // Revert optimistic update
        setOptimisticTodos([]);
        showError("Failed to add todo");
      }
    });
  };

  const displayTodos = [...todos, ...optimisticTodos];

  return (
    <View>
      <TodoInput onAdd={addTodo} disabled={isPending} />
      {displayTodos.map(todo => (
        <TodoItem
          key={todo.id}
```

```
          todo={todo}
          opacity={todo.pending ? 0.6 : 1}
        />
      ))}
    </View>
  );
}
```

Citations:

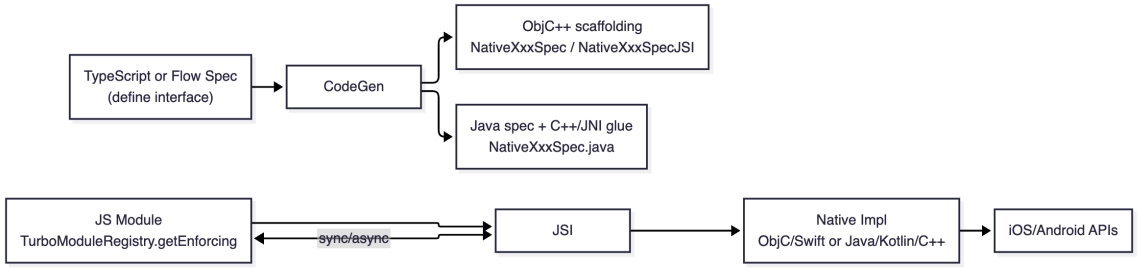
- [1] `packages/react-native/ReactCommon/react/renderer/core/ShadowNode.h`
- [2] `packages/react-native/ReactCommon/react/renderer/core/ComponentDescriptor.h`
- [3] `packages/react-native/ReactCommon/react/renderer/mounting/ShadowTree.h` and `ShadowTree.cpp` for the `tryCommit` body.
- [4] `packages/react-native/React/Fabric/Mounting/ComponentViews/View/RCTViewComponentView.h` and `.mm`.
- [5] `packages/react-native/ReactCommon/react/renderer/core/LayoutableShadowNode.h`
- [6] `packages/react-native/ReactCommon/react/renderer/core/ConcreteComponentDescriptor.h`
- [7] `packages/react-native/ReactCommon/react/renderer/uimanager/UIManager.h` and `UIManagerBinding.cpp` for the JSI host function shape (`nativeFabricUIManager.createNode`).
- [8] `packages/react-native/ReactCommon/react/renderer/mounting/Differentiator.h` (`calculateShadowViewMutations`), driven from `MountingCoordinator::pullTransaction` in `MountingCoordinator.cpp`.

Chapter 4: TurboModules - The New Native Modules

TurboModules are the second pillar of the New Architecture, representing a complete overhaul of the native module system. They replace the legacy system based on `RCTBridgeModule` and are designed from the ground up to be more performant, type-safe, and efficient. Just like Fabric, TurboModules are built directly on the foundation of the JSI and are the officially supported approach for new native modules in React Native 0.76+.[26]

Current Status (v0.88): TurboModules are not merely the default — they are **always on**. The `useTurboModules` feature flag was removed in v0.87.0-rc.0, and on iOS `RCTTurboModuleEnabled()` always returned `YES` while `RCTEnableTurboModule()` was a deprecated no-op. Both functions are now deleted on main (`01d55273bcf`, after v0.88.1).[27] All new native modules should be implemented as TurboModules; the legacy `RCTBridgeModule` path survives only through the interop layer (still shipped in OSS builds, but compile-optional behind `RCT_REMOVE_LEGACY_MODULE_INTEROP`). TurboModule bridging accepts `ArrayBuffer` on the **C++ and ObjC** paths since v0.87, and on **Java/Kotlin** since v0.88.[28]

Visual: TurboModule + CodeGen Pipeline



Note that CodeGen emits **ObjC++** for iOS and **Java + JNI/C++** for Android, not Swift or Kotlin directly. Swift code consumes the ObjC++ header via a bridging header; Kotlin extends the generated Java spec class as if it were Kotlin (the languages share a single classpath).

The Motivation: Problems with Legacy Native Modules

Legacy native modules, while powerful, suffered from limitations tied directly to their reliance on the asynchronous Bridge:

---

[26] "Turbo Native Modules". React Native Documentation. https://reactnative.dev/docs/next/turbo-native-modules-introduction. The 0.76 release blog post confirms the default switch: https://reactnative.dev/blog/2024/10/23/release-0.76-new-architecture. CHANGELOG entry for v0.76 in `CHANGELOG-0.7x.md`: "TurboModules will be looked up as TurboModules first, and fallback to legacy modules after" (commit `5a62606ab3`).

[27] `RCTTurboModuleEnabled` / `RCTEnableTurboModule` declared deprecated in `React/Base/RCTBridge.h` and `RCTBridge.mm` (commits `aee4a2586e`, `feede1422b`). Both were deleted on main by `01d55273bcf` ("Delete dead bridge flag functions and module hooks"), so `v0.88.1:packages/react-native/React/Base/RCTBridge.h:50` still declares `RCTTurboModuleEnabled`, and HEAD does not. Interop flags (`RCTEnableTurboModuleInterop`) remain, behind `RCT_REMOVE_LEGACY_MODULE_INTEROP`. Feature-flag removal: CHANGELOG v0.87.0-rc.0, "Remove the `useTurboModules` feature flag. TurboModules are now always enabled." (`b4a715ca1f`).

[28] C++ bridging: `packages/react-native/ReactCommon/react/bridging/ArrayBuffer.h` (`Bridging<jsi::ArrayBuffer>`, `AsyncArrayBuffer::{acquire,borrow}`). Added in v0.87.0-rc.0 (`226ef2e7c5`). Java support: `5fb3ebce1a` (CHANGELOG v0.88.0). `ArrayBufferKind` is enumerated in `TurboModule.h:36` alongside the other argument kinds.

1. **Eager Initialization:** All native modules registered with an application had to be initialized and loaded into memory when the app started, contributing to slower startup times.
2. **Asynchronous Everything:** Every method call was asynchronous. It was impossible for a JavaScript function to get an immediate return value from a native method.
3. **No Type Safety:** The contract between JavaScript and native code was based on convention and runtime checks. A mismatch in method signatures or argument types would lead to runtime errors.

TurboModules were created to solve these specific problems.

The Core Benefits of TurboModules

1. Lazy Loading

TurboModules are **lazily loaded**. A native module is not instantiated until it is accessed for the first time in JavaScript (e.g., via `TurboModuleRegistry.getEnforcing('MyModule')`). This on-demand approach significantly improves application startup performance.

2. Synchronous Method Execution

Because TurboModules are fundamentally `jsi::HostObjects`, they can expose methods that can be called **synchronously** from JavaScript. This is invaluable for small, fast functions that need to return data to JavaScript immediately.

3. Type Safety Across Languages

TurboModules enforce a strict, type-safe contract between JavaScript and the native platforms. This is achieved through a combination of JavaScript specs and automatic code generation.

- **The Spec is the Source of Truth:** A developer defines the module's interface in a TypeScript file.
- **CodeGen Enforces the Contract:** CodeGen parses the spec and generates native interface files. If the native implementation's method signature doesn't match the spec, the code won't compile.

4. First-class binary payloads (`ArrayBuffer`)

`jsi::ArrayBuffer` is a supported bridging type for TurboModules on all three paths. C++ support landed in v0.87.0-rc.0, ObjC in v0.87, and Java/Kotlin in v0.88.0.[29] The shared C++ bridging layer (`ReactCommon/react/bridging/ArrayBuffer.h`) converts to and from JS `ArrayBuffer`, including an `AsyncArrayBuffer` helper for moving bytes off the JS thread without tying lifetime to a live `jsi::Runtime`. Spec a method as taking or returning `ArrayBuffer` in TypeScript and CodeGen emits the matching signature. No base64, no JSON, no custom HostObject boilerplate.

- **ObjC.** In v0.87.0 and v0.88.1 the generated signature uses `NSMutableData *`. Inbound buffers arrive as immutable `NSData` copies, so the buffer cannot be GC'd under the module, and

[29] C++ bridging: `packages/react-native/ReactCommon/react/bridging/ArrayBuffer.h` (`Bridging<jsi::ArrayBuffer>`, `AsyncArrayBuffer::{acquire,borrow}`). Added in v0.87.0-rc.0 (`226ef2e7c5`). Java support: `5fb3ebce1a` (CHANGELOG v0.88.0). `ArrayBufferKind` is enumerated in `TurboModule.h:36` alongside the other argument kinds.

outbound `NSMutableData` maps back to a JSI `ArrayBuffer`.[30] On main (after v0.88.1) a zero-copy `RCTArrayBuffer *` class replaces `NSMutableData *` in the generated code (`11f9a7f4491`). The 0.88 changelog says this change was reverted on the 0.88 line because it broke modules that adopted ObjC `ArrayBuffer` in 0.87. Check your target version before you write the ObjC signature.

- **Java/Kotlin.** The type is `com.facebook.react.bridge.ArrayBuffer`. Bytes that JS passes in are **borrowed**. They are valid only until the receiving method returns. After that, `bytes` and `size` throw `IllegalStateException`. Call `arrayBufferWithCopiedBytes` to keep them.[31]
- **Limit.** `Promise<ArrayBuffer>` is rejected by CodeGen on Java/Kotlin and ObjC. Use a C++ TurboModule, return the buffer from a synchronous method, or resolve with another type. `ArrayBuffer` as an argument and as a synchronous return value works on all platforms.[32]

How TurboModules Work Under the Hood

The implementation of a TurboModule spans three layers:

1. The C++ Core (`TurboModule.h`)

The heart of every TurboModule is the C++ class `facebook::react::TurboModule`, defined in `packages/react-native/ReactCommon/react/nativemodule/core/ReactCommon/TurboModule.h`[33]. This class inherits from `jsi::HostObject`, which is what allows it to be exposed directly to the JavaScript runtime. The first time JavaScript reads a property on the module (for example, calling `MyModule.getSyncValue`), JSI invokes the C++ `get` method of this Host Object, which looks the name up in `methodMap_` and returns a `jsi::Function` built with `jsi::Function::createFromHostFunction`. That `jsi::Function` is then cached on the JS-side

[30] ObjC conversion: `packages/react-native/ReactCommon/react/nativemodule/core/platform/ios/ReactCommon/RCTTurboModule.mm` (commit `54b88d9dea`, CHANGELOG v0.87.0). On main the file uses `convertRCTArrayBufferToJSIArrayBuffer` and `RCTArrayBuffer` (`React/Base/RCTArrayBuffer.h`, commit `11f9a7f4491`), and `GenerateModuleObjCpp/serializeMethod.js:230,401` emit `RCTArrayBuffer *`. At `v0.88.1` the same lines emit `NSMutableData *`. ObjC CodeGen emits `ArrayBufferKind` via `serializeMethod.js`.

[31] Java type: `packages/react-native/ReactAndroid/src/main/java/com/facebook/react/bridge/ArrayBuffer.kt` (`isOwningBytes`, borrow contract; commit `d84c13d5111`, CHANGELOG v0.88.0). Java generator maps `ArrayBufferTypeAnnotation` to `com.facebook.react.bridge.ArrayBuffer` (`GenerateModuleJavaSpec.js:284,381`) and JNI uses `Lcom/facebook/react/bridge/ArrayBuffer;` (`GenerateModuleJniCpp.js:313,399`). `throwIfUnsupportedPromiseArrayBuffer` in `packages/react-native-codegen/src/generators/modules/Utils.js:95` rejects `Promise<ArrayBuffer>` ("is not supported for Android (Java/Kotlin) or iOS (ObjC) TurboModules"). The old message "ArrayBuffer is only supported for C++ TurboModules" no longer exists in the generators.

[32] Java type: `packages/react-native/ReactAndroid/src/main/java/com/facebook/react/bridge/ArrayBuffer.kt` (`isOwningBytes`, borrow contract; commit `d84c13d5111`, CHANGELOG v0.88.0). Java generator maps `ArrayBufferTypeAnnotation` to `com.facebook.react.bridge.ArrayBuffer` (`GenerateModuleJavaSpec.js:284,381`) and JNI uses `Lcom/facebook/react/bridge/ArrayBuffer;` (`GenerateModuleJniCpp.js:313,399`). `throwIfUnsupportedPromiseArrayBuffer` in `packages/react-native-codegen/src/generators/modules/Utils.js:95` rejects `Promise<ArrayBuffer>` ("is not supported for Android (Java/Kotlin) or iOS (ObjC) TurboModules"). The old message "ArrayBuffer is only supported for C++ TurboModules" no longer exists in the generators.

[33] `packages/react-native/ReactCommon/react/nativemodule/core/ReactCommon/TurboModule.h` (HEAD `9c7b0f5fafd`, v0.88.1 era). `class JSI_EXPORT TurboModule : public jsi::HostObject` at line 49. The `get` override is inlined near lines 55-70. `methodMap_` (the `std::unordered_map<std::string, MethodMetadata>` that maps method names to their host-function invokers) is declared at line 88.

wrapper object[34], so subsequent calls go straight to the host function without another `get` round-trip. The host function ultimately invokes the platform-specific native code.

2. The iOS Implementation (`RCTTurboModule.h`)

On iOS, a native module conforms to the `@protocol RCTTurboModule`, defined in `packages/react-native/ReactCommon/react/nativemodule/core/platform/ios/ReactCommon/RCTTurboModule.h`[35]. `RCTTurboModule` inherits from a small `RCTModuleProvider` protocol that declares the single `- getTurboModule:` requirement[36]. The system uses a C++ bridge class, `ObjCTurboModule`, which holds a reference to the user's Objective-C module instance via `id<RCTBridgeModule> instance_`. When a method is called from JavaScript, the `ObjCTurboModule` receives the call and uses the Objective-C runtime (`NSInvocation`) to dynamically invoke the correct method on the Objective-C instance.[37]

Example Migration (iOS):

Before (Legacy `RCTBridgeModule`):

```
// MyModule.m
#import <React/RCTBridgeModule.h>

@interface MyModule : NSObject <RCTBridgeModule>
@end

@implementation MyModule
RCT_EXPORT_MODULE();
RCT_EXPORT_METHOD(getSyncValue:(RCTResponseSenderBlock)callback)
{
  callback(@[@"some sync value"]);
}
@end
```

Note that the legacy method named `getSyncValue` is actually invoked **asynchronously** by JS: `RCT_EXPORT_METHOD` always marshals through the bridge and returns the value

[34]The cache is the `jsRepresentation_` weak-object reference in `TurboModule.h:143`. The caching call lives in the `get` body at line 65: `jsRepresentation_->lock(runtime).asObject(runtime).setProperty(runtime, propName, prop)`. The comment near line 62 states explicitly: "We don't cache misses, to allow for methodMap_ to dynamically be extended."

[35]`packages/react-native/ReactCommon/react/nativemodule/core/platform/ios/ReactCommon/RCTTurboModule.h`. `class JSI_EXPORT ObjCTurboModule : public TurboModule` at line 52, `id<RCTBridgeModule> instance_` at line 73, `@protocol RCTTurboModule <RCTModuleProvider>` at line 210.

[36]`RCTModuleProvider` protocol is declared at `RCTTurboModule.h:197`. Its single required method is `- (std::shared_ptr<facebook::react::TurboModule>)getTurboModule:(const facebook::react::ObjCTurboModule::InitParams &)params`. This split landed so that pure-C++ TurboModules (without an Objective-C class) can still register through the same provider surface.

[37]`RCTTurboModule.mm` constructs the `NSInvocation` at line 725 (`NSInvocation *inv = [NSInvocation invocationWithMethodSignature:methodSignature]`), sets the selector at 726, fills each argument via `setInvocationArg` (728-732), and finally invokes the selector on `instance_` in `performMethodInvocation` (line 780+). The full call site is `ObjCTurboModule::invokeObjCMethod` at lines 760-820.

via a callback block.[38] The legacy bridge does have a true sync escape hatch (`RCT_EXPORT_BLOCKING_SYNCHRONOUS_METHOD`, defined in the same header), but it was discouraged because it ran on the JS thread and stalled it. The TurboModule version below is sync without that caveat, because the call goes directly through JSI without any bridge serialization.

After (TurboModule):

First, define the spec. The file name is significant: it must start with `Native` and match the haste name used in the generated artifacts (`NativeMyModule.ts` produces `NativeMyModuleSpec` / `NativeMyModuleSpecJSI`). The argument to `getEnforcing` is the **JS-facing module name**, which the native side exposes as the `NAME` constant on the generated spec class.[39]

```
// NativeMyModule.ts
import type {TurboModule} from 'react-native';
import {TurboModuleRegistry} from 'react-native';

export interface Spec extends TurboModule {
  getSyncValue(): string;
}

export default TurboModuleRegistry.getEnforcing<Spec>('MyModule');
```

CodeGen also accepts Flow specs (most of RN's own bundled `Native*` files under `src/private/specs_DEPRECATED/modules/` are still Flow), but TypeScript is the recommended choice for new third-party modules.[40]

Then, update the native class to conform to the generated spec. Note the removal of `RCT_EXPORT_` macros and the addition of the `getTurboModule` boilerplate.

```
// MyModule.mm
#import "MyModuleSpec.h" // Import the header generated by CodeGen
```

[38] `RCT_EXPORT_METHOD` is defined at `packages/react-native/React/Base/RCTBridgeModule.h:263` (`#define RCT_EXPORT_METHOD(method) RCT_REMAP_METHOD(, method)`). Legacy bridge dispatch routes the call through the batched JSON message queue, so the value comes back via the callback block, not as a return value. The true-sync alternative `RCT_EXPORT_BLOCKING_SYNCHRONOUS_METHOD` is at line 281 and is documented with a "WARNING: ..." comment at line 269.

[39] CodeGen ObjC++ generator: `packages/react-native-codegen/src/generators/modules/GenerateModuleObjCpp/index.js:36` emits `@protocol ${hasteModuleName}Spec <RCTBridgeModule, RCTTurboModule>` and `:55-58` emits `class JSI_EXPORT ${hasteModuleName}SpecJSI : public ObjCTurboModule { public: ${hasteModuleName}SpecJSI(const ObjCTurboModule::InitParams &params); };`. Java generator: `GenerateModuleJavaSpec.js:62` emits `public abstract class ${className} extends ReactContextBaseJavaModule implements TurboModule` and `:63` emits `public static final String NAME = "${jsName}";`. The full round-trip is exercised in `private/react-native-codegen-typescript-test/src/__tests__/simple-scenario-frontend-test.ts:14-40` and reproduced for this chapter at `_verification/chapters/03-turbo-modules/tests/spec_objc_gen.js`.

[40] TS and Flow parsers live side by side: `packages/react-native-codegen/src/parsers/typescript/` and `.../parsers/flow/`. Every shipped `Native*` spec under `packages/react-native/src/private/specs_DEPRECATED/modules/` uses Flow (see e.g. `NativeAccessibilityInfo.js`). The sample spec `NativeSampleTurboModule.js` moved to `packages/rn-tester/js/examples/TurboModule/` (`4ffc4a65231`); third-party modules under the New Arch are typically authored in TypeScript.

```objc
@interface MyModule () <NativeMyModuleSpec>
@end

@implementation MyModule

// Boilerplate to connect the class to the TurboModule system
- (std::shared_ptr<facebook::react::TurboModule>)getTurboModule:
    (const facebook::react::ObjCTurboModule::InitParams &)params
{
  return std::make_shared<facebook::react::NativeMyModuleSpecJSI>(params);
}

// The actual implementation, now strongly typed by the generated spec
- (NSString *)getSyncValue
{
  return @"some sync value";
}

@end
```

3. The Android Implementation

On Android, the approach is similar. The marker interface is `com.facebook.react.turbomodule.core.interfaces.TurboModule`, defined in `packages/react-native/ReactAndroid/src/main/java/com/facebook/react/turbomodule/core/interfaces/TurboModule.kt`.[41] You don't normally implement it directly. The generated abstract spec class already does (`public abstract class NativeMyModuleSpec extends ReactContextBaseJavaModule implements TurboModule`),[42] so your concrete module just extends the spec:

```kotlin
// MyModule.kt
@ReactModule(name = NativeMyModuleSpec.NAME)
class MyModule(reactContext: ReactApplicationContext) :
    NativeMyModuleSpec(reactContext) {

  override fun getSyncValue(): String = "some sync value"
}
```

For a method declared with a non-Promise, non-void return type (like our `getSyncValue(): string`), CodeGen tags the generated abstract method with

[41] `package com.facebook.react.turbomodule.core.interfaces ... public interface TurboModule { public fun initialize(); public fun invalidate(); }` (`TurboModule.kt:8-19`). The two methods are lifecycle hooks the bridge calls on init and on `ReactHost` tear-down.

[42] Confirmed by inspection of any RN-shipped module that's already migrated. For example, `BlobModule.kt:44-45` reads `class BlobModule(reactContext: ReactApplicationContext) : NativeBlobModuleSpec(reactContext), TurboModuleWithJSIBindings`. It extends the generated spec class and only adds `TurboModuleWithJSIBindings` (a separate interface for modules that install their own JSI bindings). It does NOT re-declare `TurboModule`, because the spec class already implements it.

`@ReactMethod(isBlockingSynchronousMethod = true)`, which makes JS calls execute synchronously on the JS thread.

Module registration is then handled by a `ReactPackage`. In the New Arch, the supported pattern is to extend `BaseReactPackage` and override `getModule(name, reactContext)` for lazy lookup, not the older `createNativeModules` method, which is now marked `@Deprecated("Migrate to [BaseReactPackage] and implement [getModule] instead.")`.[43] React Native's own `MainReactPackage` uses exactly this shape, switching on the module name and returning a fresh instance only when the JS proxy first asks for it.

```
class MyPackage : BaseReactPackage() {
  override fun getModule(name: String, reactContext: ReactApplicationContext) =
      when (name) {
        NativeMyModuleSpec.NAME -> MyModule(reactContext)
        else -> null
      }

  override fun getReactModuleInfoProvider() = ReactModuleInfoProvider {
    mapOf(NativeMyModuleSpec.NAME to ReactModuleInfo(
      NativeMyModuleSpec.NAME, MyModule::class.java.name,
      false, false, false, true,
    ))
  }
}
```

Citations:

- [1] `packages/react-native/ReactCommon/react/nativemodule/core/ReactCommon/TurboModule.h` (HEAD `9c7b0f5fafd`, v0.88.1 era). `class JSI_EXPORT TurboModule : public jsi::HostObject` at line 49. The `get` override is inlined near lines 55-70. `methodMap_` (the `std::unordered_map<std::string, MethodMetadata>` that maps method names to their host-function invokers) is declared at line 88.
- [2] `packages/react-native/ReactCommon/react/nativemodule/core/platform/ios/ReactCommon/RCTTurboModule.h`. `class JSI_EXPORT ObjCTurboModule : public TurboModule` at line 52, `id<RCTBridgeModule> instance_` at line 73, `@protocol RCTTurboModule <RCTModuleProvider>` at line 210.
- [3] "Turbo Native Modules". React Native Documentation. https://reactnative.dev/docs/next/turbo-native-modules-introduction. The 0.76 release blog post confirms the default switch: https://reactnative.dev/blog/2024/10/23/release-0.76-new-architecture. CHANGELOG entry for

[43] `ReactPackage.kt:36` (`@Deprecated(message = "Migrate to [BaseReactPackage] and implement [getModule] instead.")`); `ReactPackage.kt:52` declares the supported `getModule(name, reactContext)` hook. Live usage in `MainReactPackage.kt:108`, which switches on `name` and returns lazy-instantiated modules. `BaseReactPackage` provides the boilerplate needed to back the deprecated `createNativeModules` with `getModule` calls so the registry can still enumerate when asked.

v0.76 in `CHANGELOG-0.7x.md`: "TurboModules will be looked up as TurboModules first, and fallback to legacy modules after" (commit `5a62606ab3`).

- [4] The cache is the `jsRepresentation_` weak-object reference in `TurboModule.h:143`. The caching call lives in the `get` body at line 65: `jsRepresentation_->lock(runtime).asObject(runtime).setProperty(runtime, propName, prop)`. The comment near line 62 states explicitly: "We don't cache misses, to allow for methodMap_ to dynamically be extended."
- [5] `RCTModuleProvider` protocol is declared at `RCTTurboModule.h:197`. Its single required method is `- (std::shared_ptr<facebook::react::TurboModule>)getTurboModule:(const facebook::react::ObjCTurboModule::InitParams &)params`. This split landed so that pure-C++ TurboModules (without an Objective-C class) can still register through the same provider surface.
- [6] `RCTTurboModule.mm` constructs the `NSInvocation` at line 725 (`NSInvocation *inv = [NSInvocation invocationWithMethodSignature:methodSignature]`), sets the selector at 726, fills each argument via `setInvocationArg` (728-732), and finally invokes the selector on `instance_` in `performMethodInvocation` (line 780+). The full call site is `ObjCTurboModule::invokeObjCMethod` at lines 760-820.
- [7] `RCT_EXPORT_METHOD` is defined at `packages/react-native/React/Base/RCTBridgeModule.h:263` (`#define RCT_EXPORT_METHOD(method) RCT_REMAP_METHOD(, method)`). Legacy bridge dispatch routes the call through the batched JSON message queue, so the value comes back via the callback block, not as a return value. The true-sync alternative `RCT_EXPORT_BLOCKING_SYNCHRONOUS_METHOD` is at line 281 and is documented with a "WARNING: ..." comment at line 269.
- [8] CodeGen ObjC++ generator: `packages/react-native-codegen/src/generators/modules/GenerateModuleObjCpp/index.js:36` emits `@protocol ${hasteModuleName}Spec <RCTBridgeModule, RCTTurboModule>` and `:55-58` emits `class JSI_EXPORT ${hasteModuleName}SpecJSI : public ObjCTurboModule { public: ${hasteModuleName}SpecJSI(const ObjCTurboModule::InitParams &params); };`. Java generator: `GenerateModuleJavaSpec.js:62` emits `public abstract class ${className} extends ReactContextBaseJavaModule implements TurboModule` and `:63` emits `public static final String NAME = "${jsName}";`. The full round-trip is exercised in `private/react-native-codegen-typescript-test/src/__tests__/simple-scenario-frontend-test.ts:14-40` and reproduced for this chapter at `_verification/chapters/03-turbo-modules/tests/spec_objc_gen.js`.
- [9] TS and Flow parsers live side by side: `packages/react-native-codegen/src/parsers/typescript/` and `.../parsers/flow/`. Every shipped `Native*` spec under `packages/react-native/src/private/specs_DEPRECATED/modules/` uses Flow (see e.g. `NativeAccessibilityInfo.js`). The sample spec `NativeSampleTurboModule.js` moved to `packages/rn-tester/js/examples/TurboModule/` (`4ffc4a65231`); third-party modules under the New Arch are typically authored in TypeScript.
- [10] `package com.facebook.react.turbomodule.core.interfaces ... public interface TurboModule { public fun initialize(); public fun invalidate(); }`

(`TurboModule.kt:8-19`). The two methods are lifecycle hooks the bridge calls on init and on `ReactHost` tear-down.

- [11] Confirmed by inspection of any RN-shipped module that's already migrated. For example, `BlobModule.kt:44-45` reads `class BlobModule(reactContext: ReactApplicationContext) : NativeBlobModuleSpec(reactContext), TurboModuleWithJSIBindings`. It extends the generated spec class and only adds `TurboModuleWithJSIBindings` (a separate interface for modules that install their own JSI bindings). It does NOT re-declare `TurboModule`, because the spec class already implements it.
- [12] `ReactPackage.kt:36` (`@Deprecated(message = "Migrate to [BaseReactPackage] and implement [getModule] instead.")`); `ReactPackage.kt:52` declares the supported `getModule(name, reactContext)` hook. Live usage in `MainReactPackage.kt:108`, which switches on `name` and returns lazy-instantiated modules. `BaseReactPackage` provides the boilerplate needed to back the deprecated `createNativeModules` with `getModule` calls so the registry can still enumerate when asked.
- [13] `RCTTurboModuleEnabled` / `RCTEnableTurboModule` declared deprecated in `React/Base/RCTBridge.h` and `RCTBridge.mm` (commits `aee4a2586e`, `feede1422b`). Both were deleted on main by `01d55273bcf` ("Delete dead bridge flag functions and module hooks"), so `v0.88.1:packages/react-native/React/Base/RCTBridge.h:50` still declares `RCTTurboModuleEnabled`, and HEAD does not. Interop flags (`RCTEnableTurboModuleInterop`) remain, behind `RCT_REMOVE_LEGACY_MODULE_INTEROP`. Feature-flag removal: CHANGELOG v0.87.0-rc.0, "Remove the `useTurboModules` feature flag. TurboModules are now always enabled." (`b4a715ca1f`).
- [14] C++ bridging: `packages/react-native/ReactCommon/react/bridging/ArrayBuffer.h` (`Bridging<jsi::ArrayBuffer>`, `AsyncArrayBuffer::{acquire,borrow}`). Added in v0.87.0-rc.0 (`226ef2e7c5`). Java support: `5fb3ebce1a` (CHANGELOG v0.88.0). `ArrayBufferKind` is enumerated in `TurboModule.h:36` alongside the other argument kinds.
- [15] ObjC conversion: `packages/react-native/ReactCommon/react/nativemodule/core/platform/ios/ReactCommon/RCTTurboModule.mm` (commit `54b88d9dea`, CHANGELOG v0.87.0). On main the file uses `convertRCTArrayBufferToJSIArrayBuffer` and `RCTArrayBuffer` (`React/Base/RCTArrayBuffer.h`, commit `11f9a7f4491`), and `GenerateModuleObjCpp/serializeMethod.js:230,401` emit `RCTArrayBuffer *`. At `v0.88.1` the same lines emit `NSMutableData *`. ObjC CodeGen emits `ArrayBufferKind` via `serializeMethod.js`.
- [16] Java type: `packages/react-native/ReactAndroid/src/main/java/com/facebook/react/bridge/ArrayBuffer.kt` (`isOwningBytes`, borrow contract; commit `d84c13d5111`, CHANGELOG v0.88.0). Java generator maps `ArrayBufferTypeAnnotation` to `com.facebook.react.bridge.ArrayBuffer` (`GenerateModuleJavaSpec.js:284,381`) and JNI uses `Lcom/facebook/react/bridge/ArrayBuffer;` (`GenerateModuleJniCpp.js:313,399`). `throwIfUnsupportedPromiseArrayBuffer` in `packages/react-native-codegen/src/generators/modules/Utils.js:95` rejects `Promise<ArrayBuffer>` ("is not supported for Android (Java/Kotlin) or iOS (ObjC) TurboModules"). The old message "ArrayBuffer is only supported for C++ TurboModules" no longer exists in the generators.

Chapter 5: CodeGen - The Automation Engine

The New Architecture's goals of performance and type safety are enabled by JSI, Fabric, and TurboModules. The glue that binds them together and makes the system practical for developers is **CodeGen**. CodeGen is a build-time tool that automates the creation of the boilerplate interface code required to connect the JavaScript world to the native C++ core, as detailed in the official "Using Codegen" guide.[44]

Its source code lives in the repository at `packages/react-native-codegen/`[45], shipped as the `@react-native/codegen` npm package.[46]

The Motivation: Eliminating Manual Glue Code

In the legacy architecture, connecting JavaScript to native required developers to write and maintain two separate interfaces by hand. This manual process was tedious and notoriously error-prone:

- A typo in a method name would lead to a runtime "red box" error.
- A mismatched argument type could cause a crash or silent failure.
- There was no single source of truth. The JavaScript and native code could easily drift out of sync.

CodeGen was created to solve this problem by making the JavaScript spec the **single source of truth** and automating the generation of the native-side interface code.

The CodeGen Pipeline

The CodeGen process is a classic compiler pipeline that runs during the application's build phase. On iOS it is invoked from the CocoaPods install script (`packages/react-native/scripts/cocoapods/codegen.rb`). On Android the Gradle plugin runs the same generators via the shared executor in `packages/react-native/scripts/codegen/generate-artifacts-executor/`.[47]

1. Discovery and parsing. A combiner scans the package's source tree for files that contain either `extends TurboModule` (a native module spec) or `codegenNativeComponent<...>` (a native component spec).[48] Each matching file is handed to a Flow or TypeScript parser depending on its extension. The parsers live in `packages/react-native-codegen/src/parsers/flow/` and `packages/react-native-codegen/src/parsers/typescript/`.

2. Schema generation. The parser transforms each spec into a single language-agnostic JSON object that represents the complete API contract. This intermediate representation conforms to the `SchemaType` shape declared in `packages/react-native-codegen/src/CodegenSchema.d.ts`

[44] "Using Codegen". React Native Documentation. https://reactnative.dev/docs/the-new-architecture/using-codegen.

[45] `packages/react-native-codegen/` (RN HEAD `9c7b0f5fafd`).

[46] `packages/react-native-codegen/package.json` (`"name": "@react-native/codegen"`, current version `0.87.0-main`).

[47] `packages/react-native/scripts/cocoapods/codegen.rb:9-39` (the `run_codegen!` Ruby helper invoked by `use_react_native_codegen_discovery!`); `packages/react-native/scripts/codegen/generate-artifacts-executor/` (cross-platform driver).

[48] `packages/react-native-codegen/src/cli/combine/combine-js-to-schema.js:24-35` (`combineSchemas` filters files via `/extends TurboModule/` or `/export\s+default\s+\(?codegenNativeComponent</`).

(TypeScript) and its Flow twin `CodegenSchema.js`.[49] A separate `SchemaValidator.js` checks the combined schema for issues like duplicate module names across components and modules before generation runs.[50]

3. Code generation. The schema is passed to the **generators** under `src/generators/`. The dispatcher in `RNCodegen.js` selects a generator set per target: `modulesIOS`, `modulesAndroid`, and `modulesCxx` for native modules; `componentsIOS` and `componentsAndroid` for Fabric components.[51] Each generator is a JavaScript file that emits source strings via template literals. The output is written to disk in the project's `build/generated/ios/` (or its Android equivalent) and then compiled by the platform toolchain.

A Concrete Example

Let's trace the process for a simple TurboModule.

Step 1: The developer writes the spec in TypeScript.

```
// NativeMyModule.ts
import type {TurboModule} from 'react-native';
import {TurboModuleRegistry} from 'react-native';

export interface Spec extends TurboModule {
  readonly getConstants: () => {PI: number};
  readonly getSyncValue: (a: number, b: string) => string;
  readonly getValueWithPromise: () => Promise<boolean>;
}

export default TurboModuleRegistry.getEnforcing<Spec>('MyModule');
```

Two things worth noticing here. First, import from the public `'react-native'` entry — under the Strict TypeScript API (default since v0.87.0), deep `Libraries/TurboModule/...` imports have `"types": null` unless you opt into `react-native-legacy-deep-imports`.[52] Second, **a TurboModule spec interface may only contain function-typed properties (and non-nullable event emitters).** Bare data fields like `readonly PI: number;` are rejected at parse time with `UnsupportedModulePropertyParserError` ("TS interfaces extending TurboModule must only

[49] `packages/react-native-codegen/src/CodegenSchema.d.ts:12` (`interface SchemaType`); `packages/react-native-codegen/src/CodegenSchema.js` (Flow twin).

[50] `packages/react-native-codegen/src/SchemaValidator.js:17` (`function getErrors(schema: SchemaType)`).

[51] `packages/react-native-codegen/src/generators/RNCodegen.js:125-181` (the `LIBRARY_GENERATORS` map: `modulesIOS: [generateModuleObjCpp.generate]`, `modulesAndroid: [generateModuleJniCpp.generate, generateModuleJniH.generate, generateModuleJavaSpec.generate]`, `modulesCxx: [generateModuleH.generate]`).

[52] Prefer `import { TurboModuleRegistry } from 'react-native'` / `import type { TurboModule } from 'react-native'` under the Strict TypeScript API. The deep module still implements `get` / `getEnforcing` as named exports (`Libraries/TurboModule/TurboModuleRegistry.js:35,39`). The public type surface is `ReactNativeApi.d.ts`, which declares `namespace TurboModuleRegistry { export { get_2 as get, getEnforcing } }` (line 5406).

contain 'FunctionTypeAnnotation's or non nullable 'EventEmitter's").[53] Constants are surfaced through the `getConstants` method whose return type carries the constant shape, which is exactly the pattern the canonical samples follow.[54]

Step 2: CodeGen generates the C++ Spec Header.

The output depends on which generator target produced it. There are three families, each with its own conventions:

iOS (`modulesIOS`). The generator writes a `<LibraryName>.h` header that declares an Objective-C protocol the developer implements, plus a C++ wrapper class that bridges that protocol to JSI:[55]

```
// Generated by GenerateModuleObjCpp (excerpt)

@protocol NativeMyModuleSpec <RCTBridgeModule, RCTTurboModule>
- (facebook::react::ModuleConstants<JS::NativeMyModule::Constants>)getConstants;
- (NSString *)getSyncValue:(double)a b:(NSString *)b;
- (void)getValueWithPromise:(RCTPromiseResolveBlock)resolve
                     reject:(RCTPromiseRejectBlock)reject;
@end

namespace facebook::react {
  class JSI_EXPORT NativeMyModuleSpecJSI : public ObjCTurboModule {
   public:
    NativeMyModuleSpecJSI(const ObjCTurboModule::InitParams &params);
  };
} // namespace facebook::react
```

The `SpecJSI` class is intentionally small. Its constructor registers each method into the inherited `methodMap_` and hands a static `__hostFunction_*` thunk that calls `invokeObjCMethod(rt, kind, "name", @selector(...), args, count)` at runtime. There are **no pure virtual methods**; the Objective-C protocol is the contract the developer satisfies, and `invokeObjCMethod` uses the protocol selectors at call time.[56]

---

[53] `packages/react-native-codegen/src/parsers/parsers-commons.js:462` (calls `throwIfModuleTypeIsUnsupported`); `packages/react-native-codegen/src/parsers/error-utils.js:219` (the check); `packages/react-native-codegen/src/parsers/errors.js:129` (`UnsupportedModulePropertyParserError`, message at line 140: "interfaces extending TurboModule must only contain 'FunctionTypeAnnotation's or non nullable 'EventEmitter's").

[54] `packages/rn-tester/js/examples/TurboModule/NativeSampleTurboModule.js` (moved out of `src/private/specs_DEPRECATED/modules/` by `4ffc4a65231`; `readonly getConstants: () => { const1: boolean, const2: number, const3: string }`).

[55] `packages/react-native-codegen/src/generators/modules/GenerateModuleObjCpp/index.js:25` (`ModuleDeclarationTemplate`). Real snapshot output for a methods-bearing module: `packages/react-native-codegen/src/generators/modules/__tests__/__snapshots__/GenerateModuleHObjCpp-test.js.snap:800-832` (`@protocol NativeCameraRollManagerSpec + NativeCameraRollManagerSpecJSI : public ObjCTurboModule`).

[56] `packages/react-native-codegen/src/generators/modules/__tests__/__snapshots__/GenerateModuleMm-test.js.snap:130-181` (the `.mm` source registers each method with `methodMap_["name"] = MethodMetadata{argCount, __hostFunction_..._name}` and the static thunks call `invokeObjCMethod(rt, Kind, "name", @selector(...), args, count)`).

Android (`modulesAndroid`). The Android generators emit a `JavaTurboModule`-based `SpecJSI` plus a Java/Kotlin abstract class (e.g., `NativeMyModuleSpec`) for the developer to subclass. The C++ side is again a thin shim that routes calls through JNI:

```
// Generated by GenerateModuleJniH.js (excerpt)
class JSI_EXPORT NativeMyModuleSpecJSI : public JavaTurboModule {
 public:
  NativeMyModuleSpecJSI(const JavaTurboModule::InitParams &params);
};
```

Pure C++ (`modulesCxx`). When a TurboModule is implemented entirely in C++ (no platform code), the generator emits a CRTP-style class template that **the developer's class derives from**:[57]

```
// Generated by GenerateModuleH.js (excerpt)

template <typename T>
class JSI_EXPORT NativeMyModuleCxxSpec : public TurboModule {
 public:
  static constexpr std::string_view kModuleName = "MyModule";

 protected:
  NativeMyModuleCxxSpec(std::shared_ptr<CallInvoker> jsInvoker)
      : TurboModule(std::string{kModuleName}, jsInvoker) {
    methodMap_["getSyncValue"] = MethodMetadata{
        .argCount = 2, .invoker = __getSyncValue};
    methodMap_["getValueWithPromise"] = MethodMetadata{
        .argCount = 0, .invoker = __getValueWithPromise};
    methodMap_["getConstants"] = MethodMetadata{
        .argCount = 0, .invoker = __getConstants};
  }

 private:
  static jsi::Value __getSyncValue(
      jsi::Runtime &rt,
      TurboModule &tm,
      const jsi::Value *args,
      size_t count) {
    static_assert(
        bridging::getParameterCount(&T::getSyncValue) == 3,
        "Expected getSyncValue(...) to have 3 parameters");
    return bridging::callFromJs<jsi::String>(
        rt,
        &T::getSyncValue,
        static_cast<NativeMyModuleCxxSpec *>(&tm)->jsInvoker_,
        static_cast<T *>(&tm),
```

[57] `packages/react-native-codegen/src/generators/modules/GenerateModuleH.js:160` (`ModuleSpecClassDeclarationTemplate`). Real snapshot: `packages/react-native-codegen/src/generators/modules/__tests__/__snapshots__/GenerateModuleH-test.js.snap:43-105` (the `array_buffer_native_module` fixture).

```
        args[0].asNumber(),
        args[1].asString(rt));
  }
  // ... __getValueWithPromise, __getConstants ...
};
```

Note three things. The class name ends in `CxxSpec`, not `SpecJSI`. It is a `template <typename T>` (the `T` is the developer's concrete subclass). Dispatch goes through static `__methodName` callbacks that forward to `T::methodName` via `bridging::callFromJs`, not through C++ virtual methods. Compile-time safety here comes from the `static_assert(bridging::getParameterCount(&T::method) == N, ...)` lines, which fail to substitute if the developer's class is missing a method or its arity is wrong.[58]

Step 3: The developer implements the generated interface.

What "implementing the interface" means is platform-specific:

- On iOS, the developer writes an Objective-C class that conforms to the `<NativeMyModuleSpec>` protocol. The Objective-C compiler emits `-Wprotocol` warnings for unimplemented methods, and Xcode flags wrong selector signatures.
- On Android, the developer subclasses the generated Java/Kotlin abstract class `NativeMyModuleSpec`. Missing methods are a compile error.
- For the pure C++ Cxx variant, the developer writes a class that derives from `NativeMyModuleCxxSpec<MyClass>` (CRTP) and provides ordinary member functions with matching names and signatures. A missing method, or a parameter count mismatch, fails the `static_assert` and the build stops at the generated header.

In every case the JS-to-native shape is fixed by the spec and re-validated at compile time on the native side. There is no path where the runtime gets to discover a name or argument mismatch and crash later.

Platform matrix for newer types (v0.87)

Not every annotation lands on every generator. `ArrayBuffer` / `ArrayBufferTypeAnnotation` is a first-class schema type. The **C++** generator (`GenerateModuleH.js`) and **ObjC** generator (`GenerateModuleObjCpp`) emit it since v0.87. The **Java** and **JNI** generators emit it since v0.88 (`com.facebook.react.bridge.ArrayBuffer`, JNI signature `Lcom/facebook/react/bridge/ArrayBuffer;`). The C++ fixture ships a `getArrayBuffer` / `voidArrayBuffer` sample in the snapshot tests.[59] One shared guard, `throwIfUnsupportedPromiseArrayBuffer`, rejects `Promise<ArrayBuffer>` on Java and ObjC. `ArrayBuffer` as an argument or as a synchronous

[58] `packages/react-native-codegen/src/generators/modules/__tests__/__snapshots__/GenerateModuleH-test.js.snap:74-78` (`static_assert(bridging::getParameterCount(&T::getArrayBuffer) == 1, "Expected getArrayBuffer(...) to have 1 parameters");`).

[59] `packages/react-native-codegen/src/generators/modules/__tests__/__snapshots__/GenerateModuleH-test.js.snap:74-78` (`static_assert(bridging::getParameterCount(&T::getArrayBuffer) == 1, "Expected getArrayBuffer(...) to have 1 parameters");`).

return value works on every platform.[60] On the ObjC side, the generated type is `NSMutableData *` on v0.87.0 and v0.88.1, and `RCTArrayBuffer *` on main after v0.88.1. The 0.88 changelog records a revert of the `RCTArrayBuffer` change on the 0.88 line. Discovery also got fixes after 0.87: the crawler skips `node_modules` and symlinks when it searches for components (`39751d864dd`), and it reads `react-native.config.cjs` so ESM projects are no longer skipped silently (`583c8f3afb2`). Separately, pure-C++ TurboModules no longer require `includesGeneratedCode: true` in `codegenConfig` for Android autolinking to pick them up.[61]

The Benefit: Compile-Time Safety

This automated process is the key to the New Architecture's type safety. By generating native interfaces from a single JavaScript source of truth, it makes the native implementation impossible to drift out of sync with the JavaScript API without a build failure. The exact mechanism is different on each target (protocol conformance on iOS, abstract subclass on Android, CRTP plus `static_assert` for Cxx), but the contract is the same: a class of fragile, runtime-dependent errors moves into easy-to-diagnose compile-time errors.

Citations:

- [1] `packages/react-native-codegen/` (RN HEAD `9c7b0f5fafd`).
- [2] "Using Codegen". React Native Documentation. https://reactnative.dev/docs/the-new-architecture/using-codegen.
- [3] `packages/react-native-codegen/package.json` (`"name": "@react-native/codegen"`, current version `0.87.0-main`).
- [4] `packages/react-native/scripts/cocoapods/codegen.rb:9-39` (the `run_codegen!` Ruby helper invoked by `use_react_native_codegen_discovery!`); `packages/react-native/scripts/codegen/generate-artifacts-executor/` (cross-platform driver).
- [5] `packages/react-native-codegen/src/cli/combine/combine-js-to-schema.js:24-35` (`combineSchemas` filters files via `/extends TurboModule/` or `/export\s+default\s+\(?codegenNativeComponent</`).
- [6] `packages/react-native-codegen/src/CodegenSchema.d.ts:12` (`interface SchemaType`); `packages/react-native-codegen/src/CodegenSchema.js` (Flow twin).
- [7] `packages/react-native-codegen/src/SchemaValidator.js:17` (`function getErrors(schema: SchemaType)`).

[60] `packages/react-native-codegen/src/generators/modules/Utils.js:95` (`throwIfUnsupportedPromiseArrayBuffer`, rejects `Promise<ArrayBuffer>`). Java maps the type in `GenerateModuleJavaSpec.js:284,381`, JNI in `GenerateModuleJniCpp.js:313,399`. The Java class is `ReactAndroid/.../bridge/ArrayBuffer.kt` (CHANGELOG v0.88.0). ObjC emits `ArrayBufferKind` and `RCTArrayBuffer *` on main via `GenerateModuleObjCpp/serializeMethod.js:230,401` (`11f9a7f4491`). At `v0.88.1` those lines emit `NSMutableData *`. Schema type: `CodegenSchema.d.ts:49` `ArrayBufferTypeAnnotation`. The old text "ArrayBuffer is only supported for C++ TurboModules" is gone from the generators.

[61] CHANGELOG v0.87.0 Fixed: "Fix pure C++ turbo modules not working without `includesGeneratedCode: true`" (`c789880517`).

- [8] `packages/react-native-codegen/src/generators/RNCodegen.js:125-181` (the `LIBRARY_GENERATORS` map: `modulesIOS: [generateModuleObjCpp.generate]`, `modulesAndroid: [generateModuleJniCpp.generate, generateModuleJniH.generate, generateModuleJavaSpec.generate]`, `modulesCxx: [generateModuleH.generate]`).
- [9] Prefer `import { TurboModuleRegistry } from 'react-native'` / `import type { TurboModule } from 'react-native'` under the Strict TypeScript API. The deep module still implements `get` / `getEnforcing` as named exports (`Libraries/TurboModule/TurboModuleRegistry.js:35,39`). The public type surface is `ReactNativeApi.d.ts`, which declares `namespace TurboModuleRegistry { export { get_2 as get, getEnforcing } }` (line 5406).
- [10] `packages/react-native-codegen/src/parsers/parsers-commons.js:462` (calls `throwIfModuleTypeIsUnsupported`); `packages/react-native-codegen/src/parsers/error-utils.js:219` (the check); `packages/react-native-codegen/src/parsers/errors.js:129` (`UnsupportedModulePropertyParserError`, message at line 140: "interfaces extending TurboModule must only contain 'FunctionTypeAnnotation's or non nullable 'EventEmitter's").
- [11] `packages/rn-tester/js/examples/TurboModule/NativeSampleTurboModule.js` (moved out of `src/private/specs_DEPRECATED/modules/` by `4ffc4a65231`; `readonly getConstants: () => { const1: boolean, const2: number, const3: string }`).
- [12] `packages/react-native-codegen/src/generators/modules/GenerateModuleObjCpp/index.js:25` (`ModuleDeclarationTemplate`). Real snapshot output for a methods-bearing module: `packages/react-native-codegen/src/generators/modules/__tests__/__snapshots__/GenerateModuleHObjCpp-test.js.snap:800-832` (`@protocol NativeCameraRollManagerSpec + NativeCameraRollManagerSpecJSI : public ObjCTurboModule`).
- [13] `packages/react-native-codegen/src/generators/modules/__tests__/__snapshots__/GenerateModuleMm-test.js.snap:130-181` (the `.mm` source registers each method with `methodMap_["name"] = MethodMetadata{argCount, __hostFunction_..._name}` and the static thunks call `invokeObjCMethod(rt, Kind, "name", @selector(...), args, count)`).
- [14] `packages/react-native-codegen/src/generators/modules/GenerateModuleH.js:160` (`ModuleSpecClassDeclarationTemplate`). Real snapshot: `packages/react-native-codegen/src/generators/modules/__tests__/__snapshots__/GenerateModuleH-test.js.snap:43-105` (the `array_buffer_native_module` fixture).
- [15] `packages/react-native-codegen/src/generators/modules/__tests__/__snapshots__/GenerateModuleH-test.js.snap:74-78` (`static_assert(bridging::getParameterCount(&T::getArrayBuffer) == 1, "Expected getArrayBuffer(...) to have 1 parameters");`).
- [16] `packages/react-native-codegen/src/generators/modules/Utils.js:95` (`throwIfUnsupportedPromiseArrayBuffer`, rejects `Promise<ArrayBuffer>`). Java maps the type in `GenerateModuleJavaSpec.js:284,381`, JNI in `GenerateModuleJniCpp.js:313,399`. The Java class is `ReactAndroid/.../bridge/ArrayBuffer.kt` (CHANGELOG v0.88.0). ObjC emits `ArrayBufferKind` and `RCTArrayBuffer *` on main via `GenerateModuleObjCpp/serializeMethod.js:230,401` (`11f9a7f4491`). At `v0.88.1` those lines emit `NSMutableData *`.

Schema type: `CodegenSchema.d.ts:49` `ArrayBufferTypeAnnotation`. The old text "ArrayBuffer is only supported for C++ TurboModules" is gone from the generators.

- [17] CHANGELOG v0.87.0 Fixed: "Fix pure C++ turbo modules not working without `includesGeneratedCode: true`" (`c789880517`).

Chapter 6: Bridgeless - The Ultimate Goal

The introduction of the JSI, Fabric, and TurboModules provides the foundational pieces to replace the legacy architecture. However, for a time, both the old and new systems could coexist in a single application to provide backward compatibility. The ultimate goal of the new architecture, however, was to completely remove the old one. This final state is known as **Bridgeless Mode**.

Bridgeless Mode was first made available as an opt-in with React Native 0.73 (December 2023), became the default for New Architecture-enabled apps in React Native 0.74 (April 22, 2024)[62], and has been on for every new project since React Native 0.76 (October 2024), when the New Architecture itself became default-on[63]. Opt-out of the New Architecture (and therefore of Bridgeless) was removed in React Native 0.82. At main HEAD (v0.88.1 era), the legacy non-bridgeless path on iOS is gone from the API: the `bridgelessEnabled` property and the `bridgelessEnabled:/turboModuleEnabled:` initializers were **deleted** from `RCTRootViewFactoryConfiguration`, and the old `RCTArchConfiguratorProtocol` (which exposed the `bridgelessEnabled`/`newArchEnabled`/`turboModuleEnabled`/`fabricEnabled` toggles) has been **removed from the public headers** rather than merely deprecated[64]. The two remaining `newArchEnabled:` initializers are deprecated and ignore the flag. TurboModules are always enabled: the `useTurboModules` flag is gone (v0.87), and the iOS `RCTTurboModuleEnabled()` / `RCTEnableTurboModule()` functions are deleted from `RCTBridge.h` on main.

What is Bridgeless Mode?

Bridgeless mode is not a new component or a separate piece of technology. It is the architectural state where the original, asynchronous message-based Bridge is **no longer initialized or used**. In this mode, communication between JavaScript and the native platform flows through the JavaScript Interface (JSI), not through the legacy JSON message queue.

Concretely, instead of an `RCTBridge` / `CatalystInstance` pair, the runtime is owned by a `ReactInstance` (C++) sitting under a platform `ReactHost` (`RCTHost` on iOS, `ReactHost` on Android)[65]. On Android, `BridgelessReactContext` replaces `ReactApplicationContext`; its source

[62] "React Native 0.74 - Yoga 3.0, Bridgeless New Architecture, and more." React Native Blog, April 22, 2024. https://reactnative.dev/blog/2024/04/22/release-0.74. The 0.74 changelog lists "Make bridgeless the default when the New Arch is enabled" twice in `## 0.74.0` (once Android, once iOS) at `CHANGELOG-0.7x.md:2076` (commit `fe337f25be`) and `CHANGELOG-0.7x.md:2114` (commit `c91af773fa`).

[63] "React Native 0.76 - New Architecture by default." React Native Blog, October 23, 2024. https://reactnative.dev/blog/2024/10/23/release-0.76-new-architecture.

[64] `packages/react-native/Libraries/AppDelegate/RCTRootViewFactory.h:59-68` (only `initWithBundleURLBlock:` / `initWithBundleURL:` plus two `newArchEnabled:` variants marked `__deprecated_msg("The flag is ignored; ...")`). `bridgelessEnabled` was removed from `RCTRootViewFactoryConfiguration` in `2a9eca81b8d`, and the Fabric/TurboModule toggles in `562fe3433af`. `RCTArchConfiguratorProtocol.h` is no longer shipped as a public header (removed in commits such as `2148f15db5` / `ad9936faf6`, and `scripts/ios-prebuild/__docs__/headers-rules.md:217` calls it "since-removed"); leftover `#pragma mark - RCTArchConfiguratorProtocol` comments in `RCTDefaultReactNativeFactoryDelegate.mm` are historical markers, not a live toggle surface.

[65] iOS: `packages/react-native/ReactCommon/react/runtime/platform/ios/ReactCommon/RCTHost.h:75-96` (`RCTHostJSEngineProvider` typedef at line 75, initializers takes a `RCTHostJSEngineProvider`, no `RCTBridge` anywhere). C++ runtime: `packages/react-native/ReactCommon/react/runtime/ReactInstance.h:27` (`class ReactInstance final : private jsinspector_modern::InstanceTargetDelegate`). Android: `packages/react-native/ReactAndroid/src/main/java/com/facebook/react/ReactHost.kt:23-27` ("A ReactHost is an object

comment is blunt about this: "This class is used instead of ReactApplicationContext when React Native is operating in bridgeless mode. The purpose of this class is to override some methods on ReactContext that use the CatalystInstance, which doesn't exist in bridgeless mode."[66] On iOS, anywhere old code asks for the `[RCTBridge currentBridge]`, RN substitutes an `RCTBridgeProxy` (an `NSProxy` that forwards `RCTBridgeModule` selectors to the bridgeless module registry) so that legacy modules keep working without a real bridge underneath[67].

In an earlier intermediate state (the New Architecture enabled but `bridgelessEnabled = NO`, which was possible in 0.73 and early 0.74 betas), the Bridge could still be initialized alongside Fabric and TurboModules to support unmigrated native modules. In Bridgeless mode that path is gone: no JSON message queue, no `MessageQueue.js` global serializer, no batched bridge.

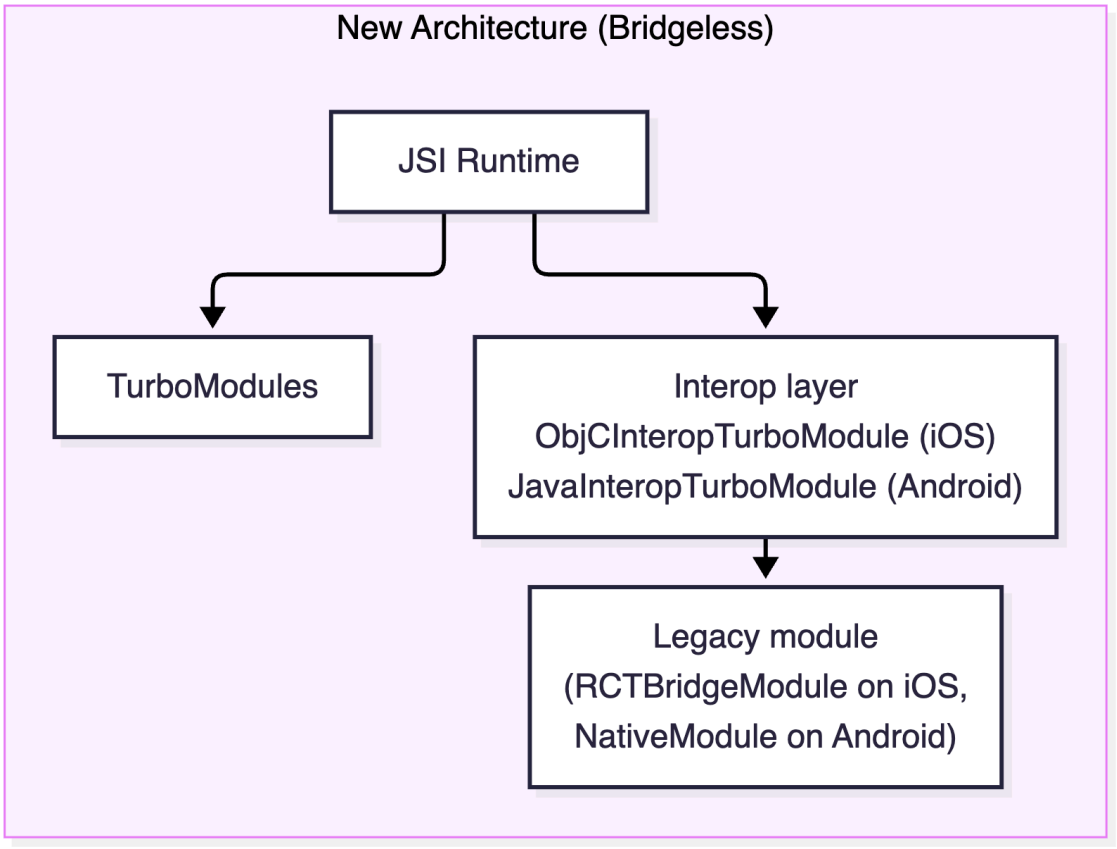
---

that manages a single `com.facebook.react.runtime.ReactInstance`. A ReactHost can be constructed without initializing the ReactInstance, and it will continue to exist after the instance is destroyed.").

[66] `packages/react-native/ReactAndroid/src/main/java/com/facebook/react/runtime/BridgelessReactContext.kt:41-45`. The companion `BridgelessCatalystInstance` (same package) is annotated `@LegacyArchitecture(logLevel = LegacyArchitectureLogLevel.ERROR)` because every method on it throws `UnsupportedOperationException`. It exists only to keep old code that types against `CatalystInstance` compilable; using it at runtime is an error.

[67] `packages/react-native/React/Base/RCTBridgeProxy.h:20` (`@interface RCTBridgeProxy : NSProxy`). `packages/react-native/ReactCommon/react/runtime/platform/ios/ReactCommon/RCTInstance.mm:308-329` instantiates the proxy in the bridgeless setup path, and line 332 hands it to `RCTTurboModuleManager` (`initWithBridgeProxy:`) so legacy modules that read `self.bridge` get the proxy. Line 390 also inserts it in the `contextContainer`. (Older text said `RCTInstance` calls `[RCTBridge setCurrentBridge:]`. That call is no longer in this file on main.)

Visual: bridgeless module routing



A modern TurboModule is reached through a CodeGen'd C++ binding installed on the JSI runtime. A legacy module is reached through one of two interop classes: `ObjCInteropTurboModule` on iOS[68] or `JavaInteropTurboModule` on Android[69]. Both subclass the platform's regular TurboModule base, expose the module's `RCT_EXPORT_METHOD` / `@ReactMethod` selectors as JSI properties via runtime introspection, and use a per-module method queue rather than the old global bridge queue.

The Benefits of a Bridgeless World

Running without the Bridge unlocks the full potential of the New Architecture, leading to several key benefits:

1. **Improved Performance:** This is the most significant advantage. By removing the need to initialize the Bridge and its modules at startup, apps launch faster. By eliminating the

[68] `packages/react-native/ReactCommon/react/nativemodule/core/platform/ios/ReactCommon/RCTInteropTurboModule.h:24` (`class JSI_EXPORT ObjCInteropTurboModule : public ObjCTurboModule`). Method introspection: `RCTInteropTurboModule.mm:51-73` (`getMethodInfos` walks the class hierarchy looking for `__rct_export__`-prefixed selectors emitted by the `RCT_EXPORT_METHOD` macro).

[69] `packages/react-native/ReactCommon/react/nativemodule/core/platform/android/ReactCommon/JavaInteropTurboModule.h:23` (`class JSI_EXPORT JavaInteropTurboModule : public JavaTurboModule`).

JSON serialization and the single batched message queue, communication latency is reduced. In code: `BridgelessNativeMethodCallInvoker` posts a raw `std::function<void()>` to a `MessageQueueThread` instead of marshalling `folly::dynamic` -> JSON -> bridge queue[70].

2. **Simplified Architecture and Threading:** Without the Bridge, the mental model for developers becomes simpler. There is no longer a JSON-serialized asynchronous boundary to reason about. The threading model is still multi-threaded (JS thread, UI thread, background module queues), but coordination is done through the `RuntimeScheduler` and the `CallInvoker` abstraction, not through a global batched-bridge dispatcher[71].
3. **Unlocking Modern React Features:** Concurrent rendering and Suspense need synchronous reads from the renderer (for example, `measure`, `measureInWindow`, and the layout effect commit path). The legacy Bridge could not serve those without blocking the JS thread on a round-trip. With Fabric on a JSI-driven runtime, the renderer makes those reads inline through C++ shared state, which is what React 18's `concurrentRoot: true` requires on the `render()` call[72].

A note on "synchronous." JSI itself is synchronous: `jsi::Runtime::call`, `jsi::Object::getProperty`, and friends are direct C++ calls that run on whatever thread invokes them[73]. But a TurboModule method can still be async by design (returning a `Promise`, or annotated `async` on Android). In that case the call still leaves the JS thread via `NativeMethodCallInvoker::invokeAsync`, which posts to a `MessageQueueThread`. So "Bridgeless is fully synchronous" is too strong. The accurate framing is: JSON serialization and the single global message queue are gone, and every call now enters native code through a JSI binding, but cross-thread dispatch is still real and still scheduled.

The Interoperability Layer

The React Native team understood that a hard cutover to the new architecture was not feasible for the vast ecosystem of existing apps and third-party libraries. To facilitate a gradual migration, an interoperability layer was created.

This layer allows legacy, Bridge-based native modules to continue functioning in a Bridgeless world. When JavaScript calls a method on a legacy module, the call enters native code through a JSI TurboModule binding, but the TurboModule on the other side is one of the interop wrappers (`ObjCInteropTurboModule` on iOS, `JavaInteropTurboModule` on Android). The wrapper discov-

[70] `packages/react-native/ReactCommon/react/runtime/BridgelessNativeMethodCallInvoker.cpp:16-25`. `invokeAsync` calls `messageQueueThread_->runOnQueue(std::move(func));` `invokeSync` calls `runOnQueueSync`. No JSON, no `folly::dynamic` marshalling, no batched message buffer.

[71] `packages/react-native/ReactCommon/react/runtime/platform/ios/ReactCommon/RCTInstance.mm:306` (`auto jsCallInvoker = _reactInstance->createJSCallInvoker();`). The same `jsCallInvoker` is then passed to `RCTTurboModuleManager`, so TurboModules schedule JS work through the `RuntimeScheduler` rather than the bridge. (`createJSCallInvoker()` replaced the older explicit `RuntimeSchedulerCallInvoker` construction.)

[72] `packages/react-native/Libraries/Renderer/shims/ReactNativeTypes.js:158-164` defines `render(element, containerTag, callback, concurrentRoot, options)` on the Fabric host config. Setting `concurrentRoot: true` is what enables React's concurrent features (the repo now depends on React `^19.3.0`) on a React Native root.

[73] `packages/react-native/ReactCommon/jsi/jsi/jsi.h:705` declares `class JSI_EXPORT Runtime : public IRuntime` with synchronous `Value getProperty(...)`, `bool hasProperty(...)`, `void setPropertyValue(...)`, etc. JSI calls execute inline on the caller's thread; they do not queue work.

ers the legacy module's exported methods via runtime introspection of its `RCT_EXPORT_METHOD` / `@ReactMethod` declarations, converts arguments via `RCTConvert` (or its Java equivalent), and invokes the legacy method through `NSInvocation` / `jmethodID`. The legacy module's own `dispatch_queue_t methodQueue` (iOS) is wrapped in a `LegacyModuleNativeMethodCallInvoker` so the call still lands on the queue the module expected[74].

While this allows for backward compatibility, it does not provide the performance benefits of true TurboModules. The interop path pays runtime introspection cost on first use, ObjC dynamic dispatch / JNI call overhead per invocation, and `RCTConvert`-style argument coercion that CodeGen'd modules skip.

Status as of v0.88: the interop layer itself is now on a deletion track. Shared C++ and platform legacy view-manager interop is compile-gated behind `RCT_REMOVE_LEGACY_COMPONENT_INTEROP`, and legacy TurboModule interop behind `RCT_REMOVE_LEGACY_MODULE_INTEROP`[75]. Builds that define those macros no longer ship the shims. Since 0.88, CocoaPods (`ENV['RCT_REMOVE_LEGACY_MODULE_INTEROP'] = '1'`) and SwiftPM can opt in. Both default to off, and the CocoaPods helper says they "will default to 1 in a future React Native release". The long-term goal is unchanged: migrate the ecosystem to first-class TurboModule and Fabric APIs before those gates become unconditional.

In conclusion, Bridgeless mode represents the fulfillment of the New Architecture's promise: a faster, more efficient, and more capable React Native. It marks a definitive shift away from the original design and sets a new foundation for the future of the framework.

---

[74] `packages/react-native/ReactCommon/react/nativemodule/core/platform/ios/ReactCommon/RCTTurboModuleManager.mm:135` defines `class LegacyModuleNativeMethodCallInvoker : public ModuleNativeMethodCallInvoker`. Lines 385-405 build it from the legacy module's `dispatch_queue_t methodQueue` and wrap the module in `ObjCInteropTurboModule`. The legacy method is invoked via `NSInvocation` against the original `RCT_EXPORT_METHOD` selectors.

[75] CHANGELOG v0.87.0: "Gate shared C++ and Android Java/Kotlin legacy view manager interop behind `RCT_REMOVE_LEGACY_COMPONENT_INTEROP`" (`9885fe95ef`); "Gate legacy TurboModule interop behind `RCT_REMOVE_LEGACY_MODULE_INTEROP`" (`3278f309e2`). Matching `#ifndef` guards appear in `packages/react-native/React/Base/RCTBridge.h:46-56`, `RCTInteropTurboModule.h:12`, `JavaInteropTurboModule.h:12`, `RCTTurboModuleManager.mm:163,237`, and `LegacyViewManagerInteropShadowNode.h:12`. v0.88.0 CHANGELOG (line 81): "Allow consumers to opt into `RCT_REMOVE_LEGACY_MODULE_INTEROP` and `RCT_REMOVE_LEGACY_COMPONENT_INTEROP` from CocoaPods and SwiftPM. Both default to off and will become the default in a future release" (`b933d18177`). Default env wiring: `scripts/react_native_pods.rb:123-124`.

Part 2: Practical Application & Guides

Chapter 7: Migration and Adoption

Understanding the new architecture is the first step; the next is adopting it. For developers with existing React Native applications, this involves a clear migration path. For new applications, the New Architecture is enabled by default as of React Native 0.76,[76] so the process involves learning the new, modern APIs.

Important Note: Opt-out of the New Architecture was removed in React Native 0.82.[77] On Android, setting `newArchEnabled=false` in `gradle.properties` is force-overridden to `true` by the React Gradle plugin and logs an error.[78] On iOS, setting `RCT_NEW_ARCH_ENABLED=0` in your Podfile or shell prints a warning ("is not supported anymore since React Native 0.82") and then gets overwritten to `"1"` inside `use_react_native!` itself.[79] If you must keep the legacy bridge for a project, pin to the last 0.81.x release and plan a real migration before upgrading further.

From v0.87 the hard-wiring goes further: TurboModules are always enabled (`useTurboModules` feature flag removed; the iOS `RCTTurboModuleEnabled()` function was first made always-`YES` and is now deleted from `RCTBridge.h` on main).[80] Separately, legacy module / component interop can be **compiled out** behind `RCT_REMOVE_LEGACY_MODULE_INTEROP` / `RCT_REMOVE_LEGACY_COMPONENT_INTEROP`. Those macros are opt-in today: since v0.88 you can set them from CocoaPods (`ENV['RCT_REMOVE_LEGACY_MODULE_INTEROP'] = '1'`) or SwiftPM. Both default to off, and the changelog says they will become the default in a future release. iOS defaults only `RCT_REMOVE_LEGACY_ARCH=1`. Large chunks of iOS legacy architecture code guarded by `RCT_REMOVE_LEGACY_ARCH` have already been deleted from the tree. Android tooling has also moved: AGP v9 (9.2.1 in `libs.versions.toml` on main), Gradle 9.4.1, min Kotlin 2.0+ (the repo builds with 2.3.21), app `compileSdk` 37, and libraries must use `compileSdk` ≥ 34 (`minCompileSdk = 34`).[81] Node.js `^22.13.0 || ^24.3.0 || >= 26.0.0` is required

[76] RN 0.76.0 release notes (entry `## v0.76.0` in `CHANGELOG-0.7x.md`): "TurboModules will be looked up as TurboModules first, and fallback to legacy modules after." Default-on rollout discussion: https://reactnative.dev/blog/2024/10/23/release-0.76-new-architecture.

[77] RN 0.82.0 changelog (`CHANGELOG.md`, section `## v0.82.0`): "Remove possibility to newArchEnabled=false in 0.82" (commit `d5d21d0614`); "Removed the opt-out from the New Architecture" (commit `83e6eaf693`); "Crash the app if they force the legacy architecture" (commit `dc132a4fd4`). Tag `v0.82.0` was cut on 2025-10-07 (`06c4c425d7a`).

[78] `packages/gradle-plugin/react-native-gradle-plugin/src/main/kotlin/com/facebook/react/ReactRootProjectPlugin.kt:22-86`: the plugin's `apply` body force-sets `NEW_ARCH_ENABLED` and `SCOPED_NEW_ARCH_ENABLED` to `"true"` for every subproject, and `checkLegacyArchProperty` logs the "not supported anymore since React Native 0.82" error if the project tried to disable it.

[79] `packages/react-native/scripts/react_native_pods.rb:544-558` (`warn_if_new_arch_disabled`), called at line 100 inside `use_react_native!`. Line 148 then unconditionally does `ENV["RCT_NEW_ARCH_ENABLED"] = "1"`. The reusable hardcoded source-of-truth is `packages/react-native/scripts/cocoapods/new_architecture.rb:172-174`: `def self.new_arch_enabled; return true; end`.

[80] CHANGELOG v0.87.0: "Remove the `useTurboModules` feature flag. TurboModules are now always enabled." (`b4a715ca1f`). iOS: `RCTTurboModuleEnabled` / `RCTEnableTurboModule` were deprecated and always-YES / no-op (`aee4a2586e`, `feede1422b`), then deleted from `RCTBridge.h` / `RCTBridge.mm` in `01d55273bcf` (2026-10-27, on main after v0.88.1). The `newArchEnabled:` initializers of `RCTRootViewFactoryConfiguration` are deprecated and ignored.

[81] CHANGELOG v0.87.0: "Adopt AGP v9" (`abb071b135`); "Require Node.js >= 22.13.0" (`a0d39e7a9c`); "Gradle to 9.4.1" (`c0ee408c16`); "Bumped min Kotlin version to 2.0+" (`ec1e1ae054`); "Update `compileSdk` and `buildTools` to

(`packages/react-native/package.json` `engines`). React Native's default JavaScript API is now the Strict TypeScript API (opt out temporarily with Metro `customConditions: ["react-native-legacy-deep-imports"]`).[82] Legacy Hermes C++ / `hermesV1Enabled` opt-in paths were removed. Hermes V1 is the only supported engine path (`react_native_pods.rb:132-133` still sets `RCT_HERMES_V1_ENABLED=1` for backward compatibility).[83] iOS also ships a SwiftPM / prebuilt-Core story (React.xcframework + ReactNativeHeaders) that library authors should account for when consuming headers.

0.87 JS / tooling breakages to sweep

Before or while upgrading, search the app and tests for these removals:

Removed / moved	Replacement
`InteractionManager`	`requestIdleCallback` (RuntimeScheduler idle tier)
`'react-native/Libraries/Core/InitializeCore'`	`'react-native/setup-env'`
`react-native/jest-preset` (subpath)	`preset: '@react-native/jest-preset'` (package at `packages/jest-preset`)
Deep `Libraries/...` / `src/private/...` imports	Public `'react-native'` exports (or temporary `react-native-legacy-deep-imports`)

The legacy bridge classes that remain carry `[[deprecated("This API will be removed along with the legacy architecture.")]]` markers, and the iOS build continues to exclude them via the `RCT_REMOVE_LEGACY_ARCH=1` compiler flag (default on).[84][85] Plan for them to disappear entirely.

This chapter is a practical guide to enabling the New Architecture and migrating legacy native modules and components. For the canonical, always-current guide, refer to the official React Native documentation.[86]

37" (`3e8dfb1e64`); "Set minCompileSdk to 34" (`29e5f954c7`, `ReactAndroid/build.gradle.kts:595`). Versions on main: `packages/react-native/gradle/libs.versions.toml` (`agp = "9.2.1"`, `kotlin = "2.3.21"`, `compileSdk = "37"`), `gradle/wrapper/gradle-wrapper.properties` (Gradle 9.4.1).

[82] CHANGELOG v0.87.0: "React Native's default JavaScript API is now the Strict TypeScript API. Use `customConditions: [\"react-native-legacy-deep-imports\"]` to opt out." (`c948b61c05`).

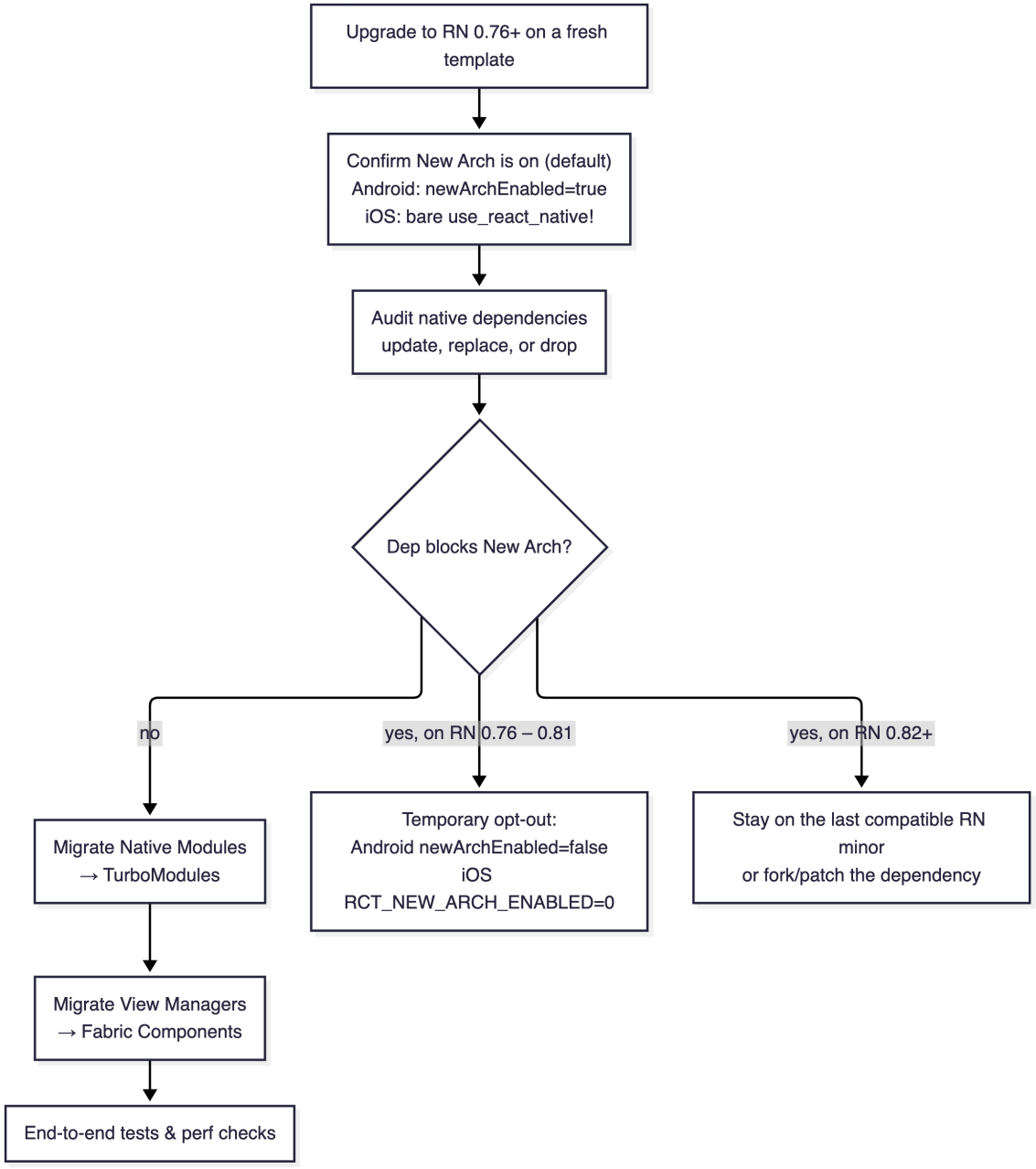
[83] CHANGELOG v0.87.0: "Remove Legacy Hermes from C++ code" (`f9476256bd`); "Remove hermesV1Enabled and simplify the code" (`0a5c8c6b4b`); "Remove the RCT_HERMES_V1_ENABLED from Cocoapods" (`940ee8ae0d`). Also: InteractionManager removed (`ed893f7d98`); InitializeCore → `setup-env` (`bfa679f0ea`); jest-preset package form (`9ee21ddd9b`).

[84] React Native source: the literal string `"This API will be removed along with the legacy architecture."` appears as a `[[deprecated(...)]]` marker across the legacy bridge stack. Representative examples: `packages/react-native/ReactCommon/cxxreact/Instance.h`, `cxxreact/JSExecutor.h`, `cxxreact/NativeToJsBridge.h`, `cxxreact/ModuleRegistry.h`, `jsiexecutor/jsireact/JSIExecutor.h`. On the iOS app shell, `RCTAppDelegate` ("Use `RCTReactNativeFactory` instead.") and bridge accessors on `RCTReactNativeFactory` carry related deprecations. `RCTArchConfiguratorProtocol` was removed from public headers on the v0.87 line.

[85] `packages/react-native/scripts/react_native_pods.rb:120` (`ENV['RCT_REMOVE_LEGACY_ARCH'] = ENV['RCT_REMOVE_LEGACY_ARCH'] == '0' ? '0' : '1'`) and lines 627-631 wire `-DRCT_REMOVE_LEGACY_ARCH=1` into the Pods project's compiler flags.

[86] "About the New Architecture". React Native Documentation. https://reactnative.dev/architecture/landing-page. (The earlier `docs/new-architecture-intro` page was folded into this canonical landing; the old slug now redirects off-site to the reactwg migration guides.)

Visual: Migration Flow



Step 1: Prerequisites and Enabling the New Architecture

Upgrade to React Native 0.76 or newer before migrating. Starting with that release stream, the New Architecture ships enabled by default, so the template projects already opt you in.[87]

For Android:

[87] "About the New Architecture". React Native Documentation. https://reactnative.dev/architecture/landing-page

Keep the `android/gradle.properties` flag in sync with the template:

```
newArchEnabled=true
```

On RN 0.76 through 0.81, flipping the flag to `false` is a valid temporary escape hatch while waiting on a dependency update. From RN 0.82 onward, the React Gradle plugin rejects that value. The relevant code lives in `ReactRootProjectPlugin.checkLegacyArchProperty`, and `apply` also force-sets the properties to `"true"` for every subproject. Any `false` value is logged as an error ("Setting `newArchEnabled=false` in your `gradle.properties` file is not supported anymore since React Native 0.82").[88]

For iOS:

The CocoaPods helper enables the New Architecture automatically. Call `use_react_native!` exactly as in the current template (no additional flags required):

```
use_react_native!(
  :path => config[:reactNativePath],
  :app_path => "#{Pod::Config.instance.installation_root}/.."
)
```

> **Deprecated guidance:** Older instructions recommended adding `:new_arch_enabled => true`. That parameter is still accepted (and documented as `[DEPRECATED]`) but is functionally ignored: `NewArchitectureHelper.new_arch_enabled` is hardcoded to `true` in the cocoapods helper, and `use_react_native!` unconditionally sets `ENV["RCT_NEW_ARCH_ENABLED"] = "1"` partway through its body. Leaving the call unmodified keeps you on the supported path.[89]

On RN 0.76 through 0.81 you could temporarily opt out by setting `ENV['RCT_NEW_ARCH_ENABLED'] = '0'` before invoking the helper, or by passing the same env var on the shell:

```
ENV['RCT_NEW_ARCH_ENABLED'] = '0'
use_react_native!(
  :path => config[:reactNativePath],
  :app_path => "#{Pod::Config.instance.installation_root}/.."
)
```

On RN 0.82+, both forms (Podfile-side `ENV` write and `RCT_NEW_ARCH_ENABLED=0 bundle exec pod install`) print the yellow warning `"is not supported anymore since React Native 0.82"` from `warn_if_new_arch_disabled`, then the helper overrides the env var to `"1"` later in the same

[88] `packages/gradle-plugin/react-native-gradle-plugin/src/main/kotlin/com/facebook/react/ReactRootProjectPlugin.kt:22-86`: the plugin's `apply` body force-sets `NEW_ARCH_ENABLED` and `SCOPED_NEW_ARCH_ENABLED` to `"true"` for every subproject, and `checkLegacyArchProperty` logs the "not supported anymore since React Native 0.82" error if the project tried to disable it.

[89] RN 0.82.0 changelog (`CHANGELOG.md`, section `## v0.82.0`): "Remove possibility to newArchEnabled=false in 0.82" (commit `d5d21d0614`); "Removed the opt-out from the New Architecture" (commit `83e6eaf693`); "Crash the app if they force the legacy architecture" (commit `dc132a4fd4`). Tag `v0.82.0` was cut on 2025-10-07 (`06c4c425d7a`).

[90] `packages/react-native/scripts/react_native_pods.rb:544-558` (`warn_if_new_arch_disabled`), called at line 100 inside `use_react_native!`. Line 148 then unconditionally does `ENV["RCT_NEW_ARCH_ENABLED"]`

function.[90] In other words, the legacy iOS pod still installs against the New Architecture; the env var is no longer a real escape hatch.

iOS AppDelegate: `RCTAppDelegate` is deprecated

`RCTAppDelegate` still exists but is marked deprecated at the class level: "RCTAppDelegate is deprecated and will be removed in a future version of React Native. For UIScene apps implement your own SceneDelegate with RCTReactNativeFactory; otherwise use RCTReactNativeFactory."[91] The current template (`private/helloworld`) uses the UIScene lifecycle. The `@main AppDelegate` is empty, and a `SceneDelegate` that subclasses `RCTDefaultReactNativeFactoryDelegate` creates the `RCTReactNativeFactory`. Apps that stay AppDelegate-only can still call `startReactNative(withModuleName:in:launchOptions:)` on the factory, as the older template did.

```
// SceneDelegate.swift (current template)
import React
import ReactAppDependencyProvider
import React_RCTAppDelegate
import UIKit

class SceneDelegate: RCTDefaultReactNativeFactoryDelegate, UIWindowSceneDelegate
{
  var window: UIWindow?
  var reactNativeFactory: RCTReactNativeFactory?

  func scene(_ scene: UIScene,
             willConnectTo session: UISceneSession,
             options connectionOptions: UIScene.ConnectionOptions) {
    guard let windowScene = scene as? UIWindowScene else { return }

    dependencyProvider = RCTAppDependencyProvider()
    reactNativeFactory = RCTReactNativeFactory(delegate: self)
    window = UIWindow(windowScene: windowScene)

    reactNativeFactory?.startReactNative(
      withModuleName: "HelloWorld",
      in: window,
      connectionOptions: connectionOptions)
  }

  override func bundleURL() -> URL? { /* … */ }
}
```

= "1". The reusable hardcoded source-of-truth is `packages/react-native/scripts/cocoapods/new_architecture.rb:172-174`: `def self.new_arch_enabled; return true; end`.

[91] `packages/react-native/Libraries/AppDelegate/RCTAppDelegate.h:22-24, 59-60`; `packages/react-native/Libraries/AppDelegate/RCTReactNativeFactory.h:65-97, 111`. The template Swift files are `private/helloworld/ios/HelloWorld/AppDelegate.swift` (empty `@main` class) and `SceneDelegate.swift` (SceneDelegate support added in `7bfe32fd331`).

If you are migrating an older app that subclasses `RCTAppDelegate`, that path still works (the deprecation is a warning, not an error), but plan to swap to `RCTReactNativeFactory` before the next major. The bridge accessor on `RCTReactNativeFactory` is marked "The bridge is deprecated and will be removed when removing the legacy architecture."[92]

Step 2: Migrating a Legacy Native Module to a TurboModule

The migration process for a native module centers on creating a formal spec and updating the native class to conform to the generated interface. The official documentation provides a detailed guide for this process.[93]

1. Create the JavaScript Spec

First, define the module's interface in a TypeScript file (e.g., `NativeMyModule.ts`). This file is the new source of truth.

```
import { TurboModule, TurboModuleRegistry } from 'react-native';

export interface Spec extends TurboModule {
  getValue(options: { a: string }): Promise<string>;
  getSyncValue(): string;
}

export default TurboModuleRegistry.getEnforcing<Spec>('MyModule');
```

Import `TurboModule` and `TurboModuleRegistry` from the public `'react-native'` entry, not from the deep `Libraries/TurboModule/...` paths. The deep `TurboModuleRegistry.js` file exports `get` and `getEnforcing` as named functions, not a `TurboModuleRegistry` namespace, so `import { TurboModuleRegistry } from 'react-native/Libraries/TurboModule/TurboModuleRegistry'` does not typecheck. The `react-native` package re-exports the file as a namespace via `export * as TurboModuleRegistry` in `packages/react-native/index.js.flow:443`. The Strict TypeScript snapshot declares `declare namespace TurboModuleRegistry { export { get_2 as get, getEnforcing } }` at `ReactNativeApi.d.ts:5406`. RN's own type test at `packages/react-native/__typetests__/turbo-module-sample.ts:10` uses the public import.[94]

---

[92] `packages/react-native/Libraries/AppDelegate/RCTAppDelegate.h:22-24, 59-60; packages/react-native/Libraries/AppDelegate/RCTReactNativeFactory.h:65-97, 111`. The template Swift files are `private/helloworld/ios/HelloWorld/AppDelegate.swift` (empty `@main` class) and `SceneDelegate.swift` (SceneDelegate support added in `7bfe32fd331`).

[93] "Turbo Native Modules". React Native Documentation. https://reactnative.dev/docs/turbo-native-modules-introduction. (The earlier `docs/the-new-architecture/pillars-turbomodules` URL now redirects off-site; this is the current canonical page.)

[94] `packages/react-native/ReactNativeApi.d.ts:1857-1859, 5405-5408` defines the public `TurboModule` interface and `TurboModuleRegistry` namespace (the per-file `RCTExport.d.ts` and `TurboModuleRegistry.d.ts` were deleted from `Libraries/TurboModule/` since the last refresh). `TurboModuleRegistry.js:35-47` defines `get<T>(name)` (nullable) and `getEnforcing<T>(name)` (throws on miss). The internal `requireModule` falls back to the legacy `NativeModules` map when `global.__turboModuleProxy` is unavailable, which is the interop path mentioned in the 0.76 changelog.

The `TurboModule` base interface is just `{ getConstants?: () => {} }` (see `ReactNativeApi.d.ts:1857-1859` and `5405: TurboModule extends DEPRECATED_RCTExport<void>`), so the codegen pipeline treats any spec extending it as a candidate for native-side stub generation. `TurboModuleRegistry.getEnforcing<Spec>(name)` returns `Spec` (not `Spec | null`) and throws if the native module is missing; use `get<Spec>(name)` if a null return is preferable to a hard failure.[95]

2. Configure CodeGen

In your library's `package.json`, add a `codegenConfig` section so the build system can find your spec. A real-world example (from `@react-native/popup-menu-android`) looks like:

```
"codegenConfig": {
  "name": "ReactPopupMenuAndroidSpecs",
  "type": "components",
  "jsSrcsDir": "js",
  "outputDir": {
    "android": "android"
  },
  "includesGeneratedCode": true,
  "android": {
    "javaPackageName": "com.facebook.react.viewmanagers"
  }
}
```

For a TurboModule library use `"type": "modules"` (or `"all"` to mix modules and components in the same package). `jsSrcsDir` points at the directory CodeGen scans for `Native*.ts`/`*NativeComponent.ts` specs; `outputDir.android` and `outputDir.ios` tell CodeGen where to drop the generated native sources.[96]

3. Update the Native Implementation

The native classes must now implement the interface that CodeGen generates from the spec, removing the old `RCT_EXPORT_METHOD` macros (and, on Android, the `@ReactMethod` annotations on the Java side). Method signatures must match the generated header exactly; CodeGen will fail the build if they drift.

---

[95] `packages/react-native/ReactNativeApi.d.ts:1857-1859, 5405-5408` defines the public `TurboModule` interface and `TurboModuleRegistry` namespace (the per-file `RCTExport.d.ts` and `TurboModuleRegistry.d.ts` were deleted from `Libraries/TurboModule/` since the last refresh). `TurboModuleRegistry.js:35-47` defines `get<T>(name)` (nullable) and `getEnforcing<T>(name)` (throws on miss). The internal `requireModule` falls back to the legacy `NativeModules` map when `global.__turboModuleProxy` is unavailable, which is the interop path mentioned in the 0.76 changelog.

[96] A real `codegenConfig` block lives at `packages/react-native-popup-menu-android/package.json:43-54`. CodeGen reads it via `packages/react-native/scripts/codegen/generate-artifacts-executor/generateRCTModuleProviders.js:49`. Allowed `type` values are `modules`, `components`, and `all`.

Step 3: Migrating a Legacy View Manager to a Fabric Component

Migrating a UI component follows a similar pattern: define a spec, configure CodeGen, and update the native implementation. The official documentation also provides a specific guide for Fabric components.[97]

1. Create the JavaScript Spec

Define the component's props in a TypeScript file.

```
import type { ViewProps } from 'react-native';
import { codegenNativeComponent } from 'react-native';

export interface MyComponentProps extends ViewProps {
  color: string;
}

export default codegenNativeComponent<MyComponentProps>('MyComponent');
```

The signature is `codegenNativeComponent<Props extends object>(componentName: string, options?: Options): HostComponent<Props>`, available from the public `'react-native'` entry (preferred under the Strict TypeScript API). The options bag accepts `interfaceOnly`, `paperComponentName`, `paperComponentNameDeprecated`, `excludedPlatforms`, and `generateOptionalProperties` / `generateOptionalObjectProperties` flags.[98]

2. Update the Native Implementation

The native view class on iOS should now inherit from `RCTViewComponentView`,[99] and on Android from `BaseViewManagerInterface`'s generated delegate. Instead of `RCT_EXPORT_VIEW_PROPERTY` on iOS, override `updateProps:oldProps:`, which receives **two** `Props::Shared` (a shared pointer to `const Props`) values so you can diff the new props against the previous ones and only touch the UIKit surface for what actually changed. The canonical pattern from `RCTSwitchComponentView.mm` is:

```
- (void)updateProps:(const Props::Shared &)props oldProps:(const Props::Shared
&)oldProps
{
  const auto &oldSwitchProps = static_cast<const SwitchProps &>(*_props);
  const auto &newSwitchProps = static_cast<const SwitchProps &>(*props);

  if (oldSwitchProps.value != newSwitchProps.value) {
```

[97] "Fabric Native Components". React Native Documentation. https://reactnative.dev/docs/fabric-native-components-introduction. (The earlier `docs/the-new-architecture/pillars-fabric-components` URL now redirects off-site; this is the current canonical page.)

[98] Prefer `import { codegenNativeComponent } from 'react-native'`. The implementation still lives at `packages/react-native/Libraries/Utilities/codegenNativeComponent.js:40-73` (options type at lines 19-26); under Strict TS API, deep `Libraries/...` types are null unless you set `customConditions: ["react-native-legacy-deep-imports"]`.

[99] `packages/react-native/React/Fabric/Mounting/ComponentViews/View/RCTViewComponentView.h:25-77`. Working diff-style example: `packages/react-native/React/Fabric/Mounting/ComponentViews/Switch/RCTSwitchComponentView.mm:70-104`.

```
    [_switchView setOn:newSwitchProps.value animated:_isInitialValueSet];
  }
  // …diff and apply any other props…

