React Native Development with Expo Best Practices (2025)

Developing React Native apps with Expo in 2025 comes with a mature toolset and established best practices. This comprehensive guide covers essential practices for performance optimization, maintainability, security, scalability, tooling, and Expo-specific tips, with code examples highlighting best practices and common anti-patterns. The advice is geared towards junior and mid-level developers using Expo's managed workflow and EAS (Expo Application Services).

Performance Optimization

Building a smooth, responsive app requires careful attention to performance. Key strategies include minimizing unnecessary work, optimizing list rendering, reducing bundle size, and avoiding expensive operations on the main thread:

- Minimize Startup Work & Lazy Load: Load only what's necessary at launch. Defer loading of secondary screens or heavy modules until needed. By lazy-loading non-critical components or using dynamic `import()`, you reduce initial load time[1]. Expo's file-based router can split the JS bundle by routes using React Suspense, so only navigated screens load their code[2]. This yields faster startup and smaller initial bundle size (especially beneficial for web or large apps).

- Keep Bundle Size in Check: A smaller JavaScript bundle parses and executes faster. Remove unused libraries, prefer lightweight packages, and use tools to analyze bundle size. Expo provides Expo Atlas, a bundle analysis tool that identifies large assets and dependencies[3][4]. You can run it via `EXPO_ATLAS=true npx expo start` or from the dev menu to pinpoint bloat. Compress

images and other assets (Expo's `optimize` command or using WebP images) to shrink bundle size for production[5].

- Avoid Unnecessary Re-renders: Optimize React rendering to prevent wasted work. Use React.memo or PureComponent for components that don't always need updating, and useCallback/useMemo hooks to preserve function and value references between renders. Ensure list items and expensive subtrees are memoized so they don't all re-render when parent state changes. Choose state management wisely – global state changes can trigger many renders, so lift state only as needed and consider context or libraries for fine-grained updates[6].

- Efficient List Rendering: Use virtualization for long lists. FlatList (or its modern alternatives FlashList/LegendList) is preferred over mapping an array in a `ScrollView`[7][8]. Virtualized lists render windows of items and recycle components, drastically cutting memory usage and render time compared to a naive `map` approach[9]. FlashList (by Shopify) can often handle thousands of items at 60 FPS by optimizing rendering and requiring an `estimatedItemSize` for efficiency[10][11]. In short, never render large lists without a virtualization strategy – a poorly optimized list of just 100 items can drop the app to 30 FPS, while a well-optimized list can smoothly handle far more[12].

- Offload Heavy Work from the UI Thread: Expensive computations or large JSON parsing should not run on the main JavaScript thread. Avoid blocking JS with long loops or heavy calculations, as it will freeze the UI[13]. If such work is necessary on the client, use strategies like chunking the work with `setTimeout` or the React 18 concurrent `startTransition` API to yield to UI updates[14][15]. Leverage JSI and Worklets (e.g. via Reanimated 3/4 or the `react-native-worklets` library) to run code on a separate thread when possible[16]. Always prefer doing large data processing on a backend server if you can, sending the client only the results.

- Optimized Animations: Use React Native Reanimated for complex animations and gestures. Reanimated runs animation logic on a native UI thread via worklets, preventing janky animations caused by JS thread delays[17]. Avoid heavy layout calculations or state updates during animations; let Reanimated or animated styles handle it off-thread. Aim for a consistent 60 FPS for fluid UX – use the Performance Monitor to watch JS and UI frame rates during development[18]. If frames drop, profile which operations are costly (using tools like the Flame Chart in Flipper or React DevTools profiler).

Profiling and Monitoring: Always profile first, optimize second[19]. Use Expo's built-in Performance Monitor (open the developer menu with Cmd+Ctrl+Z in Expo Go or dev build and enable FPS monitor) to catch dropped frames and high memory usage early[18]. Integrate profiling tools like Flipper's React DevTools plugin or Sentry's performance monitoring to identify actual bottlenecks in production[19][20]. Focus on fixes that move the needle (e.g. a slow list or image load) rather than micro-optimizing trivial renders.