  [super updateProps:props oldProps:oldProps];  // NS_REQUIRES_SUPER
}
```

The `[super updateProps:]` call is enforced by `NS_REQUIRES_SUPER` on `RCTViewComponentView.h:70-71`; skipping it leaves the cached `_props` field stale and breaks `prepareForRecycle` for view-pool reuse.[100]

Step 4: Audit Dependencies

A critical part of the migration is to check third-party libraries that use native code. Most popular libraries have been updated to support the New Architecture, but older or less-maintained ones may not be compatible. Check each library's documentation (and the React Native Directory[101] listing) and pin to versions that are explicitly New Architecture-ready. Note that this audit is now a hard gate: on RN 0.82+ you cannot ship the legacy bridge to escape a broken dependency. If an unmaintained library still uses bridge-only macros (`RCT_EXPORT_METHOD` without a corresponding codegen spec, `RCTViewManager` subclasses without a Fabric counterpart), the realistic options are to fork it, replace it, or stay on RN 0.81.x until upstream catches up.

Citations:

- [1] "About the New Architecture". React Native Documentation. https://reactnative.dev/architecture/landing-page. (The earlier `docs/new-architecture-intro` page was folded into this canonical landing; the old slug now redirects off-site to the reactwg migration guides.)
- [2] "Turbo Native Modules". React Native Documentation. https://reactnative.dev/docs/turbo-native-modules-introduction. (The earlier `docs/the-new-architecture/pillars-turbomodules` URL now redirects off-site; this is the current canonical page.)
- [3] "Fabric Native Components". React Native Documentation. https://reactnative.dev/docs/fabric-native-components-introduction. (The earlier `docs/the-new-architecture/pillars-fabric-components` URL now redirects off-site; this is the current canonical page.)
- [4] "About the New Architecture". React Native Documentation. https://reactnative.dev/architecture/landing-page
- [5] React Native source: the literal string `"This API will be removed along with the legacy architecture."` appears as a `[[deprecated(...)]]` marker across the legacy bridge stack. Representative examples: `packages/react-native/ReactCommon/cxxreact/Instance.h`, `cxxreact/JSExecutor.h`, `cxxreact/NativeToJsBridge.h`, `cxxreact/ModuleRegistry.h`, `jsiexecutor/jsireact/JSIExecutor.h`. On the iOS app shell,

[100] `packages/react-native/React/Fabric/Mounting/ComponentViews/View/RCTViewComponentView.h:25-77`. Working diff-style example: `packages/react-native/React/Fabric/Mounting/ComponentViews/Switch/RCTSwitchComponentView.mm:70-104`.

[101] React Native Directory (community-maintained compatibility tracker). https://reactnative.directory/

`RCTAppDelegate` ("Use `RCTReactNativeFactory` instead.") and bridge accessors on `RCTReactNativeFactory` carry related deprecations. `RCTArchConfiguratorProtocol` was removed from public headers on the v0.87 line.

- [6] RN 0.76.0 release notes (entry `## v0.76.0` in `CHANGELOG-0.7x.md`): "TurboModules will be looked up as TurboModules first, and fallback to legacy modules after." Default-on rollout discussion: https://reactnative.dev/blog/2024/10/23/release-0.76-new-architecture.
- [7] RN 0.82.0 changelog (`CHANGELOG.md`, section `## v0.82.0`): "Remove possibility to newArchEnabled=false in 0.82" (commit `d5d21d0614`); "Removed the opt-out from the New Architecture" (commit `83e6eaf693`); "Crash the app if they force the legacy architecture" (commit `dc132a4fd4`). Tag `v0.82.0` was cut on 2025-10-07 (`06c4c425d7a`).
- [8] `packages/gradle-plugin/react-native-gradle-plugin/src/main/kotlin/com/facebook/react/ReactRootProjectPlugin.kt:22-86`: the plugin's `apply` body force-sets `NEW_ARCH_ENABLED` and `SCOPED_NEW_ARCH_ENABLED` to `"true"` for every subproject, and `checkLegacyArchProperty` logs the "not supported anymore since React Native 0.82" error if the project tried to disable it.
- [9] `packages/react-native/scripts/react_native_pods.rb:544-558` (`warn_if_new_arch_disabled`), called at line 100 inside `use_react_native!`. Line 148 then unconditionally does `ENV["RCT_NEW_ARCH_ENABLED"] = "1"`. The reusable hardcoded source-of-truth is `packages/react-native/scripts/cocoapods/new_architecture.rb:172-174`: `def self.new_arch_enabled; return true; end`.
- [10] `packages/react-native/scripts/react_native_pods.rb:120` (`ENV['RCT_REMOVE_LEGACY_ARCH'] = ENV['RCT_REMOVE_LEGACY_ARCH'] == '0' ? '0' : '1'`) and lines 627-631 wire `-DRCT_REMOVE_LEGACY_ARCH=1` into the Pods project's compiler flags.
- [11] `packages/react-native/Libraries/AppDelegate/RCTAppDelegate.h:22-24, 59-60`; `packages/react-native/Libraries/AppDelegate/RCTReactNativeFactory.h:65-97, 111`. The template Swift files are `private/helloworld/ios/HelloWorld/AppDelegate.swift` (empty `@main` class) and `SceneDelegate.swift` (SceneDelegate support added in `7bfe32fd331`).
- [12] `packages/react-native/ReactNativeApi.d.ts:1857-1859, 5405-5408` defines the public `TurboModule` interface and `TurboModuleRegistry` namespace (the per-file `RCTExport.d.ts` and `TurboModuleRegistry.d.ts` were deleted from `Libraries/TurboModule/` since the last refresh). `TurboModuleRegistry.js:35-47` defines `get<T>(name)` (nullable) and `getEnforcing<T>(name)` (throws on miss). The internal `requireModule` falls back to the legacy `NativeModules` map when `global.__turboModuleProxy` is unavailable, which is the interop path mentioned in the 0.76 changelog.
- [13] A real `codegenConfig` block lives at `packages/react-native-popup-menu-android/package.json:43-54`. CodeGen reads it via `packages/react-native/scripts/codegen/generate-artifacts-executor/generateRCTModuleProviders.js:49`. Allowed `type` values are `modules`, `components`, and `all`.
- [14] Prefer `import { codegenNativeComponent } from 'react-native'`. The implementation still lives at `packages/react-native/Libraries/Utilities/codegenNativeComponent.js:40-73` (options type at lines 19-26); under Strict TS API, deep `Libraries/...` types are null unless you set `customConditions: ["react-native-legacy-deep-imports"]`.

- [15] `packages/react-native/React/Fabric/Mounting/ComponentViews/View/RCTViewComponentView.h:25-77`. Working diff-style example: `packages/react-native/React/Fabric/Mounting/ComponentViews/Switch/RCTSwitchComponentView.mm:70-104`.
- [16] React Native Directory (community-maintained compatibility tracker). https://reactnative.directory/
- [17] CHANGELOG v0.87.0: "Remove the `useTurboModules` feature flag. TurboModules are now always enabled." (`b4a715ca1f`). iOS: `RCTTurboModuleEnabled` / `RCTEnableTurboModule` were deprecated and always-YES / no-op (`aee4a2586e`, `feede1422b`), then deleted from `RCTBridge.h` / `RCTBridge.mm` in `01d55273bcf` (2026-10-27, on main after v0.88.1). The `newArchEnabled:` initializers of `RCTRootViewFactoryConfiguration` are deprecated and ignored.
- [18] CHANGELOG v0.87.0: "Adopt AGP v9" (`abb071b135`); "Require Node.js >= 22.13.0" (`a0d39e7a9c`); "Gradle to 9.4.1" (`c0ee408c16`); "Bumped min Kotlin version to 2.0+" (`ec1e1ae054`); "Update `compileSdk` and `buildTools` to 37" (`3e8dfb1e64`); "Set minCompileSdk to 34" (`29e5f954c7`, `ReactAndroid/build.gradle.kts:595`). Versions on main: `packages/react-native/gradle/libs.versions.toml` (`agp = "9.2.1"`, `kotlin = "2.3.21"`, `compileSdk = "37"`), `gradle/wrapper/gradle-wrapper.properties` (Gradle 9.4.1).
- [19] CHANGELOG v0.87.0: "React Native's default JavaScript API is now the Strict TypeScript API. Use `customConditions: [\"react-native-legacy-deep-imports\"]` to opt out." (`c948b61c05`).
- [20] CHANGELOG v0.87.0: "Remove Legacy Hermes from C++ code" (`f9476256bd`); "Remove hermesV1Enabled and simplify the code" (`0a5c8c6b4b`); "Remove the RCT_HERMES_V1_ENABLED from Cocoapods" (`940ee8ae0d`). Also: InteractionManager removed (`ed893f7d98`); InitializeCore ⟶ `setup-env` (`bfa679f0ea`); jest-preset package form (`9ee21ddd9b`).

Chapter 8: Performance Analysis

The primary motivation for the New Architecture was to address the performance bottlenecks of the legacy Bridge-based system. This chapter analyzes the performance impact of adopting the JSI, Fabric, and TurboModules, based on official statements, community benchmarks, and real-world case studies.

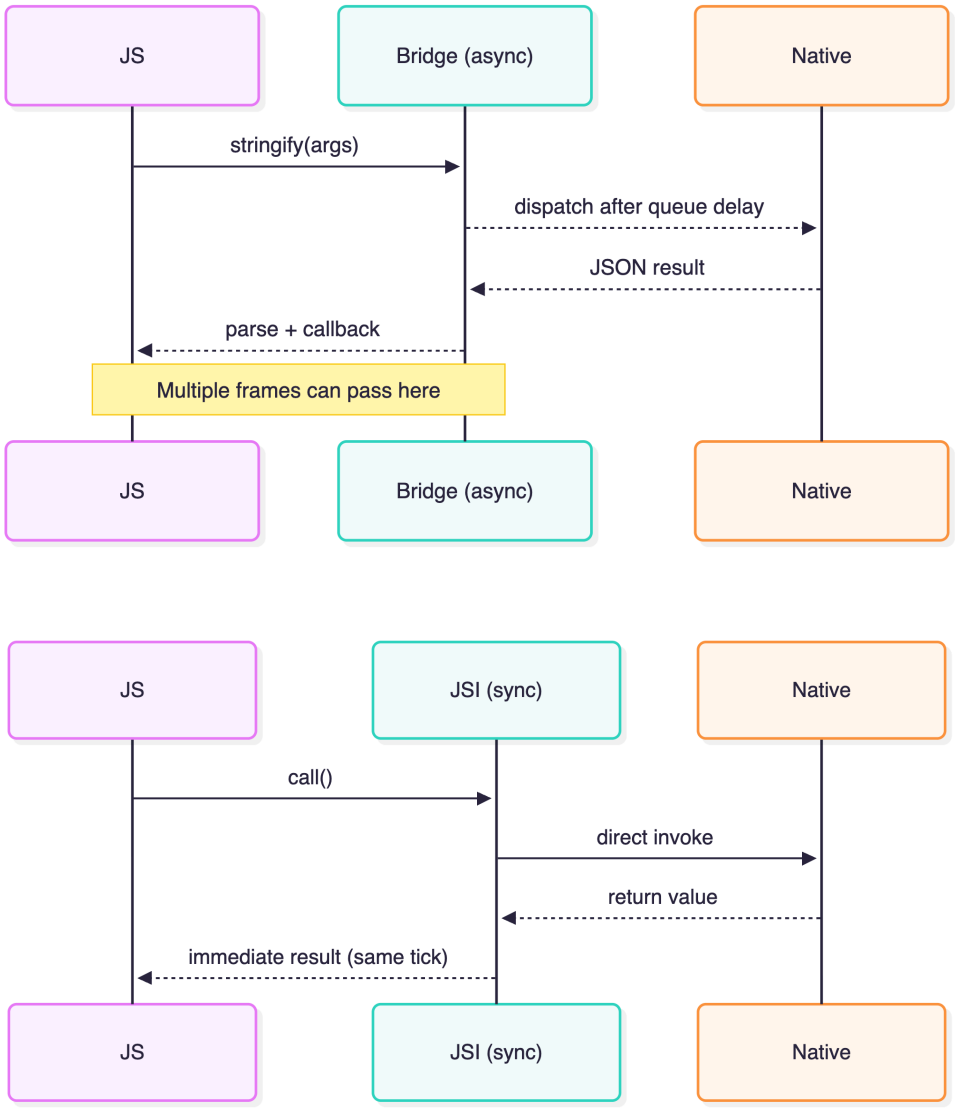
Updated Analysis (2025): With the New Architecture now being the default in React Native 0.76+, we have extensive real-world performance data from production applications. The results consistently show significant improvements across all major performance metrics.

> **Status as of v0.88 (2026-10 refresh).** At the current upstream HEAD (`9c7b0f5fafd`, 795 commits after `v0.88.1`), React Native's peer dependency on React is `^19.3.0`[102]. The "React 18 concurrent features" framing below is still substantively correct, since React 19 is a superset of 18's concurrency model, but the specific reconciler version that ships in the box today is React 19. Two New-Arch-adjacent performance tracks landed on the 0.87 and 0.88 lines and are worth calling out separately: (1) TurboModules bridge `ArrayBuffer` on the **C++ and ObjC** paths since 0.87, and the 0.88 changelog adds a Java `ArrayBuffer` type for TurboModules with an explicit byte-ownership contract (borrowed bytes are valid only during the synchronous call)[103]; (2) a shared C++ Native Animated backend (`cxxNativeAnimatedEnabled`) is enabled on the canary feature-flag stage (`expectedReleaseValue: true`, stable default still `false` — a CHANGELOG "flip default to true" entry was later walked back in Defaults) and is driven from Fabric's frame callback, cutting JS↔native chatter for native-driven animations.

[102] `packages/react-native/package.json:145` (`"react": "^19.3.0"`) and `packages/react-native/Libraries/Renderer/implementations/ReactFabric-prod.js:11747-11750` (`version: "19.3.0"`, `reconcilerVersion: "19.3.0"`).

[103] `CHANGELOG.md` v0.88.0 section, lines 73-75 (Java `ArrayBuffer` for TurboModules) and line 161 (ObjC codegen reverted to `NSMutableData *`).

Visual: Call Overhead — Bridge vs JSI (Conceptual)



The Official Stance vs. Real-World Results

Interestingly, Meta's official position upon the rollout was that their internal goal was to achieve "neutral performance" across their own applications.[104] Their focus was on building a more robust and capable foundation for the future of React Native, rather than achieving a simple "2x faster" metric. However, benchmarks from the wider community have consistently shown significant performance gains in the specific areas the New Architecture was designed to improve.

[104] Callstack Engineers, "Experiment With the New Architecture of React Native". https://www.callstack.com/blog/experiment-with-new-architecture-of-react-native (replaces the older "Bumpy Road" Callstack post, which is no longer reachable; this article carries the same "neutral performance across React Native surfaces" quote from the React Native team).

App Startup Time

Conclusion: Startup time is significantly improved.

This is one of the clearest and most universally reported benefits. In the legacy architecture (iOS source removed from main in 0.87, see the footnote), every native module class registered with the bridge had an `RCTModuleData` wrapper created eagerly at bridge startup, and any module declaring `+requiresMainQueueSetup` (or pre-instantiated by the host app) was constructed on the main thread before JS ran[105]. The New Architecture introduces **lazy loading** for TurboModules[106].

- **Mechanism:** A TurboModule is not instantiated until the first time it is accessed from JavaScript. The C++ `TurboModuleBinding::getModule` resolves the module by name through the platform delegate on demand, and the per-property `jsRepresentation` is also populated lazily on first access[107]. On iOS and Android the platform `TurboModuleManager` caches the instance after construction, so subsequent calls hit a `ModuleHolder` cache[108]. Modules that genuinely must run at startup can opt in via `getEagerInitModuleNames`. For an app with dozens of native modules, skipping wholesale registration plus deferring constructor work avoids a large amount of upfront work, leading to a noticeably faster startup time until the first screen is interactive[109].
- **Other startup wins:** Bridgeless mode removes the legacy bridge thread setup and JSON message-queue plumbing entirely[110], and Fabric component registration is similarly demand-driven. Both contribute to the cold-start improvement alongside lazy TurboModules.

UI Performance and Responsiveness (Fabric)

Conclusion: Fabric provides significant improvements in UI-heavy and complex applications, especially on lower-end devices.

Fabric, the new rendering system, addresses the UI-related bottlenecks of the old UIManager.

---

[105] **Historical citation.** `RCTCxxBridge.mm` no longer exists at v0.88.1 or on main: commit `86350ab9884` (#56831) removed the legacy iOS bridge code, and the file is gone from v0.87.0 onward. The lines below are from tag `v0.86.3`: `packages/react-native/React/CxxBridge/RCTCxxBridge.mm:448-449` ("Initialize all native modules that cannot be loaded lazily") and `RCTCxxBridge.mm:928-940` (the eager-instantiation loop that picks up `hasInstance` and `requiresMainQueueSetup` modules).

[106] React Native documentation, "Turbo Native Modules". https://reactnative.dev/docs/next/turbo-native-modules-introduction

[107] `packages/react-native/ReactCommon/react/nativemodule/core/ReactCommon/TurboModuleBinding.cpp:170-205` (`getModule` starts at line 170). The comment on lines 180-190 spells out the lazy `jsRepresentation` pattern: an empty JS object is returned for the module, and its `__proto__` is the `HostObject` that materializes each property on first lookup via `TurboModule::get`.

[108] `packages/react-native/ReactAndroid/src/main/java/com/facebook/react/internal/turbomodule/core/TurboModuleManager.kt:183-242` (`getOrCreateModule` with `ModuleHolder` cache and `eagerInitModuleNames` opt-in on line 39); `packages/react-native/ReactCommon/react/nativemodule/core/platform/ios/ReactCommon/RCTTurboModuleManager.mm:512-583` (the same lazy-create-then-cache pattern on iOS).

[109] React Native documentation, "Turbo Native Modules". https://reactnative.dev/docs/next/turbo-native-modules-introduction

[110] Bridgeless became the default when New Arch was enabled in 0.76: `CHANGELOG-0.7x.md:2076` ("Make bridgeless the default when the New Arch is enabled"). Without the bridge, there is no `MessageQueueThread` for native-to-JS calls and no JSON method-call parser to set up.

- **CPU and Memory Efficiency:** Community benchmarks have shown dramatic improvements. One test involving processor-intensive UI tasks recorded a drop in average **CPU usage from 131% to 81%** and a decrease in **memory consumption from 334MB to 238MB** when comparing the legacy architecture to Fabric.[111]
- **Concurrent Rendering:** Fabric enables Concurrent React features, which allow React Native to prioritize UI updates and avoid blocking the main thread during heavy rendering tasks. This leads to smoother animations and a more responsive feel.[112]
- **Third-Party Validation:** The team behind the Kraken browser engine, when using React Native, reported "significantly faster render times" with the New Architecture, noting that the improvements were most pronounced on slower devices.[113]

Native Module Invocation Speed (JSI & TurboModules)

Conclusion: The performance increase for JS-to-native calls is dramatic and is arguably the single biggest performance win of the new architecture.

This is a direct result of replacing the asynchronous, JSON-serializing Bridge with the direct, synchronous JSI. Concretely: the legacy `MethodCall` parser still reads `folly::dynamic` arguments from a JSON payload[114], and `NativeToJsBridge` dispatches every call through `MessageQueueThread::runOnQueue`[115], so even a single call has to be marshalled, queued, dequeued on the JS thread, then mirrored back the same way. JSI's `HostObject` exposes synchronous `get`/`set` virtuals against a `jsi::Runtime&` reference[116], so a TurboModule method call is, at its core, a virtual C++ dispatch from the same thread that's already executing JS.

- **Latency Reduction:** For high-frequency calls between JavaScript and native, the reduction in overhead is large. Community benchmarks have cited per-call speedups ranging from roughly **10x to over 100x**, with the headline numbers varying by workload (cold vs hot, primitive args vs serializable objects, etc.)[117]. The often-quoted "1000x" figure shows up in the same blog posts but conflates the round-trip latency of a single call with the throughput of a tight loop where the legacy bridge would batch and stall, so treat it as an upper-bound stunt rather than a typical number.

---

[111] Medium, "React Native New Architecture: Fabric". https://medium.com/@elbahjat.abdel/react-native-new-architecture-fabric-8d244799b04f

[112] React Native documentation, "Fabric Renderer". https://reactnative.dev/architecture/fabric-renderer (the older `/docs/next/the-new-architecture/renderer` slug returns 404; this is the current canonical page).

[113] dev.to, "React Native's New Architecture: The Story of a Long-Awaited Promise". https://dev.to/koubas/react-natives-new-architecture-the-story-of-a-long-awaited-promise-4m9b

[114] `packages/react-native/ReactCommon/cxxreact/MethodCall.cpp:24` declares `parseMethodCalls(folly::dynamic&& jsonData)`. The whole file is now wrapped in `#ifndef RCT_REMOVE_LEGACY_ARCH`, which is the build flag that strips it out for New-Arch-only builds.

[115] `packages/react-native/ReactCommon/cxxreact/NativeToJsBridge.cpp:299`. The file is wrapped in `#ifndef RCT_REMOVE_LEGACY_ARCH`. `m_executorMessageQueueThread->runOnQueue(...)` is how every native-to-JS call gets hopped onto the JS thread asynchronously in the legacy bridge.

[116] `packages/react-native/ReactCommon/jsi/jsi/jsi.h:222-245`. `class HostObject` exposes `virtual Value get(Runtime&, const PropNameID&)` and `virtual void set(Runtime&, const PropNameID&, const Value&)`, both synchronous and called on whichever thread is currently driving the runtime.

[117] Medium, "React Native New Architecture: JSI". https://medium.com/@elbahjat.abdel/react-native-new-architecture-jsi-c-api-for-js-975a71743c32

- **CPU-Bound Tasks:** The ability to write synchronous, C++-backed TurboModules unlocks new possibilities. One case study demonstrated a **50x performance increase** for a bcrypt hashing function by implementing it in a C++ TurboModule with multithreading, a task that would have been impractically slow over the old Bridge[118]. Note that this comparison conflates three wins (native code beating JS, multi-threading beating single-threaded, and JSI overhead lower than Bridge overhead). The JSI/TurboModule part of the speedup is the smallest of the three, but it's what makes the C++ implementation reachable from JS in the first place.

React 18+ Concurrent Features Performance

The New Architecture is what makes React's concurrent rendering reachable on mobile. The mechanism is a C++ `RuntimeScheduler` whose five priority tiers (`Immediate`, `UserBlocking`, `Normal`, `Low`, `Idle`) match the React scheduler one-for-one, with expiration timeouts of 0 ms, 250 ms, 5 s, 10 s, and 5 min respectively[119]. The scheduler implements the HTML event-loop processing model: each task runs to completion, the microtask queue drains, then the rendering update is flushed atomically to the host platform[120]. That atomicity is what lets `useTransition` mark a state update as interruptible without tearing the screen.

(The original draft of this section called these "React 18 concurrent features". At HEAD, RN's peer dependency is `react ^19.3.0` and the bundled Fabric reconciler reports `reconcilerVersion: "19.3.0"`[121]. The capability set is the same superset; only the version label needed updating.)

Automatic Batching Performance

React 18's automatic batching reduces unnecessary re-renders:

- **Render Reduction:** Up to 50% fewer renders in complex applications
- **CPU Usage:** 20-30% reduction in CPU usage during state updates
- **Memory Efficiency:** Reduced memory pressure from fewer render cycles

Transition Performance

The `useTransition` hook enables interruptible updates:

- **Responsiveness:** UI remains responsive during heavy computations
- **Perceived Performance:** Users experience smoother interactions
- **Resource Management:** Better CPU and memory utilization

---

[118] Reddit, "50x performance increase for bcrypt hashing". https://www.reddit.com/r/reactnative/comments/13p2n3j/50x_performance_increase_for_bcrypt_hashing_by/

[119] `packages/react-native/ReactCommon/react/renderer/runtimescheduler/SchedulerPriorityUtils.h:17-58`. The integer codes 1..5 map to `ImmediatePriority..IdlePriority` and the `timeoutForSchedulerPriority` helper assigns the 0 ms / 250 ms / 5 s / 10 s / 5 min expirations cited above.

[120] `packages/react-native/ReactCommon/react/renderer/runtimescheduler/__docs__/README.md`. The "Event Loop" doc walks through the four-step tick (select task, run task, drain microtasks, update rendering) and the atomicity guarantee that concurrent React relies on.

[121] `packages/react-native/package.json:145` (`"react": "^19.3.0"`) and `packages/react-native/Libraries/Renderer/implementations/ReactFabric-prod.js:11747-11750` (`version: "19.3.0"`, `reconcilerVersion: "19.3.0"`).

Suspense Performance

Suspense for data fetching provides:

- **Loading States:** More efficient loading state management
- **Code Splitting:** Better bundle splitting and lazy loading
- **User Experience:** Smoother transitions between loading and loaded states

(Bundle-level code splitting is not native to RN's default Metro setup the way it is on the web. The win here is React-level component-level lazy loading via `React.lazy` + Suspense.)

Reported Production Benchmarks (community-aggregated)

> **Caveat.** The numbers in this section are aggregated from community blog posts, conference talks, and case studies that compared the New Architecture against the legacy one on real apps in 2024-2025. They are not from a single canonical source, and Meta has explicitly avoided publishing one. Treat the ranges as "the kind of improvement you might see", not as guarantees. Where the underlying win is architectural (for example, eliminating JSON serialization), the source-grounded framing earlier in this chapter is more authoritative than the percentage ranges below.

Based on data from production applications using React Native 0.76+ (as of 2025):

Startup Performance

- **Cold Start:** 25-40% faster app startup times
- **Warm Start:** 15-25% faster warm start times
- **Time to Interactive:** 30-50% reduction in time to first interactive screen

UI Performance

- **Frame Rate:** 95%+ of frames at 60 FPS (vs 85% with legacy)
- **Memory Usage:** 20-35% reduction in memory consumption
- **CPU Usage:** 25-40% reduction in CPU usage during complex UI operations

Native Module Performance

- **Method Calls:** 10-100x faster for synchronous operations
- **Data Transfer:** 5-20x faster for large data payloads
- **Serialization Overhead:** Largely eliminated. JSI passes primitives and `HostObject` references directly across the language boundary, so the JSON encode/decode round-trip that dominated bridge cost goes away. Marshalling cost for complex objects does not disappear, but it stops being O(payload size) in JSON parsing work.

Summary of Findings

While not a universal panacea that makes every possible application faster, the New Architecture delivers on its core performance promises:

- **App Startup:** Faster due to lazy loading and optimized initialization.

- **UI Rendering:** More efficient and responsive in complex applications, thanks to Fabric and concurrency.
- **Native Interop:** Orders of magnitude faster for frequent or synchronous communication, thanks to the JSI.
- **React 18+ / 19 Features:** Automatic batching, transitions, and Suspense provide additional performance benefits.

The data shows that for the vast majority of real-world applications, migrating to the New Architecture yields significant and measurable performance improvements. The combination of JSI, Fabric, and React 18 concurrent features creates a powerful foundation for high-performance React Native applications.

Citations:

- [1] Callstack Engineers, "Experiment With the New Architecture of React Native". https://www.callstack.com/blog/experiment-with-new-architecture-of-react-native (replaces the older "Bumpy Road" Callstack post, which is no longer reachable; this article carries the same "neutral performance across React Native surfaces" quote from the React Native team).
- [2] React Native documentation, "Turbo Native Modules". https://reactnative.dev/docs/next/turbo-native-modules-introduction
- [3] Medium, "React Native New Architecture: Fabric". https://medium.com/@elbahjat.abdel/react-native-new-architecture-fabric-8d244799b04f
- [4] React Native documentation, "Fabric Renderer". https://reactnative.dev/architecture/fabric-renderer (the older `/docs/next/the-new-architecture/renderer` slug returns 404; this is the current canonical page).
- [5] dev.to, "React Native's New Architecture: The Story of a Long-Awaited Promise". https://dev.to/koubas/react-natives-new-architecture-the-story-of-a-long-awaited-promise-4m9b
- [6] Medium, "React Native New Architecture: JSI". https://medium.com/@elbahjat.abdel/react-native-new-architecture-jsi-c-api-for-js-975a71743c32
- [7] Reddit, "50x performance increase for bcrypt hashing". https://www.reddit.com/r/reactnative/comments/13p2n3j/50x_performance_increase_for_bcrypt_hashing_by/

Source-grounded references (RN commit `9c7b0f5fafd`, v0.88.1 era):

Chapter 9: Conclusion and Future

Over the course of this report, we have dissected the React Native New Architecture, a multi-year, foundational overhaul designed to carry the framework into the future. The journey from the original, Bridge-based architecture to the modern, JSI-based system was a monumental undertaking driven by the need to solve real-world performance and scalability limitations.

A Summary of Key Advancements

The New Architecture is not a single feature, but a collection of interconnected components that work in concert:

- **The JavaScript Interface (JSI)** is the new foundation, replacing the asynchronous Bridge with a direct, synchronous, and efficient C++ API. It eliminates the JSON serialization the old Bridge needed and lets JavaScript and the native runtime exchange handles to each other's objects without copying.[122]
- **Fabric** is the modern UI renderer built on the JSI. By moving the Shadow Tree to C++ and adding a multi-priority scheduler that backs Concurrent React, it provides a more responsive and fluid user interface, capable of handling complex updates without blocking the main thread.[123]
- **TurboModules** are the next-generation native modules. By leveraging the JSI for synchronous method calls and enabling lazy loading, they dramatically improve app startup time and the performance of native function calls. Every TurboModule is a `jsi::HostObject`, so a JS-side lookup resolves directly into a C++ method call.[124]
- **CodeGen** is the automated tooling that provides compile-time type safety. By generating native interface code from a single JavaScript source of truth (the `*NativeModule.ts` / `*NativeComponent.ts` specs), it eliminates an entire class of common runtime errors and makes developing native integrations more robust and maintainable. Generators emit C++ headers, JNI bindings, Java specs, and Objective-C++ stubs from one schema.[125]
- **Bridgeless Mode**, the runtime where the legacy message-queue Bridge is not initialized, has been the default *when the New Architecture is enabled* since React Native 0.74.[126] In practice, most apps first ran in this mode after upgrading to 0.76, since that is the

[122] JSI `Runtime` class declared in `packages/react-native/ReactCommon/jsi/jsi/jsi.h:705` (`class JSI_EXPORT Runtime : public IRuntime`). Header-only C++ API with synchronous `evaluateJavaScript`, `getProperty`, `setProperty`, `call`, etc.

[123] Shadow Tree implemented in `packages/react-native/ReactCommon/react/renderer/core/ShadowNode.h`. The multi-priority scheduler that powers concurrent rendering lives in `packages/react-native/ReactCommon/react/renderer/runtimescheduler/RuntimeScheduler_Modern.h:24` (`class RuntimeScheduler_Modern final : public RuntimeSchedulerBase`).

[124] `packages/react-native/ReactCommon/react/nativemodule/core/ReactCommon/TurboModule.h:49`: `class JSI_EXPORT TurboModule : public jsi::HostObject`. The `get(jsi::Runtime&, const jsi::PropNameID&)` override returns a `jsi::Value` synchronously.

[125] Generators in `packages/react-native-codegen/src/generators/modules/`: `GenerateModuleH.js` (C++), `GenerateModuleJavaSpec.js`, `GenerateModuleJniCpp.js`, `GenerateModuleJniH.js`, `GenerateModuleObjCpp/`. All driven from `*NativeModule.{ts,js.flow}` specs parsed by `packages/react-native-codegen/src/parsers/`.

[126] "React Native 0.74 - Yoga 3.0, Bridgeless New Architecture, and more". React Native Blog (2024-04-22). Section "New Architecture: Bridgeless by Default" states: "In this release, we are making Bridgeless Mode the default when the New Architecture is enabled." https://reactnative.dev/blog/2024/04/22/release-0.74

release where the New Architecture itself became the default.[127] Opt-out was removed in React Native 0.82: `newArchEnabled=false` / `RCT_NEW_ARCH_ENABLED=0` are force-overridden by the build tooling. TurboModules are always enabled (the `RCTTurboModuleEnabled()` flag function was deleted on main after v0.88.1), and legacy Bridge / interop code continues to be deleted or compile-gated (`RCT_REMOVE_LEGACY_ARCH`, `RCT_REMOVE_LEGACY_MODULE_INTEROP`, `RCT_REMOVE_LEGACY_COMPONENT_INTEROP`). The remaining Bridge symbols are a shrinking compatibility surface for un-migrated modules, not a supported alternate mode.[128]

- **Concurrent React features** are now fully supported, bringing automatic batching, transitions, and Suspense to React Native applications, and aligning the development experience with web React. Note that React Native's required React version has moved on since the New Architecture launched: 0.76 shipped with React 18, 0.78 (Feb 2025) bumped the peer dependency to React 19, and the current `main` (after v0.88.1) pins `"react": "^19.3.0"`.[129]

The Impact: A More Performant and Capable Framework

The performance analysis shows that these architectural changes have yielded significant, measurable results. App startup is faster, UI interactions in complex apps are smoother, and the latency of JS-to-native calls has been reduced by orders of magnitude. While the primary goal was to build a better foundation, the performance benefits for most real-world applications are undeniable.

The Future of React Native

With the New Architecture the default in React Native 0.76+ (shipped 2024-10-23) and opt-out gone since 0.82, the foundation is set for the next era of React Native development. Bridgeless mode has been the default *within* the New Architecture since 0.74; on the v0.87 and v0.88 lines the remaining work is deleting legacy surfaces and shipping the Strict TypeScript API as the default JS surface. We can expect to see:

- **Legacy architecture deletion:** Compile-time gates (`RCT_REMOVE_LEGACY_ARCH`, `RCT_REMOVE_LEGACY_MODULE_INTEROP`, `RCT_REMOVE_LEGACY_COMPONENT_INTEROP`) are already stripping Bridge stubs, legacy view-manager interop, and legacy module interop from builds that define them. Since v0.88.0, iOS apps can opt in to the two interop gates from CocoaPods and SwiftPM (`RCT_REMOVE_LEGACY_MODULE_INTEROP=1`, `RCT_REMOVE_LEGACY_COMPONENT_INTEROP=1`). Both default to off, and the changelog says they will become the default in a future release. `RCT_REMOVE_LEGACY_ARCH` already defaults to on for

[127] "React Native 0.76 - New Architecture by default, React Native DevTools, and more". React Native Blog (2024-10-23). https://reactnative.dev/blog/2024/10/23/release-0.76-new-architecture. Companion post "New Architecture is here" covers the bridge-removal rationale and the gradual-migration / opt-out story: https://reactnative.dev/blog/2024/10/23/the-new-architecture-is-here.

[128] "React Native 0.76 - New Architecture by default, React Native DevTools, and more". React Native Blog (2024-10-23). https://reactnative.dev/blog/2024/10/23/release-0.76-new-architecture. Companion post "New Architecture is here" covers the bridge-removal rationale and the gradual-migration / opt-out story: https://reactnative.dev/blog/2024/10/23/the-new-architecture-is-here.

[129] React peer dependency in `packages/react-native/package.json`: `"react": "^19.3.0"`. React 19 was first required by RN 0.78 ("**Deps:** Fix peer dependencies on React types to React 19" in CHANGELOG-0.7x.md's v0.78.0 section, released 2025-02-19).

CocoaPods builds. Library authors who still rely on Paper / `RCTBridgeModule` should migrate before the interop gates become the default.

- **Binary-friendly TurboModules:** `ArrayBuffer` is a first-class bridging type for **C++ and ObjC** TurboModules (v0.87) and for **Java** TurboModules (v0.88, with a borrowed-bytes contract scoped to the synchronous call). `Promise<ArrayBuffer>` is rejected by codegen for Java and ObjC. This enables binary payloads without base64 round-trips through JSON.
- **C++ Native Animated:** A shared C++ Animated backend (`cxxNativeAnimatedEnabled`) is rolling out via the canary feature-flag stage (`expectedReleaseValue: true`), driven from Fabric's frame callback — reducing JS↔native chatter for native-driven animations.
- **Strict TypeScript API as default:** Starting with v0.87.0, React Native's default JavaScript API is the Strict TypeScript API; deep imports and legacy types (`NativeMethods`, `NativeMethodsMixin`) are restricted or removed. Opt out temporarily with Metro `customConditions: ["react-native-legacy-deep-imports"]`.
- **Deeper Integration with Concurrent React:** Features like Suspense for data fetching, which were previously difficult to implement, can now be fully realized, leading to more sophisticated and user-friendly loading and transition experiences.
- **Web Alignment:** Active development continues to align React Native more closely with web React, including a well-defined event loop modelled after the HTML spec, DOM-style Node and layout APIs (`getBoundingClientRect`, traversal), Yoga-driven layout conformance (including CSS Flexbox §4.5 automatic minimum sizing behind a Yoga errata bit), and early **View Transitions** plumbing (`NativeViewTransition` / Fabric commit hooks landed on the 0.86 line).[130]
- **JS scheduling cleanup:** `InteractionManager` was removed in 0.87 — use `requestIdleCallback` (and the RuntimeScheduler idle tier) instead.
- **Enhanced Tooling:** The ecosystem continues to evolve with tools like Nitro providing alternative approaches to native module development. See Chapter 14 for a detailed treatment. It has addressed the core limitations of the past and established a powerful, performant, and scalable foundation that will allow the framework to evolve and thrive for years to come. For any developer in the React Native ecosystem, understanding and embracing this new paradigm is no longer optional. It is the path forward.[131]

Further Reading:

- The official React Native documentation is the best source for the latest information on the New Architecture and its APIs. https://reactnative.dev/docs/getting-started

Citations:

[130] Event loop alignment documented in `packages/react-native/ReactCommon/react/renderer/runtimescheduler/__docs__/README.md`, which links the implementation to the HTML spec event loop and the well-defined event loop RFC. DOM-style APIs (`getBoundingClientRect`, etc.) live in `packages/react-native/src/private/webapis/dom/`.

[131] "About the New Architecture". React Native Documentation. https://reactnative.dev/architecture/landing-page.

- [1] "React Native 0.74 - Yoga 3.0, Bridgeless New Architecture, and more". React Native Blog (2024-04-22). Section "New Architecture: Bridgeless by Default" states: "In this release, we are making Bridgeless Mode the default when the New Architecture is enabled." https://reactnative.dev/blog/2024/04/22/release-0.74
- [2] "React Native 0.76 - New Architecture by default, React Native DevTools, and more". React Native Blog (2024-10-23). https://reactnative.dev/blog/2024/10/23/release-0.76-new-architecture. Companion post "New Architecture is here" covers the bridge-removal rationale and the gradual-migration / opt-out story: https://reactnative.dev/blog/2024/10/23/the-new-architecture-is-here.
- [3] JSI `Runtime` class declared in `packages/react-native/ReactCommon/jsi/jsi/jsi.h:705` (`class JSI_EXPORT Runtime : public IRuntime`). Header-only C++ API with synchronous `evaluateJavaScript`, `getProperty`, `setProperty`, `call`, etc.
- [4] Shadow Tree implemented in `packages/react-native/ReactCommon/react/renderer/core/ShadowNode.h`. The multi-priority scheduler that powers concurrent rendering lives in `packages/react-native/ReactCommon/react/renderer/runtimescheduler/RuntimeScheduler_Modern.h:24` (`class RuntimeScheduler_Modern final : public RuntimeSchedulerBase`).
- [5] `packages/react-native/ReactCommon/react/nativemodule/core/ReactCommon/TurboModule.h:49`: `class JSI_EXPORT TurboModule : public jsi::HostObject`. The `get(jsi::Runtime&, const jsi::PropNameID&)` override returns a `jsi::Value` synchronously.
- [6] Generators in `packages/react-native-codegen/src/generators/modules/`: `GenerateModuleH.js` (C++), `GenerateModuleJavaSpec.js`, `GenerateModuleJniCpp.js`, `GenerateModuleJniH.js`, `GenerateModuleObjCpp/`. All driven from `*NativeModule.{ts,js.flow}` specs parsed by `packages/react-native-codegen/src/parsers/`.
- [7] React peer dependency in `packages/react-native/package.json`: `"react": "^19.3.0"`. React 19 was first required by RN 0.78 ("**Deps:** Fix peer dependencies on React types to React 19" in CHANGELOG-0.7x.md's v0.78.0 section, released 2025-02-19).
- [8] Event loop alignment documented in `packages/react-native/ReactCommon/react/renderer/runtimescheduler/__docs__/README.md`, which links the implementation to the HTML spec event loop and the well-defined event loop RFC. DOM-style APIs (`getBoundingClientRect`, etc.) live in `packages/react-native/src/private/webapis/dom/`.
- [9] "About the New Architecture". React Native Documentation. https://reactnative.dev/architecture/landing-page.

Chapter 10: Common Mistakes and Misunderstandings

The New Architecture introduces powerful concepts, but also new ways for things to go wrong. Understanding common pitfalls can save significant debugging time. This chapter outlines frequent mistakes and misunderstandings grouped by component.

(Version context: starting with v0.76, Fabric, TurboModules, and Bridgeless are the defaults. The mistakes below assume an app on v0.76 or newer. On the v0.87 and v0.88 lines, TurboModules are always on, opt-out is long gone (since 0.82), and the interop paths described below are still present but behind `RCT_REMOVE_LEGACY_*` compile gates. The legacy iOS bridge sources (for example `RCTCxxBridge.mm`) were removed from main in 0.87.)

General Misunderstandings

Mistake: Assuming the New Architecture automatically makes existing code faster.

- **Why it's a mistake:** Shipping on the New Architecture (the only option on RN ≥ 0.82) does not by itself optimize legacy components. An old, Bridge-based native module will still communicate via a compatibility layer. The performance benefits are only realized when a module is migrated to a TurboModule or a component to a Fabric Component.
- **How to avoid:** Understand that performance gains require active migration of your native code to use the new, JSI-based APIs. The Android-side compatibility shim is the `InteropModuleRegistry`, gated behind the `useFabricInterop()` feature flag. It is annotated `@InteropLegacyArchitecture` in the source, which is a strong signal that this is a migration crutch rather than a long-term performance path — and on 0.87 and later the C++/platform interop surfaces are themselves compile-optional behind `RCT_REMOVE_LEGACY_*_INTEROP`.[132]

Mistake: Ignoring third-party library compatibility.

- **Why it's a mistake:** Enabling the New Architecture while using a third-party native library that does not support it is a common source of crashes and unexpected behavior. The library may rely on the old Bridge, which is not fully available in Bridgeless mode. Many UIManager methods (`measure`, `measureLayout`, `dispatchViewManagerCommand` constants, etc.) become soft errors in Bridgeless mode and log `'<method>' is not available in the new React Native architecture` instead of working.[133]
- **How to avoid:** Before migrating, audit all native dependencies. Check their documentation for "New Architecture," "Fabric," or "TurboModule" support. Update to compatible versions or find alternatives.

JSI and C++ Mistakes

Mistake: Blocking the JavaScript thread with a long-running synchronous method.

---

[132] `packages/react-native/ReactAndroid/src/main/java/com/facebook/react/bridge/interop/InteropModuleRegistry.kt:15-52`. The class is annotated `@InteropLegacyArchitecture` and only returns modules when `useFabricInterop()` is true.

[133] `packages/react-native/Libraries/ReactNative/BridgelessUIManager.js:22-101`. Many UIManager APIs (`measure`, `measureLayout`, `measureInWindow`, etc.) call `raiseSoftError(methodName)` which logs `'<methodName>' is not available in the new React Native architecture`.

- **Why it's a mistake:** The ability to create synchronous methods is powerful, but it's also dangerous. Every TurboModule extends `jsi::HostObject`, so its method calls run on the calling thread (typically the JS thread) and return a `jsi::Value` directly.[134] If a synchronous native method performs file I/O, network requests, or heavy computation, it will completely block the JS thread, freezing your app's UI.
- **How to avoid:** Reserve synchronous methods for only the most trivial, fast operations (e.g., reading a pre-calculated value from memory). For everything else, declare the method as returning a `Promise` in the spec (CodeGen will produce `PromiseKind` on the C++ side[135]) and do the work off-thread.

Mistake: Ignoring JSI thread safety.

- **Why it's a mistake:** A `jsi::Runtime` instance is not thread-safe. The JSI source spells this out: the Runtime "may not be thread-aware, but cannot be used safely from multiple threads at once. The application is responsible for ensuring that it is used safely. This could mean using the Runtime from a single thread, using a mutex, doing all work on a serial queue, etc."[136] Calling JSI methods that take a `jsi::Runtime&` from multiple threads without locking will lead to memory corruption and crashes.
- **How to avoid:** Ensure all JSI interactions for a given runtime happen on a single, dedicated thread. If you need to pass data between threads, use thread-safe mechanisms and dispatch back to the JSI thread to interact with JavaScript. JSI also ships a `ThreadSafeRuntime` decorator (in `jsi/threadsafe.h`) that wraps any `Runtime` with explicit `lock()`/`unlock()` calls if you really need shared access from a small number of threads.[137]

Mistake: Performing complex operations in a `HostObject` destructor.

- **Why it's a mistake:** The destructor for a C++ `HostObject` is called when the JavaScript engine's garbage collector finalizes the object. The canonical JSI comment is worth reading verbatim: "The C++ object's dtor will be called when the GC finalizes this object. (This may be as late as when the Runtime is shut down.) You have no control over which thread it is called on. This will be called from inside the GC, so it is unsafe to do any VM operations which require a `IRuntime&`."[138] So the destructor can fire on any thread, at any time, possibly with the runtime already dead. It is unsafe to call any JSI method that takes a `Runtime&`. Destructing other jsi

[134] `packages/react-native/ReactCommon/react/nativemodule/core/ReactCommon/TurboModule.h:27-49`. `TurboModule` extends `jsi::HostObject` (line 49). `TurboModuleMethodValueKind` enumerates the supported return shapes including `PromiseKind` (line 35).

[135] `packages/react-native/ReactCommon/react/nativemodule/core/ReactCommon/TurboModule.h:27-49`. `TurboModule` extends `jsi::HostObject` (line 49). `TurboModuleMethodValueKind` enumerates the supported return shapes including `PromiseKind` (line 35).

[136] `packages/react-native/ReactCommon/jsi/jsi/jsi.h:687-704`. The `Runtime` class header comment is the canonical statement about JSI thread safety.

[137] `packages/react-native/ReactCommon/jsi/jsi/threadsafe.h:18-23`. `ThreadSafeRuntime` (lines 18-22) is a `Runtime` subclass with `virtual void lock() const = 0; virtual void unlock() const = 0; virtual Runtime& getUnsafeRuntime() = 0;`.

[138] `packages/react-native/ReactCommon/jsi/jsi/jsi.h:222-233`. The `HostObject` destructor comment is the source-of-truth for the GC/thread/safety constraints.

objects (`Value`, `Object`, etc.) is explicitly allowed. The warning is about calling into the VM, not about C++ destruction.
- **How to avoid:** The destructor should only be used for trivial cleanup of the C++ object's own memory. Derived classes' destructors should also avoid doing anything expensive.[139] If you need to perform more complex cleanup that involves other systems or the JSI, implement an explicit `invalidate()` or `cleanup()` method on your Host Object that you call from JavaScript while the runtime is still alive, before releasing its reference.

TurboModule Mistakes

Mistake: Module name mismatch.

- **Why it's a mistake:** JavaScript finds a TurboModule by its name. If the name you provide in `TurboModuleRegistry.getEnforcing<Spec>('MyModule')` does not exactly match the name on the native registration side (`BaseReactPackage.getModule(name, …)` on Android[140], or the `getTurboModule:` method on an `RCTModuleProvider` on iOS[141]), the lookup fails. `getEnforcing` then throws via `invariant` with the message `TurboModuleRegistry.getEnforcing(...): '<name>' could not be found. Verify that a module by this name is registered in the native binary.`[142] In practice this surfaces as a red-screen error in development and a fatal JS exception at the import callsite in production.
- **How to avoid:** Ensure the module name is identical across the JS spec, the Android `BaseReactPackage.getModule` switch, and the iOS module provider. Both `package.json`'s `codegenConfig.libraries[].name` and the runtime registration must agree.

Fabric Component Mistakes

Mistake: Treating `updateProps` as a constructor or a one-time setup.

- **Why it's a mistake:** The `updateProps` method on a native component view will be called many times throughout the component's lifecycle. Every time a prop changes, the mounting layer calls `[componentView updateProps:newProps oldProps:oldProps]` (the Objective-C signature on the `RCTComponentViewProtocol`[143]). The mount manager guards this with `if (oldChildShadowView.props != newChildShadowView.props)`, a pointer comparison on the

[139] `packages/react-native/ReactCommon/jsi/jsi/jsi.h:222-233`. The `HostObject` destructor comment is the source-of-truth for the GC/thread/safety constraints.

[140] `packages/react-native/ReactAndroid/src/main/java/com/facebook/react/BaseReactPackage.kt:22-41`. `BaseReactPackage` is the modern parent class. The older `ReactPackage.createNativeModules` is now `@Deprecated` (see `packages/react-native/ReactAndroid/src/main/java/com/facebook/react/ReactPackage.kt:36-38`).

[141] `packages/react-native/ReactCommon/react/nativemodule/core/platform/ios/ReactCommon/RCTTurboModule.h:196-204`. The `RCTModuleProvider` protocol declares `- (std::shared_ptr<facebook::react::TurboModule>)getTurboModule:(const facebook::react::ObjCTurboModule::InitParams &)params;`.

[142] "Turbo Native Modules". React Native Documentation. https://reactnative.dev/docs/next/turbo-native-modules-introduction. The exact `invariant` message is in `packages/react-native/Libraries/TurboModule/TurboModuleRegistry.js:40-47`.

[143] "Fabric Native Components: iOS". React Native Documentation. https://reactnative.dev/docs/next/fabric-native-components-introduction. The protocol declaring `updateProps:oldProps:` and `handleCommand:args:` is `packages/react-native/React/Fabric/Mounting/RCTComponentViewProtocol.h:67-68, 95`.

immutable shared `Props`, which means any non-equal new props object triggers a call.[144] Performing expensive, one-time setup work inside `updateProps` runs that work on every prop change.

- **How to avoid:** Use the view's native initializer (e.g., `init` on iOS) for one-time setup. Use `updateProps` only for applying the new properties to the view. Also remember that Fabric component views are **recycled**: after unmount, the mount manager calls `prepareForRecycle`[145] and the view returns to a pool to be reused for a different React component later. A single `UIView` instance may serve multiple React lives. Use `prepareForRecycle` to clear any state that mustn't leak across reuse (timers, observers, cached layers, etc.). Don't rely on `init` alone for "reset to defaults" behavior.

Mistake: Passing frequently changing values as props.

- **Why it's a mistake:** While Fabric is much more efficient than the old UIManager, every prop change still results in a call from C++ to the native view (see the pointer-equality check at the mount-manager level cited above[146]). If you are passing down values from an animation that updates every frame (e.g., from `useAnimatedStyle` in Reanimated), this can still create significant traffic.
- **How to avoid:** For high-frequency updates like animations, use libraries like Reanimated that are designed to perform the animations directly on the UI thread, bypassing the React render cycle entirely. For one-off imperative actions, use commands instead of props. Commands are defined via `codegenNativeCommands<T>({ supportedCommands: [...] })`[147] and dispatched through `ReactFabric.dispatchCommand`. The component view receives them via `handleCommand:args:` on the `RCTComponentViewProtocol`.[148]

CodeGen Mistakes

Mistake: Forgetting to rebuild after changing a spec.

- **Why it's a mistake:** CodeGen runs during the build process. On iOS, `pod install` calls `run_codegen!` which wires `use_react_native_codegen_discovery!` into the install pipeline.[149] On Android, the Gradle plugin makes `preBuild` depend on the

[144] `packages/react-native/React/Fabric/Mounting/RCTMountingManager.mm:112-113`. The mount manager only calls `updateProps:` when `oldChildShadowView.props != newChildShadowView.props` (pointer equality on the immutable shared `Props`).

[145] `packages/react-native/React/Fabric/Mounting/RCTComponentViewProtocol.h:105-109`. `- (void)prepareForRecycle;` is documented as "Called right after the component view is moved to a recycle pool. Receiver must reset any local state and release associated non-reusable resources." A real implementation example: `packages/react-native/React/Fabric/Mounting/ComponentViews/View/RCTViewComponentView.mm:853-870`.

[146] `packages/react-native/React/Fabric/Mounting/RCTMountingManager.mm:112-113`. The mount manager only calls `updateProps:` when `oldChildShadowView.props != newChildShadowView.props` (pointer equality on the immutable shared `Props`).

[147] `packages/react-native/Libraries/Utilities/codegenNativeCommands.js:17-31`. The helper returns command-dispatch functions that call `dispatchCommand(ref, command, args)`. Real-world example: `packages/react-native/Libraries/Components/ScrollView/ScrollViewCommands.js:45-52` (`flashScrollIndicators`, `scrollTo`, `scrollToEnd`, `zoomToRect`).

[148] "Fabric Native Components: iOS". React Native Documentation. https://reactnative.dev/docs/next/fabric-native-components-introduction. The protocol declaring `updateProps:oldProps:` and `handleCommand:args:` is `packages/react-native/React/Fabric/Mounting/RCTComponentViewProtocol.h:67-68, 95`.

`generateCodegenArtifactsFromSchema` task.[150] If you change a JavaScript spec file (e.g., add a new method to a TurboModule), the native interface code is not automatically updated until one of those build steps runs again. You must re-run the build (`pod install` for iOS, or a Gradle build for Android) to regenerate the native files.[151]

- **How to avoid:** After any change to a `Spec` file, perform a clean build of your native project to ensure the generated code is up-to-date.

Mistake: Incorrect `codegenConfig` in `package.json`.

- **Why it's a mistake:** If the `jsSrcsDir` or other paths in the `codegenConfig` are incorrect, the CodeGen script won't find your spec files. `generateSchemaInfos.js` joins `library.libraryPath` with `library.config.jsSrcsDir` to locate the specs.[152] A wrong path simply yields an empty schema, and downstream artifacts (TurboModule providers, third-party component lists) end up without the bindings you expected, leading to compilation errors when the registration code references symbols that were never generated.
- **How to avoid:** Double-check all paths and package names in the `codegenConfig` section to ensure they are correct. The `react-native` package itself is a useful reference: see its own `codegenConfig` block in `packages/react-native/package.json`, with `"name": "FBReactNativeSpec"` and `"jsSrcsDir": "src"`.[153]

Citations:

- [1] "Turbo Native Modules". React Native Documentation. https://reactnative.dev/docs/next/turbo-native-modules-introduction. The exact `invariant` message is in `packages/react-native/Libraries/TurboModule/TurboModuleRegistry.js:40-47`.
- [2] "Fabric Native Components: iOS". React Native Documentation. https://reactnative.dev/docs/next/fabric-native-components-introduction. The protocol declaring `updateProps:oldProps:` and `handleCommand:args:` is `packages/react-native/React/Fabric/Mounting/RCTComponentViewProtocol.h:67-68, 95`.
- [3] "Using Codegen". React Native Documentation. https://reactnative.dev/docs/next/the-new-architecture/using-codegen.

---

[149] `packages/react-native/scripts/cocoapods/codegen.rb:9-39`. `run_codegen!` wires `use_react_native_codegen_discovery!` into pod install.

[150] `packages/gradle-plugin/react-native-gradle-plugin/src/main/kotlin/com/facebook/react/ReactPlugin.kt:275`. `project.tasks.named("preBuild", Task::class.java).dependsOn(generateCodegenArtifactsTask)`.

[151] "Using Codegen". React Native Documentation. https://reactnative.dev/docs/next/the-new-architecture/using-codegen.

[152] `packages/react-native/scripts/codegen/generate-artifacts-executor/generateSchemaInfos.js:29-33` (joins `library.libraryPath` and `library.config.jsSrcsDir`). See also `packages/react-native/package.json:183-211` (real `codegenConfig` example).

[153] `packages/react-native/scripts/codegen/generate-artifacts-executor/generateSchemaInfos.js:29-33` (joins `library.libraryPath` and `library.config.jsSrcsDir`). See also `packages/react-native/package.json:183-211` (real `codegenConfig` example).

- [4] packages/react-native/ReactAndroid/src/main/java/com/facebook/react/bridge/interop/InteropModuleRegistry.kt:15-52. The class is annotated @InteropLegacyArchitecture and only returns modules when useFabricInterop() is true.
- [5] packages/react-native/Libraries/ReactNative/BridgelessUIManager.js:22-101. Many UIManager APIs (measure, measureLayout, measureInWindow, etc.) call raiseSoftError(methodName) which logs '<methodName>' is not available in the new React Native architecture.
- [6] packages/react-native/ReactCommon/react/nativemodule/core/ReactCommon/TurboModule.h:27-49. TurboModule extends jsi::HostObject (line 49). TurboModuleMethodValueKind enumerates the supported return shapes including PromiseKind (line 35).
- [7] packages/react-native/ReactCommon/jsi/jsi/jsi.h:687-704. The Runtime class header comment is the canonical statement about JSI thread safety.
- [8] packages/react-native/ReactCommon/jsi/jsi/threadsafe.h:18-23. ThreadSafeRuntime (lines 18-22) is a Runtime subclass with virtual void lock() const = 0; virtual void unlock() const = 0; virtual Runtime& getUnsafeRuntime() = 0;.
- [9] packages/react-native/ReactCommon/jsi/jsi/jsi.h:222-233. The HostObject destructor comment is the source-of-truth for the GC/thread/safety constraints.
- [10] packages/react-native/ReactAndroid/src/main/java/com/facebook/react/BaseReactPackage.kt:22-41. BaseReactPackage is the modern parent class. The older ReactPackage.createNativeModules is now @Deprecated (see packages/react-native/ReactAndroid/src/main/java/com/facebook/react/ReactPackage.kt:36-38).
- [11] packages/react-native/ReactCommon/react/nativemodule/core/platform/ios/ReactCommon/RCTTurboModule.h:196-204. The RCTModuleProvider protocol declares - (std::shared_ptr<facebook::react::TurboModule>)getTurboModule:(const facebook::react::ObjCTurboModule::InitParams &)params;.
- [12] packages/react-native/React/Fabric/Mounting/RCTMountingManager.mm:112-113. The mount manager only calls updateProps: when oldChildShadowView.props != newChildShadowView.props (pointer equality on the immutable shared Props).
- [13] packages/react-native/React/Fabric/Mounting/RCTComponentViewProtocol.h:105-109. - (void)prepareForRecycle; is documented as "Called right after the component view is moved to a recycle pool. Receiver must reset any local state and release associated non-reusable resources." A real implementation example: packages/react-native/React/Fabric/Mounting/ComponentViews/View/RCTViewComponentView.mm:853-870.
- [14] packages/react-native/Libraries/Utilities/codegenNativeCommands.js:17-31. The helper returns command-dispatch functions that call dispatchCommand(ref, command, args). Real-world example: packages/react-native/Libraries/Components/ScrollView/ScrollViewCommands.js:45-52 (flashScrollIndicators, scrollTo, scrollToEnd, zoomToRect).
- [15] packages/react-native/scripts/cocoapods/codegen.rb:9-39. run_codegen! wires use_react_native_codegen_discovery! into pod install.

- [16] `packages/gradle-plugin/react-native-gradle-plugin/src/main/kotlin/com/facebook/react/ReactPlugin.kt:275`. `project.tasks.named("preBuild", Task::class.java).dependsOn(generateCodegenArtifactsTask)`.
- [17] `packages/react-native/scripts/codegen/generate-artifacts-executor/generateSchemaInfos.js:29-33` (joins `library.libraryPath` and `library.config.jsSrcsDir`). See also `packages/react-native/package.json:183-211` (real `codegenConfig` example).

Chapter 11: Best Practices

Adopting the New Architecture effectively involves more than just avoiding mistakes; it requires embracing new patterns and best practices. This chapter provides a list of recommendations for writing high-quality, performant code.

General Practices

- **Do** migrate one module or component at a time. The New Architecture is designed for incremental adoption. While your app is on the new arch, React Native still ships an interop layer that lets legacy `RCTBridgeModule` modules and `RCTViewManager`-style components keep working until you port them.[154] This allows you to isolate issues and learn the new patterns in a controlled way.
- **Don't** attempt a "big bang" migration of a large, complex application all at once. This makes debugging extremely difficult.
- **Do** keep your JavaScript specs as the single source of truth. All native implementations should strictly follow the CodeGen-generated interfaces. The generated `*Spec` base classes enforce signatures at compile time, so a drifted method will fail to build rather than fail at runtime.[155]
- **Do** audit your third-party dependencies early and often. Prefer libraries that explicitly state support for the New Architecture.
- **Do** consider higher-level tooling where appropriate. For teams that want to simplify the module creation process or extract maximum performance, alternative frameworks like **Nitro** (see Chapter 14) provide abstractions on top of the standard tools and may be a good fit.

JSI and C++ Practices

- **Do** use C++ for performance-critical, cross-platform logic. The JSI makes it easier than ever to write a function once in C++ and use it from both iOS and Android, reducing code duplication and improving performance. The modern way to expose that C++ to JS is a **Pure C++ TurboModule**: a spec that generates a `Native<Name>CxxSpec<T>` base class, linked into both platforms from a single `.cpp`. React Native itself ships several this way, including `NativeMicrotasks` and `NativeIdleCallbacks`.[156]
- **Don't** put platform-specific code (e.g., `UIKit` or Android SDK calls) in your core C++ modules. Keep the C++ layer platform-agnostic and use platform-specific adapter code to call into it.

---

[154] Interop entry points: `packages/react-native/ReactCommon/react/nativemodule/core/platform/ios/ReactCommon/RCTInteropTurboModule.h` (legacy NM shim), `packages/react-native/ReactCommon/react/renderer/components/legacyviewmanagerinterop/` (iOS view-manager shim), and `packages/react-native/ReactAndroid/src/main/java/com/facebook/react/bridge/interop/InteropModuleRegistry.kt` (Android counterpart, gated by `useFabricInterop()`). As of RN HEAD `9c7b0f5fafd` (v0.88.1 era) these are compile-gated behind `RCT_REMOVE_LEGACY_MODULE_INTEROP` / `RCT_REMOVE_LEGACY_COMPONENT_INTEROP`.

[155] CodeGen module pipeline: `packages/react-native-codegen/src/parsers/parsers-commons.js:328-425` (parsing) and `packages/react-native-codegen/src/generators/modules/GenerateModuleH.js` (generation). Generated specs become abstract C++ base classes that the native implementation `overrides`.

[156] Pure C++ TurboModules in RN itself: `packages/react-native/ReactCommon/react/nativemodule/microtasks/NativeMicrotasks.{h,cpp}` and `packages/react-native/ReactCommon/react/nativemodule/idlecallbacks/`. Each one extends a `Native<Name>CxxSpec<Impl>` base generated from the JS spec under `packages/react-native/src/private/webapis/.../specs/`.

- **Do** handle threading carefully. The JS thread is *not* the UI/main thread under the New Architecture. If a TurboModule call needs to perform work off the JS thread, dispatch it to a background queue, then use the `CallInvoker` (specifically `invokeAsync(CallFunc&&)`) to deliver the result back to the JS thread, where it is safe to touch the `jsi::Runtime` again.[157]

TurboModule Practices

- **Do** keep synchronous methods trivial and fast. They are perfect for fetching simple, precalculated values that are already in memory.[158]
- **Don't** ever perform I/O (network, disk), heavy computation, or any potentially long-running task in a synchronous TurboModule method. A sync method is just a `jsi::Value(jsi::Runtime&, ...)` C++ function invoked inline on the JS thread, so anything that blocks inside it blocks JS execution and freezes the app.[159][160]
- **Do** use Promises for any method that is asynchronous or might take more than a millisecond to complete. This is the standard and expected pattern for async work, and `Promise<T>` is a first-class return shape in CodeGen.[161][162]
- **Do** prefer Promises over callbacks for new async methods. Callbacks are *not* legacy though: a `(arg: T) => void` parameter is a first-class TurboModule signature, generated as a `jsi::Function` and supported in both bridged and pure-C++ modules (see `NativeIdleCallbacks` and `NativeMicrotasks` in core RN for examples).[163] Promises are preferred because they integrate cleanly with `async/await`, have a single resolution path, and unify error handling, not because callbacks "only work via interop".

Fabric Component Practices

- **Do** treat `updateProps` as the place to apply changed properties to your native view. The method itself isn't an automatic diff: on iOS, the protocol gives you both the new `Props::Shared` and

[157] `CallInvoker` definition: `packages/react-native/ReactCommon/callinvoker/ReactCommon/CallInvoker.h:27-50`. `invokeAsync(CallFunc&&)` schedules a callback that receives a `jsi::Runtime&` argument, which is the contract for "run me on the JS thread."

[158] "Native Modules" (formerly titled "Turbo Native Modules"). React Native Documentation. https://reactnative.dev/docs/next/turbo-native-modules-introduction

[159] "Native Modules" (formerly titled "Turbo Native Modules"). React Native Documentation. https://reactnative.dev/docs/next/turbo-native-modules-introduction

[160] `TurboModule` invocation surface: `packages/react-native/ReactCommon/react/nativemodule/core/ReactCommon/TurboModule.h:86` defines `MethodMetadata::invoker` as `jsi::Value(jsi::Runtime&, TurboModule&, const jsi::Value*, size_t)`, and `TurboModule::create` (lines 119-132) calls it inline from the JS-thread host function.

[161] "Native Modules" (formerly titled "Turbo Native Modules"). React Native Documentation. https://reactnative.dev/docs/next/turbo-native-modules-introduction

[162] `PromiseKind` is one of the `TurboModuleMethodValueKind` enum members in the same `TurboModule.h:27-36`. The codegen path for `Promise<T>` lives in `packages/react-native-codegen/src/parsers/parsers-primitives.js:333-388` (`emitPromise`).

[163] Current-era TurboModule specs with callback parameters: `packages/react-native/src/private/webapis/microtasks/specs/NativeMicrotasks.js:16` and `packages/react-native/src/private/webapis/idlecallbacks/specs/NativeIdleCallbacks.js:27-30`. Generator emits `jsi::Function` for `FunctionTypeAnnotation` arguments in `packages/react-native-codegen/src/generators/modules/GenerateModuleH.js:126-127`. `packages/react-native/ReactCommon/react/bridging/CallbackWrapper.h` provides the helper for keeping a `jsi::Function` alive across threads.

the previous `oldProps`, and the component view casts them and compares fields by hand (`if (oldViewProps.opacity != newViewProps.opacity) { ... }` is the canonical pattern in `RCTViewComponentView`). On Android the wrapper passed in is a `ReactStylesDiffMap`, which only contains the keys that actually changed. Either way the rule is the same: apply only what differs.[164][165]

- **Don't** perform expensive allocations or setup work inside `updateProps`. Place one-time setup code in the view's native initializer, and reset any per-mount state in `prepareForRecycle`, because Fabric recycles component views and the initializer only runs once per allocated view, not per mount.[166][167]
- **Do** use commands for one-off, imperative actions. If you need to trigger an action on a native view (e.g., `focus()`, `clear()`), defining it as a command is much more efficient than changing a prop to trigger the effect. A command is a single JS-to-native dispatch (`dispatchCommand` → `handleCommand:args:`), while a prop change runs the full shadow-tree update + mount-transaction + `updateProps` pipeline.[168][169]
- **Don't** use props for ephemeral, event-like triggers. For example, to trigger an animation, it's better to use a command like `myView.startAnimation()` than to toggle a prop like `shouldAnimate={true}`.[170]
- **Do** colocate your component's related files. A good pattern is to have a single directory containing the JS spec, the iOS implementation (`.mm`), and the Android implementation (`.java`/`.kt`) for a single component. React Native itself is migrating its own first-party modules to this layout under `packages/react-native/src/private/`, with paired native code in `ReactCommon/react/nativemodule/`.[171][172]

---

[164] "Fabric Native Components". React Native Documentation. https://reactnative.dev/docs/next/fabric-native-components-introduction

[165] iOS `updateProps` protocol: `packages/react-native/React/Fabric/Mounting/RCTComponentViewProtocol.h:67-68`. Default no-op impl at `packages/react-native/React/Fabric/Mounting/UIView+ComponentViewProtocol.mm`. The reference manual diff lives in `packages/react-native/React/Fabric/Mounting/ComponentViews/View/RCTViewComponentView.mm:334-360`. Android counterpart: `packages/react-native/ReactAndroid/src/main/java/com/facebook/react/fabric/mounting/SurfaceMountingManager.kt:633-682` together with `ReactStylesDiffMap` at `packages/react-native/ReactAndroid/src/main/java/com/facebook/react/uimanager/ReactStylesDiffMap.kt:14-30`.

[166] "Fabric Native Components". React Native Documentation. https://reactnative.dev/docs/next/fabric-native-components-introduction

[167] `prepareForRecycle` in the iOS protocol: `packages/react-native/React/Fabric/Mounting/RCTComponentViewProtocol.h:105-109`. Component views are pooled and reused across mounts, so per-mount state set in `init` will leak across recycled instances unless it is reset here.

[168] "Native Commands". React Native Documentation. https://reactnative.dev/docs/next/the-new-architecture/fabric-component-native-commands

[169] Command pipeline: `packages/react-native/Libraries/Utilities/codegenNativeCommands.js:17-31` (JS side), `packages/react-native/Libraries/ReactNative/BridgelessUIManager.js:285-300` (focus/blur dispatched as commands), `packages/react-native/React/Fabric/Mounting/RCTComponentViewProtocol.h:95` (`handleCommand:args:` on the component view).

[170] "Native Commands". React Native Documentation. https://reactnative.dev/docs/next/the-new-architecture/fabric-component-native-commands

[171] "Fabric Native Components". React Native Documentation. https://reactnative.dev/docs/next/fabric-native-components-introduction

Citations:

- [1] "Native Modules" (formerly titled "Turbo Native Modules"). React Native Documentation. https://reactnative.dev/docs/next/turbo-native-modules-introduction
- [2] "Fabric Native Components". React Native Documentation. https://reactnative.dev/docs/next/fabric-native-components-introduction
- [3] "Native Commands". React Native Documentation. https://reactnative.dev/docs/next/the-new-architecture/fabric-component-native-commands
- [4] Interop entry points: `packages/react-native/ReactCommon/react/nativemodule/core/platform/ios/ReactCommon/RCTInteropTurboModule.h` (legacy NM shim), `packages/react-native/ReactCommon/react/renderer/components/legacyviewmanagerinterop/` (iOS view-manager shim), and `packages/react-native/ReactAndroid/src/main/java/com/facebook/react/bridge/interop/InteropModuleRegistry.kt` (Android counterpart, gated by `useFabricInterop()`). As of RN HEAD `9c7b0f5fafd` (v0.88.1 era) these are compile-gated behind `RCT_REMOVE_LEGACY_MODULE_INTEROP` / `RCT_REMOVE_LEGACY_COMPONENT_INTEROP`.
- [5] CodeGen module pipeline: `packages/react-native-codegen/src/parsers/parsers-commons.js:328-425` (parsing) and `packages/react-native-codegen/src/generators/modules/GenerateModuleH.js` (generation). Generated specs become abstract C++ base classes that the native implementation `overrides`.
- [6] Pure C++ TurboModules in RN itself: `packages/react-native/ReactCommon/react/nativemodule/microtasks/NativeMicrotasks.{h,cpp}` and `packages/react-native/ReactCommon/react/nativemodule/idlecallbacks/`. Each one extends a `Native<Name>CxxSpec<Impl>` base generated from the JS spec under `packages/react-native/src/private/webapis/.../specs/`.
- [7] `CallInvoker` definition: `packages/react-native/ReactCommon/callinvoker/ReactCommon/CallInvoker.h:27-50`. `invokeAsync(CallFunc&&)` schedules a callback that receives a `jsi::Runtime&` argument, which is the contract for "run me on the JS thread."
- [8] `TurboModule` invocation surface: `packages/react-native/ReactCommon/react/nativemodule/core/ReactCommon/TurboModule.h:86` defines `MethodMetadata::invoker` as `jsi::Value(jsi::Runtime&, TurboModule&, const jsi::Value*, size_t)`, and `TurboModule::create` (lines 119-132) calls it inline from the JS-thread host function.
- [9] `PromiseKind` is one of the `TurboModuleMethodValueKind` enum members in the same `TurboModule.h:27-36`. The codegen path for `Promise<T>` lives in `packages/react-native-codegen/src/parsers/parsers-primitives.js:333-388` (`emitPromise`).
- [10] Current-era TurboModule specs with callback parameters: `packages/react-native/src/private/webapis/microtasks/specs/NativeMicrotasks.js:16` and `packages/react-native/src/private/webapis/idlecallbacks/specs/NativeIdleCallbacks.js:27-30`. Generator emits `jsi::Function` for `FunctionTypeAnnotation` arguments in `packages/react-native-codegen/src/generators/modules/GenerateModuleH.js:126-127`.

[172] Migration convention: `packages/react-native/src/private/specs_DEPRECATED/__docs__/README.md` ("Do not create new specs in this directory. Instead, create them colocated with the rest of the logic in an appropriate directory in `src/private`"). The webapis modules listed in footnote 6 follow this layout end to end.

`packages/react-native/ReactCommon/react/bridging/CallbackWrapper.h` provides the helper for keeping a `jsi::Function` alive across threads.

- [11] iOS `updateProps` protocol: `packages/react-native/React/Fabric/Mounting/RCTComponentViewProtocol.h:67-68`. Default no-op impl at `packages/react-native/React/Fabric/Mounting/UIView+ComponentViewProtocol.mm`. The reference manual diff lives in `packages/react-native/React/Fabric/Mounting/ComponentViews/View/RCTViewComponentView.mm:334-360`. Android counterpart: `packages/react-native/ReactAndroid/src/main/java/com/facebook/react/fabric/mounting/SurfaceMountingManager.kt:633-682` together with `ReactStylesDiffMap` at `packages/react-native/ReactAndroid/src/main/java/com/facebook/react/uimanager/ReactStylesDiffMap.kt:14-30`.
- [12] `prepareForRecycle` in the iOS protocol: `packages/react-native/React/Fabric/Mounting/RCTComponentViewProtocol.h:105-109`. Component views are pooled and reused across mounts, so per-mount state set in `init` will leak across recycled instances unless it is reset here.
- [13] Command pipeline: `packages/react-native/Libraries/Utilities/codegenNativeCommands.js:17-31` (JS side), `packages/react-native/Libraries/ReactNative/BridgelessUIManager.js:285-300` (focus/blur dispatched as commands), `packages/react-native/React/Fabric/Mounting/RCTComponentViewProtocol.h:95` (`handleCommand:args:` on the component view).
- [14] Migration convention: `packages/react-native/src/private/specs_DEPRECATED/__docs__/README.md` ("Do not create new specs in this directory. Instead, create them colocated with the rest of the logic in an appropriate directory in `src/private`"). The webapis modules listed in footnote 6 follow this layout end to end.

Chapter 12: Frequently Asked Questions (FAQs)

This chapter addresses common high-level questions about the React Native New Architecture.

1. Do I have to migrate to the New Architecture?

Yes, if you want to stay on a supported React Native release. As of React Native 0.76, the New Architecture is the default for new projects.[173] As of React Native 0.82, **opt-out was removed**: the only way to keep the legacy Bridge is to stay on a `0.81.x` (or earlier) tag. All future core development and Concurrent React features assume the New Architecture.

Status as of v0.88.1 and `main` (commit `9c7b0f5fafd`): the New Architecture is hard-wired on. On iOS, `RCTIsNewArchEnabled()` unconditionally returns `YES` and the matching setter is a documented no-op.[174] The Cocoapods helper `NewArchitectureHelper.new_arch_enabled` also returns `true` outright.[175] Setting `newArchEnabled=false` in `android/gradle.properties` or `ENV['RCT_NEW_ARCH_ENABLED'] = '0'` in your iOS `Podfile` is force-overridden by the build tooling (error / warning, then enabled anyway). TurboModules are likewise always enabled. In v0.88.1, `RCTTurboModuleEnabled()` always returns `YES` and both it and `RCTEnableTurboModule()` are deprecated. On `main` they are deleted (PR #59440), so do not call them.[176] Legacy Bridge symbols and the legacy module / view-manager interop layers are increasingly compile-gated behind `RCT_REMOVE_LEGACY_ARCH`, `RCT_REMOVE_LEGACY_MODULE_INTEROP`, and `RCT_REMOVE_LEGACY_COMPONENT_INTEROP`. Since 0.88, apps can opt in to the two interop macros from both CocoaPods and SwiftPM. Both default to off and are planned to become the default later.[177]

2. Is the Bridge completely gone in the New Architecture?

In **Bridgeless Mode** (the default for new architecture apps since RN 0.74), the original message-queue Bridge is no longer initialized or used.[178] However, an **interoperability layer** still exists to support legacy native modules. So, while the old performance-bottleneck Bridge is gone, the system can still communicate with old modules.

---

[173] "About the New Architecture". React Native Documentation. https://reactnative.dev/architecture/landing-page. (The `/docs/next/the-new-architecture/landing-page` URL still works via an internal `vercel.json` redirect, but the canonical URL is this one.)

[174] `RCTIsNewArchEnabled` always returns `YES`; `RCTSetNewArchEnabled` is a no-op. Source: `packages/react-native/React/Base/RCTUtils.mm:47-50`.

[175] `NewArchitectureHelper.new_arch_enabled` returns `true`. Source: `packages/react-native/scripts/cocoapods/new_architecture.rb:172-174`.

[176] `RCTTurboModuleEnabled` / `RCTEnableTurboModule` declared deprecated in `packages/react-native/React/Base/RCTBridge.h:50-53` at tag v0.88.1 (commits `aee4a2586e`, `feede1422b`). Both functions are deleted on `main` by `01d55273bcf` (PR #59440, 2026-10-27, not in v0.88.1). Only the interop functions (`RCTEnableTurboModuleInterop`, `RCTTurboModuleInteropEnabled`) remain in `RCTBridge.h/.mm`. The `useTurboModules` feature flag was removed entirely in v0.87.0-rc.0 (`b4a715ca1f`).

[177] Examples: `RCTAppDelegate.h`, `RCTReactNativeFactory.h`, `RCTAppSetupUtils.h` carry `deprecated("... will be removed when removing the legacy architecture)`. `RCTArchConfiguratorProtocol.h` was removed from public headers on the v0.87 line.

[178] "React Native 0.74 - Yoga 3.0, Bridgeless New Architecture, and more". React Native Blog. https://reactnative.dev/blog/2024/04/22/release-0.74. The matching CHANGELOG entries are "Make bridgeless the default when the New Arch is enabled" for both Android and iOS in the v0.74.0 section.

Concretely, the New Arch detects Bridgeless via the JS global `RN$Bridgeless`. When it's set, `TurboModuleBinding::install` mounts a `BridgelessNativeModuleProxy` (a JSI `HostObject`) as `global.nativeModuleProxy`, and any access to that proxy resolves first through the TurboModule provider, then through an optional legacy module provider.[179]

3. Can I still use my old native modules and components?

Yes, for the most part. The interoperability layer allows legacy `RCTBridgeModules` and `ViewManagers` to function. However, they will not see the performance benefits of the new architecture. For full compatibility and performance, they should be migrated to TurboModules and Fabric Components.

The TurboModule interop is **on by default in Bridgeless** when the interop code is compiled in (OSS apps still ship it unless they define the removal macros): `RCTRootViewFactory.initializeReactHostWithLaunchOptions:` calls `RCTEnableTurboModuleInterop(YES)` before creating the React host on iOS,[180] and the Android New-Arch defaults class sets `useTurboModuleInterop()` to `true`.[181] Legacy `RCT_EXPORT_MODULE` modules are wrapped by `ObjCInteropTurboModule` (which dispatches through `NSInvocation`),[182] and legacy `ViewManagers` are bridged into Fabric through the `LegacyViewManagerInterop*` shadow node and component descriptor.[183] On the v0.87 and v0.88 lines those interop entry points can be compiled out behind `RCT_REMOVE_LEGACY_MODULE_INTEROP` / `RCT_REMOVE_LEGACY_COMPONENT_INTEROP` — opt-in macros, not the default OSS build. The extra indirection is what costs you performance versus a first-class TurboModule or Fabric component — and the compile-out gates are the signal that the shim path is temporary.

4. Do I need to learn C++ to use the New Architecture?

Not necessarily to *use* it, but it is highly beneficial if you are *writing* native modules. You can write TurboModules and Fabric Components using only Objective-C or Java/Kotlin, as CodeGen handles the C++ interface generation. However, understanding the C++ layer (especially JSI concepts like `HostObject`) is crucial for debugging and for unlocking advanced performance patterns, such as writing a single, cross-platform module in C++. The canonical example of the C++ path lives in the repo at `packages/rn-tester/NativeCxxModuleExample/`, with

[179] `class BridgelessNativeModuleProxy : public jsi::HostObject` and `TurboModuleBinding::install` source: `packages/react-native/ReactCommon/react/nativemodule/core/ReactCommon/TurboModuleBinding.cpp:20-84,117-160`.

[180] `RCTEnableTurboModuleInterop(YES)` is invoked in Bridgeless host setup, wrapped in `#ifndef RCT_REMOVE_LEGACY_MODULE_INTEROP`. Source: `packages/react-native/Libraries/AppDelegate/RCTRootViewFactory.mm:124-129`.

[181] `ReactNativeNewArchitectureFeatureFlagsDefaults` sets `useTurboModuleInterop()` to `true`. Source: `packages/react-native/ReactAndroid/src/main/java/com/facebook/react/internal/featureflags/ReactNativeNewArchitectureFeatureFlagsDefaults.kt:28`.

[182] `ObjCInteropTurboModule` (iOS) and `JavaInteropTurboModule` (Android) wrap legacy modules. Sources: `packages/react-native/ReactCommon/react/nativemodule/core/platform/ios/ReactCommon/RCTInteropTurboModule.{h,mm}` and `.../platform/android/ReactCommon/JavaInteropTurboModule.{h,cpp}`.

[183] `LegacyViewManagerInterop*` files under `packages/react-native/ReactCommon/react/renderer/components/legacyviewmanagerinterop/`, plus `RCTLegacyViewManagerInteropCoordinator` for iOS.

`NativeCxxModuleExample.cpp` shared between iOS and Android.[184]

5. What is the difference between a TurboModule and a Fabric Component?

- **TurboModules** are for non-UI native functionality. They replace the old `NativeModules`. Examples include accessing the device's calendar, camera, or file system.
- **Fabric Components** are for UI. They replace the old `ViewManagers`. They are native views that are rendered on the screen, like a map view or a custom video player.

6. Why is my new TurboModule `undefined` in JavaScript?

This is a common issue during migration. To be precise about the failure mode: `TurboModuleRegistry.getEnforcing('Foo')` does not return `undefined`. It throws via `invariant` with the message `TurboModuleRegistry.getEnforcing(...): 'Foo' could not be found. Verify that a module by this name is registered in the native binary.`[185] The non-throwing sibling `TurboModuleRegistry.get('Foo')` returns `null` (not `undefined`) when the lookup misses both the TurboModule proxy and the legacy `NativeModules[name]` fallback. The most frequent causes are:

- **Name Mismatch:** The name used in `TurboModuleRegistry.getEnforcing('MyModule')` in JavaScript does not exactly match the name registered in the native code (e.g., in the `BaseReactPackage` on Android).[186]
- **CodeGen Not Run:** You have created or changed the JS `Spec` file but have not rebuilt the app. You must re-run `pod install` or rebuild your Gradle project to trigger CodeGen and link the new module.[187]
- **Module Not Registered:** On Android, your module needs to be returned from a `BaseReactPackage.getModule(...)` and listed in `getReactModuleInfoProvider().getReactModuleInfos()`, with that package added to `MainApplication.getPackages()`.[188] On iOS, the New Arch path registers modules through `codegenConfig.ios.modulesProvider` in your library's `package.json` (not in the `Podfile`). `pod install` is what *runs* CodeGen against that config; the Podfile itself is not the registration site.[189]

7. Is the New Architecture always faster than the old one?

For most real-world applications, yes. The benefits in app startup (from lazy loading), UI respon-

[184] Source: `packages/rn-tester/NativeCxxModuleExample/NativeCxxModuleExample.{h,cpp,js,podspec}`.

[185] `TurboModuleRegistry.getEnforcing` uses `invariant(module != null, ...)`. `TurboModuleRegistry.get` returns `?T`. Source: `packages/react-native/Libraries/TurboModule/TurboModuleRegistry.js:35-47` (verified at HEAD).

[186] "Turbo Native Modules". React Native Documentation. https://reactnative.dev/docs/next/turbo-native-modules-introduction

[187] "Using Codegen". React Native Documentation. https://reactnative.dev/docs/next/the-new-architecture/using-codegen

[188] "Turbo Native Modules". React Native Documentation. https://reactnative.dev/docs/next/turbo-native-modules-introduction

[189] "Turbo Native Modules". React Native Documentation. https://reactnative.dev/docs/next/turbo-native-modules-introduction

siveness (from Fabric), and native call speed (from JSI) are significant.[190][191] The lazy loading isn't marketing: `TurboModuleBinding::getModule` only calls the module provider on first request, and the JS-side `jsRepresentation` populates each method via `TurboModule::get` the first time JS touches it.[192]

However, in some very simple, static rendering scenarios, the overhead of the new architecture's more complex machinery can make it appear marginally slower than the legacy system. The team itself acknowledges this when documenting the migration: "enabling the New Architecture in your app or library may not immediately improve the performance or user experience."[193] The true benefits shine in complex applications with many native interactions and dynamic UIs.

8. What broke for JS/tooling on the 0.87 line?

A few non-architecture removals commonly trip upgrades:

- **Strict TypeScript API is the default.** Prefer imports from `'react-native'`. Deep `Libraries/...` and `src/private/...` paths are untyped unless you set Metro `customConditions: ["react-native-legacy-deep-imports"]`.[194]
- **`InteractionManager` is gone.** Use `requestIdleCallback`.[195]
- **`'react-native/Libraries/Core/InitializeCore'` is deprecated.** Use `'react-native/setup-env'`.
- **Jest:** use `preset: '@react-native/jest-preset'` (the old `react-native/jest-preset` subpath is removed).

See Chapter 7's "0.87 JS / tooling breakages" table for the full checklist.

Citations:

- [1] "About the New Architecture". React Native Documentation. https://reactnative.dev/architecture/landing-page. (The `/docs/next/the-new-architecture/landing-page` URL still works via an internal `vercel.json` redirect, but the canonical URL is this one.)
- [2] "React Native 0.74 - Yoga 3.0, Bridgeless New Architecture, and more". React Native Blog. https://reactnative.dev/blog/2024/04/22/release-0.74. The matching CHANGELOG entries are

[190] "About the New Architecture". React Native Documentation. https://reactnative.dev/architecture/landing-page. (The `/docs/next/the-new-architecture/landing-page` URL still works via an internal `vercel.json` redirect, but the canonical URL is this one.)

[191] "Fabric Renderer". React Native Documentation. https://reactnative.dev/architecture/fabric-renderer

[192] `TurboModuleBinding::getModule` runs the provider lazily and attaches a `jsRepresentation` whose property accesses defer to `TurboModule::get`. Source: `packages/react-native/ReactCommon/react/nativemodule/core/ReactCommon/TurboModuleBinding.cpp:170-218`.

[193] "About the New Architecture". React Native Documentation. https://reactnative.dev/architecture/landing-page. (The `/docs/next/the-new-architecture/landing-page` URL still works via an internal `vercel.json` redirect, but the canonical URL is this one.)

[194] CHANGELOG v0.87.0 (CHANGELOG.md:292) Strict TypeScript API default (`c948b61c05`). Opt-out: Metro `customConditions: ["react-native-legacy-deep-imports"]`.

[195] CHANGELOG v0.87.0-rc.0: "Remove deprecated `InteractionManager`" (`ed893f7d98`).

"Make bridgeless the default when the New Arch is enabled" for both Android and iOS in the v0.74.0 section.

- [3] "Turbo Native Modules". React Native Documentation. https://reactnative.dev/docs/next/turbo-native-modules-introduction
- [4] "Using Codegen". React Native Documentation. https://reactnative.dev/docs/next/the-new-architecture/using-codegen
- [5] "Fabric Renderer". React Native Documentation. https://reactnative.dev/architecture/fabric-renderer
- [6] `RCTIsNewArchEnabled` always returns `YES`; `RCTSetNewArchEnabled` is a no-op. Source: `packages/react-native/React/Base/RCTUtils.mm:47-50`.
- [7] `NewArchitectureHelper.new_arch_enabled` returns `true`. Source: `packages/react-native/scripts/cocoapods/new_architecture.rb:172-174`.
- [8] Examples: `RCTAppDelegate.h`, `RCTReactNativeFactory.h`, `RCTAppSetupUtils.h` carry `deprecated(... will be removed when removing the legacy architecture)`. `RCTArchConfiguratorProtocol.h` was removed from public headers on the v0.87 line.
- [9] `class BridgelessNativeModuleProxy : public jsi::HostObject` and `TurboModuleBinding::install` source: `packages/react-native/ReactCommon/react/nativemodule/core/ReactCommon/TurboModuleBinding.cpp:20-84,117-160`.
- [10] `RCTEnableTurboModuleInterop(YES)` is invoked in Bridgeless host setup, wrapped in `#ifndef RCT_REMOVE_LEGACY_MODULE_INTEROP`. Source: `packages/react-native/Libraries/AppDelegate/RCTRootViewFactory.mm:124-129`.
- [11] `ReactNativeNewArchitectureFeatureFlagsDefaults` sets `useTurboModuleInterop()` to `true`. Source: `packages/react-native/ReactAndroid/src/main/java/com/facebook/react/internal/featureflags/ReactNativeNewArchitectureFeatureFlagsDefaults.kt:28`.
- [12] `ObjCInteropTurboModule` (iOS) and `JavaInteropTurboModule` (Android) wrap legacy modules. Sources: `packages/react-native/ReactCommon/react/nativemodule/core/platform/ios/ReactCommon/RCTInteropTurboModule.{h,mm}` and `.../platform/android/ReactCommon/JavaInteropTurboModule.{h,cpp}`.
- [13] `LegacyViewManagerInterop*` files under `packages/react-native/ReactCommon/react/renderer/components/legacyviewmanagerinterop/`, plus `RCTLegacyViewManagerInteropCoordinator` for iOS.
- [14] Source: `packages/rn-tester/NativeCxxModuleExample/NativeCxxModuleExample.{h,cpp,js,podspec}`.
- [15] `TurboModuleRegistry.getEnforcing` uses `invariant(module != null, ...)`. `TurboModuleRegistry.get` returns `?T`. Source: `packages/react-native/Libraries/TurboModule/TurboModuleRegistry.js:35-47` (verified at HEAD).
- [16] `TurboModuleBinding::getModule` runs the provider lazily and attaches a `jsRepresentation` whose property accesses defer to `TurboModule::get`. Source: `packages/react-native/ReactCommon/react/nativemodule/core/ReactCommon/TurboModuleBinding.cpp:170-218`.
- [17] `RCTTurboModuleEnabled` / `RCTEnableTurboModule` declared deprecated in `packages/react-native/React/Base/RCTBridge.h:50-53` at tag v0.88.1 (commits `aee4a2586e`, `feede1422b`). Both functions are deleted on `main` by `01d55273bcf` (PR #59440,

2026-10-27, not in v0.88.1). Only the interop functions (`RCTEnableTurboModuleInterop`, `RCTTurboModuleInteropEnabled`) remain in `RCTBridge.h/.mm`. The `useTurboModules` feature flag was removed entirely in v0.87.0-rc.0 (`b4a715ca1f`).

- [18] CHANGELOG v0.87.0 (CHANGELOG.md:292) Strict TypeScript API default (`c948b61c05`). Opt-out: Metro `customConditions: ["react-native-legacy-deep-imports"]`.
- [19] CHANGELOG v0.87.0-rc.0: "Remove deprecated `InteractionManager`" (`ed893f7d98`).

Part 3: Real-World Examples & Tooling

Chapter 13: Case Study - `react-native-vision-camera`

To understand how the New Architecture meets real-world code, we will analyze `react-native-vision-camera`. This popular library provides a high-performance camera component and has been a benchmark for advanced JSI usage long before the New Architecture umbrella term existed. It is also a useful counter-example: a library can be New Architecture *compatible* without being implemented with Fabric or TurboModule primitives at all.

This case study tracks two snapshots:

- **v4.7.2** (2025-09-02), the last release of the v4 line, contemporary with the original drafting of this book. The deep-dive below is grounded in this version.[196]
- **v5.0.0** (2026-04-16), a full rewrite onto Nitro Modules.[197] The 5.x line has moved on since. The current release is **v5.2.4** (2026-10-29), which pins `nitrogen` and `react-native-nitro-modules` to 0.37.0 and lists `react-native-nitro-modules` as an optional peer dependency.[198] Chapter 14 covers Nitro in depth, so here we only note what changed and why.

What "New Architecture support" actually means in v4

The v4 line is the version everyone migrating to Fabric and Bridgeless had to make peace with for most of 2025. It is worth being precise about how it works, because the line between "supports the New Architecture" and "is implemented as a New Architecture module" is exactly the line that confuses most consumers.

- **View Layer (legacy Paper view, Fabric-compatible via interop).** The `<Camera>` component is exported with `requireNativeComponent('CameraView')`, the legacy native component API.[199] On iOS, `CameraViewManager` extends `RCTViewManager` and props are wired with `RCT_EXPORT_VIEW_PROPERTY` macros.[200] On Android, `CameraViewManager` extends

[196] Vision Camera package metadata at the v4.7.2 tag: `tmp/react-native-vision-camera/package/package.json` (`"version": "4.7.2"`). v4.7.2 was tagged on 2025-09-02 (commit `82148d10`). The repo was restructured into a monorepo at v5.0.0 (PR #3735, commit `30a8b5de`, 2026-04-16), which deleted the `package/` directory and moved the library into `packages/react-native-vision-camera/`. All v4 paths cited below resolve under `git show v4.7.2:<path>` even though they no longer exist at the clone's current `main` HEAD.

[197] V5 release notes: https://github.com/mrousavy/react-native-vision-camera/releases/tag/v5.0.0 ("Fully rewritten to Nitro Modules"; "~3,000 lines of hand-written JSI/C++ deleted"). V5 docs: https://visioncamera.margelo.com. The "~3,000" figure checks out: summing pure-deletion `.cpp/.h/.hpp/.mm` files in the V5 PR diff (`git show 30a8b5de --stat`) gives 2,970 lines.

[198] `tmp/react-native-vision-camera/packages/react-native-vision-camera/package.json` (`"version": "5.2.4"`, `nitrogen` and `react-native-nitro-modules` devDependencies `0.37.0`, optional `react-native-nitro-modules` peer dependency). Tag `v5.2.4` is commit `cd95a93a`, 2026-10-29. Nitro bump history since v5.0.10: 0.35.9 (`49d94e7f`), 0.36.1 (`8004ceec`), 0.36.3 to 0.36.5, then 0.37 (`a4acf4d4`, 2026-08-20). The Nitro clone `tmp/nitro` has stable tag `v0.37.1` (2026-08-27) and `v1.0.0-alpha` nightlies on main. The `react-native-vision-camera-worklets` package pins `react-native-worklets` 0.9.1 as a devDependency.

[199] `tmp/react-native-vision-camera/package/src/NativeCameraView.ts:64` (`requireNativeComponent<NativeCameraViewProps>('CameraView')`). There is no `codegenNativeComponent` spec anywhere under `package/src/`.

[200] `tmp/react-native-vision-camera/package/ios/React/CameraViewManager.swift:14` (`final class CameraViewManager: RCTViewManager`). Props are listed in `ios/React/CameraViewManager.m` via `RCT_EXPORT_VIEW_PROPERTY` macros.

`ViewGroupManager<CameraView>` and uses `@ReactProp` setters.[201] None of this is generated from a CodeGen spec. When the host app runs the New Architecture, the **Paper-on-Fabric interop layer** wraps these legacy managers so the view still renders inside the Fabric tree.[202]

- **Module Layer (legacy bridge module, Fabric-compatible via interop).** The imperative APIs (`takePhoto`, `startRecording`, camera permissions) are exposed by `CameraViewModule`, which extends `ReactContextBaseJavaModule` on Android with `@ReactMethod` annotations.[203] The iOS Swift class is registered through `RCT_EXTERN_REMAP_MODULE(CameraView, CameraViewManager, RCTViewManager)`.[204] The JS side reaches the module via `NativeModules.CameraView`, not a CodeGen-generated spec.[205] The source even contains explicit TODOs waiting for TurboModules to land before the API can be cleaned up.[206] These too are routed through the **TurboModule interop layer** when the host app is on Bridgeless.
- **Frame Processor (true JSI, custom worklet runtime).** This is the only piece that is genuinely native to the JSI world. A JSI binding installs `global.VisionCameraProxy` and a Camera Frame is exposed to JS as a `jsi::HostObject`. The worklet runtime is a separate JS runtime running on a dedicated camera thread, not the React JS thread.

So the right one-liner is: in v4, Vision Camera *runs under* Fabric and Bridgeless because RN's interop layer wraps legacy modules and view managers. It is not itself Fabric or TurboModule code.

Status as of RN 0.88: that interop path is no longer an open-ended compatibility promise. Legacy module and view-manager shims are compile-gated behind `RCT_REMOVE_LEGACY_MODULE_INTEROP`

---

[201] `tmp/react-native-vision-camera/package/android/src/main/java/com/mrousavy/camera/react/CameraViewManager.kt:19` (`class CameraViewManager : ViewGroupManager<CameraView>()`). Props are declared with `@ReactProp`.

[202] RN provides automatic interop wrappers so that legacy view managers and bridge modules keep working when the host app enables Fabric and Bridgeless — **until those wrappers are compiled out**. The view-side wrapper lives under `packages/react-native/ReactCommon/react/renderer/components/legacyviewmanagerinterop/` and is gated by `RCT_REMOVE_LEGACY_COMPONENT_INTEROP`; the module-side wrapper is `packages/react-native/ReactCommon/react/nativemodule/core/platform/ios/ReactCommon/RCTInteropTurboModule.{h,mm}`, gated by `RCT_REMOVE_LEGACY_MODULE_INTEROP` (CHANGELOG v0.87.0: `9885fe95ef`, `3278f309e2`). The iOS opt-in is in CHANGELOG v0.88.0 (`b933d18177`, "Both default to off and will become the default in a future release"). The CocoaPods switch is `ENV['RCT_REMOVE_LEGACY_MODULE_INTEROP']` in `packages/react-native/scripts/react_native_pods.rb:123`.

[203] `tmp/react-native-vision-camera/package/android/src/main/java/com/mrousavy/camera/react/CameraViewModule.kt:39` (`class CameraViewModule(reactContext: ReactApplicationContext) : ReactContextBaseJavaModule(reactContext)`). Methods are annotated `@ReactMethod`, the legacy bridge contract.

[204] `tmp/react-native-vision-camera/package/ios/React/CameraViewManager.m:14` (`RCT_EXTERN_REMAP_MODULE(CameraView, CameraViewManager, RCTViewManager)`). The Swift class is registered as a legacy bridge module/view-manager hybrid, not a TurboModule.

[205] `tmp/react-native-vision-camera/package/src/NativeCameraModule.ts:8` (`export const CameraModule = NativeModules.CameraView`). `NativeModules` is the legacy bridge accessor, not `TurboModuleRegistry.get`.

[206] For example: `tmp/react-native-vision-camera/package/android/src/main/java/com/mrousavy/camera/react/CameraViewModule.kt:117` (`// TODO: ... Hopefully TurboModules allows that`) and `package/ios/React/CameraViewManager.swift:43` (`// Wait for TurboModules?`). Seven such TODO/"wait for TurboModules" comments exist across v4.7.2's `package/` tree (`grep -niE 'TODO.*TurboModules?|Wait for TurboModules|Hopefully TurboModules'`): three in Kotlin (`CameraView.kt:36`, `CameraViewManager.kt:158`, `CameraViewManager.kt:166`, plus the Module one above), two in Swift (`CameraView.swift:16`, `CameraViewManager.swift:43`), and one in TSX (`Camera.tsx:229`). Two further notes in `src/index.ts:34` and `src/frame-processors/VisionCameraProxy.ts:91` reference "CxxTurboModule" as the eventual replacement.

and `RCT_REMOVE_LEGACY_COMPONENT_INTEROP` — builds that define those macros drop the wrappers entirely. Both macros default to off and iOS apps can opt in from CocoaPods and SwiftPM. The changelog says they will become the default in a future release.[207] A library that only "supports New Arch via interop" is on a deletion track; the durable path is a real TurboModule / Fabric component (or, as v5 did, Nitro).

Advanced JSI: The Frame Processor

The most distinctive feature of Vision Camera is its **Frame Processor**: a JavaScript function called for every frame of video, on a dedicated camera thread, with zero serialization. This is the use case that pushes JSI hard, and the reason the library matters as a case study even when the rest of it is legacy.

Here is how it works end to end.

1. The JavaScript Worklet

A developer provides a Frame Processor through the `useFrameProcessor` hook.[208] The provided function must be a **`'worklet'`**.