Example – List Rendering Anti-Pattern: Below is a comparison of a poorly optimized list vs. an optimized approach:

```
// ❌ Bad: Doing heavy work in each renderItem (causes frame drops)
<FlatList
  data={items}
  renderItem={({ item, index }) => {
    // Expensive calculations on every item render:
    const formattedDate = new Date(item.timestamp).toLocaleDateString();
    const isEven = (index % 2 === 0);
    const statusColor = item.status === 'active' ? 'green' : 'red';
    return <ListItem item={item} date={formattedDate} even={isEven} color={statusColor}
/>;
  }}
/>
```

In the bad example above, each list item recomputes the date format and colors on every render, work that scales with list size. This can bog down scrolling[21][22].

```
// ✅ Good: Pre-compute and memoize list data and items
import { FlashList } from '@shopify/flash-list';
const EfficientList = ({ items, navigation }) => {
  // Do heavy work once with useMemo:
  const processedItems = useMemo(() => items.map((item, idx) => ({
    ...item,
    formattedDate: new Date(item.timestamp).toLocaleDateString(),
    isEven: idx % 2 === 0,
    statusColor: item.status === 'active' ? 'green' : 'red'
  })), [items]);
  // Use FlashList for better performance and React.memo for list items:
  return (
    <FlashList
      data={processedItems}
      renderItem={({ item }) => <ListItem item={item} navigation={navigation} />}
      estimatedItemSize={80}
    />
  );
};
// Separate memoized ListItem component (only re-renders if its props change)
const ListItem = React.memo(({ item, navigation }) => (
  <TouchableOpacity style={[styles.item, item.isEven && styles.evenItem]}
    onPress={() => navigation.navigate('Detail', { id: item.id })}>
    <Text style={styles.title}>{item.title}</Text>
    <Text style={styles.date}>{item.formattedDate}</Text>
    <View style={[styles.status, { backgroundColor: item.statusColor }]} />
  </TouchableOpacity>
));
```

In the good example, expensive computations are done once up front (using useMemo), and the list uses FlashList for virtualization. Each item is rendered via a memoized

component, so scrolling is smooth and items don’t re-render unless their props actually change[23][24]. This approach eliminates redundant work and significantly improves list performance[25].

Maintainability

Maintainable code is organized, readable, and easy to extend or refactor. Expo projects benefit from clear structure and conventions that help developers navigate the codebase:

- Project Structure: Organize your project into logical folders. Common practice is to have a top-level `src/` (or similar) containing folders for components, screens, navigation, assets, etc. For example[26], you might have:

  ```
  src/
  ├── components/    # Reusable UI components
  ├── screens/       # Screen components (pages)
  ├── navigation/    # Navigation configuration (if not using Expo Router)
  ├── hooks/         # Custom hooks
  ├── context/       # Context providers
  ├── utils/         # Utility functions
  ├── assets/        # Images, fonts
  └── styles/        # Global styles or theming
  ```

  Grouping by type (as above) is intuitive for smaller apps[27]. As apps grow, consider *feature-based* structure: e.g. group files by feature or domain (each feature folder contains its own components, hooks, etc.). This modular approach scales better for larger projects by reducing coupling between unrelated features.

- Expo Router Structure: If using Expo's file-based Router (Expo Router), follow its conventions. Place your screens in the `app/` directory as routes. For example: `app/index.tsx` for the home screen, `app/profile/[id].tsx` for a dynamic route, etc. Group routes in nested folders for subpaths or modal groups as needed. Use lowercase or kebab-case file names for routes (since they may become URL paths in web/deep links). Expo Router auto-generates navigation based on this structure, which keeps routing declarative and avoids a lot of boilerplate. Keep

non-screen components (like smaller UI pieces) in a components/ folder to avoid cluttering app/. The Expo blog recommends organizing by features within app/ for clarity and scalability (e.g. app/(auth)/login.tsx grouping screens)[28].

- Naming Conventions: Consistent naming makes the codebase understandable. Use PascalCase for React component files and their component names (e.g. LoginScreen.tsx containing function LoginScreen() {}) and use camelCase for utility modules, hooks, and other non-component files[29]. This convention arises naturally if you name files after their default export (components export a PascalCase component, other files export camelCase functions)[29]. Folder names are often lowercase (or kebab-case) for consistency. For example, a service module might be apiClient.ts (camelCase file exporting functions), whereas a component is UserCard.tsx (PascalCase file/component). Ensure your naming is descriptive – e.g. ProfileScreen vs. Screen1, useAuth vs. miscFunctions. Avoid abbreviations that aren't obvious. Consistency is more important than the specific casing style – pick a convention and apply it uniformly across the project[30][31].

- Code Organization & Modularity: Strive for *small, focused components and functions*. Each component should ideally handle one idea or UI section. Break down large screens into reusable child components when possible, and move complex logic to custom hooks or utility functions. This separation makes components more readable and easier to test. For example, complex form state management can live in a hook (e.g. useLoginForm) used by a LoginScreen component, keeping the component JSX clean. Similarly, avoid duplicating code – factor out common UI patterns into shared components (e.g. a <Button> component used throughout) and common logic into helpers or hooks. A rule of thumb: if a file exceeds a few hundred lines or is doing multiple things, consider splitting it.

- Folder and File Structure Consistency: Whatever structure you choose, keep it consistent. If you add a new feature, place its files in the agreed location. New developers should be able to guess where something might be. For example, if you have a features/ directory, each subfolder could contain its own components, hooks, services etc., mirroring a standard template. Consistency extends to file naming – e.g., decide whether to use index.tsx files in component folders or not, and stick with that choice.

- Comments and Documentation: Write self-explanatory code where possible (clear naming, simple logic), but also include comments for non-obvious code. JSDoc or TS doc comments on complex functions, and high-level comments at the top of modules explaining their purpose, can be very helpful. For a team project, consider a README.md in the project or docs describing the overall architecture and conventions.

- Styling Practices: Keep styling maintainable by using StyleSheet or styled-components consistently. If using React Native StyleSheets or Emotion, consider having a separate file for styles (e.g. ComponentName.styles.ts) or define them at the bottom of the component file to separate styling from logic. Using a consistent system for spacing, colors, etc. (possibly via a theme or constants file) helps maintain visual consistency.

- Type Safety: Expo supports TypeScript out of the box. Embrace TypeScript for more maintainable code: define proper types for component props and app data models (e.g. a User interface). This catches errors early and serves as documentation for other devs. Keep common types in a types/ or @types directory if they are shared widely.

By following these maintainability practices – logical structure, clear naming, modular code, and consistency – your Expo project will be easier to navigate and extend. A well-structured codebase means new team members can quickly find and understand code, and you reduce the chance of errors when modifying or adding features.

Security

Security is critical even in client-side apps. In React Native (and Expo), remember that your compiled app code can be inspected, so never assume anything in the app is truly hidden from a determined user. Adopt these best practices to protect sensitive data and guard against vulnerabilities:

- Never Hardcode Secrets or API Keys: Do not include secrets (API keys, private tokens, passwords, etc.) directly in your app's source code or in plaintext configuration. Anything in the JavaScript bundle can be extracted by anyone with access to the app package[32]. This means even environment variables bundled into the app are not secure. Expo's environment variable system differentiates between public and private env vars. Only use `EXPO_PUBLIC_*` variables for non-sensitive config (like public API endpoints or feature flags). Never put secrets in variables that will be embedded in the JS bundle[33]. For truly sensitive credentials, move them off the client: use a secure backend or serverless function as an intermediary. For example, if your app needs an API key to call a third-party service, have your server hold the key and expose an endpoint your app can call, rather than baking the key into the app[34]. The React Native security guidelines emphasize routing calls through a backend as the safest approach for secret management[35].

- Use Secure Storage for Sensitive Data: If the app must handle sensitive user data (like auth tokens, refresh tokens, or personal info), store it securely. Do not use AsyncStorage for secrets – AsyncStorage data is stored unencrypted on the device's storage, which can be extracted if the device is compromised or if someone gets the app data (Android backups, rooted devices, etc.)[36]. Instead, use Expo SecureStore (Expo's secure storage API) or similar, which leverages the OS keychain/key store to encrypt data at rest[37][38]. For example, use `await SecureStore.setItemAsync('authToken', token)` to save a token – this stores it in iOS Keychain or Android Encrypted Shared Preferences, making it much harder to

extract than a plaintext file. Only keep what's absolutely necessary on the device, and clear sensitive data on logout.

- Environment Variables & EAS Secrets: Expo and EAS provide ways to supply configuration at build time without exposing it in source control. Use `.env` files for configuration that can be public or that you need for build-time (and prefix truly public ones with `EXPO_PUBLIC_`). For any secret values needed during build (e.g. API keys for config plugins, or credentials your app might use to configure something), use EAS Secret environment variables. EAS Secrets are stored securely on Expo's servers and injected at build time, but they are not exposed in the published JS bundle. For example, you might set an API key as an EAS Secret and consume it in your app config (app.config.js) to insert into Info.plist at build time. This way the binary has the key, but you avoid committing it to source control. *Note:* If the key is in the binary (e.g. in the Info.plist or AndroidManifest), users could still dig it out, so ideally treat even those as non-sensitive or use OS-level protections. In summary, don't rely on just hiding something in JS – if you truly need to secure an API secret, don't include it in the app at all[34].