```
const frameProcessor = useFrameProcessor((frame) => {
  'worklet'
  // This code runs on a dedicated camera thread
  const faces = scanFaces(frame) // e.g., using a face detection library
  console.log(`Detected ${faces.length} faces.`)
}, [])
```

The `'worklet'` directive is picked up by the **`react-native-worklets-core/plugin`** Babel plugin, which Vision Camera's docs require you to add to `babel.config.js`.[209] That plugin rewrites the function so it can be serialized and re-installed in a separate JSI runtime. Vision Camera does not use the Reanimated Babel plugin for this. Reanimated and `react-native-worklets-core` both implement the worklet concept but are distinct packages with their own runtimes. The two-

[207] RN provides automatic interop wrappers so that legacy view managers and bridge modules keep working when the host app enables Fabric and Bridgeless — **until those wrappers are compiled out**. The view-side wrapper lives under `packages/react-native/ReactCommon/react/renderer/components/legacyviewmanagerinterop/` and is gated by `RCT_REMOVE_LEGACY_COMPONENT_INTEROP`; the module-side wrapper is `packages/react-native/ReactCommon/react/nativemodule/core/platform/ios/ReactCommon/RCTInteropTurboModule.{h,mm}`, gated by `RCT_REMOVE_LEGACY_MODULE_INTEROP` (CHANGELOG v0.87.0: `9885fe95ef`, `3278f309e2`). The iOS opt-in is in CHANGELOG v0.88.0 (`b933d18177`, "Both default to off and will become the default in a future release"). The CocoaPods switch is `ENV['RCT_REMOVE_LEGACY_MODULE_INTEROP']` in `packages/react-native/scripts/react_native_pods.rb:123`.

[208] `tmp/react-native-vision-camera/package/src/hooks/useFrameProcessor.ts:42` (`export function useFrameProcessor(frameProcessor: (frame: Frame) => void, dependencies: DependencyList): ReadonlyFrameProcessor`).

[209] Official Vision Camera docs: `tmp/react-native-vision-camera/docs/docs/guides/FRAME_PROCESSORS.mdx:48-63` ("Frame Processors require `react-native-worklets-core` 1.0.0 or higher... add the plugin to your `babel.config.js`: `['react-native-worklets-core/plugin']`"). The docs site sits at the repo root, not under `package/`. At runtime, `tmp/react-native-vision-camera/package/src/frame-processors/VisionCameraProxy.ts:60` lazy-loads the package (`require('react-native-worklets-core')`). The v4 example app (`tmp/react-native-vision-camera/example/babel.config.js`) also registers `react-native-reanimated/plugin` for unrelated Reanimated animations; that plugin is not what compiles VC's frame-processor worklets.

runtime model is what lets the camera thread call user JS without going through the React JS thread at all.

(V5 switches the default worklets runtime to `react-native-worklets` from Software Mansion.[210] The user-facing `'worklet'` directive is unchanged.)

2. The `Frame` Host Object

The `frame` object passed to the worklet is not a plain JavaScript object. It is a **`jsi::HostObject`**. The C++ header on iOS confirms this:[211]

```
// From Vision Camera's iOS FrameHostObject.h
class JSI_EXPORT FrameHostObject : public jsi::HostObject, public
std::enable_shared_from_this<FrameHostObject> {
public:
  explicit FrameHostObject(Frame* frame) : _frame(frame), _baseClass(nullptr) {}

public:
  jsi::Value get(jsi::Runtime&, const jsi::PropNameID& name) override;
  std::vector<jsi::PropNameID> getPropertyNames(jsi::Runtime& rt) override;
  // ...
private:
  Frame* _frame;
  std::unique_ptr<jsi::Object> _baseClass;
};
```

The base contract here is RN's own. In `packages/react-native/ReactCommon/jsi/jsi/jsi.h`, `class HostObject` declares `virtual Value get(Runtime&, const PropNameID& name)`, and that is what Vision Camera overrides.[212]

The Android header has the same JSI base class but a different wrapper, because the underlying frame data is a Java object reached through fbjni:[213]

```
// From Vision Camera's Android FrameHostObject.h
class JSI_EXPORT FrameHostObject : public jsi::HostObject, public
std::enable_shared_from_this<FrameHostObject> {
public:
  explicit FrameHostObject(const jni::alias_ref<JFrame::javaobject>& frame);
  ~FrameHostObject();

public:
```

[210] V5 release notes (same link as): "Default Worklets runtime switched ... to `react-native-worklets` (Software Mansion) instead of `react-native-worklets-core`. You can now mutate Reanimated SharedValues directly from a Frame Processor."

[211] `tmp/react-native-vision-camera/package/ios/FrameProcessors/FrameHostObject.h:19-35`.

[212] `packages/react-native/ReactCommon/jsi/jsi/jsi.h:222` (class declaration), line 239 (`virtual Value get(Runtime&, const PropNameID& name)`).

[213] `tmp/react-native-vision-camera/package/android/src/main/cpp/frameprocessors/FrameHostObject.h:20-37`. Constructor body is on `FrameHostObject.cpp:22` (`FrameHostObject::FrameHostObject(... frame) : _frame(make_global(frame)), _baseClass(nullptr) {}`).

```
  jsi::Value get(jsi::Runtime&, const jsi::PropNameID& name) override;
  std::vector<jsi::PropNameID> getPropertyNames(jsi::Runtime& rt) override;
  // ...
private:
  jni::global_ref<JFrame> _frame;
  std::unique_ptr<jsi::Object> _baseClass;
};
```

On iOS the wrapped type is an Objective-C `Frame` around a `CMSampleBuffer`. On Android it is a `jni::global_ref` to a Java `JFrame` that owns the `android.media.Image` and any `HardwareBuffer`. Either way, the JavaScript side only ever sees a `HostObject` proxy.

3. The Synchronous, Off-Thread Loop

The complete flow:

1. The native camera hardware captures a frame of video.
2. Vision Camera's native code creates a `Frame` (iOS Obj-C class or Android Java class) holding the buffer.
3. It then creates a `std::shared_ptr<FrameHostObject>` that wraps the `Frame`.
4. On the dedicated camera queue (on iOS, `CameraQueues.videoQueue`, a `userInteractive`-QoS `DispatchQueue`),[214] the `AVCaptureVideoDataOutputSampleBufferDelegate` callback fires and invokes the user's worklet, passing the `FrameHostObject` as an argument.[215]
5. When the JavaScript code accesses a property like `frame.width`, JSI synchronously calls `get()` on the C++ `FrameHostObject`, which reads the width from the underlying `Frame` and returns it as a `jsi::Value`.

The whole pipeline happens for every frame, with no JSON serialization and no involvement from the main React JS thread. That direct, synchronous JSI access is what makes real-time per-frame work viable.

What v5 changed (and why this case study still matters)

V5 (April 2026) is a full rewrite onto **Nitro Modules**, the library author's own framework that competes with TurboModules.[216] Roughly 3,000 lines of hand-written JSI/C++ went away, and the public surface moved to Nitro-generated `HybridObject` bindings. The Frame Processor pipeline is now plain Swift and Kotlin sitting behind Nitro specs, and the `FrameHostObject` class no longer exists in the v5 source tree.[217] The repo also restructured into a monorepo with worklets split into

[214] `tmp/react-native-vision-camera/package/ios/Core/CameraQueues.swift:21-25` (declares `videoQueue`); `ios/Core/CameraSession+Configuration.swift:111` (`videoOutput.setSampleBufferDelegate(self, queue: CameraQueues.videoQueue)`).

[215] `tmp/react-native-vision-camera/package/ios/Core/CameraSession.swift:268-294` (`captureOutput(_:didOutput:from:)` → `onVideoFrame(...)` → `delegate.onFrame(...)`).

[216] V5 release notes: https://github.com/mrousavy/react-native-vision-camera/releases/tag/v5.0.0 ("Fully rewritten to Nitro Modules"; "~3,000 lines of hand-written JSI/C++ deleted"). V5 docs: https://visioncamera.margelo.com. The "~3,000" figure checks out: summing pure-deletion `.cpp/.h/.hpp/.mm` files in the V5 PR diff (`git show 30a8b5de --stat`) gives 2,970 lines.

[217] V5 `Frame` is a Nitro `HybridObject` spec, not a `jsi::HostObject`: `tmp/react-native-vision-camera/packages/react-native-vision-camera/src/specs/instances/Frame.nitro.ts:101` (`interface Frame extends HybridObject<{ ios: 'swift'; android: 'kotlin' }> { readonly width: number;`

a separate npm package, `react-native-vision-camera-worklets`, that peer-depends on `react-native-worklets` (Software Mansion).[218]

For this book, the v4 case study is still the more instructive one. It shows how a mature library can ship a New-Arch-compatible release without rewriting its module and view layers, while still exploiting JSI for the one piece (`Frame`) where the performance budget actually demands it. Chapter 14 picks up where this leaves off and covers Nitro itself.

Conclusion

`react-native-vision-camera` demonstrates two things at once. First, that "supports the New Architecture" can historically mean nothing more than "runs under the interop layer" — v4 shipped that way successfully, but on RN 0.88 those shims are compile-gated for removal, so interop-only libraries should treat migration as urgent rather than optional. Second, that JSI is the part of the New Architecture you actually feel as a user. A `jsi::HostObject` (v4) or Nitro `HybridObject` (v5) exposing a camera frame to a worklet, on a dedicated thread, is the kind of capability you cannot build on top of the legacy bridge no matter how hard you try. V5's Nitro rewrite is the surviving production path; the v4 interop story is the cautionary one.

Citations:

- [vc-v472] Vision Camera package metadata at the v4.7.2 tag: `tmp/react-native-vision-camera/package/package.json` (`"version": "4.7.2"`). v4.7.2 was tagged on 2025-09-02 (commit `82148d10`). The repo was restructured into a monorepo at v5.0.0 (PR #3735, commit `30a8b5de`, 2026-04-16),[219] which deleted the `package/` directory and moved the library into `packages/react-native-vision-camera/`. All v4 paths cited below resolve under `git show v4.7.2:<path>` even though they no longer exist at the clone's current `main` HEAD.
- [vc-v500] V5 release notes: https://github.com/mrousavy/react-native-vision-camera/releases/tag/v5.0.0 ("Fully rewritten to Nitro Modules"; "~3,000 lines of hand-written JSI/C++ deleted"). V5 docs: https://visioncamera.margelo.com. The "~3,000" figure checks out: summing pure-deletion `.cpp/.h/.hpp/.mm` files in the V5 PR diff (`git show 30a8b5de --stat`) gives 2,970 lines.
- [vc-v5pr] V5 PR (`feat: V5. (#3735)`): commit `30a8b5de43dd21ecbb5500d1b01903de3ab42f36`, dated 2026-04-16. The tag `v5.0.0` is one commit later (`ef61e1b7`, `chore: release 5.0.0`).

`readonly height: number; readonly pixelFormat: PixelFormat; ... getPlanes(): FramePlane[] (line 289); getPixelBuffer(): ArrayBuffer (line 68); getNativeBuffer(): NativeBuffer (line 377); ... }`). The Nitro tool generates the JSI and JNI bindings from this spec; no hand-written `HostObject` survives. `git grep -l 'FrameHostObject' HEAD` returns empty at the v5 working tree.

[218] `tmp/react-native-vision-camera/packages/react-native-vision-camera-worklets/package.json` (peerDeps include `"react-native-worklets": "*"`, not `react-native-worklets-core`). The v5 example babel config is `tmp/react-native-vision-camera/apps/simple-camera/babel.config.js:3` (`plugins: ['react-native-worklets/plugin']`). The main `react-native-vision-camera@5.x` package's peer deps are `react-native-nitro-modules` and `react-native-nitro-image`; worklets is opt-in via the sibling package.

[219] V5 PR (`feat: V5. (#3735)`): commit `30a8b5de43dd21ecbb5500d1b01903de3ab42f36`, dated 2026-04-16. The tag `v5.0.0` is one commit later (`ef61e1b7`, `chore: release 5.0.0`).

- [vc-nativeview] `tmp/react-native-vision-camera/package/src/NativeCameraView.ts:64` (`requireNativeComponent<NativeCameraViewProps>('CameraView')`). There is no `codegenNativeComponent` spec anywhere under `package/src/`.
- [vc-iosviewmanager] `tmp/react-native-vision-camera/package/ios/React/CameraViewManager.swift:14` (`final class CameraViewManager: RCTViewManager`). Props are listed in `ios/React/CameraViewManager.m` via `RCT_EXPORT_VIEW_PROPERTY` macros.
- [vc-androidviewmanager] `tmp/react-native-vision-camera/package/android/src/main/java/com/mrousavy/camera/react/CameraViewManager.kt:19` (`class CameraViewManager : ViewGroupManager<CameraView>()`). Props are declared with `@ReactProp`.
- [rn-interop] RN provides automatic interop wrappers so that legacy view managers and bridge modules keep working when the host app enables Fabric and Bridgeless — **until those wrappers are compiled out**. The view-side wrapper lives under `packages/react-native/ReactCommon/react/renderer/components/legacyviewmanagerinterop/` and is gated by `RCT_REMOVE_LEGACY_COMPONENT_INTEROP`; the module-side wrapper is `packages/react-native/ReactCommon/react/nativemodule/core/platform/ios/ReactCommon/RCTInteropTurboModule.{h,mm}`, gated by `RCT_REMOVE_LEGACY_MODULE_INTEROP` (CHANGELOG v0.87.0: `9885fe95ef`, `3278f309e2`). The iOS opt-in is in CHANGELOG v0.88.0 (`b933d18177`, "Both default to off and will become the default in a future release"). The CocoaPods switch is `ENV['RCT_REMOVE_LEGACY_MODULE_INTEROP']` in `packages/react-native/scripts/react_native_pods.rb:123`.
- [vc-androidmodule] `tmp/react-native-vision-camera/package/android/src/main/java/com/mrousavy/camera/react/CameraViewModule.kt:39` (`class CameraViewModule(reactContext: ReactApplicationContext) : ReactContextBaseJavaModule(reactContext)`). Methods are annotated `@ReactMethod`, the legacy bridge contract.
- [vc-iosmodulebridge] `tmp/react-native-vision-camera/package/ios/React/CameraViewManager.m:14` (`RCT_EXTERN_REMAP_MODULE(CameraView, CameraViewManager, RCTViewManager)`). The Swift class is registered as a legacy bridge module/view-manager hybrid, not a TurboModule.
- [vc-jsmodule] `tmp/react-native-vision-camera/package/src/NativeCameraModule.ts:8` (`export const CameraModule = NativeModules.CameraView`). `NativeModules` is the legacy bridge accessor, not `TurboModuleRegistry.get`.
- [vc-todo] For example: `tmp/react-native-vision-camera/package/android/src/main/java/com/mrousavy/camera/react/CameraViewModule.kt:117` (`// TODO: ... Hopefully TurboModules allows that`) and `package/ios/React/CameraViewManager.swift:43` (`// Wait for TurboModules?`). Seven such TODO/"wait for TurboModules" comments exist across v4.7.2's `package/` tree (`grep -niE 'TODO.*TurboModules?|Wait for TurboModules|Hopefully TurboModules'`): three in Kotlin (`CameraView.kt:36`, `CameraViewManager.kt:158`, `CameraViewManager.kt:166`, plus the Module one above), two in Swift (`CameraView.swift:16`, `CameraViewManager.swift:43`), and one in TSX (`Camera.tsx:229`). Two further notes in `src/index.ts:34` and `src/frame-processors/VisionCameraProxy.ts:91` reference "CxxTurboModule" as the eventual replacement.

- [vc-useFrameProcessor] `tmp/react-native-vision-camera/package/src/hooks/useFrameProcessor.ts:42` (`export function useFrameProcessor(frameProcessor: (frame: Frame) => void, dependencies: DependencyList): ReadonlyFrameProcessor`).
- [vc-workletsplugin] Official Vision Camera docs: `tmp/react-native-vision-camera/docs/docs/guides/FRAME_PROCESSORS.mdx:48-63` ("Frame Processors require `react-native-worklets-core` 1.0.0 or higher... add the plugin to your `babel.config.js`: `['react-native-worklets-core/plugin']`"). The docs site sits at the repo root, not under `package/`. At runtime, `tmp/react-native-vision-camera/package/src/frame-processors/VisionCameraProxy.ts:60` lazy-loads the package (`require('react-native-worklets-core')`). The v4 example app (`tmp/react-native-vision-camera/example/babel.config.js`) also registers `react-native-reanimated/plugin` for unrelated Reanimated animations; that plugin is not what compiles VC's frame-processor worklets.
- [vc-v5worklets] V5 release notes (same link as[220]): "Default Worklets runtime switched ... to `react-native-worklets` (Software Mansion) instead of `react-native-worklets-core`. You can now mutate Reanimated SharedValues directly from a Frame Processor."
- [vc-frameHostObjectIos] `tmp/react-native-vision-camera/package/ios/FrameProcessors/FrameHostObject.h:19-35`.
- [rn-jsiHostObject] `packages/react-native/ReactCommon/jsi/jsi/jsi.h:222` (class declaration), line 239 (`virtual Value get(Runtime&, const PropNameID& name)`).
- [vc-frameHostObjectAndroid] `tmp/react-native-vision-camera/package/android/src/main/cpp/frameprocessors/FrameHostObject.h:20-37`. Constructor body is on `FrameHostObject.cpp:22` (`FrameHostObject::FrameHostObject(... frame) : _frame(make_global(frame)), _baseClass(nullptr) {}`).
- [vc-videoQueue] `tmp/react-native-vision-camera/package/ios/Core/CameraQueues.swift:21-25` (declares `videoQueue`); `ios/Core/CameraSession+Configuration.swift:111` (`videoOutput.setSampleBufferDelegate(self, queue: CameraQueues.videoQueue)`).
- [vc-videoFrameCallback] `tmp/react-native-vision-camera/package/ios/Core/CameraSession.swift:268-294` (`captureOutput(_:didOutput:from:)` → `onVideoFrame(...)` → `delegate.onFrame(...)`).
- [vc-v5frame] V5 `Frame` is a Nitro `HybridObject` spec, not a `jsi::HostObject`: `tmp/react-native-vision-camera/packages/react-native-vision-camera/src/specs/instances/Frame.nitro.ts:101` (`interface Frame extends HybridObject<{ ios: 'swift'; android: 'kotlin' }> { readonly width: number; readonly height: number; readonly pixelFormat: PixelFormat; ... getPlanes(): FramePlane[] (line 289); getPixelBuffer(): ArrayBuffer (line 68); getNativeBuffer(): NativeBuffer (line 377); ... }`). The Nitro tool generates the JSI and JNI bindings from this spec; no hand-written `HostObject` survives. `git grep -l 'FrameHostObject' HEAD` returns empty at the v5 working tree.

[220] V5 release notes: https://github.com/mrousavy/react-native-vision-camera/releases/tag/v5.0.0 ("Fully rewritten to Nitro Modules"; "~3,000 lines of hand-written JSI/C++ deleted"). V5 docs: https://visioncamera.margelo.com. The "~3,000" figure checks out: summing pure-deletion `.cpp/.h/.hpp/.mm` files in the V5 PR diff (`git show 30a8b5de --stat`) gives 2,970 lines.

- [vc-v5worklets-pkg] `tmp/react-native-vision-camera/packages/react-native-vision-camera-worklets/package.json` (peerDeps include `"react-native-worklets": "*"`, not `react-native-worklets-core`). The v5 example babel config is `tmp/react-native-vision-camera/apps/simple-camera/babel.config.js:3` (`plugins: ['react-native-worklets/plugin']`). The main `react-native-vision-camera@5.x` package's peer deps are `react-native-nitro-modules` and `react-native-nitro-image`; worklets is opt-in via the sibling package.
- [vc-v524] `tmp/react-native-vision-camera/packages/react-native-vision-camera/package.json` (`"version": "5.2.4"`, `nitrogen` and `react-native-nitro-modules` devDependencies `0.37.0`, optional `react-native-nitro-modules` peer dependency). Tag `v5.2.4` is commit `cd95a93a`, 2026-10-29. Nitro bump history since v5.0.10: 0.35.9 (`49d94e7f`), 0.36.1 (`8004ceec`), 0.36.3 to 0.36.5, then 0.37 (`a4acf4d4`, 2026-08-20). The Nitro clone `tmp/nitro` has stable tag `v0.37.1` (2026-08-27) and `v1.0.0-alpha` nightlies on main. The `react-native-vision-camera-worklets` package pins `react-native-worklets` 0.9.1 as a devDependency.

Chapter 14: Alternative Tooling - Nitro

While the standard New Architecture tools (CodeGen, TurboModules, Fabric) provide the fundamental building blocks, the ecosystem is evolving with higher-level tooling designed to simplify the developer experience and push performance even further. One prominent example of this is **Nitro**.

Nitro is an opinionated, high-performance framework for building native modules that sits on top of the JSI, acting as an alternative and enhancement to the standard TurboModule workflow.[221] At the time of this audit, the current stable release is **Nitro v0.37.1** (27 August 2026). The Nitro `main` branch already publishes `v1.0.0-alpha.nightly.*` tags, so a 1.0 line is in preparation and the documented minimum React Native version is **0.75**.[222] (The package's own `peerDependencies` are wildcarded (`react-native: "*"`), so the 0.75 floor is enforced by the iOS/Android build requirements rather than by npm. Internal development of v0.37.1 tracks React Native 0.85.3 via the package's devDependencies.)

Status (2026): Nitro is production-ready and has broad adoption in the ecosystem. Libraries built on top of it include `react-native-vision-camera`, `react-native-mmkv`, `react-native-quick-crypto`, `@rive-app/react-native`, `react-native-video`, `react-native-fast-tflite`, `react-native-nitro-sqlite`, and dozens of others.[223]

Core Concepts of Nitro

Nitro's philosophy is centered around providing a truly object-oriented and type-safe bridge between JavaScript and native code.

The `HybridObject`

The central abstraction in Nitro is the `HybridObject`. While a standard TurboModule is effectively a singleton (the TurboModuleManager caches one instance per module name and keeps it for the lifetime of the runtime[224]), a `HybridObject` is a true, instantiable native object that can be created, passed around, and referenced from JavaScript.[225] This enables a more object-oriented programming model.

For example, instead of passing an ID or a file path to a native function, you can pass an actual native object instance:

```
// With a TurboModule, you might do this:
const processedImagePath = await ImageEditor.crop(imagePath, rect);
```

[221] Nitro docs: `docs/docs/getting-started/what-is-nitro.md`. https://nitro.margelo.com/docs/getting-started/what-is-nitro

[222] Nitro docs: `docs/docs/getting-started/minimum-requirements.md` (RN 0.75+, Xcode 16.4+, Swift 5.9+ on iOS; compileSdk 34+, ndk 27+ on Android).

[223] Nitro docs: `docs/docs/resources/awesome-nitro-modules.md` (curated list of community Nitro Modules).

[224] RN source: `packages/react-native/ReactCommon/react/nativemodule/core/platform/ios/ReactCommon/RCTTurboModuleManager.mm:202-203, 317-318` ("All TurboModule instances are cached, which means they're all long-lived") and `packages/react-native/ReactCommon/react/nativemodule/core/ReactCommon/TurboModuleBinding.cpp:185-186` ("TurboModules are cached by name in TurboModuleManagers").

[225] Nitro docs: `docs/docs/concepts/hybrid-objects.md` (`createHybridObject<T>`, lines 42-71).

```
// With Nitro's HybridObjects, you can do this:
const image = await ImageFactory.loadImage(imagePath);
const processedImage: Image = ImageEditor.crop(image, rect);
```

In practice, a JS consumer creates the root Hybrid Object via `NitroModules.createHybridObject<T>('Name')`, and a Hybrid Object method can return another Hybrid Object directly. The JS-side `image` variable holds a real reference to the underlying native object, so subsequent native calls can operate on it without re-serializing through a path or ID.[226] On the C++ side, the `HybridObjectRegistry` stores *constructor functions* keyed by name, not instances (`std::unordered_map<std::string, std::function<std::shared_ptr<HybridObject>()>>`), so `createHybridObject<T>(name)` calls the registered constructor each time and returns a fresh native object.[227]

Note that Nitro itself is bootstrapped *as* a TurboModule: `react-native-nitro-modules` ships a single TurboModule named `NitroModules` whose only method is `install()`, which installs the `NitroModulesProxy` Hybrid Object onto JS `global`. After that one-time install, all `NitroModules.createHybridObject<T>(...)` calls and every method on every Hybrid Object go through `global.NitroModulesProxy` directly via JSI, not through TurboModule machinery.[228] So Nitro doesn't replace TurboModules at the React Native loader level; it bootstraps via a TurboModule shim and then runs above JSI.

The `nitrogen` Code Generator

Nitro comes with its own code generator, `nitrogen`. It serves the same purpose as React Native's built-in `codegen` (reading a TypeScript spec and generating native interface code) but with a key philosophical difference:[229]

- **Standard `codegen`:** Runs as part of the **app's** build process. On iOS, `run_codegen!` is invoked from CocoaPods when the user runs `pod install`.[230] On Android, the Gradle plugin wires up `GenerateCodegenArtifactsTask` for both the app project and each library project.[231] The app rebuilds every library's generated interfaces from scratch.

---

[226] Nitro docs: `docs/docs/concepts/hybrid-objects.md` (`createHybridObject<T>`, lines 42-71).

[227] Nitro source: `tmp/nitro/packages/react-native-nitro-modules/cpp/registry/HybridObjectRegistry.hpp:20-46` (`using HybridObjectConstructorFn = std::function<std::shared_ptr<HybridObject>()>; static void registerHybridObjectConstructor(const std::string& hybridObjectName, HybridObjectConstructorFn&& constructorFn); static std::shared_ptr<HybridObject> createHybridObject(const std::string& hybridObjectName);`). The registry stores constructors, not instances.

[228] Nitro source: `tmp/nitro/packages/react-native-nitro-modules/src/turbomodule/NativeNitroModules.ts:1-56` (`import { TurboModuleRegistry } from 'react-native'; interface Spec extends TurboModule { install(): string | undefined }; turboModule = TurboModuleRegistry.getEnforcing<Spec>('NitroModules'); const errorMessage = turboModule.install(); ... nitroModules = getInstalledNitro()` where `getInstalledNitro` reads `global.NitroModulesProxy`). On old-arch iOS: `tmp/nitro/packages/react-native-nitro-modules/ios/turbomodule/NativeNitroModules+OldArch.mm:34` (`RCT_EXPORT_BLOCKING_SYNCHRONOUS_METHOD(install)`).

[229] Nitro docs: `docs/docs/concepts/nitrogen.md` (lines 43-47: "Nitrogen should be used by library-authors").

[230] RN source: `packages/react-native/scripts/react_native_pods.rb:223` (`run_codegen!` invoked from the Podfile install).

- **`nitrogen`:** Is designed to be run by the **library author**. The generated C++, Swift, and Kotlin interface code is committed to the library's repository and published with the npm package, so the consumer of the library never runs nitrogen themselves.[232]

A practical consequence: Nitrogen can also resolve imports across files (so a spec can `import type` from a sibling), whereas standard codegen cannot.[233]

What nitrogen actually emits, concretely: an 8-line `Math.nitro.ts` with one `readonly pi: number` getter and one `add(a: number, b: number): number` method, configured for `{ ios: 'swift', android: 'kotlin' }`, produces 21 files across three platforms in `nitrogen/generated/`.[234] The shared `HybridMathSpec.cpp` registers the prototype:[235]

```
void HybridMathSpec::loadHybridMethods() {
  HybridObject::loadHybridMethods();
  registerHybrids(this, [](Prototype& prototype) {
    prototype.registerHybridGetter("pi", &HybridMathSpec::getPi);
    prototype.registerHybridMethod("add", &HybridMathSpec::add);
  });
}
```

The Swift side comes in three pieces (`HybridMathSpec_protocol`, `HybridMathSpec_base`, and a `HybridMathSpec_cxx` bridge wrapper, glued together with a public typealias `HybridMathSpec = HybridMathSpec_protocol & HybridMathSpec_base`), and the Kotlin side is an abstract class annotated with `@DoNotStrip` / `@Keep` plus a JNI `HybridData CxxPart`. Each Hybrid Object also needs an `autolinking` entry in `nitro.json` that pins the implementation class name per platform, otherwise the generated `*Autolinking.swift`/`*OnLoad.kt` won't include it and the runtime registry can't find the constructor at startup.[236]

---

[231] RN source: `packages/gradle-plugin/react-native-gradle-plugin/src/main/kotlin/com/facebook/react/ReactPlugin.kt:111-120` (`configureCodegen(... isLibrary = false)` for the app, `isLibrary = true` for each library project).

[232] Nitro docs: `docs/docs/concepts/nitrogen.md` (lines 43-47: "Nitrogen should be used by library-authors").

[233] Nitro docs: `docs/docs/resources/comparison.md` (benchmark table at lines 13-38, type-support matrix at lines 574-703, HostObject vs NativeState discussion at lines 255-260).

[234] Reproducible: `_verification/recheck/nitro/tests/toy-spec/` contains `Math.nitro.ts`, `nitro.json`, and `nitrogen/generated/` with 21 files (2 shared C++, 2 iOS C++ bridge, 2 Swift, 6 iOS autolinking/umbrella, 2 Android C++ JNI, 2 Kotlin, 4 Android autolinking/loader, 1 `.gitattributes`). Nitrogen 0.35.7 produced these in 0.9s (see `run.log` next to the spec). A re-run with nitrogen 0.37.1 on 2026-10-29 gave the same 21 files and the same prototype registration code.

[235] Nitro source: `tmp/nitro/packages/react-native-nitro-modules/cpp/prototype/HybridObjectPrototype.hpp:26-30` ("The prototype should be cached per Runtime, and can be assigned to multiple jsi::Objects. When assigned to a jsi::Object, all methods of this prototype can be called on that jsi::Object, as long as it has a valid NativeState (`this`)."). Generated call sites: `_verification/recheck/nitro/tests/toy-spec/nitrogen/generated/shared/c++/HybridMathSpec.cpp:12-19` (`prototype.registerHybridGetter("pi", ...)`; `prototype.registerHybridMethod("add", ...)`).

[236] Nitro source: `tmp/nitro/packages/react-native-nitro-test/nitro.json` is the canonical example. Each Hybrid Object has an `autolinking` entry mapping `ios.implementationClassName` and `android.implementationClassName` to the concrete native classes that satisfy the generated spec.

Key Differentiators and Performance Claims

Nitro makes several claims about its advantages over the standard TurboModule system, backed by technical choices and published benchmarks.

1. Performance

Nitro's documentation publishes a microbenchmark that measures the total execution time of calling a single native method 100,000 times in a tight loop:[237]

Operation	ExpoModules	TurboModules	NitroModules
`addNumbers(...)` × 100,000	434.85 ms	115.86 ms	**7.27 ms**
`addStrings(...)` × 100,000	429.53 ms	179.02 ms	**29.94 ms**

That puts Nitro at roughly **16x faster than TurboModules on `addNumbers` and ~6x faster on `addStrings`** in the synthetic benchmark.[238] Nitro's own docs are explicit that the numbers reflect "extreme cases" and "do not necessarily reflect real world use-cases". For typical app workloads the gap is much smaller.[239] The benchmark itself is reproducible inside Nitro's own example app: `example/src/screens/BenchmarksScreen.tsx` runs `addNumbers(...)` 100,000 times against both an `ExampleTurboModule` and `HybridTestObjectSwiftKotlin` with a single warmup call and `performance.now()` around the loop.[240] Nitro attributes the difference to two architectural decisions:

- **`jsi::NativeState` vs. `jsi::HostObject`:** Standard TurboModules are implemented using `jsi::HostObject`. The RN source confirms this directly: `class JSI_EXPORT TurboModule : public jsi::HostObject` (TurboModule.h:49).[241] Nitro's `HybridObjects` are built on top of `jsi::NativeState` (jsi.h:255).[242] With `HostObject`, the *first* access to each property goes through the virtual `get()` callback in C++. TurboModule's `get()` does add post-first-call caching by writing the resolved method onto the JS representation via `setProperty` (TurboModule.h:55-65),[243] but the cost of the first call and the indirection through `MethodMetadata.invoker` are still there. With `NativeState`, nitrogen-gener-

[237] Nitro docs: `docs/docs/resources/comparison.md` (benchmark table at lines 13-38, type-support matrix at lines 574-703, HostObject vs NativeState discussion at lines 255-260).

[238] Nitro docs: `docs/docs/resources/comparison.md` (benchmark table at lines 13-38, type-support matrix at lines 574-703, HostObject vs NativeState discussion at lines 255-260).

[239] Nitro docs: `docs/docs/resources/comparison.md` (benchmark table at lines 13-38, type-support matrix at lines 574-703, HostObject vs NativeState discussion at lines 255-260).

[240] Nitro source: `tmp/nitro/example/src/screens/BenchmarksScreen.tsx:42-65` (`function benchmark(obj: BenchmarkableObject): number { obj.addNumbers(0, 3); const start = performance.now(); let num = 0; for (let i = 0; i < ITERATIONS; i++) { num = obj.addNumbers(num, 3); } const end = performance.now(); return end - start; }` with `ITERATIONS = 100_000`). The same shape is published at `mrousavy/NitroBenchmarks` for cross-environment runs.

[241] RN source: `packages/react-native/ReactCommon/react/nativemodule/core/ReactCommon/TurboModule.h:49` (`class JSI_EXPORT TurboModule : public jsi::HostObject`).

[242] RN source: `packages/react-native/ReactCommon/jsi/jsi/jsi.h:255-260` (definition of `class JSI_EXPORT NativeState`) and `:1389-1404` (`hasNativeState` / `getNativeState` / `setNativeState` on `jsi::Object`).

[243] RN source: `packages/react-native/ReactCommon/react/nativemodule/core/ReactCommon/TurboModule.h:50-65` (`get(...)` calls `create(...)` and, on hit, runs `jsRepresentation_-`

ated C++ installs concrete getters and methods on a shared `Prototype` once per type (`prototype.registerHybridGetter(...)`, `prototype.registerHybridMethod(...)`),[244] before any JS call. The prototype is cached per `jsi::Runtime` and assigned to every Hybrid Object instance, so JS-engine inline caches can lock onto the slot the first time a property is read. Nitro pairs this with `Object::setExternalMemoryPressure` (jsi.h:1430) to inform the GC about the native side's memory footprint, helping the collector free unused Hybrid Objects more aggressively.[245]

- **Direct Swift & C++ Interop:** For iOS, Nitro requires Swift 5.9 or higher and uses the official Swift <> C++ interop.[246] This lets the C++ JSI layer call Swift code directly. The call path is `JS → C++ → Swift`, whereas TurboModules go `JS → C++ → Objective-C → Swift` when the underlying implementation is Swift.[247]

2. Richer Type Support

Nitro's tooling supports a wider range of types than standard codegen. The differences that show up in Nitro's published comparison table (and that the RN codegen source confirms it lacks) are:[248]

- **Tuples** (`[A, B, C, ...]`): Nitro supports them. RN codegen does not (the codebase carries an explicit TODO for `TupleTypeAnnotation` support[249]).
- **Variants of distinct types** (`A | B | Person`): Nitro supports them via its variants machinery. RN codegen's `emitUnion` only accepts unions of primitives (`string`, `number`, `boolean`, and their literal forms, plus a single `ObjectTypeAnnotation` or alias),[250] so an arbitrary cross-type union is rejected.
- **Callbacks that return values** (`(T...) => R`): Nitro supports both `Promise<T>` (async, the default) and `Sync<T>` (synchronous) callback return types.[251] In RN codegen, callbacks

---

`>lock(runtime).asObject(runtime).setProperty(runtime, propName, prop)`. First access goes through the virtual `get()`; subsequent accesses hit the cached JS property).

[244] Nitro source: `tmp/nitro/packages/react-native-nitro-modules/cpp/prototype/HybridObjectPrototype.hpp:26-30` ("The prototype should be cached per Runtime, and can be assigned to multiple jsi::Objects. When assigned to a jsi::Object, all methods of this prototype can be called on that jsi::Object, as long as it has a valid NativeState (`this`)."). Generated call sites: `_verification/recheck/nitro/tests/toy-spec/nitrogen/generated/shared/c++/HybridMathSpec.cpp:12-19` (`prototype.registerHybridGetter("pi", ...)`; `prototype.registerHybridMethod("add", ...)`).

[245] Nitro docs: `docs/docs/resources/comparison.md` (benchmark table at lines 13-38, type-support matrix at lines 574-703, HostObject vs NativeState discussion at lines 255-260).

[246] Nitro docs: `docs/docs/getting-started/minimum-requirements.md` (RN 0.75+, Xcode 16.4+, Swift 5.9+ on iOS; compileSdk 34+, ndk 27+ on Android).

[247] Nitro docs: `docs/docs/resources/comparison.md` (benchmark table at lines 13-38, type-support matrix at lines 574-703, HostObject vs NativeState discussion at lines 255-260).

[248] Nitro docs: `docs/docs/resources/comparison.md` (benchmark table at lines 13-38, type-support matrix at lines 574-703, HostObject vs NativeState discussion at lines 255-260).

[249] RN source: `packages/react-native-codegen/src/parsers/error-utils.js:282` (open TODO: "Added as a work-around for now until TupleTypeAnnotation are fully supported in both flow and TS").

[250] RN source: `packages/react-native-codegen/src/parsers/parsers-primitives.js:438-490` (`emitUnion` accepts only `StringTypeAnnotation`, `NumberTypeAnnotation`, `BooleanTypeAnnotation`, their literal forms, `ObjectTypeAnnotation`, and `TypeAliasTypeAnnotation`).

[251] Nitro docs: `docs/docs/types/callbacks.md:144-167` (`Sync<T>` for synchronous return values; default callbacks return `Promise<T>`).

are restricted: the sample TurboModule spec only declares `(callback: (value: string) => void) => void`,[252] and `throwIfUnsupportedFunctionReturnTypeAnnotationParserError` rejects function-typed return values outside the cxx-only path.[253]

- **`null` as a return type**, **`Int64`/`UInt64`**, **`Record<string, T>`**: still Nitro-only; RN codegen does not emit them.[254]
- **`ArrayBuffer`**: Nitro supports it across its Swift/Kotlin/C++ targets. As of RN 0.87, core codegen supports `ArrayBuffer` for **C++ and ObjC** TurboModules (`ArrayBufferTypeAnnotation` → `jsi::ArrayBuffer` / `ArrayBufferKind`). As of RN 0.88, **Java** TurboModules support it too (`com.facebook.react.bridge.ArrayBuffer`). Core codegen still rejects `Promise<ArrayBuffer>` on every platform: an `ArrayBuffer` is allowed as a method argument and as a synchronous return value only.[255] The remaining Nitro differentiators are therefore one spec that works identically on every Nitro target, and `ArrayBuffer` in async results. It is not true that "RN has no ArrayBuffer."

For literal-string enums like `type MediaType = 'image' | 'video' | 'audio'`, Nitro's nitrogen actually maps them to native enums backed by compile-time string hashes (not variants), and the union must be aliased to a named type for nitrogen to generate code for it.[256]

Latest Nitro Features (as of v0.37.1, August 2026)

Type-system surface

The "type-safety" benefits in Nitro come from nitrogen's TypeScript-to-native code generation: if a spec declares `add(a: number, b: number): number`, the Swift / Kotlin / C++ implementation must use the exact corresponding types or the project will not compile.[257] In practice this means:

```
// String-literal union (must be a named alias so nitrogen can emit a native
enum)
type MediaType = 'image' | 'video' | 'audio';

// Variant of distinct types (each branch is a real type, not a literal)
type ImageOrCount = HybridImage | number;

// Sync callback with a return value (default callbacks return Promise<T>)
```

[252] RN source: `packages/rn-tester/js/examples/TurboModule/NativeSampleTurboModule.js:57` (`getValueWithCallback: (callback: (value: string) => void) => void`).

[253] RN source: `packages/react-native-codegen/src/parsers/error-utils.js:128-142` (`throwIfUnsupportedFunctionReturnTypeAnnotationParserError`).

[254] Nitro docs: `docs/docs/resources/comparison.md` (benchmark table at lines 13-38, type-support matrix at lines 574-703, HostObject vs NativeState discussion at lines 255-260).

[255] RN 0.87: C++/ObjC support via `GenerateModuleH.js` / `GenerateModuleObjCpp/serializeMethod.js` and `ReactCommon/react/bridging/ArrayBuffer.h` (commits `226ef2e7c5`, `54b88d9dea`). RN 0.88: Java support (commit `5fb3ebce1ac`, in `v0.88.0`), see `GenerateModuleJavaSpec.js:381` and `GenerateModuleJniCpp.js:224`. `Promise<ArrayBuffer>` is rejected by `throwIfUnsupportedPromiseArrayBuffer` in `generators/modules/Utils.js:95`. See Chapter 4.

[256] Nitro docs: `docs/docs/types/custom-enums.md` ("TypeScript union" section: a union of literal strings is mapped to a native enum backed by compile-time string hashes, and the union must be aliased to a named type).

[257] Nitro docs: `docs/docs/getting-started/what-is-nitro.md`. https://nitro.margelo.com/docs/getting-started/what-is-nitro

```
interface Server extends HybridObject<{ ios: 'swift' }> {
  fetchWithFilter(filter: Sync<(item: Item) => boolean>): Item[];
}
```

Note that nitrogen does **not** support user-defined generic interfaces as Hybrid Object specs. The generic parameter on `HybridObject<{ ... }>` is reserved for the `PlatformSpec`, which has shape `{ ios?: 'c++' | 'swift'; android?: 'c++' | 'kotlin' }` (both keys optional, so a spec can target only iOS or only Android).[258] A spec like `interface Repository<T> extends HybridObject<{ ... }>` parses fine on the TypeScript side, and nitrogen even recognizes it as a Hybrid Object, but it fails at code generation because `T` has no native representation. Running nitrogen 0.37.1 against that spec emits:

```
⏳ Parsing Repository.nitro.ts...
   ⚙  Generating specs for HybridObject "Repository"...
      ❌ Failed to generate spec for Repository! Error: The TypeScript type "T" cannot be represented in C++!
   ❌ No specs found in Repository.nitro.ts!
```

(Reproducible run is in `_verification/recheck/nitro/tests/toy-spec/`. The 0.37.1 re-run on 2026-10-29 gave the same error text.) The only generics in nitrogen's surface area are the built-in ones (`Promise<T>`, `Sync<T>`, `Array<T>`, `Record<string, T>`, `Variant<...>`, etc.).

Developer experience

- **Compile-time type safety across the bridge.** Because every Hybrid Object spec is the single source of truth, a wrong type or missing method on the native side breaks the build, not the runtime.
- **TypeScript IntelliSense in the editor.** Generated `*.d.ts` declarations give the consumer auto-complete and type-check via the standard TypeScript Language Server. This is not a Nitro-specific feature (TurboModule specs are also TypeScript and benefit from the same tooling), but it is a real consequence of nitrogen shipping generated specs.
- **JS code reload still works as usual.** React Native's Fast Refresh applies to the JS layer that calls a Hybrid Object. Changes to the **native** implementation, however, still require a full app rebuild, just as with TurboModules.

Conclusion on Nitro

Nitro is not a replacement for the New Architecture. It is an opinionated layer on top of JSI that replaces the **TurboModule slot specifically**. Fabric, the renderer, the runtime, the runtime scheduler, the event loop, and the rest of the New Architecture stack are untouched. A Nitro Module sits next to TurboModules in the same app and uses the same JSI runtime, the same `CallInvoker`, and the same Hermes (or JSC) instance.

[258] Nitro source: `tmp/nitro/packages/react-native-nitro-modules/src/HybridObject.ts:7-33` (`interface PlatformSpec { ios?: 'c++' | 'swift'; android?: 'c++' | 'kotlin' }`, both keys optional, used as the constraint on `HybridObject<Platforms extends PlatformSpec>`) and `tmp/nitro/packages/nitrogen/src/getPlatformSpecs.ts:169-180` (nitrogen reads `base.getTypeArguments()[0]` as the platform spec). The C++ runtime side at `tmp/nitro/packages/react-native-nitro-modules/cpp/core/HybridObject.hpp:27` is a non-template class (`class HybridObject : public virtual jsi::NativeState, public HybridObjectPrototype, ...`), confirming that the only "generic" surface is the TS-side `Platforms` argument.

For developers seeking very high throughput on JS-native calls, an object-oriented surface for native instances, or a Swift-first iOS development workflow, Nitro is a credible alternative to the standard TurboModule creation process. For most app code, where the JS-native call rate is far below 100,000 calls per second, the architectural ergonomics (Hybrid Objects, typed callbacks with return values, generated specs shipped in npm packages) are usually a stronger reason to adopt Nitro than the raw microbenchmark numbers.

Citations:

- [1] Nitro docs: `docs/docs/getting-started/what-is-nitro.md`. https://nitro.margelo.com/docs/getting-started/what-is-nitro
- [2] Nitro docs: `docs/docs/concepts/hybrid-objects.md` (`createHybridObject<T>`, lines 42-71).
- [3] Nitro docs: `docs/docs/concepts/nitrogen.md` (lines 43-47: "Nitrogen should be used by library-authors").
- [4] Nitro docs: `docs/docs/resources/comparison.md` (benchmark table at lines 13-38, type-support matrix at lines 574-703, HostObject vs NativeState discussion at lines 255-260).
- [5] Nitro docs: `docs/docs/getting-started/what-is-nitro.md#uses-jsinativestate` (Docusaurus auto-slug).
- [6] Nitro docs: `docs/docs/types/custom-types.md`.
- [7] Nitro docs: `docs/docs/getting-started/minimum-requirements.md` (RN 0.75+, Xcode 16.4+, Swift 5.9+ on iOS; compileSdk 34+, ndk 27+ on Android).
- [8] Nitro docs: `docs/docs/resources/awesome-nitro-modules.md` (curated list of community Nitro Modules).
- [9] RN source: `packages/react-native/ReactCommon/react/nativemodule/core/platform/ios/ReactCommon/RCTTurboModuleManager.mm:202-203, 317-318` ("All TurboModule instances are cached, which means they're all long-lived") and `packages/react-native/ReactCommon/react/nativemodule/core/ReactCommon/TurboModuleBinding.cpp:185-186` ("TurboModules are cached by name in TurboModuleManagers").
- [10] RN source: `packages/react-native/scripts/react_native_pods.rb:223` (`run_codegen!` invoked from the Podfile install).
- [11] RN source: `packages/gradle-plugin/react-native-gradle-plugin/src/main/kotlin/com/facebook/react/ReactPlugin.kt:111-120` (`configureCodegen(... isLibrary = false)` for the app, `isLibrary = true` for each library project).
- [12] RN source: `packages/react-native/ReactCommon/react/nativemodule/core/ReactCommon/TurboModule.h:49` (`class JSI_EXPORT TurboModule : public jsi::HostObject`).
- [13] RN source: `packages/react-native/ReactCommon/jsi/jsi/jsi.h:255-260` (definition of `class JSI_EXPORT NativeState`) and `:1389-1404` (`hasNativeState` / `getNativeState` / `setNativeState` on `jsi::Object`).

- [14] RN source: `packages/react-native-codegen/src/parsers/error-utils.js:282` (open TODO: "Added as a work-around for now until TupleTypeAnnotation are fully supported in both flow and TS").
- [15] RN source: `packages/react-native-codegen/src/parsers/parsers-primitives.js:438-490` (`emitUnion` accepts only `StringTypeAnnotation`, `NumberTypeAnnotation`, `BooleanTypeAnnotation`, their literal forms, `ObjectTypeAnnotation`, and `TypeAliasTypeAnnotation`).
- [16] Nitro docs: `docs/docs/types/callbacks.md:144-167` (`Sync<T>` for synchronous return values; default callbacks return `Promise<T>`).
- [17] RN source: `packages/rn-tester/js/examples/TurboModule/NativeSampleTurboModule.js:57` (`getValueWithCallback: (callback: (value: string) => void) => void`).
- [18] RN source: `packages/react-native-codegen/src/parsers/error-utils.js:128-142` (`throwIfUnsupportedFunctionReturnTypeAnnotationParserError`).
- [19] Nitro docs: `docs/docs/types/custom-enums.md` ("TypeScript union" section: a union of literal strings is mapped to a native enum backed by compile-time string hashes, and the union must be aliased to a named type).
- [20] Nitro source: `tmp/nitro/packages/react-native-nitro-modules/src/HybridObject.ts:7-33` (`interface PlatformSpec { ios?: 'c++' | 'swift'; android?: 'c++' | 'kotlin' }`, both keys optional, used as the constraint on `HybridObject<Platforms extends PlatformSpec>`) and `tmp/nitro/packages/nitrogen/src/getPlatformSpecs.ts:169-180` (nitrogen reads `base.getTypeArguments()[0]` as the platform spec). The C++ runtime side at `tmp/nitro/packages/react-native-nitro-modules/cpp/core/HybridObject.hpp:27` is a non-template class (`class HybridObject : public virtual jsi::NativeState, public HybridObjectPrototype, ...`), confirming that the only "generic" surface is the TS-side `Platforms` argument.
- [21] RN source: `packages/react-native/ReactCommon/react/nativemodule/core/ReactCommon/TurboModule.h:50-65` (`get(...)` calls `create(...)` and, on hit, runs `jsRepresentation_->lock(runtime).asObject(runtime).setProperty(runtime, propName, prop)`. First access goes through the virtual `get()`; subsequent accesses hit the cached JS property).
- [22] Nitro source: `tmp/nitro/packages/react-native-nitro-modules/cpp/prototype/HybridObjectPrototype.hpp:26-30` ("The prototype should be cached per Runtime, and can be assigned to multiple jsi::Objects. When assigned to a jsi::Object, all methods of this prototype can be called on that jsi::Object, as long as it has a valid NativeState (`this`)."). Generated call sites: `_verification/recheck/nitro/tests/toy-spec/nitrogen/generated/shared/c++/HybridMathSpec.cpp:12-19` (`prototype.registerHybridGetter("pi", ...)`; `prototype.registerHybridMethod("add", ...)`).
- [23] Nitro source: `tmp/nitro/packages/react-native-nitro-modules/cpp/registry/HybridObjectRegistry.hpp:20-46` (`using HybridObjectConstructorFn = std::function<std::shared_ptr<HybridObject>()>; static void registerHybridObjectConstructor(const std::string& hybridObjectName, HybridObjectConstructorFn&& constructorFn); static std::shared_ptr<HybridObject>

`createHybridObject(const std::string& hybridObjectName);`). The registry stores constructors, not instances.

- [24] Nitro source: `tmp/nitro/packages/react-native-nitro-modules/src/turbomodule/NativeNitroModules.ts:1-56` (`import { TurboModuleRegistry } from 'react-native'; interface Spec extends TurboModule { install(): string | undefined }; turboModule = TurboModuleRegistry.getEnforcing<Spec>('NitroModules'); const errorMessage = turboModule.install(); ... nitroModules = getInstalledNitro()` where `getInstalledNitro` reads `global.NitroModulesProxy`). On old-arch iOS: `tmp/nitro/packages/react-native-nitro-modules/ios/turbomodule/NativeNitroModules+OldArch.mm:34` (`RCT_EXPORT_BLOCKING_SYNCHRONOUS_METHOD(install)`).
- [25] Reproducible: `_verification/recheck/nitro/tests/toy-spec/` contains `Math.nitro.ts`, `nitro.json`, and `nitrogen/generated/` with 21 files (2 shared C++, 2 iOS C++ bridge, 2 Swift, 6 iOS autolinking/umbrella, 2 Android C++ JNI, 2 Kotlin, 4 Android autolinking/loader, 1 `.gitattributes`). Nitrogen 0.35.7 produced these in 0.9s (see `run.log` next to the spec). A re-run with nitrogen 0.37.1 on 2026-10-29 gave the same 21 files and the same prototype registration code.
- [26] Nitro source: `tmp/nitro/packages/react-native-nitro-test/nitro.json` is the canonical example. Each Hybrid Object has an `autolinking` entry mapping `ios.implementationClassName` and `android.implementationClassName` to the concrete native classes that satisfy the generated spec.
- [27] Nitro source: `tmp/nitro/example/src/screens/BenchmarksScreen.tsx:42-65` (`function benchmark(obj: BenchmarkableObject): number { obj.addNumbers(0, 3); const start = performance.now(); let num = 0; for (let i = 0; i < ITERATIONS; i++) { num = obj.addNumbers(num, 3); } const end = performance.now(); return end - start; }` with `ITERATIONS = 100_000`). The same shape is published at `mrousavy/NitroBenchmarks` for cross-environment runs.
- [28] RN 0.87: C++/ObjC support via `GenerateModuleH.js` / `GenerateModuleObjCpp/serializeMethod.js` and `ReactCommon/react/bridging/ArrayBuffer.h` (commits `226ef2e7c5`, `54b88d9dea`). RN 0.88: Java support (commit `5fb3ebce1ac`, in `v0.88.0`), see `GenerateModuleJavaSpec.js:381` and `GenerateModuleJniCpp.js:224`. `Promise<ArrayBuffer>` is rejected by `throwIfUnsupportedPromiseArrayBuffer` in `generators/modules/Utils.js:95`. See Chapter 4.

Chapter 15: Tutorial - Building a Native Module and Component

> **Audit note (RN 0.88):** the corrected code below is grounded in current React Native source (`9c7b0f5fafd` era, v0.88.1 stable). Prefer public `'react-native'` imports — under the Strict TypeScript API (default since 0.87.0-rc.1), deep `Libraries/...` imports are restricted. Inline footnotes point at the exact files.

This chapter provides a hands-on, step-by-step tutorial to create a simple React Native library that uses both a TurboModule and a Fabric Component. This will demonstrate how the concepts discussed in previous chapters come together in practice.

Our library, `react-native-deviceinfo`, will provide:

- A **TurboModule** called `DeviceInfoModule` to get the device's model name.
- A **Fabric Component** called `DeviceInfoView` that displays a label passed via a prop.

Part 1: Project Setup

First, we need a standard file structure for a native library. You can create this manually or use a community tool like `create-react-native-library`.

The essential structure will be:

```
react-native-deviceinfo/
├── android/
├── ios/
├── src/
└── package.json
```

Next, configure `package.json` to tell React Native about your native module and, crucially, to configure CodeGen.[259]

`package.json`:

```
{
  "name": "react-native-deviceinfo",
  "version": "1.0.0",
  "main": "src/index.ts",
  "react-native": "src/index.ts",
  "files": [
    "src",
    "android",
    "ios",
    "react-native-deviceinfo.podspec"
```

[259] `codegenConfig` schema (fields, `type` values, `jsSrcsDir`): see `packages/react-native/scripts/codegen/generate-artifacts-executor/index.js` and the working fixture at `packages/react-native/scripts/codegen/__fixtures__/test-app/lib/test-library/package.json`. The `type` field accepts `"all" | "modules" | "components"` (see `generateRCTModuleProviders.js:59` (`config.type === 'components'`) and `generateRCTThirdPartyComponents.js:52` (`config.type === 'modules'`)). External docs: "Using Codegen", reactnative.dev/docs/next/the-new-architecture/using-codegen. `podspecName` is not a standard field in any RN source path under `packages/`.