- Prevent Code Injection and XSS: Although a React Native app isn't a website, you might use WebViews or evaluate dynamic content. Be cautious with any functionality that takes external input and renders it. For example, if using a `WebView` to display HTML from an API, ensure you sanitize this input on the server side to prevent any malicious script injection. Avoid using `eval` or dynamically executing code from untrusted sources. If you use messaging or deep linking, validate and sanitize inputs (e.g. allowed URL patterns). Expo's managed environment generally doesn't allow arbitrary code execution, but always be mindful of any feature (like downloading and running JS, aside from official OTA updates) as it can introduce injection risks.

- OS Security Practices: Utilize platform security features via Expo SDKs where possible. For instance, use expo-auth-session or AppAuth for secure

authentication flows (these use secure system web views for OAuth, which sandbox cookies). Use expo-local-authentication if you need biometric/PIN unlock for an app section, rather than implementing your own insecure lock. When making network requests, prefer HTTPS for all endpoints and validate SSL certificates (the underlying `fetch` in React Native does this by default). If your app deals with particularly sensitive info, consider additional encryption: for example, use the expo-crypto library to encrypt data before storing or to implement end-to-end encryption for data sent to your server.

- Be Careful with Logging and Monitoring: Avoid logging sensitive information. Tools like Sentry, LogRocket, or Firebase Crashlytics are great for error monitoring, but be mindful not to send personal data or secrets in error messages or analytics events (e.g. don't include a user's password or auth token in a crash report or console log). Strip out debug logs from production if they might reveal internal info. Expo's development builds and Expo Go show console logs, but in a production release those aren't visible; still, it's good practice not to leave verbose logs or any sensitive info logged even in dev (it could accidentally leak or degrade performance).
- Regular Updates and Auditing: Keep your Expo SDK and libraries up to date. Expo SDK updates often include security fixes (for example, updates to underlying React Native or fixes in expo modules). Leverage `expo doctor` to identify mismatched package versions that might lead to vulnerabilities. Periodically review your app's dependencies for known vulnerabilities (using tools like `npm audit`). Remove dependencies you no longer use – every extra library is additional attack surface.

By following these practices – not hardcoding secrets, using secure storage, sanitizing inputs, and staying updated – you significantly reduce security risks in your Expo app. Remember that mobile security is about layers: make it as hard as reasonably possible for attackers to get anything useful (even though a determined attacker with physical

device access or an emulator dump could analyze your app, you want to ensure they gain nothing of value easily). Always assume a savvy user can read your JS code; if something must remain truly secret, keep it on a server.

Example – Handling Secrets (Anti-Pattern vs Best Practice):

```
// ❌ Anti-Pattern: Hardcoding an API secret in the app
const API_KEY = "ABCD-1234-SECRET";  // This will be visible in the bundle!
fetch(`https://api.example.com/data?apiKey=${API_KEY}`);
```

In the above bad example, the API key is embedded in the JavaScript. A user inspecting the app bundle could find this string[32]. Instead, a better approach is to never include the secret on the client. You could store the API key on a server and call that server (which adds the key), or at minimum load it from a secure source at runtime (not ideal, as it can still be intercepted). If the API key is for a third-party service and must be in-app, consider restrictions (domain or IP whitelisting, or use a less-privileged key). For storing user tokens or passwords, never keep them in plaintext or AsyncStorage:

```
// ✅ Best Practice: Use secure storage and avoid exposing secrets
import * as SecureStore from 'expo-secure-store';
// ... after user logs in and you receive a token:
await SecureStore.setItemAsync('userToken', token);  // encrypted storage
```

Using Expo's SecureStore ensures the token is saved in the device keychain with encryption[37][38]. If the app needs to use this token, it can retrieve it and add to headers, but the raw value isn't lying around in local storage. Also, if using environment config for non-secret values, you might have in an `.env` file `EXPO_PUBLIC_API_URL="https://api.example.com"` which is fine (accessible in code via `process.env.EXPO_PUBLIC_API_URL`), but you would not put `EXPO_PUBLIC_API_KEY` for a secret – that would embed it in the app. Instead, supply secrets through EAS at build

time (and use them only in native config or server communication), or not at all on the client[33].

Scalability

As your project and team grow, it's important to structure code and utilize tools that allow the app to scale without degrading performance or developer productivity. Scalability in a React Native/Expo context involves how you architect components and state, how you split code, and how you manage a growing codebase (potentially with monorepos or modularization):

- Component Architecture and Reusability: Adopt an architecture that encourages reusability and separation of concerns. One common pattern is splitting components into Presentational and Container components (also known as "dumb" vs "smart" components)[39][40]. Presentational components focus purely on UI rendering, receiving data via props, and are often reused in multiple places. Container components handle business logic, state, and data fetching, and then render presentational components. This separation makes it easier to scale the app because each component has a single responsibility. For example, you might have a <ProductList> presentational component that renders a list layout given products and callbacks, and a container component <ProductListScreen> that fetches the products from an API or Redux store and passes them into <ProductList>. This way, the data fetching logic can change or move (e.g., to a different state management library) without requiring changes to the UI component. It also enables reusing the <ProductList> in different screens or contexts (maybe in a search screen) with different data.

- Code Splitting & Lazy Loading: As mentioned in performance, code splitting is also a scalability technique. In a large application, you don't want a single monolithic JS bundle with every feature. With Expo Router, code splitting by route is essentially automatic – each screen can be loaded on demand thanks to async import with Suspense[2]. This means the initial bundle can remain small even as your app grows. If not using Expo Router, you can still manually lazy-load modules: for instance, using React.lazy() and <Suspense> for screens or heavy

components so that they are only imported when needed. This keeps memory usage and load times more manageable as features are added. Note that on mobile, true on-demand loading after the app is bundled is limited (the entire bundle is usually loaded at startup in RN). However, you can simulate “split” bundles by organizational convention or using tools like Metro split bundles for specific sections. Also, dynamic import() can defer execution of a module until used, which can help reduce startup time (Metro will still bundle it but in a way that it can be initialized later). Overall, ensure that adding new features doesn’t linearly increase startup time – leverage lazy techniques so new code impacts only the sections of app that use it.

- Managing State at Scale: Small apps may get by with useState and prop drilling, but as the app scales, consider a more robust state management strategy. This might mean introducing Context for globally relevant state or a library like Redux, Zustand, or MobX for complex state needs. The key is to prevent state from residing at too high a level such that any change causes large re-renders. If using Redux or Zustand, structure your state into slices or stores by domain (auth, products, settings, etc.) and use selectors to retrieve just the needed piece of state in each component. This way, adding more state doesn’t overwhelm the app with re-renders. Also consider performance: tools like Redux Toolkit with createSelector (memoized selectors) or Zustand with selection functions can ensure components only re-render when relevant state changes. For asynchronous state (server data), tools like React Query (TanStack Query) or Apollo Client (for GraphQL) can help manage caching and updates in a scalable way, so that as you add more API calls, you don’t custom-write tons of effect and loading logic each time. Choose a state management approach that your team is comfortable with and that fits the app’s complexity – not every app needs Redux, but beyond a certain size, using just React Context might become difficult to optimize.

- Modularizing Features: When an app grows to have many domains or features (e.g., an e-commerce app with product listings, user profiles, orders, etc.), treat each feature as a module. Create a structure within src/ that groups feature-specific components, state, and logic. For example: src/features/Orders/ could contain its screen components, perhaps a sub-store or context, and any utils specific to orders. This modular approach means teams can work on features in isolation (helpful in larger teams) and you reduce tight coupling across the codebase. It's also a stepping stone to potentially extracting features into separate packages if needed.