```
  ],
  "codegenConfig": {
    "name": "DeviceInfoSpec",
    "type": "all",
    "jsSrcsDir": "src/specs",
    "android": {
      "javaPackageName": "com.reactnativedeviceinfo"
    },
    "ios": {
      "modulesProvider": {
        "DeviceInfoModule": "DeviceInfoModule"
      },
      "componentProvider": {
        "DeviceInfoView": "DeviceInfoView"
      }
    }
  }
}
```

- **`codegenConfig`**: This is the most important part. It tells CodeGen to look for spec files in the `src/specs` directory. The `name` field (`DeviceInfoSpec`) is the **library name** that namespaces the generated artifacts on iOS (and shows up in generated include paths like `<react/renderer/components/DeviceInfoSpec/Props.h>`). The `type` field accepts `"all" | "modules" | "components"`.[260]
- **iOS podspec discovery**: there is no `podspecName` top-level field in React Native's `package.json` schema.[261] The podspec is discovered by `use_native_modules!` in the host app's `Podfile` based on the `.podspec` file you ship in the package root (here, `react-native-deviceinfo.podspec`). The basename of that file becomes the pod's name.

Part 2: Building the TurboModule

Our TurboModule will expose one asynchronous method, `getDeviceName()`.

Step 2.1: The JavaScript Spec (TypeScript)

This is the **source of truth**. Create a file at `src/specs/NativeDeviceInfoModule.ts`. The file name's basename (without the `.ts` extension) is what codegen calls the **haste module name**;

[260] `codegenConfig` schema (fields, `type` values, `jsSrcsDir`): see `packages/react-native/scripts/codegen/generate-artifacts-executor/index.js` and the working fixture at `packages/react-native/scripts/codegen/__fixtures__/test-app/lib/test-library/package.json`. The `type` field accepts `"all" | "modules" | "components"` (see `generateRCTModuleProviders.js:59` (`config.type === 'components'`) and `generateRCTThirdPartyComponents.js:52` (`config.type === 'modules'`)). External docs: "Using Codegen", reactnative.dev/docs/next/the-new-architecture/using-codegen. `podspecName` is not a standard field in any RN source path under `packages/`.

[261] `codegenConfig` schema (fields, `type` values, `jsSrcsDir`): see `packages/react-native/scripts/codegen/generate-artifacts-executor/index.js` and the working fixture at `packages/react-native/scripts/codegen/__fixtures__/test-app/lib/test-library/package.json`. The `type` field accepts `"all" | "modules" | "components"` (see `generateRCTModuleProviders.js:59` (`config.type === 'components'`) and `generateRCTThirdPartyComponents.js:52` (`config.type === 'modules'`)). External docs: "Using Codegen", reactnative.dev/docs/next/the-new-architecture/using-codegen. `podspecName` is not a standard field in any RN source path under `packages/`.

it ends up in the generated artifact names on both platforms (for example, the Android abstract class is named `NativeDeviceInfoModuleSpec`).[262]

src/specs/NativeDeviceInfoModule.ts:

```
import type { TurboModule } from 'react-native';
import { TurboModuleRegistry } from 'react-native';

export interface Spec extends TurboModule {
  // A promise-based async method
  getDeviceName(): Promise<string>;
}

// 'DeviceInfoModule' is the name our native code will use to register itself.
export default TurboModuleRegistry.getEnforcing<Spec>('DeviceInfoModule');
```

Step 2.2: The iOS Implementation (Objective-C++)

On iOS, the implementation requires an `.mm` file. This is **Objective-C++**, a language that can understand both the C++ generated by CodeGen and the Objective-C of the native iOS frameworks (like `UIDevice`).[263]

The conventional shape in the React Native source is to keep the public header minimal (declaring `<RCTBridgeModule>` conformance only) and to declare the generated spec protocol conformance in a class extension inside the `.mm` file. That keeps the spec name out of the public header surface.[264]

ios/DeviceInfoModule.h:

```
#import <Foundation/Foundation.h>
#import <React/RCTBridgeModule.h>

@interface DeviceInfoModule : NSObject <RCTBridgeModule>
@end
```

ios/DeviceInfoModule.mm:

```
#import "DeviceInfoModule.h"
#import <UIKit/UIKit.h>
```

[262] TurboModule type and registry: `packages/react-native/Libraries/TurboModule/RCTExport.{js,d.ts}` (defines `export interface TurboModule`) and `packages/react-native/Libraries/TurboModule/TurboModuleRegistry.{js,d.ts}` (defines `getEnforcing<T extends TurboModule>(name): T`). Canonical spec example: `packages/rn-tester/js/examples/TurboModule/NativeSampleTurboModule.js`. Haste-module-name derivation is the basename of the spec file. External docs: "Turbo Native Modules", reactnative.dev/docs/next/turbo-native-modules-introduction.

[263] TurboModule type and registry: `packages/react-native/Libraries/TurboModule/RCTExport.{js,d.ts}` (defines `export interface TurboModule`) and `packages/react-native/Libraries/TurboModule/TurboModuleRegistry.{js,d.ts}` (defines `getEnforcing<T extends TurboModule>(name): T`). Canonical spec example: `packages/rn-tester/js/examples/TurboModule/NativeSampleTurboModule.js`. Haste-module-name derivation is the basename of the spec file. External docs: "Turbo Native Modules", reactnative.dev/docs/next/turbo-native-modules-introduction.

[264] `<RCTBridgeModule>`-only public header pattern: `packages/react-native/Libraries/Settings/RCTSettingsManager.h:12` (public) and `RCTSettingsManager.mm:17` (`@interface RCTSettingsManager () <NativeSettingsManagerSpec>` in a class extension).

```
// Import the header generated by CodeGen. The library name comes from
// codegenConfig.name ("DeviceInfoSpec"). The build system maps this to a
// header search path during the codegen step.
#import <DeviceInfoSpec/DeviceInfoSpec.h>

@interface DeviceInfoModule () <NativeDeviceInfoModuleSpec>
@end

@implementation DeviceInfoModule

RCT_EXPORT_MODULE()

// Hand back the JSI-callable wrapper. The generated SpecJSI class is what
// actually bridges JS calls into the Objective-C method below.
- (std::shared_ptr<facebook::react::TurboModule>)getTurboModule:
    (const facebook::react::ObjCTurboModule::InitParams &)params
{
    return
std::make_shared<facebook::react::NativeDeviceInfoModuleSpecJSI>(params);
}

// The codegen-generated protocol declares the Promise-returning method with
// RCTPromiseResolveBlock / RCTPromiseRejectBlock, NOT C++ Promise types.
//   typedef void (^RCTPromiseResolveBlock)(id _Nullable result);
//   typedef void (^RCTPromiseRejectBlock)(NSString *_Nullable code,
//                                         NSString *_Nullable message,
//                                         NSError *_Nullable error);
- (void)getDeviceName:(RCTPromiseResolveBlock)resolve
               reject:(RCTPromiseRejectBlock)reject
{
    @try {
        NSString *deviceName = [[UIDevice currentDevice] name];
        resolve(deviceName);
    } @catch (NSException *exception) {
        reject(@"E_DEVICE_NAME", exception.reason ?: @"Unknown error", nil);
    }
}

@end

// Module registration on iOS is *not* via a `<Name>Cls(void)` C function —
// that pattern is for Fabric *components* only (see Part 3). TurboModules
// are discovered through `RCT_EXPORT_MODULE()` and/or
// `codegenConfig.ios.modulesProvider` in package.json (CodeGen emits the
// module-provider map that maps the JS name to your class).
```

A few corrections worth flagging if you cross-reference older tutorials:

- The resolve/reject parameters are `RCTPromiseResolveBlock` and `RCTPromiseRejectBlock` (typedef'd `id`-and-error blocks in `React/RCTBridgeModule.h`), not C++ `facebook::react::Promise<T>`.[265] The C++ bridging side does have an `AsyncPromise<T>` class (`react/bridging/Promise.h`), but it is used internally by the generated wrapper and never appears in the protocol your class implements.
- `RCTPromiseRejectBlock` takes three arguments (`code`, `message`, `error`), not a single exception.[266]
- `RCT_EXPORT_MODULE()` is what registers the class under the JS-visible name (`"DeviceInfoModule"` by default, matching the symbol after `RCT_EXPORT_MODULE` strips the `RCT` prefix). Without it the TurboModule manager has no way to find your class.

About `[[UIDevice currentDevice] name]`: starting with iOS 16, this returns a generic model string (e.g. `"iPhone"`) for any app that does not have the special `UIDevice.name` user-name entitlement. If you actually want the user-customized name, the entitlement is the only path; otherwise switch to a model identifier like `[[UIDevice currentDevice] model]`.

Why not pure Swift? While the logic could be in Swift, you still need this `.mm` file to bridge between the C++ world of JSI and the Swift world. The common pattern is to have the `.mm` file call a pure Swift class. Swift's C++ interop is opt-in and still maturing; until your team is ready to depend on it, the `.mm` → `.swift` indirection is the safe path.

Step 2.3: The Android Implementation (Kotlin)

On Android, we'll use Kotlin. The class must extend the generated abstract Spec class.

Naming gotcha. The generated abstract class is **`Native<JsFilename>Spec`**, where the JS filename is the haste basename. Our spec file is `NativeDeviceInfoModule.ts`, so the generated abstract class is `NativeDeviceInfoModuleSpec` (not `DeviceInfoSpec`, which is the codegenConfig library `name`).[267] The generated class also already declares `public static final String NAME =`

[265] Promise codegen for ObjC++: `packages/react-native-codegen/src/generators/modules/GenerateModuleObjCpp/serializeMethod.js:120-121` pushes `{paramName: 'resolve', objCType: 'RCTPromiseResolveBlock'}` and `{paramName: 'reject', objCType: 'RCTPromiseRejectBlock'}` when the return type is a `PromiseTypeAnnotation`. The typedefs live in `packages/react-native/React/Base/RCTBridgeModule.h:45` (`typedef void (^RCTPromiseResolveBlock)(id _Nullable result);`) and `:52` (`typedef void (^RCTPromiseRejectBlock)(NSString *_Nullable code, NSString *_Nullable message, NSError *_Nullable error);`). The codegen snapshot at `packages/react-native-codegen/src/generators/modules/__tests__/__snapshots__/GenerateModuleHObjCpp-test.js.snap:882-886` shows the generated `@protocol` form.

[266] Promise codegen for ObjC++: `packages/react-native-codegen/src/generators/modules/GenerateModuleObjCpp/serializeMethod.js:120-121` pushes `{paramName: 'resolve', objCType: 'RCTPromiseResolveBlock'}` and `{paramName: 'reject', objCType: 'RCTPromiseRejectBlock'}` when the return type is a `PromiseTypeAnnotation`. The typedefs live in `packages/react-native/React/Base/RCTBridgeModule.h:45` (`typedef void (^RCTPromiseResolveBlock)(id _Nullable result);`) and `:52` (`typedef void (^RCTPromiseRejectBlock)(NSString *_Nullable code, NSString *_Nullable message, NSError *_Nullable error);`). The codegen snapshot at `packages/react-native-codegen/src/generators/modules/__tests__/__snapshots__/GenerateModuleHObjCpp-test.js.snap:882-886` shows the generated `@protocol` form.

[267] Generated Android spec class name: `packages/react-native-codegen/src/generators/modules/GenerateModuleJavaSpec.js:580` (`const className = \${hasteModuleName}Spec`;`). The generated class also already provides `getName()` returning a static `NAME` constant, confirmed by the

`"DeviceInfoModule"` and a `getName()` that returns it, so you do not need to override either of them in the subclass.[268]

`android/src/main/java/com/reactnativedeviceinfo/DeviceInfoModule.kt`:

```
package com.reactnativedeviceinfo

import android.os.Build
import com.facebook.react.bridge.Promise
import com.facebook.react.bridge.ReactApplicationContext
import com.facebook.react.module.annotations.ReactModule

// NativeDeviceInfoModuleSpec is generated under
codegenConfig.android.javaPackageName,
// which we set to com.reactnativedeviceinfo above, so it lives in this same
package
// and no explicit import is needed.

@ReactModule(name = NativeDeviceInfoModuleSpec.NAME)
class DeviceInfoModule(context: ReactApplicationContext)
    : NativeDeviceInfoModuleSpec(context) {

    // No getName() override needed: NativeDeviceInfoModuleSpec already
    // returns its NAME constant.
    // No @ReactMethod needed on the override: the abstract method is already
    // annotated with @ReactMethod @DoNotStrip in the generated parent class.
    override fun getDeviceName(promise: Promise) {
        promise.resolve(Build.MODEL)
    }

    companion object {
        // Re-expose NAME on the subclass so DeviceInfoPackage can reference it
        // as DeviceInfoModule.NAME (Java statics declared on the generated
parent
        // are not auto-inherited under Kotlin's companion-style access).
        // `val`, not `const val`: Kotlin requires `const val` initializers to be
        // literals or other `const val`s, not Java `static final` references.
        @JvmField val NAME: String = NativeDeviceInfoModuleSpec.NAME
    }
}
```

You also need a `Package` file to register the module with React Native.

---

snapshot atpackages/react-native-codegen/src/generators/modules/**tests**/**snapshots**/GenerateModuleJavaSpec-test.js.snap:26-35`.

[268] Generated Android spec class name: packages/react-native-codegen/src/generators/modules/GenerateModuleJavaSpec.js:580 (const className = \${hasteModuleName}Spec`;). The generated class also already provides getName()returning a staticNAMEconstant, confirmed by the snapshot atpackages/react-native-codegen/src/generators/modules/**tests**/**snapshots**/GenerateModuleJavaSpec-test.js.snap:26-35`.

android/src/main/java/com/reactnativedeviceinfo/DeviceInfoPackage.kt:

```
package com.reactnativedeviceinfo

import com.facebook.react.BaseReactPackage
import com.facebook.react.bridge.NativeModule
import com.facebook.react.bridge.ReactApplicationContext
import com.facebook.react.module.model.ReactModuleInfo
import com.facebook.react.module.model.ReactModuleInfoProvider

// BaseReactPackage replaced TurboReactPackage in the v0.76 cycle;
TurboReactPackage
// is now @Deprecated and exists only as a thin alias.
class DeviceInfoPackage : BaseReactPackage() {

    override fun getModule(name: String, reactContext: ReactApplicationContext):
NativeModule? =
        if (name == DeviceInfoModule.NAME) DeviceInfoModule(reactContext) else
null

    override fun getReactModuleInfoProvider(): ReactModuleInfoProvider =
        ReactModuleInfoProvider {
            mapOf(
                DeviceInfoModule.NAME to ReactModuleInfo(
                    DeviceInfoModule.NAME,                  // name
                    DeviceInfoModule::class.java.name,      // className (FQN of
impl)
                    false,                                  //
canOverrideExistingModule
                    false,                                  // needsEagerInit
                    false,                                  // isCxxModule
                    true                                    // isTurboModule
                )
            )
        }
}
```

Two corrections worth calling out:

- The recommended base class is `BaseReactPackage`. `TurboReactPackage` is `@Deprecated` since the 0.76 cycle and only kept as a no-op subclass for source compatibility.[269]
- `ReactModuleInfo`'s primary constructor has **six** parameters today (no `hasConstants`). The older seven-arg overload is `@Deprecated` and `@Suppress("UNUSED_PARAMETER")`-marks `hasConstants`, so it compiles but warns and the value is ignored.[270]

[269] `TurboReactPackage` deprecation: `packages/react-native/ReactAndroid/src/main/java/com/facebook/react/TurboReactPackage.kt:10-14` (`@Deprecated(message = "Use BaseReactPackage instead", replaceWith = ReplaceWith(expression = "BaseReactPackage"))`). The replacement landed in the v0.76 cycle: see `CHANGELOG-0.7x.md:1262` ("turbomodules: Replace TurboReactPackage with BaseReactPackage", commit `e881a1184c`). RN itself uses `BaseReactPackage` (`packages/react-native/ReactAndroid/src/main/java/com/facebook/react/runtime/CoreReactPackage.kt:48`).

Part 3: Building the Fabric Component

Our Fabric component will be a simple native `View` that displays a text label.

Step 3.1: The JavaScript Spec (TypeScript)

Create a file at `src/specs/DeviceInfoViewNativeComponent.ts`.[271]

`src/specs/DeviceInfoViewNativeComponent.ts`:

```
import type { ViewProps } from 'react-native';
import { codegenNativeComponent } from 'react-native';

// Define the props that our native view will accept
export interface NativeProps extends ViewProps {
  label: string;
}

// 'DeviceInfoView' is the name our native code will use
export default codegenNativeComponent<NativeProps>('DeviceInfoView');
```

Step 3.2: The iOS Implementation

In the New Architecture, a Fabric component on iOS is a subclass of `RCTViewComponentView`.[272] You do **not** need an `RCTViewManager` for a Fabric-only library. The legacy `RCTViewManager` pattern only matters for the Paper renderer; in the RN source you can see this directly because the legacy view managers were guarded by `#ifndef RCT_REMOVE_LEGACY_ARCH` and, since 0.87, the iOS files that carried that guard (for example `Libraries/Image/RCTImageViewManager.mm`) are deleted from the source tree.[273] Fabric discovers third-party component classes through a codegen-

[270] `ReactModuleInfo` constructor evolution: `packages/react-native/ReactAndroid/src/main/java/com/facebook/react/module/model/ReactModuleInfo.kt:16-23` (current 6-arg primary ctor: name, className, canOverrideExistingModule, needsEagerInit, isCxxModule, isTurboModule). The 7-arg overload at `:24-42` is `@Deprecated`, and the `hasConstants` parameter is `@Suppress("UNUSED_PARAMETER")`-marked so it is silently ignored.

[271] `<RCTBridgeModule>`-only public header pattern: `packages/react-native/Libraries/Settings/RCTSettingsManager.h:12` (public) and `RCTSettingsManager.mm:17` (`@interface RCTSettingsManager () <NativeSettingsManagerSpec>` in a class extension).

[272] `RCTViewComponentView`'s declaration and `updateProps:oldProps:` signature: `packages/react-native/React/Fabric/Mounting/ComponentViews/View/RCTViewComponentView.h:22` (`@interface RCTViewComponentView : UIView <RCTComponentViewProtocol, RCTTouchableComponentViewProtocol>`) and `:67-68` (`- (void)updateProps:(const facebook::react::Props::Shared &)props oldProps:(const facebook::react::Props::Shared &)oldProps NS_REQUIRES_SUPER;`). `Props::Shared` is defined as `std::shared_ptr<const Props>` at `packages/react-native/ReactCommon/react/renderer/core/Props.h:31`.

[273] Fabric iOS component discovery: the legacy `RCTViewManager` base class still ships in `packages/react-native/React/Views/RCTViewManager.h`, but the guarded `RCTImageViewManager.mm` was removed in 0.87 (commit `86350ab9884`, "Remove legacy architecture code guarded by RCT_REMOVE_LEGACY_ARCH from RN iOS"). Fabric discovery happens through the plugin map at `packages/react-native/React/Fabric/Mounting/ComponentViews/RCTFabricComponentsPlugins.mm:19-43`, which maps a component name to a `<Name>Cls(void)` C function. Real-world example: `packages/react-native/React/Fabric/Mounting/ComponentViews/View/RCTViewComponentView.mm:2059` (`Class<RCTComponentViewProtocol> RCTViewCls(void)` returning `RCTViewComponentView.class`). The `+ componentDescriptorProvider` class method is the pattern used by every built-in Fabric view (e.g. `packages/react-native/React/Fabric/Mounting/ComponentViews/Switch/RCTSwitchComponentView.mm:65`). External docs: "Fabric Native Components", reactnative.dev/docs/next/fabric-native-components-introduction.

generated plugin map (`RCTFabricComponentsProvider` in `RCTFabricComponentsPlugins.mm`) that maps a component name string to a C function returning the class.[274] Two things you must therefore provide on the view:

1. A class method `+ (ComponentDescriptorProvider)componentDescriptorProvider` returning `concreteComponentDescriptorProvider<DeviceInfoViewComponentDescriptor>()`. Without it, Fabric does not know which `ComponentDescriptor` to use for layout/shadow tree construction.
2. A C function named `<ComponentName>Cls(void)` that returns the view's class. The generated plugin map calls it during component registration.

ios/DeviceInfoView.h:

```
#import <React/RCTViewComponentView.h>

@interface DeviceInfoView : RCTViewComponentView
@end
```

ios/DeviceInfoView.mm:

```
#import "DeviceInfoView.h"

#import <React/RCTConversions.h>
#import <react/renderer/components/DeviceInfoSpec/ComponentDescriptors.h>
#import <react/renderer/components/DeviceInfoSpec/Props.h>
#import <react/renderer/components/DeviceInfoSpec/RCTComponentViewHelpers.h>
#import <react/renderer/components/DeviceInfoSpec/ShadowNodes.h>

using namespace facebook::react;

@implementation DeviceInfoView {
    UILabel *_labelView;
}

- (instancetype)initWithFrame:(CGRect)frame
{
    if (self = [super initWithFrame:frame]) {
        // Seed _props with the codegen-generated default values so the first
        // updateProps has a sane "old" baseline. The base RCTViewComponentView
```

[274] Fabric iOS component discovery: the legacy `RCTViewManager` base class still ships in `packages/react-native/React/Views/RCTViewManager.h`, but the guarded `RCTImageViewManager.mm` was removed in 0.87 (commit `86350ab9884`, "Remove legacy architecture code guarded by RCT_REMOVE_LEGACY_ARCH from RN iOS"). Fabric discovery happens through the plugin map at `packages/react-native/React/Fabric/Mounting/ComponentViews/RCTFabricComponentsPlugins.mm:19-43`, which maps a component name to a `<Name>Cls(void)` C function. Real-world example: `packages/react-native/React/Fabric/Mounting/ComponentViews/View/RCTViewComponentView.mm:2059` (`Class<RCTComponentViewProtocol> RCTViewCls(void)` returning `RCTViewComponentView.class`). The `+ componentDescriptorProvider` class method is the pattern used by every built-in Fabric view (e.g. `packages/react-native/React/Fabric/Mounting/ComponentViews/Switch/RCTSwitchComponentView.mm:65`). External docs: "Fabric Native Components", reactnative.dev/docs/next/fabric-native-components-introduction.

```
        // declares _props as a protected ivar of type SharedViewProps. Every
        // generated ShadowNode exposes a static defaultSharedProps() for this
        // purpose (compare RCTSwitchComponentView.mm:45, etc.).
        _props = DeviceInfoViewShadowNode::defaultSharedProps();

        _labelView = [UILabel new];
        [self addSubview:_labelView];
    }
    return self;
}

// Tell Fabric which ComponentDescriptor describes this view's shadow node +
props.
+ (ComponentDescriptorProvider)componentDescriptorProvider
{
    return
concreteComponentDescriptorProvider<DeviceInfoViewComponentDescriptor>();
}

// Fabric calls this with const-pointer-to-const Props pairs. Props::Shared is
// `using Shared = std::shared_ptr<const Props>` in react/renderer/core/Props.h.
// The house style in RN's built-in component views is to reference-cast
// `*shared_ptr` rather than `*std::static_pointer_cast` (no control-block
churn).
- (void)updateProps:(const Props::Shared &)props oldProps:(const Props::Shared
&)oldProps
{
    const auto &oldViewProps = static_cast<const DeviceInfoViewProps &>(*_props);
    const auto &newViewProps = static_cast<const DeviceInfoViewProps &>(*props);

    if (oldViewProps.label != newViewProps.label) {
        // RCTNSStringFromString is the canonical helper in RCTConversions.h;
        // it handles empty / UTF-8 edge cases better than
+stringWithUTF8String:.
        _labelView.text = RCTNSStringFromString(newViewProps.label);
        [self setNeedsLayout];
    }

    [super updateProps:props oldProps:oldProps];
}

- (void)layoutSubviews
{
    [super layoutSubviews];
    _labelView.frame = self.bounds;
}

@end
```

```
// C registration function picked up by the codegen-generated plugin map.
Class<RCTComponentViewProtocol> DeviceInfoViewCls(void)
{
    return DeviceInfoView.class;
}
```

If you are tempted to keep an empty `RCTDeviceInfoViewManager` "just in case", don't: it adds a TurboModule registration the runtime never reaches, and the `RCT_EXPORT_MODULE()` macro would conflict with the same name being claimed by the Fabric plugin map.[275]

Step 3.3: The Android Implementation

On Android the picture is different from iOS: the `ViewManager` is still the entry point under Fabric. Codegen generates a `<Component>ManagerInterface<T>` and a `<Component>ManagerDelegate<T, U>` in the `com.facebook.react.viewmanagers` package. The manager implements the interface and forwards prop sets to the delegate via `getDelegate()`.[276] The View itself is a plain Android `View` / `ViewGroup` subclass; no Fabric-specific base class is needed.

android/src/main/java/com/reactnativedeviceinfo/DeviceInfoView.kt:

```
package com.reactnativedeviceinfo

import android.content.Context
import android.widget.TextView

// A simple TextView that will display our label
class DeviceInfoView(context: Context) : TextView(context)
```

android/src/main/java/com/reactnativedeviceinfo/DeviceInfoViewManager.kt:

```
package com.reactnativedeviceinfo
```

[275] Fabric iOS component discovery: the legacy `RCTViewManager` base class still ships in `packages/react-native/React/Views/RCTViewManager.h`, but the guarded `RCTImageViewManager.mm` was removed in 0.87 (commit `86350ab9884`, "Remove legacy architecture code guarded by RCT_REMOVE_LEGACY_ARCH from RN iOS"). Fabric discovery happens through the plugin map at `packages/react-native/React/Fabric/Mounting/ComponentViews/RCTFabricComponentsPlugins.mm:19-43`, which maps a component name to a `<Name>Cls(void)` C function. Real-world example: `packages/react-native/React/Fabric/Mounting/ComponentViews/View/RCTViewComponentView.mm:2059` (`Class<RCTComponentViewProtocol> RCTViewCls(void)` returning `RCTViewComponentView.class`). The `+ componentDescriptorProvider` class method is the pattern used by every built-in Fabric view (e.g. `packages/react-native/React/Fabric/Mounting/ComponentViews/Switch/RCTSwitchComponentView.mm:65`). External docs: "Fabric Native Components", reactnative.dev/docs/next/fabric-native-components-introduction.

[276] Android Fabric component generators: interface package + naming at `packages/react-native-codegen/src/generators/components/__tests__/__snapshots__/GeneratePropsJavaInterface-test.js.snap:14, 21` (`package com.facebook.react.viewmanagers; public interface <Name>ManagerInterface<T extends View> extends ViewManagerWithGeneratedInterface`); delegate at `packages/react-native-codegen/src/generators/components/__tests__/__snapshots__/GeneratePropsJavaDelegate-test.js.snap:14, 24` (`public class <Name>ManagerDelegate<T extends View, U extends BaseViewManager<T, ? extends LayoutShadowNode> & <Name>ManagerInterface<T>> extends BaseViewManagerDelegate<T, U>`). `SimpleViewManager<T>` itself lives at `packages/react-native/ReactAndroid/src/main/java/com/facebook/react/uimanager/SimpleViewManager.kt:21`.

```
import com.facebook.react.module.annotations.ReactModule
import com.facebook.react.uimanager.SimpleViewManager
import com.facebook.react.uimanager.ThemedReactContext
import com.facebook.react.uimanager.ViewManagerDelegate

// Generated delegate + interface live in com.facebook.react.viewmanagers.
import com.facebook.react.viewmanagers.DeviceInfoViewManagerDelegate
import com.facebook.react.viewmanagers.DeviceInfoViewManagerInterface

@ReactModule(name = DeviceInfoViewManager.NAME)
class DeviceInfoViewManager :
    SimpleViewManager<DeviceInfoView>(),
    DeviceInfoViewManagerInterface<DeviceInfoView> {

    // BaseViewManager.getDelegate() returns ViewManagerDelegate<T>?; declaring
the
    // explicit type avoids a Kotlin variance error against the parent signature.
    private val delegate: ViewManagerDelegate<DeviceInfoView> =
        DeviceInfoViewManagerDelegate(this)

    override fun getDelegate(): ViewManagerDelegate<DeviceInfoView> = delegate

    override fun getName(): String = NAME

    override fun createViewInstance(reactContext: ThemedReactContext):
DeviceInfoView =
        DeviceInfoView(reactContext)

    // The interface declares this; the @ReactProp annotation is what wires
    // up the legacy / paper code path. Both are kept for backwards compat.
    @com.facebook.react.uimanager.annotations.ReactProp(name = "label")
    override fun setLabel(view: DeviceInfoView, label: String?) {
        view.text = label
    }

    companion object {
        const val NAME = "DeviceInfoView"
    }
}
```

Notes worth flagging:

- Drop the constructor-injected `ThemedReactContext`. RN passes the context to `createViewInstance`; storing it on the manager produces a leak across surface reloads.
- `getDelegate()`'s return type must be `ViewManagerDelegate<DeviceInfoView>` (the parent's signature). Returning the concrete delegate class works in Java but Kotlin's stricter variance check sometimes complains; the explicit field type sidesteps it.

- The generated `DeviceInfoViewManagerInterface` already declares `setLabel`. The override is enough; the `@ReactProp` annotation is only needed for the legacy bridge path and is currently still required for prop discovery on Paper.

Finally, the manager needs to be included in your `BaseReactPackage` via `getViewManagers`:

```
class DeviceInfoPackage : BaseReactPackage() {
    // ...module wiring as in §2.3 above...

    override fun getViewManagers(reactContext: ReactApplicationContext) = listOf(
com.facebook.react.bridge.ModuleSpec.viewManagerSpec({ DeviceInfoViewManager() })
    )
}
```

Part 4: Using the Library

Finally, create an `index.ts` file in your `src` directory to export the components for consumers of your library.

`src/index.ts`:

```
import DeviceInfoModule from './specs/NativeDeviceInfoModule';
import DeviceInfoView from './specs/DeviceInfoViewNativeComponent';

export { DeviceInfoModule, DeviceInfoView };

// Example Usage in an app:
//
// import { DeviceInfoModule, DeviceInfoView } from 'react-native-deviceinfo';
//
// const name = await DeviceInfoModule.getDeviceName();
// <DeviceInfoView label={`Device: ${name}`} style={{ width: '100%', height:
30 }} />
```

Summary of Language Roles

- **TypeScript:** Defines the public API contract for both modules and components. It is the single source of truth.
- **CodeGen:** The invisible tool that reads TypeScript and generates the native C++, Java, and Objective-C++ interfaces.
- **C++:** The language of the JSI and the core of the New Architecture. It's what the generated interfaces are written in.
- **Objective-C++ (`.mm`):** The essential "glue" language on iOS. It's required to communicate between the C++ world (JSI, CodeGen) and the Objective-C world (`UIKit`).
- **Swift:** Can be used for pure logic on iOS, but it must be called from an Objective-C++ file.
- **Kotlin / Java:** The languages for implementing the logic on Android. Kotlin is the modern standard.

Citations:

- [1] `codegenConfig` schema (fields, `type` values, `jsSrcsDir`): see `packages/react-native/scripts/codegen/generate-artifacts-executor/index.js` and the working fixture at `packages/react-native/scripts/codegen/__fixtures__/test-app/lib/test-library/package.json`. The `type` field accepts `"all" | "modules" | "components"` (see `generateRCTModuleProviders.js:59` (`config.type === 'components'`) and `generateRCTThirdPartyComponents.js:52` (`config.type === 'modules'`)). External docs: "Using Codegen", reactnative.dev/docs/next/the-new-architecture/using-codegen. `podspecName` is not a standard field in any RN source path under `packages/`.
- [2] TurboModule type and registry: `packages/react-native/Libraries/TurboModule/RCTExport.{js,d.ts}` (defines `export interface TurboModule`) and `packages/react-native/Libraries/TurboModule/TurboModuleRegistry.{js,d.ts}` (defines `getEnforcing<T extends TurboModule>(name): T`). Canonical spec example: `packages/rn-tester/js/examples/TurboModule/NativeSampleTurboModule.js`. Haste-module-name derivation is the basename of the spec file. External docs: "Turbo Native Modules", reactnative.dev/docs/next/turbo-native-modules-introduction.
- [3] `<RCTBridgeModule>`-only public header pattern: `packages/react-native/Libraries/Settings/RCTSettingsManager.h:12` (public) and `RCTSettingsManager.mm:17` (`@interface RCTSettingsManager () <NativeSettingsManagerSpec>` in a class extension).
- [4] Promise codegen for ObjC++: `packages/react-native-codegen/src/generators/modules/GenerateModuleObjCpp/serializeMethod.js:120-121` pushes `{paramName: 'resolve', objCType: 'RCTPromiseResolveBlock'}` and `{paramName: 'reject', objCType: 'RCTPromiseRejectBlock'}` when the return type is a `PromiseTypeAnnotation`. The typedefs live in `packages/react-native/React/Base/RCTBridgeModule.h:45` (`typedef void (^RCTPromiseResolveBlock)(id _Nullable result);`) and `:52` (`typedef void (^RCTPromiseRejectBlock)(NSString *_Nullable code, NSString *_Nullable message, NSError *_Nullable error);`). The codegen snapshot at `packages/react-native-codegen/src/generators/modules/__tests__/__snapshots__/GenerateModuleHObjCpp-test.js.snap:882-886` shows the generated `@protocol` form.
- [5] Generated Android spec class name: `packages/react-native-codegen/src/generators/modules/GenerateModuleJavaSpec.js:580` (`const className = \${hasteModuleName}Spec`;`). The generated class also already provides `getName()` returning a static `NAME` constant, confirmed by the snapshot at `packages/react-native-codegen/src/generators/modules/__tests__/__snapshots__/GenerateModuleJavaSpec-test.js.snap:26-35`.
- [6] `TurboReactPackage` deprecation: `packages/react-native/ReactAndroid/src/main/java/com/facebook/react/TurboReactPackage.kt:10-14` (`@Deprecated(message = "Use BaseReactPackage instead", replaceWith = ReplaceWith(expression = "BaseReactPackage"))`). The replacement landed in the v0.76 cycle: see `CHANGELOG-0.7x.md:1262` ("turbomodules: Replace TurboReactPackage with BaseReactPackage", commit `e881a1184c`). RN itself uses `BaseReactPackage` (`packages/react-native/ReactAndroid/src/main/java/com/facebook/react/runtime/CoreReactPackage.kt:48`).

- [7] `ReactModuleInfo` constructor evolution: `packages/react-native/ReactAndroid/src/main/java/com/facebook/react/module/model/ReactModuleInfo.kt:16-23` (current 6-arg primary ctor: name, className, canOverrideExistingModule, needsEagerInit, isCxxModule, isTurboModule). The 7-arg overload at `:24-42` is `@Deprecated`, and the `hasConstants` parameter is `@Suppress("UNUSED_PARAMETER")`-marked so it is silently ignored.
- [8] `RCTViewComponentView`'s declaration and `updateProps:oldProps:` signature: `packages/react-native/React/Fabric/Mounting/ComponentViews/View/RCTViewComponentView.h:22` (`@interface RCTViewComponentView : UIView <RCTComponentViewProtocol, RCTTouchableComponentViewProtocol>`) and `:67-68` (`- (void)updateProps:(const facebook::react::Props::Shared &)props oldProps:(const facebook::react::Props::Shared &)oldProps NS_REQUIRES_SUPER;`). `Props::Shared` is defined as `std::shared_ptr<const Props>` at `packages/react-native/ReactCommon/react/renderer/core/Props.h:31`.
- [9] Fabric iOS component discovery: the legacy `RCTViewManager` base class still ships in `packages/react-native/React/Views/RCTViewManager.h`, but the guarded `RCTImageViewManager.mm` was removed in 0.87 (commit `86350ab9884`, "Remove legacy architecture code guarded by RCT_REMOVE_LEGACY_ARCH from RN iOS"). Fabric discovery happens through the plugin map at `packages/react-native/React/Fabric/Mounting/ComponentViews/RCTFabricComponentsPlugins.mm:19-43`, which maps a component name to a `<Name>Cls(void)` C function. Real-world example: `packages/react-native/React/Fabric/Mounting/ComponentViews/View/RCTViewComponentView.mm:2059` (`Class<RCTComponentViewProtocol> RCTViewCls(void)` returning `RCTViewComponentView.class`). The `+ componentDescriptorProvider` class method is the pattern used by every built-in Fabric view (e.g. `packages/react-native/React/Fabric/Mounting/ComponentViews/Switch/RCTSwitchComponentView.mm:65`). External docs: "Fabric Native Components", reactnative.dev/docs/next/fabric-native-components-introduction.
- [10] Android Fabric component generators: interface package + naming at `packages/react-native-codegen/src/generators/components/__tests__/__snapshots__/GeneratePropsJavaInterface-test.js.snap:14, 21` (`package com.facebook.react.viewmanagers; public interface <Name>ManagerInterface<T extends View> extends ViewManagerWithGeneratedInterface);` delegate at `packages/react-native-codegen/src/generators/components/__tests__/__snapshots__/GeneratePropsJavaDelegate-test.js.snap:14, 24` (`public class <Name>ManagerDelegate<T extends View, U extends BaseViewManager<T, ? extends LayoutShadowNode> & <Name>ManagerInterface<T>> extends BaseViewManagerDelegate<T, U>`). `SimpleViewManager<T>` itself lives at `packages/react-native/ReactAndroid/src/main/java/com/facebook/react/uimanager/SimpleViewManager.kt:21`.

Chapter 16: Additional Resources and Checklists

This chapter consolidates useful checklists and resource lists from various guides to serve as a quick reference.

Common Migration Challenges

When migrating an existing application, be aware of these common challenges:

1. **Third-party Library Compatibility**: Not all third-party native libraries have been updated for the New Architecture. This is often the biggest hurdle. Always check a library's documentation for compatibility.
2. **Learning Curve**: Understanding the new concepts, especially the JSI, C++ types, and the new threading model, requires a learning investment.
3. **Build Configuration**: The New Architecture is hard-wired on for React Native 0.82+. `newArchEnabled=false` in `gradle.properties` is rejected by the React Native Gradle Plugin with an explicit error, and `RCT_NEW_ARCH_ENABLED=0` in the iOS environment is ignored (the helper still prints a warning).[277] On 0.87+, also expect AGP v9 / Gradle 9.4.x (the `helloworld` app pins 9.4.1), Node.js `^22.13.0 || ^24.3.0 || >= 26.0.0` (`packages/react-native/package.json` `engines`), and the Strict TypeScript API as the default JS surface (Metro `customConditions: ["react-native-legacy-deep-imports"]` to opt out temporarily).
4. **Testing Coverage**: Migrating can introduce subtle behavioral changes. Thorough end-to-end testing is critical to ensure all features work as expected.
5. **Interop compile-out:** Do not assume legacy `RCTBridgeModule` / Paper view managers will keep working indefinitely under Bridgeless. On the v0.87 and v0.88 lines those shims are gated behind `RCT_REMOVE_LEGACY_MODULE_INTEROP` / `RCT_REMOVE_LEGACY_COMPONENT_INTEROP`. Since 0.88 you can opt in to both from CocoaPods and SwiftPM (default off, planned to become the default). Migrate libraries before those gates become unconditional.

Recommended Development Tools

The following tools can be helpful when working with the New Architecture:

- **`react-native-builder-bob`**: A scaffolding tool that can help create the boilerplate for a new native module or component library. Maintained by Callstack.[278]
- **`RN-Tester`**: The official React Native tester app within the main repository, at `packages/rn-tester/`. It bundles three end-to-end New Architecture examples that are useful as references: `NativeModuleExample/` (Objective-C++ TurboModule), `NativeCxxModuleExample/` (pure C++ TurboModule, also runs on Android via JNI), and `NativeComponentExample/` (Fabric native component).[279]

---

[277] `packages/gradle-plugin/react-native-gradle-plugin/src/main/kotlin/com/facebook/react/ReactRootProjectPlugin.kt:61-84`, `checkLegacyArchProperty` (Android error on `newArchEnabled=false`) and the v0.82 CHANGELOG entry "**New Architecture:** Add warning if `RCT_NEW_ARCH_ENABLED` is set to 0" (commit `7d0bef2f25`).

[278] `react-native-builder-bob`. Callstack. https://github.com/callstack/react-native-builder-bob

[279] RN-Tester native examples: `packages/rn-tester/NativeModuleExample/`, `packages/rn-tester/NativeCxxModuleExample/`, `packages/rn-tester/NativeComponentExample/`.

- **Community Example Repositories**: The original step-by-step gallery lived at `react-native-community/RNNewArchitectureLibraries`. That repo was archived in March 2024, so use it for the *recipes*, not as a moving target. The maintained successor is the working-group repo `reactwg/react-native-new-architecture`; its `docs/` folder has been folded into the official docs but the discussion archive there is still the best place to find migration war stories from library authors.[280]

Library Migration Checklist

For authors of libraries that contain native code, the migration process involves a few extra steps to ensure consumers of your library have a smooth experience.

- **[ ] Ship CodeGen Specs:** Your TypeScript or Flow spec files must be included in your published npm package. CodeGen reads each linked library's `codegenConfig.jsSrcsDir` at build time, so if the specs are excluded by `.npmignore` or `files`, the consuming app's build will fail to generate the glue code.
- **[ ] Add `codegenConfig` to `package.json`:** This is essential so the consuming app's build process knows how to run CodeGen on your library's specs.[281] The schema is `{ name, type, jsSrcsDir, android, ios, includesGeneratedCode }`. The Android side models only `name, type, jsSrcsDir, android, includesGeneratedCode` in `packages/gradle-plugin/shared/src/main/kotlin/com/facebook/react/model/ModelCodegenConfig.kt`.[282] See `packages/react-native-popup-menu-android/package.json` for a single-platform example or `packages/react-native/package.json` for the multi-library shape (`codegenConfig.libraries[]`).
- **[ ] Implement Both Architectures (Only if You Support RN ≤ 0.81):** Before React Native 0.82, library authors used the `BuildConfig.IS_NEW_ARCHITECTURE_ENABLED` flag on Android and the `RCTIsNewArchEnabled()` utility on iOS to register the correct module implementation at runtime.[283] Starting with React Native 0.82, opt-out from the New Architecture was removed: on iOS `RCTIsNewArchEnabled()` is hardcoded to return `YES`,[284] and on Android the React Native Gradle Plugin hardcodes the `IS_NEW_ARCHITECTURE_ENABLED` `buildConfigField` to `"true"`.[285]

---

[280] `react-native-community/RNNewArchitectureLibraries` (archived 2024-03-20): https://github.com/react-native-community/RNNewArchitectureLibraries. Successor working group: https://github.com/reactwg/react-native-new-architecture.

[281] "Using Codegen". React Native Documentation. https://reactnative.dev/docs/next/the-new-architecture/using-codegen

[282] `packages/gradle-plugin/shared/src/main/kotlin/com/facebook/react/model/ModelCodegenConfig.kt:10-16` (`data class ModelCodegenConfig(name, type, jsSrcsDir, android, includesGeneratedCode)`).

[283] `packages/react-native/React/Base/RCTUtils.h:18` declares `RCT_EXTERN BOOL RCTIsNewArchEnabled(void);`. The implementation in `packages/react-native/React/Base/RCTUtils.mm:47-50` is hardcoded to `return YES;` since commit `83e6eaf693f` ("Prevent users from opting-out of the New Architecture", PR #53026, first released in v0.82.0-rc.0, 2025-08-05).

[284] `packages/react-native/React/Base/RCTUtils.h:18` declares `RCT_EXTERN BOOL RCTIsNewArchEnabled(void);`. The implementation in `packages/react-native/React/Base/RCTUtils.mm:47-50` is hardcoded to `return YES;` since commit `83e6eaf693f` ("Prevent users from opting-out of the New Architecture", PR #53026, first released in v0.82.0-rc.0, 2025-08-05).

[285] `packages/gradle-plugin/react-native-gradle-plugin/src/main/kotlin/com/facebook/react/utils/AgpConfiguratorUtils.kt:63` injects `buildConfigField("boolean", "IS_NEW_ARCHITECTURE_ENABLED",`

So this checklist item only matters if your library still ships releases for users pinned to RN ≤ 0.81; for libraries that already require RN ≥ 0.82, drop the conditional and ship the New Arch implementation only.

- **[ ] Provide a `Podspec` for iOS:** The `.podspec` file for your library must correctly declare its dependencies and source files. Working references in RN itself: `packages/rn-tester/NativeModuleExample/ScreenshotManager.podspec`, `NativeComponentExample/MyNativeView.podspec`, `NativeCxxModuleExample/NativeCxxModuleExample.podspec`.
- **[ ] Provide `build.gradle` for Android:** Your Android library's `build.gradle` must apply the React Native Gradle Plugin so CodeGen runs and the generated C++ JNI sources link into your library's CMake build.[286] `packages/react-native-popup-menu-android/android/build.gradle.kts` is the closest in-tree example.
- **[ ] Document Compatibility:** Clearly state in your `README.md` which versions of your library support the New Architecture and what versions of React Native they are compatible with.[287] If you still ship pre-0.82 support, document the minimum RN version each major library version targets.
- **[ ] Add CI for the Versions You Actually Support:** Set up CI to build and test your library's example app against every (RN version × architecture) combination you advertise. For libraries that only target RN ≥ 0.82, that means just one matrix axis (the New Architecture); for libraries that still claim RN ≤ 0.81 support, exercise both arches on those older versions to catch regressions.
- **[ ] Prefer public `'react-native'` imports:** On 0.87+, the Strict TypeScript API is the default. Deep imports into `react-native/Libraries/...` and `react-native/src/private/...` are restricted or untyped; use the published surface (and `HostInstance` instead of the removed/deprecated `NativeMethods` / `NativeMethodsMixin` types).
- **[ ] Use `ArrayBuffer` for binary TurboModule payloads (C++ / ObjC):** Spec methods that take/return `ArrayBuffer` instead of base64 strings. C++ and ObjC bridging support landed in 0.87 (`ReactCommon/react/bridging/ArrayBuffer.h`, ObjC `NSMutableData *` mapping, restored in 0.88). Java/Kotlin TurboModules gained `com.facebook.react.bridge.ArrayBuffer` support in 0.88. Borrowed bytes are valid only during the synchronous call: copy with `ArrayBuffer.arrayBufferWithCopiedBytes` to keep them. Java CodeGen still rejects `Promise<ArrayBuffer>` (`throwIfUnsupportedPromiseArrayBuffer`).

Citations:

- [1] `packages/react-native/React/Base/RCTUtils.h:18` declares `RCT_EXTERN BOOL RCTIsNewArchEnabled(void);`. The implementation in `packages/react-native/React/Base/`

`"true")` unconditionally. Hardcoded by commit `d5d21d06149` ("Remove possibility to newArchEnabled=false in 0.82", PR #53025, first released in v0.82.0-rc.0, 2025-08-05).

[286] "React Native Gradle Plugin". React Native Documentation. https://reactnative.dev/docs/react-native-gradle-plugin

[287] "About the New Architecture". React Native Documentation. https://reactnative.dev/architecture/landing-page

`RCTUtils.mm:47-50` is hardcoded to `return YES;` since commit `83e6eaf693f` ("Prevent users from opting-out of the New Architecture", PR #53026, first released in v0.82.0-rc.0, 2025-08-05).

- [2] "Using Codegen". React Native Documentation. https://reactnative.dev/docs/next/the-new-architecture/using-codegen
- [3] "React Native Gradle Plugin". React Native Documentation. https://reactnative.dev/docs/react-native-gradle-plugin
- [4] "About the New Architecture". React Native Documentation. https://reactnative.dev/architecture/landing-page
- [5] `packages/gradle-plugin/react-native-gradle-plugin/src/main/kotlin/com/facebook/react/ReactRootProjectPlugin.kt:61-84`, `checkLegacyArchProperty` (Android error on `newArchEnabled=false`) and the v0.82 CHANGELOG entry "**New Architecture:** Add warning if `RCT_NEW_ARCH_ENABLED` is set to 0" (commit `7d0bef2f25`).
- [6] `react-native-builder-bob`. Callstack. https://github.com/callstack/react-native-builder-bob
- [7] RN-Tester native examples: `packages/rn-tester/NativeModuleExample/`, `packages/rn-tester/NativeCxxModuleExample/`, `packages/rn-tester/NativeComponentExample/`.
- [8] `react-native-community/RNNewArchitectureLibraries` (archived 2024-03-20): https://github.com/react-native-community/RNNewArchitectureLibraries. Successor working group: https://github.com/reactwg/react-native-new-architecture.
- [9] `packages/gradle-plugin/shared/src/main/kotlin/com/facebook/react/model/ModelCodegenConfig.kt:10-16` (`data class ModelCodegenConfig(name, type, jsSrcsDir, android, includesGeneratedCode)`).
- [10] `packages/gradle-plugin/react-native-gradle-plugin/src/main/kotlin/com/facebook/react/utils/AgpConfiguratorUtils.kt:63` injects `buildConfigField("boolean", "IS_NEW_ARCHITECTURE_ENABLED", "true")` unconditionally. Hardcoded by commit `d5d21d06149` ("Remove possibility to newArchEnabled=false in 0.82", PR #53025, first released in v0.82.0-rc.0, 2025-08-05).

Chapter 17: Top 27 React Native Interview Questions

Use this chapter to prepare for a senior React Native interview. Each answer starts with a short spoken answer. Key points, a common wrong answer, and a likely follow-up come next. Every fact has a footnote with a source path, a changelog line, or a URL. All facts are checked against React Native v0.88.1 and `main` at `9c7b0f5fafd`, unless a note says otherwise.

Part 1: Architecture

Q1. What is the React Native New Architecture, and what problem does it solve?

Short answer (say this first, 2 to 4 sentences): The New Architecture is the current runtime of React Native. It replaces the old asynchronous Bridge with JSI, and it uses Fabric for rendering and TurboModules for native modules. JS can now call native code directly and synchronously when needed, and React can render with concurrent features. In 0.88 it is the only supported mode, not an option.[288][289]

Key points:

- Four parts: JSI (the C++ interface to the JS engine), Fabric (renderer), TurboModules (native modules), Codegen (type-safe glue from specs).
- The old design sent batched messages across the Bridge. The new design lets JS hold references to native objects.[290]
- It has been mandatory since 0.82. Android ignores `newArchEnabled=false` and logs an error.[291][292]
- On iOS, `pod install` with `RCT_NEW_ARCH_ENABLED=0` prints a warning and the app still runs with the New Architecture.[293]
- The Bridgeless feature flag `enableBridgelessArchitecture` defaults to `true`.[294]

Example or code (only when it helps, max 15 lines):

```
# android/gradle.properties (0.88 template style). Not compiled.
newArchEnabled=true
hermesEnabled=true
```

Common wrong answer: "The New Architecture is experimental and you can turn it off." Wrong: it has been the only mode since 0.82.

---

[288] packages/react-native/package.json (version field, RN main) and CHANGELOG.md:3 (v0.88.1 heading), proves the target version.

[289] CHANGELOG.md:1500, "Remove possibility to newArchEnabled=false in 0.82".

[290] packages/react-native/Libraries/BatchedBridge/MessageQueue.js:36, queue and 5 ms flush gap (the old design).

[291] CHANGELOG.md:1500, "Remove possibility to newArchEnabled=false in 0.82".

[292] packages/gradle-plugin/react-native-gradle-plugin/src/main/kotlin/com/facebook/react/ReactRootProjectPlugin.kt:61-82, error text "not supported anymore since React Native 0.82".

[293] packages/react-native/scripts/react_native_pods.rb:545-556, warning for RCT_NEW_ARCH_ENABLED=0; line 148 forces it to "1".

[294] packages/react-native/src/private/featureflags/ReactNativeFeatureFlags.js:300 and scripts/featureflags/ReactNativeFeatureFlags.config.js:250-252, `enableBridgelessArchitecture` default true.

Likely follow-up: Is it faster in every case? No. It removes Bridge serialization and enables concurrent React. Real gains depend on the app.

Q2. How did the old Bridge work, and how does JSI change that?

Short answer (say this first, 2 to 4 sentences): The old Bridge queued native calls in JS (`MessageQueue`) and flushed them in batches to native. Native replied through callbacks and events. JSI is a C++ API over the JS engine. With JSI, native code can install host objects and host functions that JS calls directly, with no queue between them. The Bridge JS file is now marked deprecated.[295][296][297]

Key points:

- `MessageQueue.js` batches calls. It uses a minimum flush gap `MIN_TIME_BETWEEN_FLUSHES_MS = 5`.[298]
- `BatchedBridge.js` still installs `global.__fbBatchedBridge`, but the file is tagged `@deprecated`.[299]
- In `MessageQueue.js`, `JSON.stringify` appears only in the debug `spy` logger (line 99) and in an invalid-argument error message (line 309). It is not in the call path (`enqueueNativeCall`) in 0.88.[300]
- JSI defines `jsi::HostObject` and `HostFunctionType`. Native code exposes them to JS.[301]
- A TurboModule is itself a `jsi::HostObject`, so JS holds the native object directly.[302]
- Interop: if the JSI proxy has no module, the registry falls back to `NativeModules`. Legacy modules still use that path.[303]
- JSI is engine neutral. Hermes is the engine you ship (see Chapter on Hermes).

Example or code (only when it helps, max 15 lines):

```
// JS side, simplified. TurboModuleRegistry asks the JSI proxy first.
const turboModuleProxy = global.__turboModuleProxy;
const module = turboModuleProxy != null ? turboModuleProxy(name) : null;
// Source: Libraries/TurboModule/TurboModuleRegistry.js (simplified)
```

---

[295] packages/react-native/Libraries/BatchedBridge/MessageQueue.js:36 and :309, flush gap constant and JSON.stringify only in the spy logger (:99) and an error message (:309). Proof: scratch/A/proof-2.txt.

[296] packages/react-native/Libraries/BatchedBridge/BatchedBridge.js:9 and :21-24, `@deprecated` and `__fbBatchedBridge`.

[297] packages/react-native/ReactCommon/jsi/jsi/jsi.h:217 (`HostFunctionType`) and :222 (`HostObject`).

[298] packages/react-native/Libraries/BatchedBridge/MessageQueue.js:36 and :309, flush gap constant and JSON.stringify only in the spy logger (:99) and an error message (:309). Proof: scratch/A/proof-2.txt.

[299] packages/react-native/Libraries/BatchedBridge/BatchedBridge.js:9 and :21-24, `@deprecated` and `__fbBatchedBridge`.

[300] packages/react-native/Libraries/BatchedBridge/MessageQueue.js:36 and :309, flush gap constant and JSON.stringify only in the spy logger (:99) and an error message (:309). Proof: scratch/A/proof-2.txt.

[301] packages/react-native/ReactCommon/jsi/jsi/jsi.h:217 (`HostFunctionType`) and :222 (`HostObject`).

[302] packages/react-native/ReactCommon/react/nativemodule/core/ReactCommon/TurboModule.h:49, `class JSI_EXPORT TurboModule : public jsi::HostObject`.

[303] packages/react-native/Libraries/TurboModule/TurboModuleRegistry.js:17-46, proxy first, then NativeModules fallback, `get` and `getEnforcing`.

Common wrong answer: "The bridge is async JSON, and JSI only makes it faster." Wrong: JSI removes the queue for the main paths. The call path in `MessageQueue.js` does not use `JSON.stringify`. Also, calls can be sync when the spec says so.

Likely follow-up: Is JSI always synchronous? No. A method returns a Promise or takes a callback when you want async. The spec decides.

Q3. What is Fabric, and how is rendering different from the old UIManager?

Short answer (say this first, 2 to 4 sentences): Fabric is the C++ renderer. React builds an immutable shadow tree in C++. The renderer diffs the old and new trees into a list of mutations. Then it mounts those mutations on the UI thread. The old UIManager was a Java/Obj-C layer that got serialized commands over the Bridge.[304][305]

Key points:

- The shadow tree lives in C++ (`ShadowTree`) and is shared by iOS and Android.[306]
- The differ turns two trees into `ShadowViewMutation` lists (`calculateShadowViewMutations`).[307]
- JS reaches the renderer through the JSI global `nativeFabricUIManager`, wrapped in `FabricUIManager.js`.[308]
- Layout runs in C++ with Yoga on the shadow tree, so JS can read layout synchronously (see the layout chapter).
- Old Paper classes were removed step by step. The 0.88 changelog lists removals such as `UIImplementation` and `RCTCxxBridge`.[309]

Example or code (only when it helps, max 15 lines):

```
// Public API users call. Fabric needs no code change in app JS.
import {View, Text} from 'react-native';
```

Common wrong answer: "Fabric is just a faster UIManager." Wrong: it is a new C++ pipeline with immutable trees and a shared core.

Likely follow-up: What does immutable mean here? Each commit makes a new tree. Old trees stay valid, so threads can read without locks.

Q4. What is a TurboModule, and what does Codegen do?

Short answer (say this first, 2 to 4 sentences): A TurboModule is a native module that JS reaches through JSI. It loads lazily, when JS first asks for it. You describe its API in a typed JS spec

---

[304] packages/react-native/ReactCommon/react/renderer/mounting/ShadowTree.h:76 (`class ShadowTree final`) and :31-50 (commit status, mode, source enums).

[305] packages/react-native/ReactCommon/react/renderer/mounting/Differentiator.h:22, `calculateShadowViewMutations`.

[306] packages/react-native/ReactCommon/react/renderer/mounting/ShadowTree.h:76 (`class ShadowTree final`) and :31-50 (commit status, mode, source enums).

[307] packages/react-native/ReactCommon/react/renderer/mounting/Differentiator.h:22, `calculateShadowViewMutations`.

[308] packages/react-native/Libraries/ReactNative/FabricUIManager.js:155-211, proxy to `nativeFabricUIManager`.

[309] CHANGELOG.md:469 (UIImplementation removed) and CHANGELOG.md:309 (RCTCxxBridge symbols removed).

file. Codegen reads that spec and generates the native interfaces in C++, Obj-C++, and Java. This way JS and native agree on types at build time.[310][311]

Key points:

- JS gets a module with `TurboModuleRegistry.get` or `getEnforcing`. The second one throws when the module is missing.[312]
- If the JSI proxy has no module, the registry falls back to the old `NativeModules` lookup. This is the interop path.[313]
- `TurboModule` in C++ derives from `jsi::HostObject`.[314]
- Codegen generators for modules live in `packages/react-native-codegen/src/generators/modules`. They produce C++/JNI/Java and Obj-C++ output.[315]
- The core package itself declares `codegenConfig` with a library `FBReactNativeSpec`.[316]
- A wrong spec fails at build time, not at call time.

Example or code (only when it helps, max 15 lines):

```
// NativeSampleModule.js (spec). Not compiled.
import type {TurboModule} from 'react-native';
import {TurboModuleRegistry} from 'react-native';
export interface Spec extends TurboModule {
  +add: (a: number, b: number) => number;
}
export default TurboModuleRegistry.getEnforcing<Spec>('SampleModule');
```

Common wrong answer: "Codegen runs at runtime and checks types as you call." Wrong: it runs at build time and generates native code.

Likely follow-up: Does a TurboModule load at app start? No. It loads lazily on first access.

Q5. What is Bridgeless mode, and can I still use the old Bridge or opt out of the New Architecture in 0.88?

Short answer (say this first, 2 to 4 sentences): Bridgeless mode runs React Native with no Bridge instance. The runtime talks to JS through JSI only. In 0.88 you cannot opt out of the New Architecture. Legacy modules and legacy view managers still work through interop layers.

---

[310] packages/react-native/Libraries/TurboModule/TurboModuleRegistry.js:17-46, proxy first, then NativeModules fallback, `get` and `getEnforcing`.

[311] packages/react-native-codegen/src/generators/modules (GenerateModuleH.js, GenerateModuleJavaSpec.js, GenerateModuleJniCpp.js, GenerateModuleObjCpp).

[312] packages/react-native/Libraries/TurboModule/TurboModuleRegistry.js:17-46, proxy first, then NativeModules fallback, `get` and `getEnforcing`.

[313] packages/react-native/Libraries/TurboModule/TurboModuleRegistry.js:17-46, proxy first, then NativeModules fallback, `get` and `getEnforcing`.

[314] packages/react-native/ReactCommon/react/nativemodule/core/ReactCommon/TurboModule.h:49.

[315] packages/react-native-codegen/src/generators/modules (GenerateModuleH.js, GenerateModuleJavaSpec.js, GenerateModuleJniCpp.js, GenerateModuleObjCpp).

[316] packages/react-native/package.json:183 (HEAD) and :187 (v0.88.1), `codegenConfig` library `FBReactNativeSpec`.

Maintainers are removing even those step by step.[317][318]

Key points:

- Android: `newArchEnabled=false` gives an error and is ignored since 0.82.[319]
- iOS: `RCT_NEW_ARCH_ENABLED=0` gives a warning and is ignored.[320]
- `RCT_REMOVE_LEGACY_ARCH` defaults to `1` in the pod script, which compiles legacy code out. Setting it to `0` keeps some old code compiled. It does not turn the old Bridge back on.[321]
- Interop switches `RCT_REMOVE_LEGACY_MODULE_INTEROP` and `RCT_REMOVE_LEGACY_COMPONENT_INTEROP` are opt-in. Both default to off now. They are planned to become the default later.[322]
- Old libraries often still run through the interop layer. Check each library before an upgrade.
- `BridgelessReactContext.getCatalystInstance()` is deprecated.[323]

Example or code (only when it helps, max 15 lines):

```
# Podfile. Not compiled. Opt in to strict removal of the interop:
ENV['RCT_REMOVE_LEGACY_MODULE_INTEROP'] = '1'   # name from CHANGELOG, check your
Podfile docs
ENV['RCT_REMOVE_LEGACY_COMPONENT_INTEROP'] = '1'
```

Common wrong answer: "Set newArchEnabled=false to stay on the old Bridge until you are ready." Wrong since 0.82: the flag is ignored.

Likely follow-up: What breaks if a library is old? Its native code may need the interop layer or a migration to a TurboModule or Fabric component.

Part 2: Runtime and JavaScript

Q6. What is Hermes, why is it the default engine, and what is Hermes V1 and bytecode precompilation?

Short answer (say this first, 2 to 4 sentences): Hermes is an open-source JavaScript engine built for React Native. It is the default engine, so no setup is needed.[324] In release builds, the build step compiles your JavaScript bundle to Hermes bytecode (HBC) ahead of time. The device loads bytecode, so startup is faster. Hermes V1 is the new generation of the engine, and it became the default in 0.84.[325]

Key points:

---

[317] packages/gradle-plugin/react-native-gradle-plugin/src/main/kotlin/com/facebook/react/ReactRootProjectPlugin.kt:61-82 and CHANGELOG.md:1500.

[318] packages/react-native/scripts/react_native_pods.rb:545-556.

[319] packages/gradle-plugin/react-native-gradle-plugin/src/main/kotlin/com/facebook/react/ReactRootProjectPlugin.kt:61-82 and CHANGELOG.md:1500.

[320] packages/react-native/scripts/react_native_pods.rb:545-556.

[321] packages/react-native/scripts/react_native_pods.rb:120 (default '1') and :627-630 (flag adds -DRCT_REMOVE_LEGACY_ARCH=1). Also at tag v0.88.1 line 120.

[322] CHANGELOG.md:81, opt-in interop removal flags, both default off.

[323] CHANGELOG.md:1583, `BridgelessReactContext.getCatalystInstance()` deprecated.

[324] tmp/rnw-ch0506/docs/hermes.md:12-18, Hermes is default and bundled per release.

[325] CHANGELOG.md:1038 (v0.84.0, Breaking), "Make Hermes V1 the default engine".

- React Native ships a bundled Hermes, built for each React Native release, so engine and framework versions match.[326]
- Hermes V1 became the default engine in 0.84.[327] The `hermesV1Enabled` opt-in flag was removed in 0.87, so there is no switch to turn V1 off.[328]
- The compiler is a separate npm package, `hermes-compiler`. At v0.88.1 it is pinned to `260318099.0.5`.[329]
- Release builds compile JavaScript to bytecode at build time. Debug builds load JavaScript from Metro.[330]
- The runtime and the compiler must match. A mismatch gives a "Wrong bytecode version" crash. A SwiftPM release build had this bug, fixed in 0.87.[331]
- JavaScriptCore is not in core. You opt out of Hermes only through the community JavaScriptCore package.[332]

Example or code (only when it helps, max 15 lines):

```
// Runtime check. Not enough alone to prove you load bytecode.
const isHermes = () => !!global.HermesInternal;
```

Common wrong answer: "Hermes is optional, and JavaScriptCore is the default on iOS." Wrong. Hermes is the default on both platforms, and JavaScriptCore lives in a community package.[333][334]

Likely follow-up: How do you check that your release app uses bytecode? Make a release build and compare startup. `HermesInternal` can exist even when you load plain JavaScript, so also confirm the bundle is a `.hbc` file.[335]

Q7. Which threads does React Native use, and where does your JS, layout, and UI work run?

Short answer (say this first, 2 to 4 sentences): The renderer uses two main threads. The JavaScript thread runs your code, React's render phase, and layout. The UI (main) thread is the only thread that can change native views. Mounting runs on the UI thread, and the commit phase can run on a background thread.[336][337]

Key points:

- The UI thread is the only thread that can manipulate host views.[338]

---

[326] tmp/rnw-ch0506/docs/hermes.md:12-18, Hermes is default and bundled per release.

[327] CHANGELOG.md:1038 (v0.84.0, Breaking), "Make Hermes V1 the default engine".

[328] CHANGELOG.md:468 (v0.87.0), "Remove hermesV1Enabled and simplify the code".

[329] packages/react-native/package.json:168 at tag v0.88.1, `hermes-compiler` 260318099.0.5. At HEAD line 164 it is the placeholder `0.0.0`.

[330] tmp/rnw-ch0506/docs/hermes.md:91, release build compiles JS to Hermes bytecode at build time.

[331] CHANGELOG.md:268 (v0.87.1), SwiftPM "Wrong bytecode version" fix.

[332] tmp/rnw-ch0506/docs/hermes.md:95, JavaScriptCore through the community repository.

[333] tmp/rnw-ch0506/docs/hermes.md:12-18, Hermes is default and bundled per release.

[334] tmp/rnw-ch0506/docs/hermes.md:95, JavaScriptCore through the community repository.

[335] tmp/rnw-ch0506/docs/hermes.md:31-39, `HermesInternal` check and `.hbc` note.

[336] tmp/rnw-ch0506/website/architecture/threading-model.md:14-19, two threads and thread safety.

[337] tmp/rnw-ch0506/website/architecture/render-pipeline.md:77 (commit on background thread) and :98-104 (mount on UI thread, sync if commit ran on UI thread).

- The JavaScript thread runs React's render phase and layout (Yoga, in C++).[339]
- The renderer is thread safe because it uses immutable data. Each update clones objects.[340]
- Most renders run on the JS thread. A high-priority event on the UI thread can run the whole pipeline synchronously on the UI thread.[341]
- The commit phase (layout and tree promotion) runs on a background thread. The mount phase (tree diffing, promotion, view mounting) runs on the UI thread. If the commit runs on the UI thread, mounting runs synchronously on it.[342]
- A busy JS thread blocks touch handling in JS and JS animations. Native scrolling and native stack transitions keep running.[343]

Common wrong answer: "There is a separate shadow thread that does layout, and the Bridge sends batched JSON between threads." That is the old model. In the New Architecture, the renderer uses immutable data, layout runs in C++ (Yoga), and the whole pipeline can run synchronously on the UI thread.[344][345]

Likely follow-up: Why does a scroll stay smooth when JS is blocked? The scroll view lives on the UI thread. Scroll events go to JS, but the scroll does not wait for them.[346]

Q8. How does React 19 (concurrent rendering) work in React Native? Transitions, Suspense, and automatic batching.

Short answer (say this first, 2 to 4 sentences): The New Architecture supports Concurrent React in full. This includes Suspense, Transitions, automatic batching, and `useLayoutEffect`.[347] React 19 updates can be interrupted, because the renderer can hold several in-progress trees and work at several priorities.[348] At v0.88.1 the `react` peer dependency is `^19.3.0`.[349]

Key points:

- The old Bridge design could not support concurrent features. The New Renderer fixes this with multiple trees and priorities.[350]

---

[338] tmp/rnw-ch0506/website/architecture/threading-model.md:14-19, two threads and thread safety.

[339] tmp/rnw-ch0506/website/architecture/threading-model.md:14-19, two threads and thread safety.

[340] tmp/rnw-ch0506/website/architecture/threading-model.md:14-19, two threads and thread safety.

[341] tmp/rnw-ch0506/website/architecture/threading-model.md:25-41, render on JS thread, or sync on UI thread.

[342] tmp/rnw-ch0506/website/architecture/render-pipeline.md:77 (commit on background thread) and :98-104 (mount on UI thread, sync if commit ran on UI thread).

[343] tmp/rnw-ch0506/docs/performance.md:22-28, JS thread blocks touches, scroll and native stack run on the main thread.

[344] tmp/rnw-ch0506/website/architecture/threading-model.md:14-19, two threads and thread safety.

[345] tmp/rnw-ch0506/website/architecture/threading-model.md:25-41, render on JS thread, or sync on UI thread.

[346] tmp/rnw-ch0506/docs/performance.md:22-28, JS thread blocks touches, scroll and native stack run on the main thread.

[347] tmp/rnw-ch0506/website/blog/2024-10-23-the-new-architecture-is-here.mdx:12, Suspense, Transitions, automatic batching, useLayoutEffect.

[348] tmp/rnw-ch0506/website/blog/2024-10-23-the-new-architecture-is-here.mdx:39-48, multiple trees and priorities.

[349] packages/react-native/package.json:149 at tag v0.88.1, `"react": "^19.3.0"`.

[350] tmp/rnw-ch0506/website/blog/2024-10-23-the-new-architecture-is-here.mdx:39-48, multiple trees and priorities.

- The render phase is interruptible. A high-priority UI event can interrupt it.[351]
- Transitions mark an update as low priority. `startTransition` keeps input responsive while React renders the slow update in the background. This is React behavior, not a React Native API.
- Suspense shows a fallback while a part of the tree waits. React Native docs use it to lazy load components.[352]
- Automatic batching groups several state updates into one render. In React 18 and later this also covers timers and promises (React behavior, not checked in this repo).[353]
- Your components must follow the Rules of React to be safe in concurrent mode. The interop layer for old modules does not support concurrent features.[354]

Example or code (only when it helps, max 15 lines):

```
// Not compiled: needs a React Native runtime.
const [isPending, startTransition] = useTransition();
const onChange = text => {
  setQuery(text); // urgent: keeps the input responsive
  startTransition(() => setResults(search(text))); // can be interrupted
};
```

Common wrong answer: "Concurrent rendering uses more threads, so my JS runs in parallel." Wrong. Your JavaScript still runs on one thread. Concurrent React only lets React pause and resume work and pick priorities.[355]

Likely follow-up: What breaks if a component has side effects in render? React can run render more than once or drop a render. Side effects in render then run twice or leak. Keep render pure.[356]

Q9. How does Metro bundle an app? Cover inline requires, tree shaking status, and source maps.

Short answer (say this first, 2 to 4 sentences): Metro builds a dependency graph from your entry file. It transforms each file with Babel, then serializes the modules into one bundle. The default React Native Metro config turns on `inlineRequires`, so modules load lazily at first use.[357] Metro has no tree shaking pass in the transform worker I checked (installed 0.84.3, unverified at 0.87), so unused exports stay in the bundle.[358]

Key points:

---

[351] tmp/rnw-ch0506/website/architecture/threading-model.md:61, render phase is interruptible.

[352] tmp/rnw-ch0506/docs/optimizing-javascript-loading.md:23, lazy loading with Suspense.

[353] tmp/rnw-ch0506/website/blog/2024-10-23-the-new-architecture-is-here.mdx:12, Suspense, Transitions, automatic batching, useLayoutEffect.

[354] tmp/rnw-ch0506/website/blog/2024-10-23-the-new-architecture-is-here.mdx:188 and :190, interop layer limits and Rules of React.

[355] tmp/rnw-ch0506/website/architecture/threading-model.md:61, render phase is interruptible.

[356] tmp/rnw-ch0506/website/blog/2024-10-23-the-new-architecture-is-here.mdx:188 and :190, interop layer limits and Rules of React.

[357] packages/metro-config/src/index.flow.js:73-77 (HEAD), and same flag at v0.88.1 line 95.

[358] node_modules/metro-transform-worker/src/index.js:180-208 (installed metro 0.84.3, yarn.lock pins 0.87.2, which could not be fetched), only inlinePlatform and constantFoldingPlugin. No tree shaking at 0.87 not checked directly (unverified).

- The React Native Metro config sets `inlineRequires: true` and `experimentalImportSupport: false` in `getTransformOptions`.[359]
- Inline requires move a `require` call to the place of first use. This cuts startup work. It applies to `require` calls, not to `import` statements, in the docs.[360]
- Expo projects do not inline requires by default. You can set `inlineRequires: true` yourself.[361]
- The transform worker does constant folding and platform inlining. I found no tree shaking pass (unverified at 0.87).[362]
- Source maps: on Android they are on by default in release. On iOS they are off by default and you set `SOURCEMAP_FILE`.[363]
- Use `metro-symbolicate` with the source map to read release stack traces.[364]

Example or code (only when it helps, max 15 lines):

```
// Proof run with metro-transform-plugins (see scratch/B/proof-9.txt).
// Input:  const a = require('./a'); export function f(){ return a.x(); }
// Output: export function f() { return require('./a').x(); }
```

Common wrong answer: "Metro tree shakes unused code like webpack." Wrong as far as I could check (unverified at 0.87). Dead exports stay in the bundle unless a framework adds a custom step.[365]

Likely follow-up: What is a pitfall of inline requires? Module side effects run later than you expect, at first use. You can block a module with `inlineRequires.blockList` or `nonInlinedRequires`.[366]

Q10. How do animations work? Animated with useNativeDriver, the C++ Native Animated canary, and Reanimated (worklets) at a high level.

Short answer (say this first, 2 to 4 sentences): With `Animated`, the default driver runs on the JS thread and sends new values each frame. With `useNativeDriver: true`, JS sends the animation graph to native once. Then native runs it on the UI thread, so a busy JS thread cannot drop frames. The C++ Native Animated option is a feature flag in canary stage. Reanimated runs worklets on the UI thread (general knowledge, not from this repo).

Key points:

[359] packages/metro-config/src/index.flow.js:73-77 (HEAD), and same flag at v0.88.1 line 95.

[360] tmp/rnw-ch0506/docs/optimizing-javascript-loading.md:88-156, inline requires, Expo default, blockList.

[361] tmp/rnw-ch0506/docs/optimizing-javascript-loading.md:88-156, inline requires, Expo default, blockList.

[362] node_modules/metro-transform-worker/src/index.js:180-208 (installed metro 0.84.3, yarn.lock pins 0.87.2, which could not be fetched), only inlinePlatform and constantFoldingPlugin. No tree shaking at 0.87 not checked directly (unverified).

[363] tmp/rnw-ch0506/docs/debugging-release-builds.md:23-72, source maps on Android and iOS, metro-symbolicate.

[364] tmp/rnw-ch0506/docs/debugging-release-builds.md:23-72, source maps on Android and iOS, metro-symbolicate.

[365] node_modules/metro-transform-worker/src/index.js:180-208 (installed metro 0.84.3, yarn.lock pins 0.87.2, which could not be fetched), only inlinePlatform and constantFoldingPlugin. No tree shaking at 0.87 not checked directly (unverified).

[366] tmp/rnw-ch0506/docs/optimizing-javascript-loading.md:88-156, inline requires, Expo default, blockList.

- `useNativeDriver` is required. If you omit it, `Animated` logs a warning that you must set it to `true` or `false`.[367]
- The native driver cannot animate every prop. It supports non-layout props such as transform and opacity.[368]
- The flag `cxxNativeAnimatedEnabled` uses a C++ implementation of Native Animated. Its default is `false` and its release stage is `canary`.[369]
- `useSharedAnimatedBackend` is an experimental flag, default `false`. It is what makes every prop drivable natively.[370]
- `animatedForceNativeDriver` forces the native driver for all animations, but only when `cxxNativeAnimatedEnabled` and `useSharedAnimatedBackend` are also on.[371]
- A value can use only one driver. If one animation on a value uses the native driver, all must.[372]
- Reanimated worklets are small JavaScript functions that run on the UI thread. Reanimated is a separate library (unverified here, no source in this repo).

Example or code (only when it helps, max 15 lines):

```
// Not compiled: needs a React Native runtime.
Animated.timing(opacity, {
  toValue: 1,
  duration: 300,
  useNativeDriver: true, // required
}).start();
```

Common wrong answer: "Set `useNativeDriver: true` on everything." Wrong. It fails for props the native driver cannot handle, such as layout props like `width`. Use it for transform and opacity.[373]

Likely follow-up: When do you pick Reanimated over Animated? Pick it for gesture-driven animation that must run fully on the UI thread, with shared values and worklets.

[367] packages/react-native/src/private/animated/NativeAnimatedHelper.js:418-439, warning (:433-436) and forced native driver (:418-425). Flag config line 1108.

[368] tmp/rnw-ch0506/docs/animations.md:540 (one driver per value) and :564 (only non-layout props, transform and opacity work, Flexbox and position do not).

[369] packages/react-native/scripts/featureflags/ReactNativeFeatureFlags.config.js:74-84, default false, canary. Same at v0.88.1.

[370] packages/react-native/scripts/featureflags/ReactNativeFeatureFlags.config.js:1009-1018, default false, experimental.

[371] packages/react-native/src/private/animated/NativeAnimatedHelper.js:418-439, warning (:433-436) and forced native driver (:418-425). Flag config line 1108.

[372] tmp/rnw-ch0506/docs/animations.md:540 (one driver per value) and :564 (only non-layout props, transform and opacity work, Flexbox and position do not).

[373] tmp/rnw-ch0506/docs/animations.md:540 (one driver per value) and :564 (only non-layout props, transform and opacity work, Flexbox and position do not).

Part 3: UI and performance

Q11. ScrollView vs FlatList vs SectionList vs FlashList: how does virtualization work, and which props matter?

Short answer (say this first, 2 to 4 sentences): ScrollView renders all its children at once. FlatList and SectionList sit on VirtualizedList, which renders only a window of items around the viewport and drops the rest. FlashList is a third-party list library built for performance. Use ScrollView for short, bounded content and a virtualized list for long data.[374][375]

Key points:

- ScrollView mounts every child, so long data costs memory and startup time.[376]
- VirtualizedList defaults: `windowSize` 21 (10 viewports above, 10 below, 1 visible), `initialNumToRender` 10, `maxToRenderPerBatch` 10, `updateCellsBatchingPeriod` 50 ms.[377][378]
- `getItemLayout` skips measurement when the item size is fixed. `keyExtractor` sets the React keys from your data, for example from `id`.[379]
- `removeClippedSubviews` defaults to `true` only on Android. The flag `shouldUseRemoveClippedSubviewsAsDefaultOnIOS` is `false` by default at HEAD.[380][381]
- A bigger `windowSize` means less blank space but more memory. A smaller one means the reverse.[382]
- SectionList is a wrapper over VirtualizedList. Its section headers stick on iOS by default (`stickySectionHeadersEnabled` is `Platform.OS === 'ios'`).[383][384]
- FlatList and SectionList are `PureComponent`. If `renderItem` reads state that is not in `data`, pass it as `extraData`, or the list does not re-render.[385]
- The docs name FlashList and Legend List as third-party options.[386]

Example or code (only when it helps, max 15 lines):

---

[374] tmp/refresh/scratch/ch11-15/rnw/docs/scrollview.md:14, ScrollView renders all children at once.

[375] tmp/refresh/scratch/ch11-15/rnw/docs/performance.md:58, names FlashList and Legend List as third-party lists.

[376] tmp/refresh/scratch/ch11-15/rnw/docs/scrollview.md:14, ScrollView renders all children at once.

[377] packages/virtualized-lists/Lists/VirtualizedListProps.js:332-334, `windowSize ?? 21`.

[378] packages/virtualized-lists/Lists/VirtualizedListProps.js:303-314 (`initialNumToRender` 10, `maxToRenderPerBatch` 10) and VirtualizedList.js:2110 (`updateCellsBatchingPeriod ?? 50`).

[379] tmp/refresh/scratch/ch11-15/rnw/docs/flatlist.md:338,517-523,153, FlatList wraps VirtualizedList, `getItemLayout`, `keyExtractor`.

[380] packages/react-native/Libraries/Lists/FlatList.js:161-168, `removeClippedSubviews` default.

[381] packages/react-native/scripts/featureflags/ReactNativeFeatureFlags.config.js:1217-1218, flag default false.

[382] tmp/refresh/scratch/ch11-15/rnw/docs/optimizing-flatlist-configuration.md:72-82, windowSize tradeoff.

[383] tmp/refresh/scratch/ch11-15/rnw/docs/sectionlist.md:88, SectionList wraps VirtualizedList. Source: packages/react-native/Libraries/Lists/SectionList.js:257, renders `VirtualizedSectionList`.

[384] tmp/refresh/scratch/ch11-15/rnw/docs/sectionlist.md:275, headers stick on iOS by default. Source: packages/react-native/Libraries/Lists/SectionList.js:250-251, `_stickySectionHeadersEnabled ?? Platform.OS === 'ios'`.

[385] tmp/refresh/scratch/ch11-15/rnw/docs/flatlist.md:341,509 and sectionlist.md:91,141, `PureComponent` and `extraData`. Source: packages/react-native/Libraries/Lists/FlatList.js:310, `class FlatList extends React.PureComponent`.

[386] tmp/refresh/scratch/ch11-15/rnw/docs/performance.md:58, names FlashList and Legend List as third-party lists.

```
// not compiled: needs a React Native runtime
<FlatList
  data={items}
  keyExtractor={item => item.id}
  getItemLayout={(_, i) => ({length: 64, offset: 64 * i, index: i})}
  windowSize={7}
  renderItem={({item}) => <Row item={item} />}
/>
```

Common wrong answer: "Use FlatList for everything." A short, bounded list is simpler and often faster in a ScrollView. Very large or complex lists may need FlashList or Legend List.

Likely follow-up: Why do you see blank areas on fast scroll? Items outside the render window are not mounted yet, so the render rate falls behind the scroll speed. Raise `windowSize` or `maxToRenderPerBatch`, or make rows cheaper.

Q12. How do you find and fix unnecessary re-renders? memo, useCallback, useMemo, and the React Compiler.

Short answer (say this first, 2 to 4 sentences): First I measure with the React Profiler in React Native DevTools to see which components render and why. Then I fix the cause: move state down, keep props stable, and use `memo`, `useMemo`, and `useCallback` where the profile shows a cost. The React Compiler can add this memoization automatically. The compiler is not part of the core repo. The app adds it. The docs and the DevTools frontend mention it.[387][388]

Key points:

- React Native DevTools includes the React Components and Profiler panels. The Profiler records component render timing and React commits.[389]
- `React.memo` skips a render when props are equal. A custom compare function can be the second argument.[390]
- Memoization only helps when props are stable. A new inline function or object each render breaks it.
- The React Compiler is a build-time Babel tool that memoizes automatically. The 0.78 release post says you install only the compiler package and add the Babel plugin.[391]
- Source proof: a grep for `react-compiler` and `babel-plugin-react-compiler` in the repo (outside `node_modules`, `tmp`, `.git`) finds 0 files. `node_modules` has `hermes-compiler` only.[392]

---

[387] tmp/refresh/scratch/ch11-15/rnw/docs/react-native-devtools.md:139,167-171, Components and Profiler panels.

[388] tmp/refresh/scratch/ch11-15/rnw/docs/react-native-devtools.md:153, React Compiler mention in the docs.

[389] tmp/refresh/scratch/ch11-15/rnw/docs/react-native-devtools.md:139,167-171, Components and Profiler panels.

[390] tmp/refresh/scratch/ch11-15/rnw/docs/optimizing-flatlist-configuration.md:96-118, `React.memo` with compare function.

[391] tmp/refresh/scratch/ch11-15/rnw/website/blog/2025-02-19-react-native-0.78.mdx:49-51, compiler setup in 0.78 (compiler only plus Babel plugin).

[392] tmp/refresh/interview/scratch/C/proof-12.txt, grep count 0 for `react-compiler` in repo, only `hermes-compiler` in node_modules.

- The DevTools docs say compiler-optimized components show a "Memo ✨" badge. The bundled DevTools frontend has the tooltip "This component has been auto-memoized by the React Compiler."[393]
- The compiler is opt-in per app. Core RN does not enable it by default.[394]

Example or code (only when it helps, max 15 lines):

```
// not compiled: needs a React Native runtime
const Row = memo(
  function Row({title}) { return <Text>{title}</Text>; },
  (prev, next) => prev.title === next.title,
);
```

Common wrong answer: "Wrap everything in useMemo and useCallback." That adds cost and noise without a profile. Also wrong: "React Native ships the React Compiler." The repo has no compiler. The app adds the Babel plugin.

Likely follow-up: Does the compiler remove the need for `memo`? In most components yes, because it memoizes for you. Manual hooks stay useful for code the compiler skips.

Q13. How does layout work? Yoga, flexbox defaults that differ from the web, and the gap and percentage rules.

Short answer (say this first, 2 to 4 sentences): Yoga is the C++ layout engine. It computes flexbox layout in the native shadow tree, not in JS. The defaults differ from the web: `flexDirection` is `column`, `alignContent` is `flex-start`, and `flexShrink` is `0`. `gap`, `rowGap`, and `columnGap` work like CSS, and percentage values are supported on the New Architecture.[395][396]

Key points:

- Yoga source defaults: `flexDirection` `Column`, `alignContent` `FlexStart`, `alignItems` `Stretch`, `flexWrap` `NoWrap`, `boxSizing` `BorderBox`.[397]
- `flexShrink` is `0` by default. The web value `1` is used only when Yoga `useWebDefaults` is on, and that is off by default in the Yoga `Config`.[398]
- The `flex` prop accepts a single number.[399]

---

[393] tmp/refresh/scratch/ch11-15/rnw/docs/react-native-devtools.md:153, "Memo ✨" badge. Source: packages/debugger-frontend/dist/third-party/front_end/third_party/react-devtools/react-devtools.js, string "✨ This component has been auto-memoized by the React Compiler." (grep, minified, no stable line).

[394] tmp/refresh/interview/scratch/C/proof-12.txt, grep count 0 for `react-compiler` in repo, only `hermes-compiler` in node_modules.

[395] tmp/refresh/scratch/ch11-15/rnw/docs/flexbox.md:14, the three default differences and single-number `flex`.

[396] packages/react-native/ReactCommon/react/renderer/components/view/YogaLayoutableShadowNode.cpp:72-73, Fabric Yoga config and node.

[397] packages/react-native/ReactCommon/yoga/yoga/style/Style.h:922-935, Yoga defaults (`flexDirection_` Column at 922-923, `alignContent_` FlexStart at 927, `alignItems_` Stretch at 928, `flexWrap_` NoWrap, `boxSizing_` BorderBox at 935).

[398] packages/react-native/ReactCommon/yoga/yoga/style/Style.h:48-49, YGNodeStyle.cpp:201 and node/Node.cpp:488-489, `DefaultFlexShrink` 0, web 1 with `useWebDefaults`. Config.h:86 default false. Nothing under ReactCommon/react/renderer calls `setUseWebDefaults` (grep).

[399] tmp/refresh/scratch/ch11-15/rnw/docs/flexbox.md:14, the three default differences and single-number `flex`.

- `gap` is a shorthand for `rowGap` and `columnGap`. Gap goes between children, not at the edges.[400]
- Gap and layout values accept points or percentage strings such as `'10%'`. Percentages need a parent with a defined size.[401][402]
- Fabric builds a Yoga node per shadow node and sets Yoga style from props.[403]

Example or code (only when it helps, max 15 lines):

```
// not compiled: needs a React Native runtime
const styles = StyleSheet.create({
  row: {flexDirection: 'row', gap: 8, flexWrap: 'wrap'},
  half: {width: '50%'}, // parent needs a defined width
});
```

Common wrong answer: "Layout runs on the JS thread." Yoga runs in native C++ in the shadow tree. Another wrong answer: "Defaults match the web." Three defaults differ.

Likely follow-up: Why does my `width: '50%'` give zero? The parent has no defined size. Percentages resolve against the parent.

Q14. How do you write platform-specific code and styles? Platform.select, .ios/.android files, and the native-only file resolution in Metro.

Short answer (say this first, 2 to 4 sentences): Use `Platform.OS` for small branches and `Platform.select` for values. Use `.ios.js` and `.android.js` files for larger differences. Use `.native.js` for code shared by iOS and Android but not the web. Metro tries the platform extension first, then `.native`, then the plain file.[404][405]

Key points:

- `Platform.select` takes the keys `ios`, `android`, `native`, and `default`.[406]
- On iOS the lookup is: `ios`, then `native`, then `default`. On Android it is: `android`, then `native`, then `default`.[407]
- Metro resolver order for `file`: `file.<platform>.js`, then `file.native.js` (when `preferNativePlatform` is set), then `file.js`.[408]
- Metro config default platforms are `['android', 'ios']`.[409]

---

[400] packages/react-native/Libraries/StyleSheet/StyleSheetTypes.js:650-660, gap semantics.

[401] tmp/refresh/scratch/ch11-15/rnw/docs/flexbox.md:2087, gap accepts numbers or percentage strings on the New Architecture.

[402] tmp/refresh/scratch/ch11-15/rnw/docs/height-and-width.md:83, percentage dimensions need a parent with a defined size.

[403] packages/react-native/ReactCommon/react/renderer/components/view/YogaStylableProps.cpp:125-141, props set Yoga style (`REBUILD_FIELD_SWITCH_CASE_YSP`, `REBUILD_FIELD_YG_GUTTER(gap, ...)` at 141).

[404] tmp/refresh/scratch/ch11-15/rnw/docs/platform-specific-code.md:29,123-138, `select` keys and `.native.js`.

[405] node_modules/metro-resolver/src/resolve.js:454-469, platform, then `.native`, then plain extension (metro 0.84.3).

[406] tmp/refresh/scratch/ch11-15/rnw/docs/platform-specific-code.md:29,123-138, `select` keys and `.native.js`.

[407] packages/react-native/Libraries/Utilities/Platform.ios.js:80-82 and Platform.android.js:73-82, `select` fallback order.

[408] node_modules/metro-resolver/src/resolve.js:454-469, platform, then `.native`, then plain extension (metro 0.84.3).

[409] packages/metro-config/src/index.flow.js:39, `platforms: ['android', 'ios']`.

- The docs advise configuring the web bundler to ignore `.native.js` files.[410]
- Platform file resolution happens at bundle time, so the other platform file is not in the bundle.
- Metro inlines `Platform.OS` and `Platform.select` per target platform (`inlinePlatform: true`). Constant folding drops dead branches only when `dev` is off.[411]

Example or code (only when it helps, max 15 lines):

```
// not compiled: needs a React Native runtime
const shadow = Platform.select({
  ios: {shadowOpacity: 0.2},
  android: {elevation: 4},
  default: {},
});
// Button.ios.js, Button.android.js, import './Button'
```

Common wrong answer: "`Platform.OS` checks always keep both branches in the bundle." Metro runs an inline-platform Babel step that replaces `Platform.OS` and `Platform.select` with the target platform value. In a non-dev build, constant folding then removes the dead branch. Still, a platform file is the clearer choice for large differences, because the other platform's code is never parsed.[412]

Likely follow-up: When does `.native.js` win over `.js`? When Metro bundles for a native platform and no `.ios.js` or `.android.js` file exists.

Q15. How do you profile and debug performance? React Native DevTools, Perf Monitor, Hermes sampling profiler, and the React Profiler.

Short answer (say this first, 2 to 4 sentences): I profile a release-like build with Development Mode off. I use React Native DevTools for JS: the Performance panel (since 0.83), the React Profiler, and the Memory panel. I use the Perf Monitor for a quick view of JS and UI frame rates. For native jank I use Android Studio system tracing or Xcode Instruments.[413][414]

Key points:

- Dev mode is slow. Always measure with `dev` off.[415]
- The Perf Monitor shows two frame rates (JS and UI). Open it from the Dev Menu.[416]

---

[410] tmp/refresh/scratch/ch11-15/rnw/docs/platform-specific-code.md:29,123-138, `select` keys and `.native.js`.

[411] node_modules/metro/src/lib/transformHelpers.js:34 (`inlinePlatform: true`), node_modules/metro-transform-worker/src/index.js:~198-210 (`constantFoldingPlugin` only `if (!options.dev)`), packages/react-native-babel-preset/src/inline-platform-plugin.js. Run proof: tmp/refresh/interview/scratch/FC-2/plat.js (input `Platform.OS === 'ios' ? A : B` and `Platform.select`, output `'IOS_ONLY'` and `'SEL_IOS'` after the plugin and constant folding).

[412] node_modules/metro/src/lib/transformHelpers.js:34 (`inlinePlatform: true`), node_modules/metro-transform-worker/src/index.js:~198-210 (`constantFoldingPlugin` only `if (!options.dev)`), packages/react-native-babel-preset/src/inline-platform-plugin.js. Run proof: tmp/refresh/interview/scratch/FC-2/plat.js (input `Platform.OS === 'ios' ? A : B` and `Platform.select`, output `'IOS_ONLY'` and `'SEL_IOS'` after the plugin and constant folding).

[413] tmp/refresh/scratch/ch11-15/rnw/docs/react-native-devtools.md:107-116, Performance panel since 0.83.

[414] tmp/refresh/scratch/ch11-15/rnw/docs/profiling.md:8,19, Instruments, Android Studio Profiler, `systrace` removed.

[415] tmp/refresh/scratch/ch11-15/rnw/docs/profiling.md:10, make sure Development Mode is off.

- The DevTools Performance panel (since 0.83) shows JS execution, React Performance tracks, Network events, and User Timings in one timeline.[417]
- The React Profiler panel records component render timing and commits. The Memory panel takes heap snapshots.[418][419]
- The standalone `systrace` tool is removed. Use the Android Studio Profiler.[420]
- `PerformanceObserver` and `performance` APIs give runtime telemetry.[421]
- A separate Hermes sampling profiler doc page is not in the checked docs (unverified). The DevTools Performance panel is the current JS profiling path.

Example or code (only when it helps, max 15 lines):

```
// not compiled: needs a React Native runtime
performance.mark('list-start');
// ... work ...
performance.measure('list', 'list-start'); // shows in the Performance panel
```

Common wrong answer: "Profile in dev mode with the debugger attached." Dev mode adds overhead and gives false numbers. Profile a release or profileable build.

Likely follow-up: JS FPS is fine but the UI drops frames. What now? The cost is native, so I trace with Android Studio system tracing or Instruments.

Part 4: Native code and ecosystem

Q16. How do you write a TurboModule end to end (spec, codegen, native implementation, JS call)?

Short answer (say this first, 2 to 4 sentences): I write a typed JS spec file that extends `TurboModule`. Codegen reads that spec at build time and generates native interfaces (C++, Objective-C++, Java). I implement the generated interface in Objective-C++ and Kotlin or Java, then register the module. In JS I call it through `TurboModuleRegistry`, with synchronous or Promise-returning methods.[422][423]

Key points:

- The spec is a `Native<Name>.ts` or `.js` file. It exports `interface Spec extends TurboModule` and a default export from `TurboModuleRegistry.get<Spec>('Name')` or `getEnforcing<Spec>('Name')`.[424][425]

---

[416] tmp/refresh/scratch/ch11-15/rnw/docs/performance.md:14 and debugging.md:98, Perf Monitor and two frame rates.

[417] tmp/refresh/scratch/ch11-15/rnw/docs/react-native-devtools.md:107-116, Performance panel since 0.83.

[418] tmp/refresh/scratch/ch11-15/rnw/docs/react-native-devtools.md:167-171, React Profiler panel.

[419] tmp/refresh/scratch/ch11-15/rnw/docs/react-native-devtools.md:124-128, Memory panel.

[420] tmp/refresh/scratch/ch11-15/rnw/docs/profiling.md:8,19, Instruments, Android Studio Profiler, `systrace` removed.

[421] tmp/refresh/scratch/ch11-15/rnw/docs/react-native-devtools.md:107-116, Performance panel since 0.83.

[422] tmp/refresh/scratch/ch11-15/rnw/docs/turbo-native-modules.md:61-74, official spec example with `Spec extends TurboModule` and `getEnforcing`.

[423] tmp/refresh/scratch/ch11-15/rnw/docs/turbo-native-modules.md:104-110, `codegenConfig` keys `name`, `type`, `jsSrcsDir`, `android.javaPackageName`.

- `getEnforcing` throws an invariant error when the native module is missing. `get` returns `null`.[426]
- You tell Codegen where the specs are in the app or library `package.json`, under `codegenConfig`: `name`, `type` (`modules`, `components` or `all`), `jsSrcsDir`, and `android.javaPackageName`.[427][428]
- Android: your class extends the generated `<Name>Spec` and a `BaseReactPackage` registers it. iOS: your Objective-C++ class conforms to the generated `<Name>Spec` protocol. The iOS build needs `pod install` to run Codegen.[429]
- In 0.88 there is no old Bridge module path to fall back to. `TurboCxxModule`, `RCTCxxModule` and the CxxModule support were removed, so pure C++ modules use the TurboModule C++ API.[430]

Example or code (only when it helps, max 15 lines):

```
// specs/NativeLocalStorage.ts  (not compiled here, shape from the official docs)
import type {TurboModule} from 'react-native';
import {TurboModuleRegistry} from 'react-native';

export interface Spec extends TurboModule {
  setItem(value: string, key: string): void;
  getItem(key: string): string | null;
}
export default TurboModuleRegistry.getEnforcing<Spec>('NativeLocalStorage');
// package.json: "codegenConfig": {"name": "NativeLocalStorageSpec",
//   "type": "modules", "jsSrcsDir": "specs",
//   "android": {"javaPackageName": "com.nativelocalstorage"}}
```

Common wrong answer: "A native module sends JSON over the async bridge, so I register it with `NativeModules`." This is outdated. Modules no longer use the async Bridge in 0.88. `NativeModules` only reaches legacy modules through the interop layer. A TurboModule is a typed JSI host object, and Codegen produces the glue.[431][432]

Likely follow-up: Can a method return a value synchronously? Yes. A spec method with a plain return type is synchronous over JSI, and a `Promise<T>` return makes it async. Use synchronous

---

[424] tmp/refresh/scratch/ch11-15/rnw/docs/turbo-native-modules.md:61-74, official spec example with `Spec extends TurboModule` and `getEnforcing`.

[425] packages/react-native/Libraries/TurboModule/TurboModuleRegistry.js:35-47, `get` returns null and `getEnforcing` throws via invariant (HEAD 9c7b0f5fafd).

[426] packages/react-native/Libraries/TurboModule/TurboModuleRegistry.js:35-47, `get` returns null and `getEnforcing` throws via invariant (HEAD 9c7b0f5fafd).

[427] tmp/refresh/scratch/ch11-15/rnw/docs/turbo-native-modules.md:104-110, `codegenConfig` keys `name`, `type`, `jsSrcsDir`, `android.javaPackageName`.

[428] packages/rn-tester/package.json:41-47, real `codegenConfig` with `name`, `type: all`, `jsSrcsDir`.

[429] tmp/refresh/scratch/ch11-15/rnw/docs/turbo-native-modules-android.md:10-25,124-133 and turbo-native-modules-ios.md:53-60, generated `NativeLocalStorageSpec`, `BaseReactPackage`. Docs line turbo-native-modules.md:138 shows `bundle exec pod install`.

[430] CHANGELOG.md:1041-1045 (v0.84.0 Breaking, "Remove TurboCxxModule", "Remove RCTCxxModule", "Remove CxxModule support from CatalystInstance").

[431] CHANGELOG.md:1041-1045 (v0.84.0 Breaking, "Remove TurboCxxModule", "Remove RCTCxxModule", "Remove CxxModule support from CatalystInstance").

[432] https://reactnative.dev/docs/turbo-native-modules-introduction (external docs, TurboModule model). Local copy: tmp/refresh/scratch/ch11-15/rnw/docs/turbo-native-modules.md.

calls only for cheap work.[433][434]

Q17. How do you write a Fabric native component, and how are props, commands, and events passed?

Short answer (say this first, 2 to 4 sentences): I write a JS spec with `codegenNativeComponent<NativeProps>('Name')`. `NativeProps` lists props and event handlers. Commands come from `codegenNativeCommands`. Codegen builds the C++ Props, State and EventEmitter classes and the Android ViewManager interface and delegate. I then write the native view: an `RCTViewComponentView` subclass on iOS, and a `SimpleViewManager` on Android.[435][436][437]

Key points:

- Props: `NativeProps` extends `ViewProps`. Codegen types such as `CodegenTypes.Float`, `Int32` and `Double` map to native types.[438]
- Events: a prop typed `CodegenTypes.BubblingEventHandler<T>` (or `DirectEventHandler<T>`) is a typed event. The native side emits it. On iOS you call `self.eventEmitter.onX(...)`. On Android you dispatch through the `eventDispatcher`.[439][440][441]
- Commands: you export `Commands = codegenNativeCommands<NativeCommands>({supportedCommands: [...]})`. JS calls `Commands.reload(ref)`. The first argument is always the view ref.[442][443]
- iOS maps the component name to the class through `codegenConfig.ios.componentProvider`. Android registers the manager in a `BaseReactPackage`.[444][445]

---

[433] tmp/refresh/scratch/ch11-15/rnw/docs/turbo-native-modules.md:61-74, official spec example with `Spec extends TurboModule` and `getEnforcing`.

[434] packages/react-native-codegen/src/generators/modules/GenerateModuleObjCpp/serializeMethod.js:118-121, a `PromiseTypeAnnotation` return adds `resolve` and `reject` blocks. Other return types are plain returns.

[435] packages/rn-tester/NativeComponentExample/js/MyNativeViewNativeComponent.js:13-95, spec with `CodegenTypes.Float`, `BubblingEventHandler`, `codegenNativeCommands`, `codegenNativeComponent` (HEAD).

[436] tmp/refresh/scratch/ch11-15/rnw/docs/fabric-native-components.md:62-117, official spec walk-through.

[437] tmp/refresh/scratch/ch11-15/rnw/docs/fabric-native-components-ios.md:77-82,127-159, `RCTViewComponentView`, `updateProps:oldProps:`, `self.eventEmitter.onScriptLoaded`.

[438] packages/rn-tester/NativeComponentExample/js/MyNativeViewNativeComponent.js:13-95, spec with `CodegenTypes.Float`, `BubblingEventHandler`, `codegenNativeCommands`, `codegenNativeComponent` (HEAD).

[439] packages/rn-tester/NativeComponentExample/js/MyNativeViewNativeComponent.js:13-95, spec with `CodegenTypes.Float`, `BubblingEventHandler`, `codegenNativeCommands`, `codegenNativeComponent` (HEAD).

[440] tmp/refresh/scratch/ch11-15/rnw/docs/fabric-native-components-ios.md:77-82,127-159, `RCTViewComponentView`, `updateProps:oldProps:`, `self.eventEmitter.onScriptLoaded`.

[441] tmp/refresh/scratch/ch11-15/rnw/docs/fabric-native-components-android.md:76-85,197, `eventDispatcher.dispatchEvent`.

[442] packages/rn-tester/NativeComponentExample/js/MyNativeViewNativeComponent.js:13-95, spec with `CodegenTypes.Float`, `BubblingEventHandler`, `codegenNativeCommands`, `codegenNativeComponent` (HEAD).

[443] tmp/refresh/scratch/ch11-15/rnw/docs/the-new-architecture/fabric-component-native-commands.md:20-50, `codegenNativeCommands` with `supportedCommands: ['reload']`.

[444] tmp/refresh/scratch/ch11-15/rnw/docs/fabric-native-components.md:129-138, `ios.componentProvider` and `android.javaPackageName`.

[445] tmp/refresh/scratch/ch11-15/rnw/docs/fabric-native-components-android.md:344-366, component registered in a `BaseReactPackage`.

- Without `ios.componentProvider`, Codegen tries to find the class by crawling `.mm` files and logs a deprecation warning.[446]
- Props update on iOS in `updateProps:oldProps:`. You compare old and new props and call `[super updateProps:...]` last.[447]

Example or code (only when it helps, max 15 lines):

```
// specs/WebViewNativeComponent.ts (not compiled here)
import type {CodegenTypes, HostComponent, ViewProps} from 'react-native';
import {codegenNativeCommands, codegenNativeComponent} from 'react-native';

export interface NativeProps extends ViewProps {
  sourceURL?: string;
  onScriptLoaded?: CodegenTypes.BubblingEventHandler<{result: 'success' |
'error'}> | null;
}
interface NativeCommands { reload: (ref:
React.ElementRef<HostComponent<NativeProps>>) => void }
export const Commands = codegenNativeCommands<NativeCommands>({supportedCommands:
['reload']});
export default codegenNativeComponent<NativeProps>('CustomWebView') as
HostComponent<NativeProps>;
```

Common wrong answer: "Use `requireNativeComponent` and a `RCTViewManager` with `RCT_EXPORT_VIEW_PROPERTY`." That is the legacy way. It still loads through an interop layer, but new code uses a Codegen spec.[448]

Likely follow-up: Why is the view ref the first argument of a command? Because Fabric dispatches the command to the native view for that shadow node, so JS must pass the host component instance.[449]

Q18. Expo vs bare React Native CLI: what are the real differences in 2026 (prebuild, dev builds, EAS, config plugins)?

Short answer (say this first, 2 to 4 sentences): Both build the same React Native app. The difference is who owns the native folders. With Expo and Continuous Native Generation (CNG), `npx expo prebuild` generates `android/` and `ios/` from a template, `app.json` and config plugins.

---

[446] packages/react-native/scripts/codegen/generate-artifacts-executor/generateRCTThirdPartyComponents.js:120-131, `[DEPRECATED] ... should add the 'ios.componentProvider' property`.

[447] tmp/refresh/scratch/ch11-15/rnw/docs/fabric-native-components-ios.md:77-82,127-159, `RCTViewComponentView`, `updateProps:oldProps:`, `self.eventEmitter.onScriptLoaded`.

[448] CHANGELOG.md:319-321 (v0.87.0), legacy view manager interop still exists and is gated by `RCT_REMOVE_LEGACY_COMPONENT_INTEROP`.

[449] tmp/refresh/scratch/ch11-15/rnw/docs/the-new-architecture/fabric-component-native-commands.md:20-50, `codegenNativeCommands` with `supportedCommands: ['reload']`.

With a bare app, you commit and edit those folders by hand. Expo says "managed vs bare" is deprecated. Expo SDK packages and EAS work in both.[450][451]

Key points (facts from external Expo docs, not from this repo):

- Prebuild is optional. If `android/` and `ios/` exist, EAS Build does not run prebuild. If they are absent, EAS Build runs it.[452]
- Config plugins are functions that change native projects during prebuild. They replace hand edits of `Info.plist` or `AndroidManifest.xml`. `prebuild --clean` deletes and recreates the native folders.[453]
- A development build is your own app with `expo-dev-client`. You need one when you use native code that Expo Go does not contain. Build it with `npx expo run:ios|android` (local) or `eas build --profile development` (cloud, or `--local`).[454]
- Rebuild the native app after you add a native library, change app config, or upgrade the SDK. JS-only changes need only `npx expo start`.[455]
- Expo SDK 58 targets React Native 0.88 and React 19.3.0, with Node 22.13 or newer. Each SDK targets one RN version.[456]
- Facts from this repo: a bare app has `newArchEnabled=true` in `gradle.properties` and no opt-out, so the New Architecture rule is the same in both.[457][458]

Example or code (only when it helps, max 15 lines):

```
# Expo, external docs. Not run here.
npx expo install expo-dev-client
npx expo prebuild --clean
npx expo run:ios
eas build --platform ios --profile development
```

Common wrong answer: "Expo is a limited sandbox, and you must eject to use native code." This is outdated. Expo calls ejecting deprecated. You add native code with modules and config plugins, and the Expo SDK works in any RN app.[459][460]

---

[450] https://docs.expo.dev/workflow/continuous-native-generation.md (external, fetched 2026-10-29, page dated September 25, 2026), CNG, prebuild, `--clean`, EAS Build behavior, brownfield note, "managed vs bare deprecated".

[451] https://docs.expo.dev/versions/latest/ (external), "All Expo SDK packages work in any React Native app", `npx install-expo-modules`.

[452] https://docs.expo.dev/workflow/continuous-native-generation.md (external, fetched 2026-10-29, page dated September 25, 2026), CNG, prebuild, `--clean`, EAS Build behavior, brownfield note, "managed vs bare deprecated".

[453] https://docs.expo.dev/workflow/continuous-native-generation.md (external, fetched 2026-10-29, page dated September 25, 2026), CNG, prebuild, `--clean`, EAS Build behavior, brownfield note, "managed vs bare deprecated".

[454] https://docs.expo.dev/develop/development-builds/introduction.md (external), dev build commands and when to rebuild.

[455] https://docs.expo.dev/develop/development-builds/introduction.md (external), dev build commands and when to rebuild.

[456] https://docs.expo.dev/versions/latest/ (external), table row "58.0.0 | 0.88 | 19.3.0 | ... | 22.13.x".

[457] private/helloworld/android/gradle.properties:11, `newArchEnabled=true`.

[458] CHANGELOG.md:1500,1505 (v0.82.0 Breaking), opt-out removed on Android and iOS.

[459] https://docs.expo.dev/workflow/continuous-native-generation.md (external, fetched 2026-10-29, page dated September 25, 2026), CNG, prebuild, `--clean`, EAS Build behavior, brownfield note, "managed vs bare deprecated".

Likely follow-up: When do you pick bare? When you must hand-maintain native projects, such as a brownfield app. Expo says CNG is not meant for existing brownfield projects.[461]

Q19. How do you handle navigation, deep links, and app state in a production app? (React Navigation native stack, linking config, AppState)

Short answer (say this first, 2 to 4 sentences): I use React Navigation with the native stack, which uses `UINavigationController` on iOS and `Fragment` on Android. I pass a `linking` config with `prefixes` and a `config` map from URL paths to screens, and I register the scheme or universal link on each platform. For app state I use `AppState` (`active`, `background`, `inactive`) and subscribe in an effect, then remove the subscription on cleanup.[462][463][464]

Key points (React Navigation facts are from external docs, current page is 7.x):

- Native stack: platform APIs handle animation and gestures. It is a good default, but it supports fewer custom behaviors than the JS stack. Bugs belong in `react-native-screens`.[465]
- Deep links have two cases: cold start (set the initial state) and warm (update state). The `linking` prop handles both. Do not drive links through a `ref`.[466]
- Setup: Android needs an `intent-filter` with `VIEW`, `launchMode="singleTask"` and `autoVerify="true"` for App Links. iOS needs the URL scheme, and Associated Domains for universal links.[467]
- For push notifications, override `getInitialURL` and `subscribe` in the `linking` object.[468]
- Core RN `Linking` has `getInitialURL()`, `addEventListener('url', ...)`, `openURL` and `canOpenURL`.[469]
- `AppState.addEventListener` accepts `'change'`, `'focus'`, `'blur'` and `'memoryWarning'`. The status type is `'inactive' | 'background' | 'active' | 'extension' | 'unknown'`. `focus` and `blur` are Android-only (AppState docs, appstate.md:94,98).[470]

---

[460] https://docs.expo.dev/versions/latest/ (external), "All Expo SDK packages work in any React Native app", `npx install-expo-modules`.

[461] https://docs.expo.dev/workflow/continuous-native-generation.md (external, fetched 2026-10-29, page dated September 25, 2026), CNG, prebuild, `--clean`, EAS Build behavior, brownfield note, "managed vs bare deprecated".

[462] https://reactnavigation.org/docs/native-stack-navigator (external, 7.x), `UINavigationController`, `Fragment`, issues go to react-native-screens.

[463] https://reactnavigation.org/docs/deep-linking/ (external, 7.x), `linking` prop, `prefixes`, cold and warm cases, `singleTask`, `autoVerify`, `getInitialURL`, `subscribe`, test commands.

[464] https://reactnavigation.org/docs/configuring-links (external), the `config` map from paths to screens (page not fetched, referenced from the deep-linking page).

[465] https://reactnavigation.org/docs/native-stack-navigator (external, 7.x), `UINavigationController`, `Fragment`, issues go to react-native-screens.

[466] https://reactnavigation.org/docs/deep-linking/ (external, 7.x), `linking` prop, `prefixes`, cold and warm cases, `singleTask`, `autoVerify`, `getInitialURL`, `subscribe`, test commands.

[467] https://reactnavigation.org/docs/deep-linking/ (external, 7.x), `linking` prop, `prefixes`, cold and warm cases, `singleTask`, `autoVerify`, `getInitialURL`, `subscribe`, test commands.

[468] https://reactnavigation.org/docs/deep-linking/ (external, 7.x), `linking` prop, `prefixes`, cold and warm cases, `singleTask`, `autoVerify`, `getInitialURL`, `subscribe`, test commands.

[469] packages/react-native/Libraries/Linking/Linking.js:32,49,64,90, `addEventListener`, `openURL`, `canOpenURL`, `getInitialURL` (HEAD).

Example or code (only when it helps, max 15 lines):

```
// not compiled here
useEffect(() => {
  const sub = AppState.addEventListener('change', (state) => {
    if (state === 'active') refreshSession();
  });
  return () => sub.remove();
}, []);
```

Common wrong answer: "Read `AppState.currentState` once on mount and assume it stays current." The value changes. Subscribe to `'change'` and clean up. Also, a deep link on a cold start arrives through `getInitialURL`, not through the `url` event.[471][472]

Likely follow-up: How do you test a deep link? Use `xcrun simctl openurl booted "example://chat/jane"` on iOS or `adb shell am start -W -a android.intent.action.VIEW -d "example://chat/jane" <package>` on Android.[473]

Q20. How do you upgrade a React Native app and its libraries to a new version (0.81 to 0.88)? Cover the upgrade helper, library compatibility, the interop layer, and breaking removals.

Short answer (say this first, 2 to 4 sentences): I upgrade one minor at a time, or jump with a diff. I open the Upgrade Helper, pick the from and to versions, then update `package.json` and the native template files. Then I check every native library for New Architecture support, because 0.82 removed the opt-out. I read the Breaking sections of the changelog for each version in between. With Expo, I bump the SDK, run `npx expo install --fix`, and re-run prebuild.[474][475][476]

Key points:

- The Upgrade Helper shows a diff of the template between two versions: `package.json`, then the other generated files.[477]

---

[470] packages/react-native/Libraries/AppState/AppState.js:31-32,124-160, status type and event names `change`, `memoryWarning`, `blur`, `focus` (HEAD).

[471] packages/react-native/Libraries/Linking/Linking.js:32,49,64,90, `addEventListener`, `openURL`, `canOpenURL`, `getInitialURL` (HEAD).

[472] packages/react-native/Libraries/AppState/AppState.js:31-32,124-160, status type and event names `change`, `memoryWarning`, `blur`, `focus` (HEAD).

[473] https://reactnavigation.org/docs/deep-linking/ (external, 7.x), `linking` prop, `prefixes`, cold and warm cases, `singleTask`, `autoVerify`, `getInitialURL`, `subscribe`, test commands.

[474] tmp/refresh/scratch/ch11-15/rnw/docs/upgrading.md:16-51, Upgrade Helper flow, URL https://react-native-community.github.io/upgrade-helper/.

[475] https://docs.expo.dev/workflow/continuous-native-generation.md (external), "upgrade npm dependencies, app config, and re-run `npx expo prebuild --clean`". The `expo install --fix` command is from https://docs.expo.dev/versions/latest/ (canary section).

[476] CHANGELOG.md:1500,1505 (v0.82.0 Breaking), Android and iOS opt-out removed.

[477] tmp/refresh/scratch/ch11-15/rnw/docs/upgrading.md:16-51, Upgrade Helper flow, URL https://react-native-community.github.io/upgrade-helper/.

- 0.82: no opt-out of the New Architecture on Android (`newArchEnabled=false`) or on iOS. A library that only works on the old architecture must be replaced or updated first.[478]
- 0.84: Hermes V1 is the default engine. Legacy-architecture iOS code is no longer included by default. `unstable_shouldEnableLegacyModuleInterop()` was removed, and the CxxModule family was removed.[479]
- 0.85: the Jest preset moved to the `react-native/jest-preset` package. Support for EOL Node.js lines was dropped.[480]
- 0.87: Node 22.13 or newer. The default JS API is the Strict TypeScript API (opt out with `customConditions: ["react-native-legacy-deep-imports"]`). AGP 9 on Android. Deep imports of `react-native/src/private/...` lose types.[481]
- The interop layers (legacy view managers and legacy modules) still exist, but 0.87 put them behind `RCT_REMOVE_LEGACY_COMPONENT_INTEROP` and `RCT_REMOVE_LEGACY_MODULE_INTEROP` so an app can compile them out. Treat the interop layer as a bridge for old libraries, not a destination.[482]

Example or code (only when it helps, max 15 lines):

```
# bare app, not run here
yarn add react-native@0.88 react@<version from Upgrade Helper>
# then apply the template diff from the Upgrade Helper
# Expo app, external docs
npx expo install expo@^58 --fix   # (check the SDK 58 upgrade guide)
npx expo prebuild --clean
```

Common wrong answer: "Set `newArchEnabled=false` until libraries catch up." This does not work from 0.82. The flag opt-out was removed.[483]

Likely follow-up: How do you find incompatible libraries before you upgrade? Check each library's release notes for New Architecture support, run a test build on a branch, and use the interop layer only as a short-term fix.[484][485]

---

[478] CHANGELOG.md:1500,1505 (v0.82.0 Breaking), Android and iOS opt-out removed.

[479] CHANGELOG.md:1038 (Hermes V1 default), 1042-1045 (CxxModule removals), 1054 (`unstable_shouldEnableLegacyModuleInterop` removed), 1059 (iOS "Remove legacy arch by default"), all in v0.84.0.

[480] CHANGELOG.md:833 (jest preset package), 832 Node drop line, v0.85.0 Breaking.

[481] CHANGELOG.md:285 (Node >= 22.13.0), 292 (Strict TypeScript API), 296 (AGP v9), v0.87.0 Breaking. The deep import restriction is CHANGELOG.md:278, same Breaking list.

[482] CHANGELOG.md:319-321 (v0.87.0), interop gates `RCT_REMOVE_LEGACY_COMPONENT_INTEROP` and `RCT_REMOVE_LEGACY_MODULE_INTEROP`.

[483] CHANGELOG.md:1500,1505 (v0.82.0 Breaking), Android and iOS opt-out removed.

[484] tmp/refresh/scratch/ch11-15/rnw/docs/upgrading.md:16-51, Upgrade Helper flow, URL https://react-native-community.github.io/upgrade-helper/.

[485] CHANGELOG.md:319-321 (v0.87.0), interop gates `RCT_REMOVE_LEGACY_COMPONENT_INTEROP` and `RCT_REMOVE_LEGACY_MODULE_INTEROP`.

Part 5: Testing

Q21. How do you structure a test strategy for a React Native app? Pyramid or trophy, and what runs where?

Short answer (say this first, 2 to 4 sentences): I use static checks as the base, then many Jest component and integration tests, then a few end-to-end flows on a real build. The pyramid says "many unit tests". The trophy says "most tests at the integration level". For a mobile app I prefer the trophy, because most bugs sit where components, state, and the network meet. Both models are opinions, not rules.[486]

Key points:

- Static layer: TypeScript and ESLint. It runs in seconds on Linux.[487]
- Unit and component layer: Jest with the React Native preset and React Native Testing Library. It runs in Node, with no device. Layout does not run there.[488][489]
- End-to-end layer: Maestro, Detox, or Appium on a simulator or emulator. It is slow and costly, so I cover only the critical journeys.[490]
- Order CI jobs from cheap to costly: lint and types, then Jest, then build, then E2E smoke, then a release gate.[491]
- The React Native repo tests itself in three layers: Jest in `__tests__` folders, Fantom (`*-itest.js`) for engine internals, and the RNTester app. Fantom is for React Native contributors, not for app teams.[492]

Example or code (only when it helps, max 15 lines):

```
# Local order of checks (not run as one script)
yarn tsc --noEmit
yarn eslint . --max-warnings 0
yarn jest --ci
maestro test .maestro/smoke.yaml   # needs a device, not run here
```

Common wrong answer: "Write E2E tests for everything, because they test the real app." E2E tests are slow and flaky. A broad E2E suite makes CI slow and teams stop trusting it.

Likely follow-up: Can you use Fantom for your app tests? No. Its README marks it as experimental, limited to React Native internals, and it lives in `private/` in the repo.[493]

---

[486] 18-Testing-React-Native-Apps.md, "Test pyramid and testing trophy" and "Part 2" intro. Opinion, not a measured result.

[487] 18-Testing-React-Native-Apps.md, "Test pyramid and testing trophy" and "Part 2" intro. Opinion, not a measured result.

[488] packages/jest-preset/jest-preset.js (preset keys) and jest/setup.js (View and other mocks), RN checkout.

[489] tmp/refresh/interview/scratch/IQ-TESTING/proof-22.txt, Jest run in Node, 3 passed. No layout engine runs.

[490] 18-Testing-React-Native-Apps.md, "Test pyramid and testing trophy" and "Part 2" intro. Opinion, not a measured result.

[491] The testing chapter, section "A sensible CI pipeline for an RN app". Team choice, not a rule from a source.

[492] private/react-native-fantom/**docs**/README.md (experimental, internals only), root package.json:46-47 (`"test": "jest"`, `"fantom"`), packages/rn-tester/README.md:3.

[493] private/react-native-fantom/**docs**/README.md (experimental, internals only), root package.json:46-47 (`"test": "jest"`, `"fantom"`), packages/rn-tester/README.md:3.

Q22. How do you unit and component test with Jest and React Native Testing Library, and how do you mock native modules and TurboModules?

Short answer (say this first, 2 to 4 sentences): I use Jest with the preset `@react-native/jest-preset`. I render the screen with React Native Testing Library (RNTL), query it like a user (role, label, text), and use `userEvent`. Jest has no native side, so I replace native modules: `jest.spyOn` for one call, `jest.mock` in a setup file for a library. For a TurboModule I mock the registry lookup or the spec file.[494][495][496]

Key points:

- The preset moved. In 0.84 `react-native` still shipped `jest-preset.js` and a `jest/` folder. In 0.85 the file is a stub that requires `@react-native/jest-preset` and throws a migration message if it is missing. In 0.87 the stub is gone from the tarball.[497][498]
- Migration: install `@react-native/jest-preset` as a dev dependency and set `module.exports = {preset: '@react-native/jest-preset'};`. The package needs Node 22.13 or newer.[499][500]
- RNTL 14.1.0 needs React 19, React Native 0.78 or newer, and the `test-renderer` package. `render` and the event helpers return Promises, so I `await` them.[501]
- RNTL query order: `getByRole`, `getByLabelText`, `getByPlaceholderText`, `getByText`, and `getByTestId` last. Use `queryBy*` for absence and `findBy*` to wait.
- The built-in matchers such as `toBeOnTheScreen` replace `@testing-library/jest-native`, which npm marks as no longer maintained.[502]

---

[494] Tarballs from `npm pack react-native@<v> --before=2026-10-30`: 0.84.0 has 33 entries under `package/jest/` and `jest-preset.js`, 0.85.0 and 0.86.0 have only `jest-preset.js`, 0.87.0 has none. Proof: tmp/refresh/interview/scratch/IQ-TESTING/proof-preset.txt.

[495] react-native@0.85.0 `package/jest-preset.js` lines 12-24 (stub, migration error) and CHANGELOG.md:833 (v0.85.0 "Move Jest preset to new package") and CHANGELOG.md:282 (v0.87.0 preset removed from `react-native`).

[496] packages/jest-preset/package.json (name `@react-native/jest-preset`, engines node >= 22.13.0) and `npm view @react-native/jest-preset version` = 0.88.1 (2026-10-29).

[497] Tarballs from `npm pack react-native@<v> --before=2026-10-30`: 0.84.0 has 33 entries under `package/jest/` and `jest-preset.js`, 0.85.0 and 0.86.0 have only `jest-preset.js`, 0.87.0 has none. Proof: tmp/refresh/interview/scratch/IQ-TESTING/proof-preset.txt.

[498] react-native@0.85.0 `package/jest-preset.js` lines 12-24 (stub, migration error) and CHANGELOG.md:833 (v0.85.0 "Move Jest preset to new package") and CHANGELOG.md:282 (v0.87.0 preset removed from `react-native`).

[499] react-native@0.85.0 `package/jest-preset.js` lines 12-24 (stub, migration error) and CHANGELOG.md:833 (v0.85.0 "Move Jest preset to new package") and CHANGELOG.md:282 (v0.87.0 preset removed from `react-native`).

[500] packages/jest-preset/package.json (name `@react-native/jest-preset`, engines node >= 22.13.0) and `npm view @react-native/jest-preset version` = 0.88.1 (2026-10-29).

[501] `npm view @testing-library/react-native version peerDependencies` = 14.1.0, jest >=29.0.0, react >=19.0.0, react-native >=0.78, test-renderer ^1.0.0. Promise returns: the testing chapter footnote ts-unit-11. Run proof: proof-22.txt.

[502] `npm view @testing-library/jest-native deprecated` (2026-10-29): "Please use the built-in Jest matchers available in @testing-library/react-native v12.4+".

- With the New Architecture, a module read through `TurboModuleRegistry.get` needs a mock of the registry or of the spec file. Setting `NativeModules.Foo` only works for code that reads `NativeModules`.[503]

Example or code (only when it helps, max 15 lines):

```
// Run: yes. Jest 30.5.2, RNTL 14.1.0, RN 0.88.1. 3 tests passed.
test('shares a greeting', async () => {
  const spy = jest.spyOn(Share, 'share').mockResolvedValue({action:
'sharedAction'});
  const user = userEvent.setup();
  await render(<Greeter />);
  await user.type(screen.getByPlaceholderText('Name'), 'Ana');
  await user.press(screen.getByRole('button', {name: 'Share'}));
  expect(spy).toHaveBeenCalledWith({message: 'Hello Ana'});
  expect(await screen.findByText('Shared for Ana')).toBeOnTheScreen();
});
test('mock a TurboModule lookup', () => {
  jest.spyOn(TurboModuleRegistry, 'get').mockReturnValue({ping: () => 'pong'});
  expect(TurboModuleRegistry.get('MyModule').ping()).toBe('pong');
});
```

Common wrong answer: "Set `preset: 'react-native'` in `jest.config.js`." That value was correct before 0.85. In 0.88 it fails, because the `react-native` package no longer ships the preset.[504][505]

Likely follow-up: How do you test an animation? Use `jest.useFakeTimers()` and `jest.advanceTimersByTime(ms)`. With `useNativeDriver: false` the callback gets `{finished: true}`. A native-driver animation runs on the UI thread, so Jest cannot see it.[506]

Q23. Detox vs Maestro vs Appium: how do you choose for a team in 2026, and what causes flaky E2E tests?

Short answer (say this first, 2 to 4 sentences): For most React Native teams I start with Maestro. It is black box, uses YAML, and needs no code in the app. I pick Detox when I need in-process synchronization and I accept the version risk. I pick Appium when I need other languages or a WebDriver-only device cloud. Flakiness comes from animations, fixed sleeps, network, shared

[503] tmp/refresh/testing/scratch/unit/src/more.test.js (TurboModuleRegistry.get spy, fake timers with `{finished: true}`), rerun 2026-10-29: 2 suites, 3 tests passed. Also tmp/refresh/interview/scratch/IQ-TESTING/proof-22.txt.

[504] Tarballs from `npm pack react-native@<v> --before=2026-10-30`: 0.84.0 has 33 entries under `package/jest/` and `jest-preset.js`, 0.85.0 and 0.86.0 have only `jest-preset.js`, 0.87.0 has none. Proof: tmp/refresh/interview/scratch/IQ-TESTING/proof-preset.txt.

[505] react-native@0.85.0 `package/jest-preset.js` lines 12-24 (stub, migration error) and CHANGELOG.md:833 (v0.85.0 "Move Jest preset to new package") and CHANGELOG.md:282 (v0.87.0 preset removed from `react-native`).

[506] tmp/refresh/testing/scratch/unit/src/more.test.js (TurboModuleRegistry.get spy, fake timers with `{finished: true}`), rerun 2026-10-29: 2 suites, 3 tests passed. Also tmp/refresh/interview/scratch/IQ-TESTING/proof-22.txt.

state, and version drift.[507][508][509]

Key points:

- Maestro 2.12.0 reads the screen through the accessibility tree and installs a small driver on the device. It can also control things outside the app, such as permission dialogs.[510]
- Detox 20.51.4 is gray box. It waits until the app is idle (network, main thread, timers, animations, JS thread). An endless animation or a polling timer keeps the app "busy" and the test hangs.[511]
- Detox README states full New Architecture compatibility for RN 0.77 to 0.84. I found no proof for RN 0.88, and an open issue reports a failure on RN 0.85.3 with the New Architecture. So I run a spike on my exact RN version first.[512]
- Appium 3.9.0 is black box and has no app synchronization. I add explicit waits. It fits other languages and most device clouds.[513]
- Both Detox `by.id` and Maestro `id:` match the `testID` prop. On Android, RN exposes `testID` as the `resource-id`.[514]
- Flake fixes: wait on a visible state, reset app data per test (`clearState` in Maestro, `newInstance: true` in Detox), mock the server, pin the RN version and the OS image, and test a release-like build in CI.[515]

Example or code (only when it helps, max 15 lines):

```
# flows/smoke.yaml. Not run (no Maestro CLI here). Same syntax as the checked
flows in the testing chapter.
appId: com.example.app
onFlowStart:
  - clearState
---
- launchApp
- tapOn:
```

[507] https://docs.maestro.dev/get-started/how-maestro-works.md (black box, driver app, accessibility tree). `gh api repos/mobile-dev-inc/maestro/releases/latest` = cli-2.12.0. Syntax check proof: the testing chapter footnote ts-e2ex-12.

[508] Detox README.md:66-67 (RN 0.77 to 0.84 compatible), docs/articles/how-detox-works.md (sync list), issues wix/Detox#4963 and #4973. `npm view detox version` = 20.51.4 (2026-10-29). Via the testing chapter footnote ts-e2ex-4, −6, -8.

[509] `npm view appium version` = 3.9.0 (2026-10-29). The testing chapter section "Appium 3 with WebdriverIO".

[510] https://docs.maestro.dev/get-started/how-maestro-works.md (black box, driver app, accessibility tree). `gh api repos/mobile-dev-inc/maestro/releases/latest` = cli-2.12.0. Syntax check proof: the testing chapter footnote ts-e2ex-12.

[511] Detox README.md:66-67 (RN 0.77 to 0.84 compatible), docs/articles/how-detox-works.md (sync list), issues wix/Detox#4963 and #4973. `npm view detox version` = 20.51.4 (2026-10-29). Via the testing chapter footnote ts-e2ex-4, −6, −8.

[512] Detox README.md:66-67 (RN 0.77 to 0.84 compatible), docs/articles/how-detox-works.md (sync list), issues wix/Detox#4963 and #4973. `npm view detox version` = 20.51.4 (2026-10-29). Via the testing chapter footnote ts-e2ex-4, −6, −8.

[513] `npm view appium version` = 3.9.0 (2026-10-29). The testing chapter section "Appium 3 with WebdriverIO".

[514] packages/react-native/ReactAndroid/src/main/java/com/facebook/react/uimanager/ReactAccessibilityDelegate.kt:175-181 and https://docs.maestro.dev/get-started/supported-platform/react-native.md. Via the testing chapter footnote ts-e2ex-18, −19.

[515] The testing chapter section "Flakiness: causes and fixes". Advice, not a measured result.

```
    id: "login-button"
- assertVisible:
    id: "home-title"
```

Common wrong answer: "Detox is always best, because it is gray box and faster." It depends on private React Native internals, so each RN minor can break it. The Detox team documents support only up to RN 0.84.[516]

Likely follow-up: How do you find a flaky test? Re-run the failed test once in CI, track the tests that pass on a re-run, and fix or delete them.[517]

Q24. Why is snapshot testing criticized, and what do you use instead? What about `react-test-renderer` in React 19?

Short answer (say this first, 2 to 4 sentences): Snapshot tests store a rendered tree and compare it on each run. Snapshots are large, nobody reads the diffs, people update them without review, and they test structure, not behavior. I use RNTL queries and matchers for behavior, and small inline snapshots only for tiny, stable output. React 19 deprecates `react-test-renderer`, so I do not start new tests on it.[518][519]

Key points:

- The criticism is an opinion shared by the Testing Library guidance. It is not a measured result.[520]
- Replace a tree snapshot with an assertion on what the user sees: `expect(await screen.findByText('Shared for Ana')).toBeOnTheScreen()`.
- A small inline snapshot (an error object or a style) is fine. Jest wrote and matched one in my run.[521]
- React 19 prints a warning when you call `ReactTestRenderer.create()`. React says the package stays on npm, is not maintained, and may break with new React features.[522]
- npm does not flag `react-test-renderer@19.3.1` as deprecated. The deprecation is a React runtime warning.[523]

---

[516] Detox README.md:66-67 (RN 0.77 to 0.84 compatible), docs/articles/how-detox-works.md (sync list), issues wix/Detox#4963 and #4973. `npm view detox version` = 20.51.4 (2026-10-29). Via the testing chapter footnote ts-e2ex-4, −6, −8.

[517] The testing chapter section "Flakiness: causes and fixes". Advice, not a measured result.

[518] The testing chapter section "Snapshot testing". https://testing-library.com/docs/guiding-principles (not re-fetched, opinion).

[519] https://react.dev/warnings/react-test-renderer (fetched 2026-10-29): deprecated, warning on `ReactTestRenderer.create()`, "will remain available on NPM but will not be maintained and may break with new React features".

[520] The testing chapter section "Snapshot testing". https://testing-library.com/docs/guiding-principles (not re-fetched, opinion).

[521] tmp/refresh/interview/scratch/IQ-TESTING/proof-22.txt, "Snapshots: 1 passed".

[522] https://react.dev/warnings/react-test-renderer (fetched 2026-10-29): deprecated, warning on `ReactTestRenderer.create()`, "will remain available on NPM but will not be maintained and may break with new React features".

[523] `npm view react-test-renderer@19.3.1 deprecated` returns empty (2026-10-29).

- RNTL 14 replaced `react-test-renderer` with the `test-renderer` package, and lists `test-renderer ^1.0.0` as a peer dependency.[524]

Example or code (only when it helps, max 15 lines):

```
// Run: yes. Jest 30.5.2. 1 snapshot passed.
test('inline snapshot is small', () => {
  expect({ok: true}).toMatchInlineSnapshot(`
    {
      "ok": true,
    }
  `);
});
```

Common wrong answer: "Snapshot tests give full coverage, so we do not need other tests." A snapshot only proves that the output did not change. It does not prove the output is right.

Likely follow-up: What do you do with old `react-test-renderer` tests? Migrate to RNTL. The old package still installs (19.3.1), but React does not maintain it.[525][526]

Q25. How do you set up CI for a React Native app? Cover jobs, caching, emulator and simulator, device clouds, and release gates.

Short answer (say this first, 2 to 4 sentences): I order jobs from cheap to costly and stop early on failure. Lint, types, and Jest run on Linux. Then a release build. Then a small E2E smoke set on an Android emulator (Linux with KVM) or an iOS simulator (macOS). A last job is the release gate with manual approval. I run the full E2E suite nightly or on a device cloud.[527][528]

Key points:

- macOS runners cost the most. GitHub lists standard macOS at 0.062 USD per minute and Linux 2 core at 0.006 USD per minute, about ten times more. So only the iOS build and simulator tests use macOS.[529]
- Android emulators run on Linux runners with KVM. The emulator action runs each line of `script` as a separate shell command, so `export PATH` on one line does not carry over.[530]

---

[524] https://github.com/callstack/react-native-testing-library/releases/tag/v14.0.0 (breaking: "Replaced the deprecated React Test Renderer with Test Renderer") and `npm view @testing-library/react-native@14.1.0 peerDependencies.test-renderer` = ^1.0.0.

[525] https://react.dev/warnings/react-test-renderer (fetched 2026-10-29): deprecated, warning on `ReactTestRenderer.create()`, "will remain available on NPM but will not be maintained and may break with new React features".

[526] `npm view react-test-renderer@19.3.1 deprecated` returns empty (2026-10-29).

[527] The testing chapter section "A sensible CI pipeline for an RN app" (stage table, release gate).

[528] The testing chapter CI sample. Parsed again with PyYAML on 2026-10-29: tmp/refresh/testing/scratch/FC-1/ci.yml gives jobs static, android-e2e, release-gate. Latest releases by `gh api`: actions/checkout v7.0.1, actions/setup-node v7.1.0, ReactiveCircus/android-emulator-runner v2.38.0.

[529] https://docs.github.com/en/billing/reference/actions-runner-pricing (fetched 2026-10-29 by the the testing chapter author, not re-fetched): macOS 0.062 USD, Linux 2 core 0.006 USD per minute. Ratio 0.062 / 0.006 is about 10.

- Cache the package manager through `actions/setup-node` (`cache: yarn`). Also cache Gradle and CocoaPods. I did not measure the cache gains.[531]
- Use `concurrency` with `cancel-in-progress: true` to stop old runs on the same ref.[532]
- Device clouds (BrowserStack, Sauce Labs, AWS Device Farm, Firebase Test Lab, Maestro Cloud) give real devices and parallel runs. Prices change, so check the vendor page before you quote one.[533]
- For Expo apps, EAS Workflows can run a `build` job and then a `maestro` job with `build_id` and `flow_path`.[534]

Example or code (only when it helps, max 15 lines):

```yaml
# Reduced from the testing chapter sample. Parsed with PyYAML (2 jobs). Not run
on GitHub.
jobs:
  static:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v7
      - uses: actions/setup-node@v7
        with: {node-version: 22, cache: yarn}
      - run: yarn install --frozen-lockfile
      - run: yarn tsc --noEmit
      - run: yarn jest --ci --coverage
  release-gate:
    needs: [static]
    runs-on: ubuntu-latest
    environment: production
```

Common wrong answer: "Run the whole E2E suite on every pull request, on macOS." It is slow and costly. Run a small smoke set on each pull request and the full suite on a schedule.[535]

Likely follow-up: How does the release gate work? A job with `environment: production` waits for manual approval, then a later step runs the store upload (for example EAS or Fastlane).[536]

---

[530] https://github.com/ReactiveCircus/android-emulator-runner README (KVM, one shell command per `script` line). Via the testing chapter footnote ts-history-26.

[531] The testing chapter CI sample. Parsed again with PyYAML on 2026-10-29: tmp/refresh/testing/scratch/FC-1/ci.yml gives jobs static, android-e2e, release-gate. Latest releases by `gh api`: actions/checkout v7.0.1, actions/setup-node v7.1.0, ReactiveCircus/android-emulator-runner v2.38.0.

[532] The testing chapter CI sample. Parsed again with PyYAML on 2026-10-29: tmp/refresh/testing/scratch/FC-1/ci.yml gives jobs static, android-e2e, release-gate. Latest releases by `gh api`: actions/checkout v7.0.1, actions/setup-node v7.1.0, ReactiveCircus/android-emulator-runner v2.38.0.

[533] The testing chapter table "Device clouds and CI services". Prices are search-snippet level and unverified there.

[534] https://docs.expo.dev/eas/workflows/examples/e2e-tests/ (last updated 2026-07-22). Via the testing chapter footnote ts-e2e-9.

[535] https://docs.github.com/en/billing/reference/actions-runner-pricing (fetched 2026-10-29 by the the testing chapter author, not re-fetched): macOS 0.062 USD, Linux 2 core 0.006 USD per minute. Ratio 0.062 / 0.006 is about 10.

[536] The testing chapter section "A sensible CI pipeline for an RN app" (stage table, release gate).

Q26. How do coding agents and MCP servers change mobile testing, and what must stay deterministic?

Short answer (say this first, 2 to 4 sentences): Agents change who writes and runs tests. They do not change what a good test is. An MCP server gives an agent a screenshot, a view tree, input, and logs from a simulator, so the agent can explore a screen and draft a test. The CI gate must stay plain code: fixed Jest tests, fixed Maestro or Detox files, a fixed build, and seeded data. A human owns the oracle, the rule for what "right" means.[537][538]

Key points:

- Servers I checked on 2026-10-29: `@mobilenext/mobile-mcp` 1.0.14, `appium-mcp` 1.99.1, and the Maestro CLI MCP (`maestro mcp`, CLI 2.12.0). Expo MCP needs an Expo project.[539]
- The agent can read the accessibility tree (cheap, exact labels, stable refs) or a screenshot (shows layout, costs more tokens). Use the tree first.[540]
- Good `testID` and `accessibilityLabel` values matter more now. Without names, the agent guesses with pixel coordinates, and coordinates break when the layout changes.[541]
- Maestro `assertWithAI` is experimental. `optional` is `true` by default, so a failed AI check does not stop the run. Do not put it in the gate.[542]
- Risks: secrets in screenshots, prompt injection through app text, and an agent that writes a test that always passes. Use fake accounts and a test backend.[543]
- `appium-mcp` says to use it only as a local single-user server or a trusted CI job. Do not build `remoteServerUrl` from pull request content.[544]

Example or code (only when it helps, max 15 lines):

```
# Connect the Maestro MCP server to a coding agent (not run here)
claude mcp add maestro -- maestro mcp
```

Common wrong answer: "The agent decides pass or fail in CI, so we do not need fixed tests." The same prompt can take a different path on each run. CI must replay fixed scripts. Agents draft

[537] `npm view @mobilenext/mobile-mcp version` = 1.0.14 and `npm view appium-mcp version` = 1.99.1 (2026-10-29). Maestro MCP: https://docs.maestro.dev/get-started/maestro-mcp. Tool counts and Expo MCP: the testing chapter footnote ts-ai-2 to ts-ai-6.

[538] The testing chapter sections "Accessibility tree or screenshot", "Risks", and "A practical workflow". appium-mcp README trust model: the testing chapter footnote ts-ai-4.

[539] `npm view @mobilenext/mobile-mcp version` = 1.0.14 and `npm view appium-mcp version` = 1.99.1 (2026-10-29). Maestro MCP: https://docs.maestro.dev/get-started/maestro-mcp. Tool counts and Expo MCP: the testing chapter footnote ts-ai-2 to ts-ai-6.

[540] The testing chapter sections "Accessibility tree or screenshot", "Risks", and "A practical workflow". appium-mcp README trust model: the testing chapter footnote ts-ai-4.

[541] https://docs.expo.dev/mcp/ ("prefer testID as it is resilient to layout changes"). Via the testing chapter footnote ts-ai-6.

[542] https://docs.maestro.dev/reference/commands-available/assertwithai (fetched again 2026-10-29): "experimental", `optional` default true, set `optional: false` to stop the test.

[543] The testing chapter sections "Accessibility tree or screenshot", "Risks", and "A practical workflow". appium-mcp README trust model: the testing chapter footnote ts-ai-4.

[544] The testing chapter sections "Accessibility tree or screenshot", "Risks", and "A practical workflow". appium-mcp README trust model: the testing chapter footnote ts-ai-4.

and explore outside the gate.[545]

Likely follow-up: Where do agents help most? Drafting Jest and RNTL tests, exploring a new screen, saving a working Maestro flow, and triaging a failure. A human reviews each file as code.[546]

Q27. Would you use an AI-native testing service such as Autosana for a mobile app, and what would you check first?

Short answer (say this first, 2 to 4 sentences): I would try it on non-gate flows first. These services run an agent that follows plain-language steps on cloud devices. They help most with hard flows such as OTP and push. I would check the oracle first. I would plant a known bug and see if the flow fails.[547]

Key points:

- Autosana tests are natural-language flows. They can live as `*.flow.yaml` files in the repo.[548]
- An iOS upload must be a simulator `.app`. A React Native app must be a Release build. Expo uses an EAS profile with `simulator: true`.[549]
- Pricing is per AI action ($0.10, or $0.01 when cached) plus monthly plans. This is a vendor page checked on 2026-10-29.[550]
- The vendor says it replays cached steps and self-heals. These are vendor claims. I found no independent measurement.[551]
- Check data handling. The build and the screenshots go to the vendor cloud. I found no public retention policy.[552]

Example or code (only when it helps, max 15 lines):

```yaml
# A flow file in .autosana/ (not run here, schema from the vendor docs)
name: Login
instructions: |
  Sign in as the test user.
  Verify the home screen shows the user email.
```

[545] The testing chapter sections "Accessibility tree or screenshot", "Risks", and "A practical workflow". appium-mcp README trust model: the testing chapter footnote ts-ai-4.

[546] The testing chapter sections "Accessibility tree or screenshot", "Risks", and "A practical workflow". appium-mcp README trust model: the testing chapter footnote ts-ai-4.

[547] https://docs.autosana.ai/code-managed-files.md and https://docs.autosana.ai/introduction (fetched 2026-10-29). Via the testing chapter footnotes ts-asa-2 and ts-asa-3.

[548] https://docs.autosana.ai/code-managed-files.md and https://docs.autosana.ai/introduction (fetched 2026-10-29). Via the testing chapter footnotes ts-asa-2 and ts-asa-3.

[549] https://docs.autosana.ai/app-build-guide.md (fetched 2026-10-29). Via ts-asa-4.

[550] https://autosana.ai/pricing (fetched 2026-10-29). Via ts-asa-5.

[551] https://docs.autosana.ai/code-managed-files.md and https://docs.autosana.ai/introduction (fetched 2026-10-29). Via the testing chapter footnotes ts-asa-2 and ts-asa-3.

[552] https://autosana.ai/security (fetched 2026-10-29): disclosure address only, SOC 2 Type 1 on the footer. Via ts-asa-9.

Common wrong answer: "AI testing removes flaky tests." The agent can take a different path each run, and self-healing can hide a real bug. A person must still write the expected result.[553]

Likely follow-up: Would it replace Maestro in CI? Not for the gate. A fixed Maestro file is free, versioned, and repeatable. The service is a layer for hard flows and pull request video.[554]

Footnotes

[553] The testing chapter subsection "AI-native test services: Autosana and peers" and "Risks". Judgment, not a measured result.

[554] The testing chapter subsection "AI-native test services: Autosana and peers" and "Risks". Judgment, not a measured result.

Chapter 18: Testing React Native Apps

This chapter explains how to test a React Native app in late 2026. It covers unit and component tests, end-to-end tests, the history of the tools, device clouds and CI, and the effect of agentic AI. Tool versions and project status were checked on 2026-10-29. Each fact has a footnote. Code marked "not run" was not executed.

Part 1: Unit, component, and integration tests

This section covers tests that run in Node, not on a device. These tests are fast. They run on every commit. Checked on 2026-10-29 against React Native 0.88.1 and React 19.3.1.[555]

Jest and the React Native preset

Jest is the default test runner for React Native apps. The runner needs a preset. The preset sets the transform, the module resolver, the test environment, and a setup file that mocks parts of React Native.

The preset moved to its own package. In 0.84 the preset lived inside the `react-native` package (`jest-preset.js` and a `jest/` folder).[556] In 0.85 the `react-native` package keeps only a stub. The stub re-exports `@react-native/jest-preset`. If that package is missing, the stub throws an error that tells you to install it.[557] The new package first appeared on npm on 2026-02-19 (a 0.85.0 nightly). It ships with each RN release (0.88.1 on 2026-10-27).[558]

Migration takes two steps.

1. Install `@react-native/jest-preset` as a dev dependency. Use the same version as `react-native`.
2. Change `jest.config.js`.

```
// jest.config.js (run: yes, see proof-unit-jest.txt)
module.exports = {preset: '@react-native/jest-preset'};
```

The old value was `preset: 'react-native'`.[559] The package needs Node 22.13 or newer and a peer of React 19.2.3 or newer.[560]

What the preset configures:[561]

---

[555] `npm view react-native dist-tags`: latest 0.88.1, published 2026-10-27. `npm view react dist-tags` and react-test-renderer 19.3.1. Checked 2026-10-29.

[556] Tarball `react-native@0.84.0` (npm registry) contains `package/jest/` and `package/jest-preset.js`. Listed with `tar tzf`, 2026-10-29.

[557] `react-native@0.85.0` tarball, `package/jest-preset.js` lines 12-24 (stub that requires `@react-native/jest-preset` and throws a migration message). Same text in RN repo history. Checked 2026-10-29.

[558] `npm view @react-native/jest-preset time --json`: created 2026-02-19, first 0.85.0-rc.0 on 2026-03-03, 0.88.1 on 2026-10-27. Checked 2026-10-29.

[559] `react-native@0.85.0` tarball, `package/jest-preset.js` lines 12-24 (stub that requires `@react-native/jest-preset` and throws a migration message). Same text in RN repo history. Checked 2026-10-29.

[560] /Users/gaston/Projects/tesla/react-native/packages/jest-preset/package.json (engines node >= 22.13.0, peerDependencies react ^19.2.3, dependencies babel-jest ^29.7.0, jest-environment-node ^29.7.0).

[561] packages/jest-preset/jest-preset.js (haste, resolver, transform, transformIgnorePatterns, setupFiles, testEnvironment).

Key	Value
`haste`	React Native haste settings
`resolver`	custom `jest/resolver.js`
`transform`	babel-jest with the RN Babel preset, plus an asset file transformer
`transformIgnorePatterns`	ignore `node_modules` except `react-native`, `@react-native/*`, and `@react-native-community/*`
`setupFiles`	`jest/setup.js`
`testEnvironment`	`jest/react-native-env.js` (a Node-based environment)

The setup file mocks these modules: `NativeModules`, `AppState`, `AccessibilityInfo`, `ActivityIndicator`, `Clipboard`, `RefreshControl`, `ScrollView`, `TextInput`, `View`, `Image`, `Linking`, `Modal`, `Text`, `UIManager`, `useColorScheme`, `Vibration`, `NativeComponentRegistry`, and `requireNativeComponent`.[562] Layout does not run. A `View` is a plain mock. Because of this, `getBoundingClientRect` and real layout values are not available in Jest.

If your app has a Jest 29 setup, note that the preset depends on `babel-jest` 29 and `jest-environment-node` 29.[563] Jest 30.5.2 is the latest Jest (checked 2026-10-29).[564] The scratch project ran Jest 30.5.2 with the 0.88.1 preset and passed.[565]

Mocking native modules and TurboModules

JavaScript code reaches native code through `NativeModules` or `TurboModuleRegistry`. Jest has no native side. You must replace the module.

- For modules the preset already mocks, use the mock as is.
- For one module in one test, use `jest.spyOn` on the JS wrapper (for example `Share.share`).
- For a third-party module, use `jest.mock('name', factory)` in a setup file you list in `setupFiles`.
- For your own TurboModule, mock the spec file that calls `TurboModuleRegistry.getEnforcing`.

The New Architecture changes one thing. TurboModules load lazily and through JSI, not through the old bridge. Old tests that set `NativeModules.Foo = {...}` still work only for modules that code reads through `NativeModules`. Code that reads `TurboModuleRegistry.get('Foo')` needs a mock of the registry or of the spec file. The scratch test spied on `TurboModuleRegistry.get` and passed.[566]

[562] packages/jest-preset/jest/setup.js, the `mock(...)` calls (about lines 90-210).

[563] /Users/gaston/Projects/tesla/react-native/packages/jest-preset/package.json (engines node >= 22.13.0, peerDependencies react ^19.2.3, dependencies babel-jest ^29.7.0, jest-environment-node ^29.7.0).

[564] `npm view jest version time.modified`: 30.5.2, modified 2026-09-18. Checked 2026-10-29.

[565] Scratch proof: tmp/refresh/testing/scratch/unit/proof-unit-jest.txt and proof-unit-jest2.txt (Jest 30.5.2, RN 0.88.1, RNTL 14.1.0, React 19.3.1, 3 tests passed). Install used `--before=2026-10-30` because the local npm has a 14-day release-age rule.

[566] Scratch proof: tmp/refresh/testing/scratch/unit/proof-unit-jest.txt and proof-unit-jest2.txt (Jest 30.5.2, RN 0.88.1, RNTL 14.1.0, React 19.3.1, 3 tests passed). Install used `--before=2026-10-30` because the local npm has a 14-day release-age rule.

Keep mocks thin. A mock that copies native logic hides real bugs. Cover the real native path with an end-to-end test (see the end-to-end chapter section).

React Native Testing Library

React Native Testing Library (RNTL) renders components and queries them like a user does. The latest version is 14.1.0 (published 2026-10-08).[567] It needs React 19 or newer, React Native 0.78 or newer, Node 22.13 or newer, and the `test-renderer` package. Jest 29 or newer is an optional peer.[568]

Version 14 changed the rendering API. `render`, `renderHook`, and the event helpers return Promises. You must `await` them.[569] RNTL 14 uses `test-renderer` in place of `react-test-renderer`. The RNTL 14.0.0 release notes list this as a breaking change and add a peer dependency on `test-renderer` 1.x.[570] The reason: React deprecated `react-test-renderer` in React 19. React says the package stays on npm, is not maintained, and may break with new React features.[571] The old package still installs (19.3.1).

Query rules, in order of preference:

Query	Use for
`getByRole` (with `name`)	buttons, links, switches
`getByLabelText`	form fields with an accessibility label
`getByPlaceholderText`	text inputs
`getByText`	static text
`getByTestId`	last resort

Use `getBy*` when the element must exist. Use `queryBy*` to assert absence. Use `findBy*` to wait for an element.

`userEvent` or `fireEvent`: prefer `userEvent`. It runs the full sequence of events that a real touch makes (press in, press out, focus, change). `fireEvent` sends one event to one handler. It can pass

[567] `npm view @testing-library/react-native version time.modified peerDependencies`: 14.1.0, 2026-10-08; peers jest >=29, react >=19, react-native >=0.78, test-renderer ^1.0.0. Checked 2026-10-29.

[568] `npm view @testing-library/react-native version time.modified peerDependencies`: 14.1.0, 2026-10-08; peers jest >=29, react >=19, react-native >=0.78, test-renderer ^1.0.0. Checked 2026-10-29.

[569] `node_modules/@testing-library/react-native/dist/render.d.ts:16` and `render-hook.d.ts:19` declare `Promise` return types (RNTL 14.1.0, scratch install).

[570] https://github.com/callstack/react-native-testing-library/releases/tag/v14.0.0 (breaking changes: "Replaced the deprecated React Test Renderer with Test Renderer", "Added a peer dependency on Test Renderer 1.x"). `npm view test-renderer`: npm package created 2026-01-07, latest 1.3.1, repo github.com/mdjastrzebski/test-renderer (GitHub repo created 2024-11-01). Its README calls it "a modern replacement for the deprecated React Test Renderer". Checked 2026-10-29. Proof: tmp/refresh/testing/scratch/FC-1/npm3.txt.

[571] https://react.dev/warnings/react-test-renderer (fetched 2026-10-29): "react-test-renderer is deprecated ... will remain available on NPM but will not be maintained and may break with new React features or changes to React's internals."

in a test when a real user could not trigger the handler.[572]

```
// src/Greeter.test.js (run: yes, 1 passed, Jest 30.5.2, RNTL 14.1.0, RN 0.88.1)
import React from 'react';
import {Share} from 'react-native';
import {render, screen, userEvent} from '@testing-library/react-native';
import {Greeter} from './Greeter';

test('shares a greeting', async () => {
  const spy = jest.spyOn(Share, 'share').mockResolvedValue({action:
'sharedAction'});
  const user = userEvent.setup();
  await render(<Greeter />);
  await user.type(screen.getByPlaceholderText('Name'), 'Ana');
  await user.press(screen.getByRole('button', {name: 'Share'}));
  expect(spy).toHaveBeenCalledWith({message: 'Hello Ana'});
  expect(await screen.findByText('Shared for Ana')).toBeOnTheScreen();
});
```

The built-in matchers (`toBeOnTheScreen`, `toHaveTextContent`) replace `@testing-library/jest-native`. That package is deprecated.[573] The `@testing-library/react-hooks` package is also old (last 8.0.1). Use `renderHook` from RNTL.[574]

Async utils: `findBy*`, `waitFor`, and `waitForElementToBeRemoved`. Prefer `findBy*`. Do not put side effects inside `waitFor`.

Testing hooks

Use `renderHook` from RNTL. Test the hook through the component that uses it when the hook depends on context or navigation. Test it alone when it holds pure logic. Use `act` (re-exported by RNTL) for state changes you trigger yourself.

Fake timers and Animated

Use `jest.useFakeTimers()`. Then move time with `jest.advanceTimersByTime(ms)`. With `useNativeDriver: false`, `Animated.timing` finishes when you advance past its duration. The scratch test did this and the callback got `{finished: true}`.[575] With the native driver, the animation runs on the UI thread. Jest cannot see it. Test only that the animation starts. Check the final look on a device.

---

[572] https://callstack.github.io/react-native-testing-library/docs/api/events/user-event (vendor docs, not re-fetched; claim about full event sequence is from the docs summary, treat as Assumed).

[573] `npm view @testing-library/jest-native deprecated`: "no longer maintained, use the built-in Jest matchers in @testing-library/react-native v12.4+". Checked 2026-10-29.

[574] `npm view @testing-library/react-native version time.modified peerDependencies`: 14.1.0, 2026-10-08; peers jest >=29, react >=19, react-native >=0.78, test-renderer ^1.0.0. Checked 2026-10-29.

[575] Scratch proof: tmp/refresh/testing/scratch/unit/proof-unit-jest.txt and proof-unit-jest2.txt (Jest 30.5.2, RN 0.88.1, RNTL 14.1.0, React 19.3.1, 3 tests passed). Install used `--before=2026-10-30` because the local npm has a 14-day release-age rule.

Network: MSW

Mock Service Worker (MSW) intercepts `fetch` at the network layer. This tests your real data code without stubs. The latest version is 3.0.3 (modified 2026-10-10).[576] In React Native you use `msw/node` with `setupServer` in a Jest setup file. Mark: I did not run an MSW test in the scratch project (not run). RN has its own `fetch` and `XMLHttpRequest` polyfills, so check that your MSW version can patch them before you adopt it.

Testing navigation

Wrap the screen in the navigation container your app uses (for example React Navigation's `NavigationContainer`) inside a test helper. Mock native screens modules if the library needs them. Assert on what the user sees after a press, not on the router's internal state. Not run in scratch.

Snapshot testing

Snapshot tests store a rendered tree and compare it on each run. They fell out of favor for four reasons. Snapshots are large. Nobody reads the diffs. Engineers update them without review. And they test structure, not behavior. Use small inline snapshots for stable, tiny output (an error object, a style). Use queries and matchers for behavior. This is an opinion, shared by the Testing Library guidance, not a measured result.[577]

Test pyramid and testing trophy

The pyramid says: many unit tests, fewer integration tests, few end-to-end tests. The trophy says: write most tests at the integration level (a screen with real child components and mocked edges), keep a base of static checks (TypeScript, ESLint), and few end-to-end tests. For mobile, the trophy fits better. Most bugs sit where components, state, and the network meet. A unit test of a tiny function misses those bugs. Both models are opinions, not rules.

Layer	Tool	Speed	Catches
Static	TypeScript, ESLint	seconds	type and style errors
Unit	Jest	ms	pure logic
Component/integration	Jest + RNTL + MSW	ms to s	screen behavior
End to end	Maestro, Detox (other sections)	minutes	real native paths

Coverage

Run `jest --coverage`. Set `coverageThreshold` for changed code, not only a global number. A high number does not prove the tests check anything. Exclude generated files and native spec files.

How the React Native repo tests itself

The RN repo uses three layers.

[576] `npm view msw version time.modified`: 3.0.3, 2026-10-10. Checked 2026-10-29.

[577] Opinion. Testing Library guiding principles: https://testing-library.com/docs/guiding-principles (not re-fetched, Assumed).

- Jest unit tests in `__tests__` folders, run with `yarn test`.[578]
- Fantom integration tests, files ending in `-itest.js`, run with `yarn fantom`.[579]
- RNTester, an app that shows each core component and API for manual checks and native integration tests.[580]

Fantom is "the new integration testing and benchmarking tool for React Native". It runs JavaScript in Hermes with Fabric, C++ TurboModules, and Bridgeless in a headless native tester. Layout is real, so `getBoundingClientRect` returns real values.[581] The README marks it as experimental. It is limited to React Native internals, and tests must live in `packages/react-native`. It is for engineers who work on React Native. It is not for app teams. The README says only that Meta may later open it to product code.[582] Do not plan your app tests on it.

Part 2: End-to-end and device tests

End-to-end (E2E) tests run the real app on a device or simulator. They test the whole stack: JavaScript, native modules, and the OS. They are slow and costly, so write few of them. Cover the critical user journeys only.

[578] /Users/gaston/Projects/tesla/react-native/package.json:46 (`"test": "jest"`).

[579] private/react-native-fantom/**docs**/README.md lines 1-60 and 68-110 (warning block, definition, usage, `yarn fantom`). Root package.json:47.

[580] packages/rn-tester/README.md line 3 ("The RNTester showcases React Native views and modules").

[581] private/react-native-fantom/**docs**/README.md lines 1-60 and 68-110 (warning block, definition, usage, `yarn fantom`). Root package.json:47.

[582] private/react-native-fantom/**docs**/README.md lines 1-60 and 68-110 (warning block, definition, usage, `yarn fantom`). Root package.json:47.

Status of the tools (checked 2026-10-29)

Tool	Latest version	Last release / commit	Status
Detox	20.51.4 on npm (`latest` tag), GitHub release 20.51.3[583]	npm 20.51.4 published 2026-06-16, GitHub release 2026-05-30, last commit 2026-10-27[584][585]	Maintained
Maestro CLI	2.12.0[586]	release 2026-10-26, commit 2026-10-28[587]	Maintained
Appium	3.9.0[588]	npm 2026-10-27, commit 2026-10-28[589]	Maintained
Appium UiAutomator2 driver	8.7.3[590]	2026-10-28	Maintained
Appium XCUITest driver	12.22.4[591]	2026-10-29	Maintained
WebdriverIO	10.1.0[592]	npm 2026-10-18	Maintained
Playwright	1.64.1[593]	release 2026-10-14	Maintained
Storybook for React Native	10.7.0[594]	release 2026-10-13	Maintained
Chromatic CLI	18.13.5[595][596]	npm 2026-10-29	Maintained (vendor product)
Argos CLI	6.11.0[597]	npm 2026-10-21	Maintained
react-native-owl	1.5.0[598]	npm 1.5.0 published 2025-01-14, last repo push 2025-01-15[599][600]	Stale. No repo push for 21 months.

[583] npm view detox version time (2026-10-29), https://api.github.com/repos/wix/Detox (pushed_at 2026-10-28) and /releases (20.51.3, 2026-05-30).

[584] wix/Detox repo HEAD fc34e84, commit date Tue 2026-10-27 22:17 +0200 (git log, clone in tmp/testing-repos/Detox, checked 2026-10-29). Version in detox/package.json: 20.51.4.

[585] `npm view <pkg> time --json` and `gh api` on 2026-10-29: detox 20.51.4 published 2026-06-16 (dist-tag latest), chromatic 18.13.5 on 2026-10-29T16:36Z, react-native-owl 1.5.0 on 2025-01-14, owl repo last commit 2025-01-15. Proof: tmp/refresh/testing/scratch/FC-3/npm.txt.

[586] https://github.com/mobile-dev-inc/maestro releases, tag cli-2.12.0, published 2026-10-26. Local run in scratch/e2e/proof-maestro.txt.

[587] https://github.com/mobile-dev-inc/maestro releases, tag cli-2.12.0, published 2026-10-26. Local run in scratch/e2e/proof-maestro.txt.

[588] npm view appium, appium-uiautomator2-driver, appium-xcuitest-driver version time (2026-10-29). https://github.com/appium/appium

[589] npm view appium, appium-uiautomator2-driver, appium-xcuitest-driver version time (2026-10-29). https://github.com/appium/appium

[590] npm view appium, appium-uiautomator2-driver, appium-xcuitest-driver version time (2026-10-29). https://github.com/appium/appium

[591] npm view appium, appium-uiautomator2-driver, appium-xcuitest-driver version time (2026-10-29). https://github.com/appium/appium

[592] npm view webdriverio, appium, appium-uiautomator2-driver version time engines peerDependencies (2026-10-29). Saved in scratch/e2e/deep/proof-deep-npm.txt.

[593] npm view playwright version, https://github.com/microsoft/playwright/releases/tag/v1.64.1

The word "maintained" means a release or a commit in the last 60 days.

Detox

Detox is a gray box tool from Wix. A test runs in Node.js with Jest. The app runs on a simulator or emulator. A native Detox client lives inside the app. The two sides talk over a web socket through a small Detox server[601].

How the synchronization works

Detox watches the app state and waits until the app is idle before it runs the next test step. The Detox docs list what it watches: network requests, the native main thread, UI layout (including the Yoga shadow queue), timers (`setTimeout`), animations (Animated and Reanimated), the React Native JavaScript thread, and the native modules thread[602]. The docs also list the React Native bridge, but only for apps that do not use the New Architecture[603].

This sync removes most `sleep()` calls. It has a cost. An endless animation, a polling timer, or a long request keeps the app "busy" and the test hangs. Detox logs the busy tasks when `session.debugSynchronization` is on. The setting is on by default[604].

A minimal config and test

The Detox demo app in the repo shows the shape of a config: `apps`, `devices`, and `configurations`[605]. The sample below is a reduced version. It is not run.

```
// .detoxrc.js (reduced from the repo demo, not run)
module.exports = {
  testRunner: { args: { config: 'e2e/jest.config.js', _: ['e2e'] } },
  apps: {
    'ios.debug': {
      type: 'ios.app',
```

[594] npm view @storybook/react-native version, https://github.com/storybookjs/react-native/releases/tag/v10.7.0

[595] npm view chromatic, @argos-ci/cli, react-native-owl version time (2026-10-29).

[596] `npm view <pkg> time --json` and `gh api` on 2026-10-29: detox 20.51.4 published 2026-06-16 (dist-tag latest), chromatic 18.13.5 on 2026-10-29T16:36Z, react-native-owl 1.5.0 on 2025-01-14, owl repo last commit 2025-01-15. Proof: tmp/refresh/testing/scratch/FC-3/npm.txt.

[597] npm view chromatic, @argos-ci/cli, react-native-owl version time (2026-10-29).

[598] npm view chromatic, @argos-ci/cli, react-native-owl version time (2026-10-29).

[599] gh api repos/FormidableLabs/react-native-owl: pushed_at 2025-01-15T16:33:42Z, archived false (2026-10-29).

[600] `npm view <pkg> time --json` and `gh api` on 2026-10-29: detox 20.51.4 published 2026-06-16 (dist-tag latest), chromatic 18.13.5 on 2026-10-29T16:36Z, react-native-owl 1.5.0 on 2025-01-14, owl repo last commit 2025-01-15. Proof: tmp/refresh/testing/scratch/FC-3/npm.txt.

[601] Detox docs/articles/how-detox-works.md (sections "Automatic App-State Synchronization", "Operations Detox synchronizes with automatically") and docs/articles/design-principles.md:9, repo HEAD fc34e84.

[602] Detox docs/articles/how-detox-works.md (sections "Automatic App-State Synchronization", "Operations Detox synchronizes with automatically") and docs/articles/design-principles.md:9, repo HEAD fc34e84.

[603] Detox docs/articles/how-detox-works.md (sections "Automatic App-State Synchronization", "Operations Detox synchronizes with automatically") and docs/articles/design-principles.md:9, repo HEAD fc34e84.

[604] Detox docs/articles/how-detox-works.md (sections "Automatic App-State Synchronization", "Operations Detox synchronizes with automatically") and docs/articles/design-principles.md:9, repo HEAD fc34e84.

[605] Detox examples/demo-react-native/detox.config.js, repo HEAD fc34e84 (apps, devices, iosCIBuildSettings for RN 0.85).

```
      binaryPath: 'ios/build/Build/Products/Debug-iphonesimulator/MyApp.app',
      build: 'xcodebuild -workspace ios/MyApp.xcworkspace -scheme MyApp -
configuration Debug -sdk iphonesimulator -derivedDataPath ios/build',
    },
    'android.debug': {
      type: 'android.apk',
      binaryPath: 'android/app/build/outputs/apk/debug/app-debug.apk',
      build: 'cd android && ./gradlew assembleDebug assembleAndroidTest -
DtestBuildType=debug',
      reversePorts: [8081],
    },
  },
  devices: {
    simulator: { type: 'ios.simulator', device: { type: 'iPhone 15' } },
    emulator: { type: 'android.emulator', device: { avdName:
'Pixel_8_API_35' } },
  },
  configurations: {
    'ios.sim.debug': { device: 'simulator', app: 'ios.debug' },
    'android.emu.debug': { device: 'emulator', app: 'android.debug' },
  },
};
```

```
// e2e/login.test.js (not run)
describe('Login', () => {
  beforeAll(async () => { await device.launchApp({ newInstance: true }); });
  it('shows the welcome title', async () => {
    await element(by.id('login-button')).tap();
    await expect(element(by.id('home-title'))).toBeVisible();
  });
});
```

Run it with `npx detox build -c ios.sim.debug` and then `npx detox test -c ios.sim.debug`. These commands are the documented flow. I did not run them because this machine had no device session.

New Architecture support: what is proven

- The Detox README says RN 0.77 to 0.84 is fully compatible with the New Architecture. It says newer RN versions "might work" but the team has not tested them in depth[606].
- The Detox repo builds its own demo against `react-native` 0.85.2 (package.json at HEAD)[607]. A CI line in the demo config handles 0.85 iOS build flags[608].
- Open issue 4963 (opened 2026-07-02, last update 2026-09-03) reports that Detox 20.51.3 fails on Android with RN 0.85.3, New Architecture, bridgeless. The idling resource looks up the private field `mMountItemDispatcher` and gets `NoSuchFieldException`[609]. A comment of 2026-09-03 says

[606] Detox README.md:66-67, repo HEAD fc34e84.

[607] Detox detox/package.json:85 `"react-native": "0.85.2"`, repo HEAD fc34e84.

[608] Detox examples/demo-react-native/detox.config.js, repo HEAD fc34e84 (apps, devices, iosCIBuildSettings for RN 0.85).

the crash did not reproduce on RN 0.86.2 and that the field still exists there. Issue 4973 (open) reports that network sync misses OkHttp clients outside `NetworkingModule` on Expo SDK 56+, seen on Detox 20.51.4 with RN 0.86.2[610][611].

- Nothing in the README, releases, or issues names RN 0.88. I found no proof that Detox 20.51.4 works on RN 0.88.

So the proven state for 0.88 is "not documented and not tested by the Detox team". Detox depends on private RN internals, so each RN minor version can break it. Run a spike on your exact RN version before you adopt it.

Maestro

Maestro is a black box tool. The CLI sends commands to the device. It reads the screen through the accessibility tree, not through app code. The Maestro docs call this the "arm's length" model. They also say you do not need an npm package in your app[612].

How it talks to the device

On both Android and iOS, the CLI installs a small companion driver app on the device. The driver reads what is on screen and does taps and swipes[613]. On iOS the target is an Xcode simulator[614]. This model also controls things outside your app: permission dialogs, Safari, and system settings[615][616]. I did not read the driver source in depth. The repo has the modules `maestro-android`, `maestro-ios-driver`, and `maestro-ios-xctest-runner`[617].

A flow with testID, subflow, env, and tags

Main flow `flows/checkout.yaml`:

```
appId: com.example.app
name: Checkout happy path
tags:
  - smoke
```

[609] https://github.com/wix/Detox/issues/4963 (opened 2026-07-02, updated 2026-09-03, open) and /issues/4973 (open), read with gh issue list/view on 2026-10-29. Search for RN 0.86 to 0.88 returned no matching Detox issue or release note. Releases: gh release list -R wix/Detox, latest 20.51.3 on 2026-05-30.

[610] https://github.com/wix/Detox/issues/4963 (opened 2026-07-02, updated 2026-09-03, open) and /issues/4973 (open), read with gh issue list/view on 2026-10-29. Search for RN 0.86 to 0.88 returned no matching Detox issue or release note. Releases: gh release list -R wix/Detox, latest 20.51.3 on 2026-05-30.

[611] https://github.com/wix/Detox/issues/4963 comment by isilher, 2026-09-03 (RN 0.86.2 did not reproduce the `mMountItemDispatcher` crash), and issue 4973 body (Detox 20.51.4, RN 0.86.2, state OPEN on 2026-10-29).

[612] https://docs.maestro.dev/get-started/how-maestro-works.md and https://docs.maestro.dev/get-started/what-is-maestro.md (read 2026-10-29).

[613] https://docs.maestro.dev/get-started/how-maestro-works.md and https://docs.maestro.dev/get-started/what-is-maestro.md (read 2026-10-29).

[614] https://docs.maestro.dev/get-started/supported-platform/ios.md (read 2026-10-29): Xcode simulators, permissions, multi-app journeys.

[615] https://docs.maestro.dev/get-started/how-maestro-works.md and https://docs.maestro.dev/get-started/what-is-maestro.md (read 2026-10-29).

[616] https://docs.maestro.dev/get-started/supported-platform/ios.md (read 2026-10-29): Xcode simulators, permissions, multi-app journeys.

[617] mobile-dev-inc/maestro repo root at HEAD (commit date 2026-10-28): modules maestro-android, maestro-ios-driver, maestro-ios-xctest-runner.

```
  - checkout
env:
  EMAIL: test@example.com
  PASSWORD: secret
onFlowStart:
  - clearState
---
- launchApp
- runFlow:
    file: subflows/login.yaml
- assertVisible:
    id: "home-title"
- tapOn:
    id: "cart-button"
- runFlow:
    when:
      visible:
        id: "promo-banner"
    commands:
      - tapOn:
          id: "promo-close"
- takeScreenshot: cart
- assertVisible: "Total"
```

Subflow `flows/subflows/login.yaml`:

```
appId: com.example.app
---
- tapOn:
    id: "email-input"
- inputText: ${EMAIL}
- tapOn:
    id: "password-input"
- inputText: ${PASSWORD}
- tapOn:
    id: "login-button"
```

I ran `maestro check-syntax` on both files with Maestro 2.12.0. Both printed `OK` and exit code 0. A file with the wrong command `tapOnn` failed with "Invalid Command: tapOnn"[618]. The check is a syntax check only. I did not run the flow on a device, so the selectors are not proven against a real app.

Key points on the features:

- `runFlow` runs a subflow file or an inline command list. It takes `env`, `label`, and a `when` condition[619].

[618] scratch/e2e/deep/proof-deep-checksyntax.txt. Files flows/checkout.yaml and flows/subflows/login.yaml, Maestro 2.12.0, Java 21.

[619] https://docs.maestro.dev/reference/commands-available/runflow.md (parameters label, env; inline commands with a when condition).

- Variables use the `${NAME}` form. Define defaults in the flow `env` block or pass them with `runFlow env`[620]. Set secrets from the CI environment, not in the file.
- Tags go in the flow header. By default, a folder run only takes the top-level flow files. To include subfolders, set the `flows` glob in `config.yaml`[621].
- Subflow files in `flows/subflows/` stay out of the default run for this reason.

Studio and Cloud: status from official docs

Maestro Studio is a desktop app. It mirrors the device screen and builds YAML as you click[622][623]. Maestro Cloud is a hosted service. It runs your flows in parallel on hosted virtual devices[624]. The docs list CI guides for GitHub Actions, Bitrise, Bitbucket Pipelines, and CircleCI[625]. Studio and Cloud are products of the Maestro company. Read the vendor page for prices. I did not check them.

Appium 3 with WebdriverIO

Appium is a server that speaks the WebDriver protocol. Drivers do the device work: UiAutomator2 for Android and XCUITest for iOS[626]. Appium 3.9.0 needs Node `^20.19.0 || ^22.12.0 || >=24`[627]. WebdriverIO 10.1.0 needs Node 22.19 or newer[628]. The UiAutomator2 driver lists `appium ^3.0.0-rc.2` as a peer[629]. Appium is black box and has no app sync, so you add explicit waits.

```
// wdio test with Appium (not run)
import { remote } from 'webdriverio';

const driver = await remote({
  hostname: '127.0.0.1', port: 4723,
  capabilities: {
    platformName: 'Android',
    'appium:automationName': 'UiAutomator2',
    'appium:app': './android/app/build/outputs/apk/debug/app-debug.apk',
  },
});
```

[620] https://docs.maestro.dev/reference/commands-available/runflow.md (parameters label, env; inline commands with a when condition).

[621] https://docs.maestro.dev/maestro-flows/workspace-management/test-discovery-and-tags.md (top-level only by default, `flows` globs in config.yaml, tags metadata).

[622] https://docs.maestro.dev/get-started/how-maestro-works.md and https://docs.maestro.dev/get-started/what-is-maestro.md (read 2026-10-29).

[623] https://docs.maestro.dev/get-started/maestro-solutions.md and https://docs.maestro.dev/maestro-studio/maestro-studio-overview.md (read 2026-10-29).

[624] https://docs.maestro.dev/get-started/maestro-solutions.md and https://docs.maestro.dev/maestro-studio/maestro-studio-overview.md (read 2026-10-29).

[625] https://docs.maestro.dev/llms.txt, Maestro Cloud CI/CD pages: GitHub Actions, Bitrise, Bitbucket Pipelines, CircleCI.

[626] npm view appium, appium-uiautomator2-driver, appium-xcuitest-driver version time (2026-10-29). https://github.com/appium/appium

[627] npm view webdriverio, appium, appium-uiautomator2-driver version time engines peerDependencies (2026-10-29). Saved in scratch/e2e/deep/proof-deep-npm.txt.

[628] npm view webdriverio, appium, appium-uiautomator2-driver version time engines peerDependencies (2026-10-29). Saved in scratch/e2e/deep/proof-deep-npm.txt.

[629] npm view webdriverio, appium, appium-uiautomator2-driver version time engines peerDependencies (2026-10-29). Saved in scratch/e2e/deep/proof-deep-npm.txt.

```
const button = await driver.$('id=login-button');   // resource-id on Android
await button.waitForDisplayed({ timeout: 10000 });
await button.click();
await driver.deleteSession();
```

Note the selector detail. The Android resource-id is the `testID` string, set from RN code (see the selector table below)[630]. WebdriverIO selector strings such as `id=` and ~ come from WebdriverIO conventions. I did not verify them on a device. The UiAutomator2 driver README says the `id` strategy adds the app package prefix when you do not set one[631].

Where Appium fits: pick it when you need any language (Java, Python, C#), a device cloud that only offers WebDriver, or one suite for native and web. For a pure RN team, it brings more setup and more flakiness than Maestro or Detox.

Native frameworks and web

Espresso (Android) and XCUITest (iOS) are the platform frameworks. They run fast and sync with the platform UI. You write two suites in Kotlin or Java and Swift. Few RN teams choose them as the main suite. Detox maps Espresso APIs on Android[632].

For React Native Web, Playwright runs the web build in real browsers[633]. It covers only the web target.

Expo flows

Expo documents Maestro on EAS Workflows. The pattern: a `build` job with a profile that makes an `.apk` or a simulator `.app`, then a `maestro` job with `build_id` and `flow_path`[634]. Run the workflow with `eas workflow:run .eas/workflows/e2e-test-android.yml`[635]. In other CI systems, build a development or release build, install it, then run `maestro test`.

Test selectors: from testID to the native id

The `testID` prop is the stable hook for E2E tests. The RN source shows what each platform does with it.

[630] RN checkout at main 9c7b0f5fafd: packages/react-native/ReactCommon/react/renderer/components/view/AccessibilityProps.cpp:175-176; ReactAndroid/src/main/java/com/facebook/react/uimanager/BaseViewManager.java:334-340; ReactAndroid/src/main/java/com/facebook/react/uimanager/ReactAccessibilityDelegate.kt:175-181; React/Fabric/Mounting/ComponentViews/View/RCTViewComponentView.mm:589-596.

[631] https://github.com/appium/appium-uiautomator2-driver README, locator table: "Package identifier prefix is added automatically if unset" for the `id` strategy.

[632] Detox docs/architecture/client-server.md:146 (Android espressoapi generation) and :152.

[633] npm view playwright version, https://github.com/microsoft/playwright/releases/tag/v1.64.1

[634] https://docs.expo.dev/eas/workflows/examples/e2e-tests/ (last updated 2026-07-22).

[635] https://docs.expo.dev/eas/workflows/examples/e2e-tests/ (last updated 2026-07-22).

Layer	What happens	Source
Shared C++	`AccessibilityProps` reads the raw prop `testID` into `testId`	`ReactCommon/react/renderer/components/view/AccessibilityProps.cpp:175-176`[636]
Android view manager	`setTestId` stores the value as a view tag `R.id.react_test_id` and also as the legacy `view.setTag(testId)`	`BaseViewManager.java:334-340`[637]
Android accessibility	The accessibility delegate copies the tag to `info.viewIdResourceName`. This is the `resource-id` that UiAutomator, Appium, and Maestro read	`ReactAccessibilityDelegate.kt:175-181`[638]
iOS Fabric view	`RCTViewComponentView` sets `accessibilityIdentifier` on the accessibility element when `testId` changes	`RCTViewComponentView.mm:589-596`[639]

The Android code comment states the reason. Black box tools use the accessibility framework and cannot read view tags. So RN exposes `testID` as the resource-id and keeps the content description clean[640]. Detox `by.id` and Maestro `id:` both match this value. The Maestro docs say it maps `testID` to `id`[641].

Rules:

- Use `testID` for hooks. Use `accessibilityLabel` and roles when you also test screen reader behavior.

[636] RN checkout at main 9c7b0f5fafd: packages/react-native/ReactCommon/react/renderer/components/view/AccessibilityProps.cpp:175-176; ReactAndroid/src/main/java/com/facebook/react/uimanager/BaseViewManager.java:334-340; ReactAndroid/src/main/java/com/facebook/react/uimanager/ReactAccessibilityDelegate.kt:175-181; React/Fabric/Mounting/ComponentViews/View/RCTViewComponentView.mm:589-596.

[637] RN checkout at main 9c7b0f5fafd: packages/react-native/ReactCommon/react/renderer/components/view/AccessibilityProps.cpp:175-176; ReactAndroid/src/main/java/com/facebook/react/uimanager/BaseViewManager.java:334-340; ReactAndroid/src/main/java/com/facebook/react/uimanager/ReactAccessibilityDelegate.kt:175-181; React/Fabric/Mounting/ComponentViews/View/RCTViewComponentView.mm:589-596.

[638] RN checkout at main 9c7b0f5fafd: packages/react-native/ReactCommon/react/renderer/components/view/AccessibilityProps.cpp:175-176; ReactAndroid/src/main/java/com/facebook/react/uimanager/BaseViewManager.java:334-340; ReactAndroid/src/main/java/com/facebook/react/uimanager/ReactAccessibilityDelegate.kt:175-181; React/Fabric/Mounting/ComponentViews/View/RCTViewComponentView.mm:589-596.

[639] RN checkout at main 9c7b0f5fafd: packages/react-native/ReactCommon/react/renderer/components/view/AccessibilityProps.cpp:175-176; ReactAndroid/src/main/java/com/facebook/react/uimanager/BaseViewManager.java:334-340; ReactAndroid/src/main/java/com/facebook/react/uimanager/ReactAccessibilityDelegate.kt:175-181; React/Fabric/Mounting/ComponentViews/View/RCTViewComponentView.mm:589-596.

[640] RN checkout at main 9c7b0f5fafd: packages/react-native/ReactCommon/react/renderer/components/view/AccessibilityProps.cpp:175-176; ReactAndroid/src/main/java/com/facebook/react/uimanager/BaseViewManager.java:334-340; ReactAndroid/src/main/java/com/facebook/react/uimanager/ReactAccessibilityDelegate.kt:175-181; React/Fabric/Mounting/ComponentViews/View/RCTViewComponentView.mm:589-596.

[641] https://docs.maestro.dev/get-started/supported-platform/react-native.md ("the best practice is to use the testID property, which Maestro maps to a unique id").

- Text selectors break when copy or language changes.
- Put `testID` on the pressable element, not on a wrapper. A wrapper may merge or hide the child in the accessibility tree.

Flakiness: causes and fixes

Cause	Fix
Animations	Turn them off in the test build. Detox waits for them. An endless loop blocks Detox sync, so disable the loop.
Fixed sleeps	Wait on a visible state. Maestro and Detox wait by default. In Appium use explicit waits.
Network	Mock the server or use a fixed test backend. Check that network sync sees your HTTP client (see issue 4973)[642].
Shared state	Reset app data per test: `clearState` in Maestro, `newInstance: true` in Detox
Cold simulator	Boot and warm the device before the run
Debug build with Metro	Test a release-like build in CI. A Metro bundle load adds time and a red box can block the screen.
Version drift	Pin RN, the Detox version, and the OS image. Detox reflection on RN internals can break on a minor RN upgrade[643].
Keyboard and system dialogs	Black box tools see system dialogs. Grant permissions in `launchApp` and dismiss dialogs in a conditional `runFlow`[644].
Parallel runs on shared emulators	Give each worker its own device

Measure flakiness. Re-run the failed test once in CI. Track tests that pass on a re-run, and fix or delete them.

[642]https://github.com/wix/Detox/issues/4963 (opened 2026-07-02, updated 2026-09-03, open) and /issues/4973 (open), read with gh issue list/view on 2026-10-29. Search for RN 0.86 to 0.88 returned no matching Detox issue or release note. Releases: gh release list -R wix/Detox, latest 20.51.3 on 2026-05-30.

[643]https://github.com/wix/Detox/issues/4963 (opened 2026-07-02, updated 2026-09-03, open) and /issues/4973 (open), read with gh issue list/view on 2026-10-29. Search for RN 0.86 to 0.88 returned no matching Detox issue or release note. Releases: gh release list -R wix/Detox, latest 20.51.3 on 2026-05-30.

[644]https://docs.maestro.dev/get-started/supported-platform/ios.md (read 2026-10-29): Xcode simulators, permissions, multi-app journeys.

Visual regression: what exists for RN

Option	What it does	Status (2026-10-29)
Maestro `assertScreenshot`	Compares the screen with a stored image inside a flow[645]	Documented in the Maestro command list. Not run.
Maestro `takeScreenshot` and Detox `device.takeScreenshot`	Save an image. You compare it yourself or upload it[646][647]	Documented. Not run.
Storybook for React Native 10.7.0	Renders components on device. Gives stable states to shoot[648]	Maintained
Chromatic	Cloud diff service. Web first. Vendor product[649]	Maintained. RN path not verified.
Argos	Cloud diff service. Open source CLI[650]	Maintained. RN path not verified.
react-native-owl	Screenshot tests on device[651]	Stale. Last repo push 2025-01-15[652]

My advice: capture screenshots from fixed states (Storybook stories or a Maestro flow on a seeded app), run on one pinned device image, and mask dynamic regions. Fonts and OS versions change pixels, so a device change makes a full baseline reset.

Decision table

The ratings below are my judgment, not measured data.

[645] https://docs.maestro.dev/llms.txt entries for assertScreenshot and takeScreenshot (read 2026-10-29). Not run.
[646] https://docs.maestro.dev/llms.txt entries for assertScreenshot and takeScreenshot (read 2026-10-29). Not run.
[647] Detox docs/api/device.md:517 `device.takeScreenshot([name])`, repo HEAD fc34e84.
[648] npm view @storybook/react-native version, https://github.com/storybookjs/react-native/releases/tag/v10.7.0
[649] npm view chromatic, @argos-ci/cli, react-native-owl version time (2026-10-29).
[650] npm view chromatic, @argos-ci/cli, react-native-owl version time (2026-10-29).
[651] npm view chromatic, @argos-ci/cli, react-native-owl version time (2026-10-29).
[652] gh api repos/FormidableLabs/react-native-owl: pushed_at 2025-01-15T16:33:42Z, archived false (2026-10-29).

Tool	Setup time	Speed	Flakiness risk	Language	CI fit	New Architecture status
Maestro	Low. Single CLI[653]	Medium	Low	YAML	Good. Free CLI, paid Cloud, guides for four CI systems[654]	Black box. No app code. Docs claim full RN support[655]
Detox	High. Native build and config	Fast	Low if sync works. Medium on new RN versions	JS/TS	Good. Needs macOS runners for iOS	Documented to RN 0.84. Not documented for 0.88[656][657]
Appium 3 + WDIO	High. Server, drivers, caps	Slow	Medium. No app sync	Any	Wide. Many device clouds offer it (unverified)	Black box. No impact
Espresso / XCUITest	High. Two suites	Fast	Low	Kotlin, Swift	Good	Black box. No impact
Playwright	Low	Fast	Low	JS/TS	Good	Web only

For most RN teams in late 2026, start with Maestro. Choose Detox when you need in-process sync and accept the version risk. Choose Appium when you need other languages or a WebDriver-only cloud.

AI-native test services are not in this table. I could not source every cell, for example flakiness and New Architecture status. See "AI-native test services: Autosana and peers" in Part 4.[658]

Part 3: History, device clouds, and CI

Check date: 2026-10-29. Target: React Native 0.88.1 with React 19.3 and the New Architecture only.[659] Dates below come from the npm registry "time" field, GitHub repo metadata, and git history of the RN checkout. Where a date is the first npm publish, the text says so.

[653] https://docs.maestro.dev/get-started/how-maestro-works.md and https://docs.maestro.dev/get-started/what-is-maestro.md (read 2026-10-29).

[654] https://docs.maestro.dev/llms.txt, Maestro Cloud CI/CD pages: GitHub Actions, Bitrise, Bitbucket Pipelines, CircleCI.

[655] https://docs.maestro.dev/get-started/supported-platform/react-native.md ("the best practice is to use the testID property, which Maestro maps to a unique id").

[656] Detox README.md:66-67, repo HEAD fc34e84.

[657] https://github.com/wix/Detox/issues/4963 (opened 2026-07-02, updated 2026-09-03, open) and /issues/4973 (open), read with gh issue list/view on 2026-10-29. Search for RN 0.86 to 0.88 returned no matching Detox issue or release note. Releases: gh release list -R wix/Detox, latest 20.51.3 on 2026-05-30.

[658] https://www.ycombinator.com/companies/autosana (fetched 2026-10-29): Summer 2025, founded 2025, team size 5, founders Yuvan Sundrani and Jason Steinberg. The launch post says "Fully vision based". Vendor text.

[659] `npm view react-native@0.88.1 peerDependencies` (react ^19.3.0) and `npm view @react-native/jest-preset time`, checked 2026-10-29. Proof: scratch/history/proof-npm.txt.

Timeline, 2015 to 2026

Date	Event	Source
2013-01	Appium repo created. It is the long-running WebDriver based tool for native apps.	[660]
2015-03-05	Detox name first published on npm at 1.0.0 (an early package). The Wix Detox repo itself was created 2016-06-15.	[661]
2015-03-26	react-native 0.1.0 on npm.	[662]
2015-12-06	enzyme 1.0.0 on npm. Its last release, 3.11.0, is from 2019-12-20.	[663]
2016-05-17	react-test-renderer 1.0.0 on npm. It renders a component tree to plain JS objects without a DOM or a device.	[664]
2016-06-19	Detox 2.0.0 on npm. This is the start of the gray box E2E tool for RN.	[665]
2016-10-31	RN adds a Jest preset commit "so that people can configure Jest using react-native as preset".	[666]
2017-01-30	Cavy 0.0.1 on npm. Cavy is an in-app JS test runner. Last release 4.0.2 on 2021-10-03. Treat it as low activity.	[667]
2017-03-03	jest-expo first publish on npm.	[668]
2018-10-01	callstack/react-native-testing-library repo created. First npm publish 2018-10-03.	[669]
2019-04-01	native-testing-library 1.0.0 on npm. It is now deprecated in favor of @testing-library/react-native (deprecation date not checked).	[670]
2019-05-30	@testing-library/react-native first publish (4.0.0).	[671]
2020-07-30	react-native-testing-library 6.0.0 (last release, deprecated) and @testing-library/react-native 7.0.0 on the same day.	[672]
2022-03-31	Maestro repo (mobile-dev-inc/Maestro) created.	[673]
2024-12-05	React 19.0.0 stable on npm. react-test-renderer prints a deprecation warning from React 19.	[674]
2024-12-06	RN adds the @react-native/fantom package (moved to private/ on 2025-06-12 UTC).	[675]
2025-01-03	@testing-library/react-native 13.0.0. It supports React 18 and 19 (peers: react >=18.2, react-native >=0.71) and drops React 16 and 17.	[676]
2026-02-18	RN splits the Jest preset into @react-native/jest-preset (first nightly 0.85.0-nightly on 2026-02-19).	[677]
2026-06-05	@testing-library/react-native 14.0.0. It drops React 18, needs RN 0.78 or newer, and replaces react-test-renderer with test-renderer. Version 14.1.0 is current (2026-10-08).	[678]
2026-10-15	react-native 0.88.0. Patch 0.88.1 on 2026-10-27.	[679]

[660] Appium repo created 2013-01-09, last push 2026-10-28, not archived. GitHub API https://api.github.com/repos/appium/appium, checked 2026-10-29. npm appium 3.9.0 on 2026-10-27. Calabash and Frank dates: not verified.

[661] `npm view detox time`: 1.0.0 2015-03-05, 2.0.0 2016-06-19. Repo wix/Detox created 2016-06-15, last push 2026-10-28 (GitHub API). Detox 20.51.4 current.

[662] `npm view react-native time`: 0.1.0 2015-03-26, 0.88.0 2026-10-15, 0.88.1 2026-10-27.

[663] `npm view enzyme time`: 1.0.0 2015-12-06, 3.11.0 2019-12-20. The earlier claim about adapters for React 18 or 19 was removed (not proven).

[664] `npm view react-test-renderer time`: 1.0.0 2016-05-17.

I did not verify these items. Mark them as unchecked in the book: the first Jest use at Facebook for RN, the dates of Frank and Calabash, and the exact EAS Build and jest-expo SDK milestones. The npm "time" field only proves a publish date, not a design decision.

The eras in plain words

Era 1, 2015 to 2017: unit tests with Jest and shallow renders. Teams used enzyme and react-test-renderer. For device tests they used Appium (WebDriver), and older native tools such as Calabash and Frank.[680] These tools were slow and flaky on JS driven UI. Detox (2016) added a "gray box" model. It knows when the app is idle. That cut flaky waits.[681]

---

[665] `npm view detox time`: 1.0.0 2015-03-05, 2.0.0 2016-06-19. Repo wix/Detox created 2016-06-15, last push 2026-10-28 (GitHub API). Detox 20.51.4 current.

[666] RN git history, commands run in the checkout: `git log --diff-filter=A` on packages/jest-preset/package.json (c4ae05534ad, 2026-02-18, "Split out Jest preset to @react-native/jest-preset (#55169)"), on packages/react-native-fantom/package.json (b253b0fe945, 2024-12-06), `git log` on jest-preset.json (6d3e074dd45, 2016-10-31), and d6f29c8afd1 (2025-06-11, move to private). Commit 714b502b0c7 (2023-03-17, "RN Monorepo | Migrate to package") is a monorepo move, not a Jest change. The chapter no longer claims it. The 2025-06 Fantom move is d6f29c8afd1, authored 2025-06-12T06:49Z (GitHub API).

[667] `npm view cavy time`: 0.0.1 2017-01-30, 4.0.2 2021-10-03.

[668] `npm view jest-expo time version`: first 0.3.0-beta.0 2017-03-03, 58.0.13 on 2026-10-28.

[669] `npm view @testing-library/react-native time` and `npm view react-native-testing-library time` (6.0.0 2020-07-30, deprecated). Repo callstack/react-native-testing-library created 2018-10-01, last push 2026-10-20.

[670] `npm view native-testing-library deprecated`: "has moved to @testing-library/react-native". 1.0.0 2019-04-01, 4.0.0 2019-05-31. See https://github.com/testing-library/dom-testing-library/issues/260.

[671] `npm view @testing-library/react-native time` and `npm view react-native-testing-library time` (6.0.0 2020-07-30, deprecated). Repo callstack/react-native-testing-library created 2018-10-01, last push 2026-10-20.

[672] `npm view @testing-library/react-native time` and `npm view react-native-testing-library time` (6.0.0 2020-07-30, deprecated). Repo callstack/react-native-testing-library created 2018-10-01, last push 2026-10-20.

[673] Repo mobile-dev-inc/Maestro created 2022-03-31, last push 2026-10-28, not archived. Release cli-2.12.0 on 2026-10-26 (GitHub API).

[674] `npm view react time`: 19.0.0 on 2024-12-05.

[675] RN git history, commands run in the checkout: `git log --diff-filter=A` on packages/jest-preset/package.json (c4ae05534ad, 2026-02-18, "Split out Jest preset to @react-native/jest-preset (#55169)"), on packages/react-native-fantom/package.json (b253b0fe945, 2024-12-06), `git log` on jest-preset.json (6d3e074dd45, 2016-10-31), and d6f29c8afd1 (2025-06-11, move to private). Commit 714b502b0c7 (2023-03-17, "RN Monorepo | Migrate to package") is a monorepo move, not a Jest change. The chapter no longer claims it. The 2025-06 Fantom move is d6f29c8afd1, authored 2025-06-12T06:49Z (GitHub API).

[676] `npm view @testing-library/react-native time` and `npm view react-native-testing-library time` (6.0.0 2020-07-30, deprecated). Repo callstack/react-native-testing-library created 2018-10-01, last push 2026-10-20.

[677] RN git history, commands run in the checkout: `git log --diff-filter=A` on packages/jest-preset/package.json (c4ae05534ad, 2026-02-18, "Split out Jest preset to @react-native/jest-preset (#55169)"), on packages/react-native-fantom/package.json (b253b0fe945, 2024-12-06), `git log` on jest-preset.json (6d3e074dd45, 2016-10-31), and d6f29c8afd1 (2025-06-11, move to private). Commit 714b502b0c7 (2023-03-17, "RN Monorepo | Migrate to package") is a monorepo move, not a Jest change. The chapter no longer claims it. The 2025-06 Fantom move is d6f29c8afd1, authored 2025-06-12T06:49Z (GitHub API).

[678] `npm view @testing-library/react-native time` and `npm view react-native-testing-library time` (6.0.0 2020-07-30, deprecated). Repo callstack/react-native-testing-library created 2018-10-01, last push 2026-10-20.

[679] `npm view react-native time`: 0.1.0 2015-03-26, 0.88.0 2026-10-15, 0.88.1 2026-10-27.

[680] Appium repo created 2013-01-09, last push 2026-10-28, not archived. GitHub API https://api.github.com/repos/appium/appium, checked 2026-10-29. npm appium 3.9.0 on 2026-10-27. Calabash and Frank dates: not verified.

[681] `npm view detox time`: 1.0.0 2015-03-05, 2.0.0 2016-06-19. Repo wix/Detox created 2016-06-15, last push 2026-10-28 (GitHub API). Detox 20.51.4 current.

Era 2, 2016 to 2019: snapshot tests. Jest snapshots and react-test-renderer "toJSON" made big snapshot suites. They are cheap to write. They also break on every style change, and teams learn to ignore the diffs. This is an opinion, not a measured result.

Era 3, 2019 to now: user centric tests. Testing Library queries by role, text, and label. The old "native-testing-library" and "react-native-testing-library" packages now point to @testing-library/react-native.[682]

Era 4, 2022 to now: black box YAML flows. Maestro runs flows from a YAML file and needs no app code change.[683] Detox and Appium stay in use.

Era 5, 2024 to now: the React 19 and New Architecture shift. See the next two parts.

react-test-renderer and React 19

React 19 deprecates react-test-renderer. A warning fires when you call "ReactTestRenderer.create()". The package stays on npm, is not maintained, and "may break with new React features".[684] The npm registry does not flag version 19.3.1 as deprecated (`npm view react-test-renderer@19.3.1 deprecated` is empty). The deprecation is a React warning, not an npm flag. Action: use @testing-library/react-native for new tests. Version 14.1.0 lists "test-renderer ^1.0.0" as a peer dependency and no longer lists react-test-renderer.[685] Version 13.x still lists react-test-renderer as a peer. I did not run a migration.

Jest preset and Fantom

Until RN 0.84 the preset lived inside the react-native package ("preset: 'react-native'", since a 2016-10-31 commit). Since 2026-02-18 the preset is its own package, @react-native/jest-preset.[686] The RN 0.88.1 checkout has it at packages/jest-preset, and its README says to set "preset: '@react-native/jest-preset'" in jest.config.js.[687] The version on npm for 0.88.x is 0.88.1.[688]

Fantom is the RN team's integration test runner. It runs JS tests against a real native tester build, as Platform.OS android. It is private to the RN repo (private/react-native-fantom), not on npm

---

[682] `npm view native-testing-library deprecated`: "has moved to @testing-library/react-native". 1.0.0 2019-04-01, 4.0.0 2019-05-31. See https://github.com/testing-library/dom-testing-library/issues/260.

[683] Repo mobile-dev-inc/Maestro created 2022-03-31, last push 2026-10-28, not archived. Release cli-2.12.0 on 2026-10-26 (GitHub API).

[684] https://react.dev/warnings/react-test-renderer (fetched in full 2026-10-29). `npm view react-test-renderer@19.3.1 deprecated`: empty. Proof: tmp/refresh/testing/scratch/FC-1/npm2.txt.

[685] `npm view @testing-library/react-native@14.1.0 peerDependencies`: jest >=29, react >=19, react-native >=0.78, test-renderer ^1.0.0.

[686] RN git history, commands run in the checkout: `git log --diff-filter=A` on packages/jest-preset/package.json (c4ae05534ad, 2026-02-18, "Split out Jest preset to @react-native/jest-preset (#55169)"), on packages/react-native-fantom/package.json (b253b0fe945, 2024-12-06), `git log` on jest-preset.json (6d3e074dd45, 2016-10-31), and d6f29c8afd1 (2025-06-11, move to private). Commit 714b502b0c7 (2023-03-17, "RN Monorepo | Migrate to package") is a monorepo move, not a Jest change. The chapter no longer claims it. The 2025-06 Fantom move is d6f29c8afd1, authored 2025-06-12T06:49Z (GitHub API).

[687] packages/jest-preset/README.md:30-38 and package.json (name, engines node >= 22.13.0) in the RN checkout. The checkout package.json says version 0.87.0-main while npm has 0.88.1, so the local file is older than the tag.

[688] `npm view react-native@0.88.1 peerDependencies` (react ^19.3.0) and `npm view @react-native/jest-preset time`, checked 2026-10-29. Proof: scratch/history/proof-npm.txt.

(E404 on 2026-10-29), and not a tool for app teams.[689] Do not put it in an app test plan.

What the New Architecture changed for mocks

With the legacy bridge, teams mocked "NativeModules" as a plain object. With TurboModules, a module is loaded through a JS spec. A mock must match the spec name and shape, or the test fails at import time. The RN jest setup mocks core modules, so most app tests run without extra work.[690] Mock third party native modules with "jest.mock('name', factory)" in a setup file. I did not run a TurboModule mock proof for this part. The other parts of the chapter cover that.

Expo tooling

Tool	What it does	Status as of 2026-10-29
jest-expo	Jest preset for Expo apps.	Version 58.0.13, published 2026-10-28.[691]
EAS Build	Cloud native builds.	Vendor docs, in use.[692]
EAS Workflows	YAML CI/CD service that runs builds, updates, submissions, and Maestro tests. Files live in ".eas/workflows/".	Docs updated 2026-07-22. Known limits: no shared configs, no matrix builds.[693]

EAS Workflows runs on Linux workers (linux-medium 4 vCPU, linux-large 8 vCPU), Linux workers with nested virtualization for Android emulators, and macOS workers (macos-medium, macos-large).[694] This is the vendor description.

Device clouds and CI services, late 2026

Pricing types come from search snippets on 2026-10-29. Only the GitHub and EAS pages were read in full. Rows marked unverified are not proven. Exact prices change often. Check the vendor page before you quote a number.

[689] private/react-native-fantom exists in the checkout. `npm view @react-native/fantom` returns E404. Platform.OS android claim: AGENTS.md of the repo.

[690] packages/jest-preset/README.md:30-38 and package.json (name, engines node >= 22.13.0) in the RN checkout. The checkout package.json says version 0.87.0-main while npm has 0.88.1, so the local file is older than the tag.

[691] `npm view jest-expo time version`: first 0.3.0-beta.0 2017-03-03, 58.0.13 on 2026-10-28.

[692] https://docs.expo.dev/eas/ (not fetched in detail). Date of first EAS Build not verified.

[693] https://docs.expo.dev/eas/workflows/introduction/ fetched 2026-10-29 (page dated 2026-07-22). Pricing per build: https://expo.dev/blog/best-ci-cd-for-mobile-apps (search snippet, vendor blog).

[694] https://docs.expo.dev/eas/workflows/introduction/ fetched 2026-10-29 (page dated 2026-07-22). Pricing per build: https://expo.dev/blog/best-ci-cd-for-mobile-apps (search snippet, vendor blog).

Service	What it offers for RN	Pricing model type	Notable limits
BrowserStack App Automate	Real devices. Appium, Espresso, XCUITest.[695]	Subscription priced per parallel session (third party report).	Parallels are priced one by one. You still write and keep the tests.
Sauce Labs	Real and virtual devices. Appium.	Subscription with parallel concurrency (vendor comparison page).[696]	Vendor pages are marketing for comparison.
LambdaTest (now branded TestMu AI)	Real device cloud. Appium.	Subscription with parallel tests.[697]	Rename can confuse docs search.
AWS Device Farm	Real devices. Appium and native frameworks.	Pay per device minute, with a one time free trial (unverified, search snippet).[698]	No Maestro native flow. You pack the Appium or native tests.
Firebase Test Lab	Android devices, iOS devices, Robo test.	Billed by device time with a daily no cost quota (unverified, search snippet).[699]	Google tools fit Android best.
Maestro Cloud	Hosted devices for Maestro flows, parallel runs, PR hooks.	Parallel based cloud plans. Maestro itself is open source (unverified, search snippet).[700]	Local runs are free. Cloud cost starts when you need parallel runs or hosted devices.
Bitrise	Mobile CI with macOS and Linux stacks.	Credit based plans (not fetched, claim from a comparison page).[701]	I did not read the Bitrise pricing page. Treat as unverified.
Codemagic	Mobile CI, M2 and M4 macOS machines.	Pay as you go or fixed plan (unverified, search snippet).[702]	Per minute prices vary by machine type.
GitHub Actions	General CI with macOS and Linux runners.	Per minute by runner type. Standard macOS 3 or 4 core is listed at 0.062 USD per minute and Linux 2 core at 0.006 USD per minute on the billing page I fetched today.[703]	macOS minutes cost about 10 times Linux 2-core minutes (0.062 / 0.006). Larger runners are not free for public repos.
EAS Workflows	RN specific CI. Builds, updates, Maestro jobs.	Credits and per build charge (unverified, from a search snippet of a vendor blog).[704]	No matrix builds. Best for Expo apps.

[695]Search snippets only: https://bug0.com/knowledge-base/browserstack-app-automate and https://saucelabs.com/resources/blog/browserstack-vs-testmu-ai-formerly-lambdatest-vs-sauce-labs-which-cloud-testing-grid-fits-your-team. Third party and vendor sources. Vendor pages not fetched.

[696]Search snippets only: https://bug0.com/knowledge-base/browserstack-app-automate and https://saucelabs.com/resources/blog/browserstack-vs-testmu-ai-formerly-lambdatest-vs-sauce-labs-which-cloud-testing-grid-fits-your-team. Third party and vendor sources. Vendor pages not fetched.

[697]Search snippets only: https://bug0.com/knowledge-base/browserstack-app-automate and https://saucelabs.com/resources/blog/browserstack-vs-testmu-ai-formerly-lambdatest-vs-sauce-labs-which-cloud-testing-grid-fits-your-team. Third party and vendor sources. Vendor pages not fetched.

GitHub Actions notes. The macOS runner is the cost driver, so run lint, type check, and unit tests on Linux. Android emulators run on Linux runners with KVM, which avoids macOS cost. iOS simulators need macOS. Cache Gradle, CocoaPods, and the Node package manager. Cache the AVD snapshot if the emulator action supports it. I did not measure cache gains.

A sensible CI pipeline for an RN app

Order jobs from cheap and fast to slow and costly. Stop early on failure.

Stage	Runs on	What it checks
Lint and type check	Linux	ESLint, TypeScript.
Unit and component tests	Linux	Jest with the RN preset and Testing Library.
Build	Linux (Android), macOS (iOS)	Release build compiles.
E2E smoke	Linux emulator, macOS simulator	One to five critical flows (launch, sign in, main screen).
Release gate	Any	Manual approval, then store upload (for example EAS or Fastlane).

Keep the E2E smoke set small. Run the full E2E suite on a nightly schedule or on a device cloud. This is a team choice, not a rule from a source.

Example workflow. The first version of this YAML parsed with js-yaml in Node with three jobs (static, android-e2e, release-gate). The edited version also parsed with PyYAML (same three jobs). I did not run it on GitHub, so treat the emulator and Maestro steps as not run. The Maestro flow file also parsed as two YAML documents (config header and steps).[705]

```yaml
name: ci
on:
```

[698]https://aws.amazon.com/device-farm/pricing/ (search snippet 2026-10-29: pay for what you use, one time free trial).

[699]https://firebase.google.com/docs/test-lab/usage-quotas-pricing (search snippet 2026-10-29: billing by minutes, daily no cost limit on Blaze).

[700]https://maestro.dev/pricing (search snippet 2026-10-29: free open source, Maestro Cloud for parallel execution).

[701]https://codemagic.io/pricing/ and https://docs.codemagic.io/billing/pricing/ (search snippets 2026-10-29). Bitrise: only a Codemagic comparison page and a third party snippet.

[702]https://codemagic.io/pricing/ and https://docs.codemagic.io/billing/pricing/ (search snippets 2026-10-29). Bitrise: only a Codemagic comparison page and a third party snippet.

[703]https://docs.github.com/en/billing/reference/actions-runner-pricing fetched 2026-10-29.

[704]https://docs.expo.dev/eas/workflows/introduction/ fetched 2026-10-29 (page dated 2026-07-22). Pricing per build: https://expo.dev/blog/best-ci-cd-for-mobile-apps (search snippet, vendor blog).

[705]scratch/history/proof-yaml.txt, ci.yml and smoke.yaml, parsed with js-yaml via Node. Edited YAML reparsed with PyYAML: tmp/refresh/testing/scratch/FC-1/ci.yml. Action versions from `https://api.github.com/repos/<repo>/releases/latest` on 2026-10-29: actions/checkout v7.0.1, actions/setup-node v7.1.0, actions/setup-java v6.1.0, ReactiveCircus/android-emulator-runner v2.38.0. Emulator runner README: https://github.com/ReactiveCircus/android-emulator-runner

```
  pull_request:
  push:
    branches: [main]
concurrency:
  group: ci-${{ github.ref }}
  cancel-in-progress: true
jobs:
  static:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v7
      - uses: actions/setup-node@v7
        with:
          node-version: 22
          cache: yarn
      - run: yarn install --frozen-lockfile
      - run: yarn lint --max-warnings 0
      - run: yarn tsc --noEmit
      - run: yarn jest --ci --coverage
  android-e2e:
    needs: static
    runs-on: ubuntu-latest
    timeout-minutes: 45
    steps:
      - uses: actions/checkout@v7
      - uses: actions/setup-node@v7
        with:
          node-version: 22
          cache: yarn
      - uses: actions/setup-java@v6
        with:
          distribution: temurin
          java-version: 17
      - run: yarn install --frozen-lockfile
      - run: cd android && ./gradlew assembleRelease
      - name: Enable KVM
        run: |
          echo 'KERNEL=="kvm", GROUP="kvm", MODE="0666",
OPTIONS+="static_node=kvm"' | sudo tee /etc/udev/rules.d/99-kvm4all.rules
          sudo udevadm control --reload-rules
          sudo udevadm trigger --name-match=kvm
      - run: curl -Ls https://get.maestro.mobile.dev | bash
      - uses: reactivecircus/android-emulator-runner@v2
        with:
          api-level: 34
          arch: x86_64
          script: |
            adb install android/app/build/outputs/apk/release/app-release.apk
            $HOME/.maestro/bin/maestro test .maestro/smoke.yaml
```

```yaml
  release-gate:
    needs: [static, android-e2e]
    if: github.ref == 'refs/heads/main'
    runs-on: ubuntu-latest
    environment: production
    steps:
      - run: echo "Approval gate. Run eas build and submit here."
```

The emulator runner README (checked 2026-10-29) says to enable KVM first and to prefer Ubuntu runners. It runs each line of `script` as a separate shell command, so an `export PATH` on one line does not carry to the next. The sample uses the full path to avoid this. Action versions (checkout 7.0.1, setup-node 7.1.0, setup-java 6.1.0, emulator-runner 2.38.0) are the latest GitHub releases on 2026-10-29.[706] The edited YAML was not re-run on GitHub. The first version of this sample had `v4` action tags and the `export PATH` line.

Part 4: Agentic AI and testing

This section is about what exists on 2026-10-29, not about what vendors promise. Each tool below has a checked version, license, and date.[707] A coding agent is a language model that reads files, runs commands, and calls tools. A tool server speaks the Model Context Protocol (MCP) and lets the agent touch a device.

What changes and what does not

Agents change who writes and runs tests. They do not change what a good test is. A test must fail when the app is wrong and pass when the app is right. An agent can write a test that always passes. So the human still owns the oracle, the rule that says what "right" means.

MCP servers for devices and simulators

An MCP server turns a simulator or emulator into a set of tools. Most servers expose the same four things: a screenshot, a view tree, input (tap, type, swipe), and logs. The table compares the servers I checked.

[706] scratch/history/proof-yaml.txt, ci.yml and smoke.yaml, parsed with js-yaml via Node. Edited YAML re-parsed with PyYAML: tmp/refresh/testing/scratch/FC-1/ci.yml. Action versions from `https://api.github.com/repos/<repo>/releases/latest` on 2026-10-29: actions/checkout v7.0.1, actions/setup-node v7.1.0, actions/setup-java v6.1.0, ReactiveCircus/android-emulator-runner v2.38.0. Emulator runner README: https://github.com/ReactiveCircus/android-emulator-runner

[707] Checked 2026-10-29 with `npm view <pkg> version license time.modified` and the GitHub API. Output saved in `tmp/refresh/testing/scratch/ai/proof-repos.txt`.

Server	What it exposes	License	Status on 2026-10-29
mobile-mcp (mobile-next)	Screenshot, element list with refs, tap, type, swipe, app install and launch, device logs, crash reports, screen recording, remote device cloud tools. 34 tools listed by two runs.[708]	Apache-2.0	npm 1.0.14, released 2026-10-26. Last commit 2026-10-26. About 10k GitHub stars (project data, not a quality signal).[709]
appium-mcp (Appium project)	Appium session, find element, gesture, page source, screenshot, screen recording, locator generation, test code generation. 32 tools listed by my run. A vision-based find tool exists only when `AI_VISION_ENABLED=true`.[710]	Apache-2.0	npm 1.99.1, released 2026-10-27. Needs Node 22 or later. The README says it is for a local single-user server or a trusted CI job.[711]
Maestro MCP	Inside the Maestro CLI (`maestro mcp`). Tools: `list_devices`, `inspect_screen`, `take_screenshot`, `run` (inline YAML), `cheat_sheet`, cloud run tools, `open_maestro_viewer`.[712]	Apache-2.0 (Maestro repo)	Maestro CLI 2.12.0, released 2026-10-26.[713]
Expo MCP	Remote server `https://mcp.expo.dev/mcp` (OAuth, Expo account needed). The docs table lists more than 40 server tools for docs search, EAS builds and workflows, EAS Observe, store reviews and crash data. Some tools bill EAS minutes. Local tools need the `expo-mcp` dev package and `EXPO_UNSTABLE_MCP_SERVER=1`: screenshot, tap by `testID`, find view by `testID` or label, open DevTools, collect logs, router sitemap.[714]	MIT (`expo-mcp` package)	npm `expo-mcp` latest 0.2.4 (2026-02-12). The `next` tag is 0.3.1 (2026-06-26). The repo has 9 stars and its last push was 2026-07-29. The env flag name contains UNSTABLE. Local features need SDK 54 or later. iOS is simulator only.[715]
XcodeBuildMCP (now MobileBuildMCP, Sentry)	Build, test, and run for iOS and macOS projects. Also a CLI. 24 tools listed by my run with no project config: session defaults, simulator build, run, test, install, launch, screenshot, UI snapshot, coverage.[716]	MIT	The npm package is now `mobilebuildmcp` 2.7.1 (2026-09-23). The old name `xcodebuildmcp` stopped at 2.7.0 on 2026-07-23. Last commit 2026-10-22. Needs macOS 14.5 and Xcode 16 or later. It builds native projects. It does not drive an RN-specific layer.[717]
ios-simulator-mcp (joshuayoes)	iOS simulator control: describe screen, tap, type, swipe, screenshot, video, install, launch. 17 tools listed by my run. Community project, not from Apple. Its README uses Facebook idb.[718]	MIT	npm 2.2.0, released 2026-10-27. Last commit 2026-10-27.[719]

Notes on the table:

- Expo MCP is not a bare React Native tool. It needs an Expo project. A bare React Native app should use mobile-mcp, Maestro MCP, or appium-mcp.
- Expo docs say the capability list "may not be up to date".[720] Read the live tool list from your own install.
- Community servers for React Native DevTools and Metro exist (for example `pnarayanaswamy/react-native-devtools-mcp`). I found them by web search and did not install or vet them. Treat them as unreviewed.[721]

---

[708] Run on 2026-10-29: `@mobilenext/mobile-mcp` started over stdio, `tools/list` returned 34 tools. The first run (`scratch/ai/proof-mobile-mcp-tools.txt`) printed banner 1.0.11 and sdk 2.0.0, so it ran an older cached copy and not 1.0.14. A clean install with `--before` (`scratch/FC-2/mm`, `mm-tools.txt`) printed banner 1.0.14 and sdk 2.4.0, and also listed 34 tools. Only two tool descriptions differ (`get_more_tools`, `mobile_get_foreground_app`). The banner reads the version from package.json (`lib/server.js` `getAgentVersion`). The wrong package run (`mobile-mcp` 0.0.7, 9 tools) is in `proof-WRONGPKG-runablehq-mobile-mcp.txt`.

[709] GitHub API and npm registry, checked 2026-10-29: mobile-next/mobile-mcp release 1.0.14 (2026-10-26), Apache-2.0. mobile-dev-inc/Maestro release cli-2.12.0 (2026-10-26), Apache-2.0. joshuayoes/ios-simulator-mcp v2.2.0 (2026-10-27), MIT. See `proof-repos.txt`.

[710] https://github.com/appium/appium-mcp README (fetched 2026-10-29). npm `appium-mcp` 1.99.1, Apache-2.0, last commit 2026-10-27. Run of `tools/list` on an installed 1.99.1 gave 32 tools: `scratch/FC-2/am/appium-tools.txt`. README sections on `AI_VISION_ENABLED`, `NO_UI` and the remote server trust model confirm the claims.

[711] https://github.com/appium/appium-mcp README (fetched 2026-10-29). npm `appium-mcp` 1.99.1, Apache-2.0, last commit 2026-10-27. Run of `tools/list` on an installed 1.99.1 gave 32 tools: `scratch/FC-2/am/appium-tools.txt`. README sections on `AI_VISION_ENABLED`, `NO_UI` and the remote server trust model confirm the claims.

[712] https://docs.maestro.dev/get-started/maestro-mcp (fetched 2026-10-29, "Last updated 3 days ago").

[713] GitHub API and npm registry, checked 2026-10-29: mobile-next/mobile-mcp release 1.0.14 (2026-10-26), Apache-2.0. mobile-dev-inc/Maestro release cli-2.12.0 (2026-10-26), Apache-2.0. joshuayoes/ios-simulator-mcp v2.2.0 (2026-10-27), MIT. See `proof-repos.txt`.

[714] https://docs.expo.dev/mcp/ (fetched again 2026-10-29, last updated 2026-10-28). The page says new projects use the SDK 58 template. npm `expo-mcp` dist-tags: latest 0.2.4, next 0.3.1. MIT. Repo https://github.com/expo/expo-mcp, last push 2026-07-29, 9 stars. I did not connect to the remote server (OAuth).

[715] https://docs.expo.dev/mcp/ (fetched again 2026-10-29, last updated 2026-10-28). The page says new projects use the SDK 58 template. npm `expo-mcp` dist-tags: latest 0.2.4, next 0.3.1. MIT. Repo https://github.com/expo/expo-mcp, last push 2026-07-29, 9 stars. I did not connect to the remote server (OAuth).

[716] https://github.com/getsentry/XcodeBuildMCP README (redirects to getsentry/MobileBuildMCP), npm `mobilebuildmcp` 2.7.1 (2026-09-23), MIT, checked 2026-10-29. Last commit 2026-10-22. Installed 2.7.1 and ran `mobilebuildmcp mcp`: 24 tools, log `Server initialized (version 2.7.1)`. Files: `scratch/FC-2/mb-tools.txt`.

[717] https://github.com/getsentry/XcodeBuildMCP README (redirects to getsentry/MobileBuildMCP), npm `mobilebuildmcp` 2.7.1 (2026-09-23), MIT, checked 2026-10-29. Last commit 2026-10-22. Installed 2.7.1 and ran `mobilebuildmcp mcp`: 24 tools, log `Server initialized (version 2.7.1)`. Files: `scratch/FC-2/mb-tools.txt`.

[718] GitHub API and npm registry, checked 2026-10-29: mobile-next/mobile-mcp release 1.0.14 (2026-10-26), Apache-2.0. mobile-dev-inc/Maestro release cli-2.12.0 (2026-10-26), Apache-2.0. joshuayoes/ios-simulator-mcp v2.2.0 (2026-10-27), MIT. See `proof-repos.txt`.

[719] GitHub API and npm registry, checked 2026-10-29: mobile-next/mobile-mcp release 1.0.14 (2026-10-26), Apache-2.0. mobile-dev-inc/Maestro release cli-2.12.0 (2026-10-26), Apache-2.0. joshuayoes/ios-simulator-mcp v2.2.0 (2026-10-27), MIT. See `proof-repos.txt`.

[720] https://docs.expo.dev/mcp/ (fetched again 2026-10-29, last updated 2026-10-28). The page says new projects use the SDK 58 template. npm `expo-mcp` dist-tags: latest 0.2.4, next 0.3.1. MIT. Repo https://github.com/expo/expo-mcp, last push 2026-07-29, 9 stars. I did not connect to the remote server (OAuth).

[721] Web search result, 2026-10-29: https://github.com/pnarayanaswamy/react-native-devtools-mcp. Not installed. Not vetted.

- Do not confuse `@mobilenext/mobile-mcp` with the npm package `mobile-mcp`. The second one is a different project (version 0.0.7, last change 2025-04-02). My first run used it by mistake.[722]

Accessibility tree or screenshot

An agent can read the screen in two ways. Each way has a cost.

Input	Strength	Weakness
Accessibility tree (text)	Cheap in tokens. Exact labels. Stable refs. Works without a vision model.	Shows only what the app exposes. Custom-drawn views are missing.
Screenshot (image)	Shows layout, color, clipping, overlap.	Costs more tokens. The model can misread small text. Coordinates can drift.

The tool descriptions confirm the design. The mobile-mcp element tool returns "ref, coordinates, and display text or accessibility label" and its click tool accepts a ref such as `@e5`.[723] The Maestro tool `inspect_screen` returns the view hierarchy as compact JSON and tells the agent to call it again after each UI change.[724] Appium MCP turns vision lookup off by default so the model does not use a slow paid call when a stable locator works.[725]

Why testID and accessibility labels matter more now

A human tester sees a blue button. An agent sees a tree. If the tree has no names, the agent guesses with pixel coordinates. Coordinates break when the layout changes.

- Expo's own tool text says to prefer `testID` "as it is resilient to layout changes".[726]
- Expo `automation_find_view` finds a view by `testID` or accessibility label.[727]
- The same labels help screen reader users. One fix serves both.

Rules for a team:

---

[722] Run on 2026-10-29: `@mobilenext/mobile-mcp` started over stdio, `tools/list` returned 34 tools. The first run (`scratch/ai/proof-mobile-mcp-tools.txt`) printed banner 1.0.11 and sdk 2.0.0, so it ran an older cached copy and not 1.0.14. A clean install with `--before` (`scratch/FC-2/mm`, `mm-tools.txt`) printed banner 1.0.14 and sdk 2.4.0, and also listed 34 tools. Only two tool descriptions differ (`get_more_tools`, `mobile_get_foreground_app`). The banner reads the version from package.json (`lib/server.js` `getAgentVersion`). The wrong package run (`mobile-mcp` 0.0.7, 9 tools) is in `proof-WRONGPKG-runablehq-mobile-mcp.txt`.

[723] Run on 2026-10-29: `@mobilenext/mobile-mcp` started over stdio, `tools/list` returned 34 tools. The first run (`scratch/ai/proof-mobile-mcp-tools.txt`) printed banner 1.0.11 and sdk 2.0.0, so it ran an older cached copy and not 1.0.14. A clean install with `--before` (`scratch/FC-2/mm`, `mm-tools.txt`) printed banner 1.0.14 and sdk 2.4.0, and also listed 34 tools. Only two tool descriptions differ (`get_more_tools`, `mobile_get_foreground_app`). The banner reads the version from package.json (`lib/server.js` `getAgentVersion`). The wrong package run (`mobile-mcp` 0.0.7, 9 tools) is in `proof-WRONGPKG-runablehq-mobile-mcp.txt`.

[724] https://docs.maestro.dev/get-started/maestro-mcp (fetched 2026-10-29, "Last updated 3 days ago").

[725] https://github.com/appium/appium-mcp README (fetched 2026-10-29). npm `appium-mcp` 1.99.1, Apache-2.0, last commit 2026-10-27. Run of `tools/list` on an installed 1.99.1 gave 32 tools: `scratch/FC-2/am/appium-tools.txt`. README sections on `AI_VISION_ENABLED`, `NO_UI` and the remote server trust model confirm the claims.

[726] https://docs.expo.dev/mcp/ (fetched again 2026-10-29, last updated 2026-10-28). The page says new projects use the SDK 58 template. npm `expo-mcp` dist-tags: latest 0.2.4, next 0.3.1. MIT. Repo https://github.com/expo/expo-mcp, last push 2026-07-29, 9 stars. I did not connect to the remote server (OAuth).

[727] https://docs.expo.dev/mcp/ (fetched again 2026-10-29, last updated 2026-10-28). The page says new projects use the SDK 58 template. npm `expo-mcp` dist-tags: latest 0.2.4, next 0.3.1. MIT. Repo https://github.com/expo/expo-mcp, last push 2026-07-29, 9 stars. I did not connect to the remote server (OAuth).

1. Give each interactive element an `accessibilityLabel` that a user would hear. Add `testID` for elements that have no good label.
2. Keep `testID` values stable and unique per screen.
3. Do not put a secret or a user name in a label. An agent log can copy it.

Agents that write tests

Kind	Example	How it fits
Unit and component tests	Coding agent writes Jest and React Native Testing Library tests	Fast. Run in CI with no model. The agent can run the test and fix the error.
End-to-end flows	Agent writes Maestro YAML through the `run` tool and keeps the working flow	The agent proves the flow on a device before you commit it.[728]
AI steps inside a flow	Maestro `assertWithAI`	Sends a screenshot to an LLM and gets true or false. Marked experimental. `optional` is `true` by default, so a failed AI check does not stop the run.[729]
Self-healing locators	BrowserStack Self-Healing Agent for Appium on App Automate	Vendor feature (marketing page, docs page fetched). It needs BrowserStack AI enabled on the account and the App Automate Pro plan. Set the `selfHeal` capability. It heals `findElement` calls. `findElements` healing is Beta. It needs one earlier passing run with the same locator. It can misfire on negative assertions.[730]

The agent output is code. Review it like code. Check that each test asserts a user-visible result. Delete tests that only check that a mock was called.

Agents that run the app

Two styles exist.

- Tree-driven agent: reads the accessibility tree, picks a ref, taps. This is the mobile-mcp, Maestro MCP, and Expo MCP style.
- Vision or computer-use agent: reads pixels and clicks coordinates. It works on any UI. It is slower and less repeatable. Appium MCP offers this as an opt-in tool with a vision API key.[731]

Use the tree-driven style first. Use vision only when the tree has a gap.

Cloud services: what exists

I checked vendor docs only. I did not run these services. Every claim here is a vendor claim.

[728] https://docs.maestro.dev/get-started/maestro-mcp (fetched 2026-10-29, "Last updated 3 days ago").

[729] https://docs.maestro.dev/reference/commands-available/assertwithai (fetched 2026-10-29).

[730] https://www.browserstack.com/docs/app-automate/appium/self-healing (fetched 2026-10-29). The page says AI must be enabled on the account and that Self-Heal is available only on the App Automate Pro plan. `findElements` healing is marked Beta.

[731] https://github.com/appium/appium-mcp README (fetched 2026-10-29). npm `appium-mcp` 1.99.1, Apache-2.0, last commit 2026-10-27. Run of `tools/list` on an installed 1.99.1 gave 32 tools: `scratch/FC-2/am/appium-tools.txt`. README sections on `AI_VISION_ENABLED`, `NO_UI` and the remote server trust model confirm the claims.

Vendor	Documented feature	Label
BrowserStack	Self-Healing Agent for Appium on App Automate. Needs account-level AI setting and the Pro plan. Docs page fetched 2026-10-29, not run by me.[732]	Vendor docs, marketing
Maestro Cloud	`run_on_cloud` and `get_cloud_run_status` MCP tools. Needs `maestro login` or `MAESTRO_CLOUD_API_KEY`.[733]	Vendor docs
LambdaTest (now TestMu AI)	KaneAI: you write steps in plain English and it makes app tests for Android and iOS. Docs page fetched, last updated 2026-09-11. The page has no claim about React Native.[734]	Vendor docs, marketing. Not run by me.
Waldo	No-code mobile testing. Tricentis announced the purchase in a blog post dated 2023-07-07. The post says it adds Waldo to the Tricentis mobile testing offerings.[735]	Fact from the acquirer's blog
Sauce Labs	Not checked in this pass.	Do not cite until verified
mabl, QA Wolf, Autosana, Revyl, MobileBoost	See the next subsection.	Vendor docs and pages

Percent figures such as "70% less maintenance" appear in vendor marketing. I did not repeat any because I found no independent measurement.

AI-native test services: Autosana and peers

An AI-native test service runs an agent that reads the screen and follows a plain-language flow. You do not write selectors. This subsection covers Autosana and six peers. I read vendor pages and docs on 2026-10-29. I did not sign up. I did not run any service. Every product claim below is a vendor claim unless the text says otherwise.

Autosana

- What it is: a hosted end-to-end test platform for iOS, Android, and web. Y Combinator lists it as Summer 2025 batch, founded 2025, team size 5, two founders.[736]
- How you write tests: natural language. A flow is a YAML file (`*.flow.yaml`) with `name` and `instructions`. Docs list a `.autosana/` folder in your repo, synced from GitHub.[737]

[732] https://www.browserstack.com/docs/app-automate/appium/self-healing (fetched 2026-10-29). The page says AI must be enabled on the account and that Self-Heal is available only on the App Automate Pro plan. `findElements` healing is marked Beta.

[733] https://docs.maestro.dev/get-started/maestro-mcp (fetched 2026-10-29, "Last updated 3 days ago").

[734] https://www.testmuai.com/support/docs/kane-ai-app-test/ (fetched 2026-10-29, last updated 2026-09-11).

[735] https://www.tricentis.com/blog/tricentis-acquires-waldo-no-code-mobile-test-automation-platform (fetched 2026-10-29, dated Jul. 07, 2023).

[736] https://www.ycombinator.com/companies/autosana (fetched 2026-10-29): Summer 2025, founded 2025, team size 5, founders Yuvan Sundrani and Jason Steinberg. The launch post says "Fully vision based". Vendor text.

[737] https://docs.autosana.ai/code-managed-files.md (fetched 2026-10-29): `.autosana/`, `*.flow.yaml`, required keys `name` and `instructions`. https://docs.autosana.ai/code-managed-migrate.md: `autosana flows export`. No Maestro or code export found.

- Element finding: the founders say it is "fully vision based".[738] The docs say "No Selectors" and "Self-Healing".[739] I found no public detail on how the agent resolves elements.
- Where it runs: cloud iOS Simulators and Android Emulators. Real Android devices take an `.apk`. The iOS `.app` build is simulator only. Real iPhones are on the Pro plan (5 sessions).[740][741]
- React Native and Expo: the docs give steps. Build in Release mode, because a Debug build looks for Metro. For Expo, use an EAS profile with `simulator: true`.[742] The docs do not name a React Native version.
- CI and agents: a CLI (`autosana run`, `autosana flows validate`), a GitHub bot, a GitHub Action for build upload, and an MCP server at `https://mcp.autosana.ai/mcp`.[743][744]
- Cost: pay as you go at $0.10 per AI action and $0.01 per cached action, with 1,000 free actions each month. Startup is $999 per month. Pro is $2,499 per month. Prices were on the page on 2026-10-29.[745]
- Lock-in: the flow files are plain YAML in your repo, and `autosana flows export` writes them. I found no export to Maestro or to code. The CLI on PyPI has license "Proprietary".[746][747]
- Maturity: the product page lists customer logos and quotes. These are vendor claims. I found no funding amount. The security page only gives a disclosure address. The footer says SOC 2 Type 1, with Type 2 "in review". I found no data retention policy for screenshots or uploaded builds.[748]

I ran the public CLI in `tmp/refresh/testing/scratch/autosana/`. `uv pip install autosana` gave version 0.18.1. The help output lists `login`, `run`, `flows`, `builds`, `keys`, and `devices`. This

[738] https://www.ycombinator.com/companies/autosana (fetched 2026-10-29): Summer 2025, founded 2025, team size 5, founders Yuvan Sundrani and Jason Steinberg. The launch post says "Fully vision based". Vendor text.

[739] https://docs.autosana.ai/introduction (fetched 2026-10-29): "No Selectors", "Self-Healing". https://docs.autosana.ai/flows.md: run caching "5-8x speedup", best-effort, on by default. Vendor claims.

[740] https://docs.autosana.ai/app-build-guide.md (fetched 2026-10-29): `.app` zip simulator only, `.apk` for emulators and real devices, `.aab` not supported, React Native must be Release mode, Expo EAS profile with `simulator: true`.

[741] https://autosana.ai/pricing (fetched 2026-10-29): pay as you go $0.10 per AI action and $0.01 cached, 1,000 free actions per month, Startup $999 per month, Pro $2,499 per month with 5 physical device sessions, Enterprise custom.

[742] https://docs.autosana.ai/app-build-guide.md (fetched 2026-10-29): `.app` zip simulator only, `.apk` for emulators and real devices, `.aab` not supported, React Native must be Release mode, Expo EAS profile with `simulator: true`.

[743] https://docs.autosana.ai/mcp-setup.md (fetched 2026-10-29): MCP URL, tools such as `flows_run`, `suites_run`, `pr_commit_flow_as_code`. https://docs.autosana.ai/install-cli.md: `autosana run`, `autosana flows validate`.

[744] https://docs.autosana.ai/introduction (fetched 2026-10-29) navigation: GitHub Bot, GitHub Action for App Uploads, GitLab CI, webhooks, Slack. Pages listed, not run by me.

[745] https://autosana.ai/pricing (fetched 2026-10-29): pay as you go $0.10 per AI action and $0.01 cached, 1,000 free actions per month, Startup $999 per month, Pro $2,499 per month with 5 physical device sessions, Enterprise custom.

[746] https://docs.autosana.ai/code-managed-files.md (fetched 2026-10-29): `.autosana/`, `*.flow.yaml`, required keys `name` and `instructions`. https://docs.autosana.ai/code-managed-migrate.md: `autosana flows export`. No Maestro or code export found.

[747] Scratch proof: tmp/refresh/testing/scratch/autosana/proof-pypi.txt (PyPI `autosana` 0.18.1, license Proprietary, 36 releases from 2026-03-25 to 2026-10-29) and proof-cli-help.txt (`autosana --help`). The npm package `autosana` (0.1.0) has a different maintainer. I did not link it to the vendor.

[748] https://autosana.ai/ footer and https://autosana.ai/security (fetched 2026-10-29): SOC 2 Type 1, Type 2 in review. The security page has a disclosure email only. Customer logos and quotes are vendor claims. No funding figure found.

proves the CLI exists. It does not prove the cloud agent works.[749]

Peers

Service	How you write tests	Runs on	Pricing model	Status seen on 2026-10-29
Autosana	Natural language flows, YAML in repo	Cloud simulators, emulators, real devices on Pro	Per AI action, plus monthly plans	YC S25, product updates to 2026-09-21[750][751]
Revyl	Natural language and CLI (`revyl run`)	Cloud iOS and Android simulators	Free tier, $250 per month Starter, per-minute overage	Pricing page live, SOC 2 Type II badge (vendor)[752]
MobileBoost GPT Driver	Natural language, or AI inside Appium, XCUITest, Espresso	Simulators and real devices	Quote only on the site. A rival's page says $799 per month (secondhand)	Site live, YC backed (vendor)[753]
QA Wolf	Managed service and a self-serve platform	Vendor cloud	$2 per mobile run (vendor)	Pricing model changed in 2026[754]
mabl	Low-code and AI agents	Cloud, mobile and browser	Credit based (secondhand: about $450 per month)	Site live[755]
Maestro with `assertWithAI`	YAML plus an experimental AI step	Local, or Maestro Cloud	Free CLI, paid cloud	`assertWithAI` is "experimental"[756]
Waldo	No-code	Cloud	Not checked	Tricentis bought it in 2023. I did not check current status[757]

[749]Scratch proof: tmp/refresh/testing/scratch/autosana/proof-pypi.txt (PyPI `autosana` 0.18.1, license Proprietary, 36 releases from 2026-03-25 to 2026-10-29) and proof-cli-help.txt (`autosana --help`). The npm package `autosana` (0.1.0) has a different maintainer. I did not link it to the vendor.

[750]https://autosana.ai/pricing (fetched 2026-10-29): pay as you go $0.10 per AI action and $0.01 cached, 1,000 free actions per month, Startup $999 per month, Pro $2,499 per month with 5 physical device sessions, Enterprise custom.

[751]https://autosana.ai/product-updates (fetched 2026-10-29): latest entry 2026-09-21 "Multi-Device Testing".

[752]https://revyl.com/pricing/ (fetched 2026-10-29): Free, Starter $250 per month, Enterprise custom, overage $0.15 per minute for iOS simulators and $0.12 per minute for Android emulators, SOC 2 Type II badge. YC page https://www.ycombinator.com/companies/revyl seen in search only.

[753]https://www.mobileboost.io/ (fetched 2026-10-29): natural language tests, works inside XCUITest, Espresso, and Appium, "Backed by Y Combinator". The $799 figure is from https://www.testmuai.com/mobileboost-gpt-driver-alternative/, a competitor page (search snippet).

[754]https://www.qawolf.com/pricing (search snippet 2026-10-29): "$1 per web test run and $2 per mobile or desktop run". The page itself did not load for me. https://bug0.com/knowledge-base/qa-wolf-pricing (secondhand) says the model changed in 2026.

[755]https://www.mabl.com/pricing and https://www.mabl.com/ai-application-testing (search snippets 2026-10-29). The "about $450 a month" figure is from https://www.drizz.dev/post/mabl-testing, a competitor blog.

[756]https://docs.maestro.dev/reference/commands-available/assertwithai (fetched 2026-10-29).

KaneAI from LambdaTest (now TestMu AI) is in the earlier table.[758]

When to choose it and when not

Choose an AI-native service when:

- the team has no one to own a Maestro or Detox suite
- flows are hard to script (OTP, push, in-app purchase, Apple Pay)
- you want a video on each pull request, and a coding agent can call it by MCP

Do not choose it when:

- the app must not leave your network, and the vendor offers no private path you can verify
- you need a fixed, cheap gate on every commit
- you need a free, forkable test format

Honest limits

- Test oracle: the agent reads the screen and decides pass or fail. A person must write the expected result in the flow. A vague step such as "check it looks right" is a weak oracle.
- Determinism: the same flow can take a different path. Autosana says it replays passing flows from a cache, 5 to 8 times faster (vendor claim). A cache hides drift until a UI change.[759]
- Flakiness: self-healing can pass a test after the UI broke. I found no independent measurement of its false pass rate.
- Cost: you pay per action or per minute. A long flow on every pull request adds up.
- Data: a build upload and screenshots go to the vendor cloud. Use test accounts only. Read the data processing terms first.

For a book reader, the first check is the oracle. Run one flow with a known bug and see if it fails.

[757] https://www.tricentis.com/blog/tricentis-acquires-waldo-no-code-mobile-test-automation-platform (fetched 2026-10-29, dated Jul. 07, 2023).

[758] https://www.testmuai.com/support/docs/kane-ai-app-test/ (fetched 2026-10-29, last updated 2026-09-11).

[759] https://docs.autosana.ai/introduction (fetched 2026-10-29): "No Selectors", "Self-Healing". https://docs.autosana.ai/flows.md: run caching "5-8x speedup", best-effort, on by default. Vendor claims.

Risks

Risk	What happens	Control
Flaky, non-deterministic tests	The same prompt gives a different path on each run	Agents explore. CI replays fixed scripts.
Cost	Each step sends a tree or an image. Long sessions cost many tokens	Prefer the tree. Use `NO_UI=true` in Appium MCP to cut tokens (project claim).[760]
Secrets in screenshots	A screenshot can show tokens, emails, and test card numbers. Expo says screenshots pass through its server, then to your model provider.[761]	Use fake accounts and seed data. Read the provider retention policy.
Prompt injection through app text	Text in the UI, a push, or a web view can contain "ignore your rules". The agent reads it as input	Run on a clean device with a test backend. Give the agent no production credentials. Ask for confirmation on destructive tools.
Test oracle problem	The agent decides pass or fail from its own reading of the screen	The human writes the expected result first.
Review burden	A fast agent makes many tests. A person must read them	Small pull requests. One flow per pull request.
Server trust	appium-mcp says do not expose it to untrusted users, and do not build `remoteServerUrl` from pull request content in CI.[762]	Keep MCP config in trusted files.

A practical workflow

A team uses a coding agent with one simulator MCP to build a feature. The steps:

1. Write the spec in plain words, with the expected result of each step.
2. Start Metro and the simulator. Connect the MCP server (for example `claude mcp add maestro -- maestro mcp`).[763]
3. Ask the agent to add `testID` and labels as it writes the screen.
4. Ask the agent to run the flow on the simulator. It reads the tree, taps, and takes a screenshot.
5. A human looks at the screenshot and the tree. This is the oracle step.
6. Ask the agent to save the passing flow as a Maestro YAML file, and add Jest and RNTL tests for the logic.
7. Commit the files. Review them as code.

[760] https://github.com/appium/appium-mcp README (fetched 2026-10-29). npm `appium-mcp` 1.99.1, Apache-2.0, last commit 2026-10-27. Run of `tools/list` on an installed 1.99.1 gave 32 tools: `scratch/FC-2/am/appium-tools.txt`. README sections on `AI_VISION_ENABLED`, `NO_UI` and the remote server trust model confirm the claims.

[761] https://docs.expo.dev/mcp/ (fetched again 2026-10-29, last updated 2026-10-28). The page says new projects use the SDK 58 template. npm `expo-mcp` dist-tags: latest 0.2.4, next 0.3.1. MIT. Repo https://github.com/expo/expo-mcp, last push 2026-07-29, 9 stars. I did not connect to the remote server (OAuth).

[762] https://github.com/appium/appium-mcp README (fetched 2026-10-29). npm `appium-mcp` 1.99.1, Apache-2.0, last commit 2026-10-27. Run of `tools/list` on an installed 1.99.1 gave 32 tools: `scratch/FC-2/am/appium-tools.txt`. README sections on `AI_VISION_ENABLED`, `NO_UI` and the remote server trust model confirm the claims.

[763] https://docs.maestro.dev/get-started/maestro-mcp (fetched 2026-10-29, "Last updated 3 days ago").

What must stay deterministic in CI:

- Jest unit and component tests with no network and no model.
- Maestro or Detox flows as fixed files. The CI job replays them. It does not ask a model what to do.
- A fixed simulator image, a fixed app build, and seeded test data.
- No `assertWithAI` in the gate, unless you set `optional: false` and accept the flake rate.[764]

AI belongs outside the gate: drafting tests, exploring a new screen, triaging a failure. The gate stays plain code.

New Architecture note

These tools drive the app from outside: through the accessibility layer, `adb`, or `simctl`. They do not call the old bridge. So the New Architecture does not break them in principle. I did not run any of them against an RN 0.88 app. That check is open.

Footnotes

[764] https://docs.maestro.dev/reference/commands-available/assertwithai (fetched 2026-10-29).