- Monorepo for Multiple Packages or Apps: If your project expands to multiple applications or packages, consider a monorepo. Expo has first-class support for monorepos using Yarn, npm, or PNPM workspaces[41][42]. A monorepo allows you to house, for example, a React Native app, a web app, and a shared component library all in one repository with shared code. It can also be used to maintain separate modules (like design system components, or a separate package for state logic) that the app consumes. The advantages are a single source of truth and easier code sharing across projects[41] – updates to the shared package can propagate to apps immediately. Expo SDK 52+ automatically handles Metro bundler configuration for monorepos, so it's much easier to set up than it was in the past (you typically just ensure your workspaces are defined in package.json, and Expo CLI configures watchFolders and nodeModulesPaths for you)[43]. A typical monorepo layout might be:

  ```
  root/
   package.json (with workspaces)
   apps/
    myExpoApp/
    myOtherExpoApp/
   packages/
  ```

ui-library/
utils/

Each app can import code from the packages (e.g. `import { Button } from '@myorg/ui-library'`). Monorepos can speed up development on large projects and ensure consistency across multiple apps[44]. However, they add complexity in tooling (you need to manage workspace dependencies, potentially handle different release cycles for packages, etc.). Use a monorepo if you have a clear need (e.g., multiple apps sharing a lot of code). Expo's documentation and community examples (using tools like Turborepo or Nx) provide guidance for these setups, and EAS Build supports monorepos as well.

- Scalability of CI/CD: As the project grows, automated build and deployment become crucial. Expo's EAS can scale your build process by offloading to cloud builders. Define multiple build profiles in `eas.json` (for development, staging, production builds with different configs) as needed. If your team grows, use CI pipelines (GitHub Actions, CircleCI, etc.) to run tests and then trigger EAS builds. EAS can integrate with CI by using the EAS CLI (e.g., `eas build --profile production --auto-submit`). This ensures that even as your app gets more complex, the delivery process remains consistent and does not bottleneck on any one developer's machine.

- Design System & Theming: To scale the front-end UI, it's beneficial to adopt a design system early. Use a consistent set of components (possibly with a library like NativeBase or React Native Paper, or a custom in-house library) to enforce uniform styles. Expo projects can easily use such libraries (just install and go, since Expo includes the native dependencies for many popular UI kits, or use config plugins if needed). With a design system, adding new screens or features is faster and more consistent. It also allows for global theming – for instance, switching to dark mode throughout the app by adjusting theme context in one place.

In summary, scaling an Expo app is about managing complexity through structure and tools. Use modular architecture to keep codebases manageable, use code splitting and performance techniques to keep the app fast as features grow, and consider monorepo or multi-package strategies if you have multiple projects or very large codebases. Expo's robust tooling (automatic monorepo support, EAS services, etc.) is designed to support apps from prototype stage to production at scale.

Tooling and Workflow

Using the right tools can greatly improve developer productivity and app quality. Expo and the React Native ecosystem provide a variety of tools for debugging, testing, building, and monitoring your app:

- Debugging Tools: Expo makes debugging convenient with multiple options:
- Expo Dev Menu & Logs: In development, you can shake your device (or Cmd+Ctrl+Z in simulator) to bring up the Expo dev menu. From here, enable Debug Remote JS (to use Chrome/Edge debugger) or Inspect (to use Flipper's React DevTools), and the Performance Monitor to track FPS and memory[18]. Use console.log for quick logs (accessible in the terminal or debugger console). The dev menu also has options to reload, toggle Fast Refresh, etc.
- React Developer Tools & Flipper: For a more powerful debugging experience, use Flipper (a desktop app by Meta). Expo apps (especially if you use development builds) can connect to Flipper for inspecting native logs, using React DevTools, viewing network requests, and more. Flipper is now compatible with Expo projects – for Expo Go, enabling "Debug Remote JS" lets you use DevTools, and for custom dev clients built with EAS, you can integrate Flipper plugins. Additionally, React Developer Tools can connect to the app for inspecting component trees and state/props – this is accessible via Flipper or by running react-devtools in a separate terminal.
- Breakpoints and Profiling: When debugging in Chrome, you can set breakpoints in your JavaScript code via the Sources panel, just like debugging a web app. This is invaluable for stepping through logic. For performance profiling, use the "Performance" tab in Chrome DevTools while the app is running to record a timeline and see JS thread activity. Alternatively, use Systrace for low-level RN performance profiling or tools like why-did-you-render (a library to highlight unnecessary Renders).

- Testing: A solid testing setup ensures your app remains stable as it grows:

- Unit and Component Testing: Use Jest as your testing framework, which is the default for React Native. Expo provides a Jest preset called jest-expo, which simplifies configuration (it mocks native modules for you). Set up Jest with `jest-expo` in your package.json as shown in Expo docs[45][46]. For component testing, use React Native Testing Library (@testing-library/react-native) which provides utilities to render components and assert on their behavior in a test environment[47]. Write tests for pure functions (business logic) and for components (e.g. does tapping a button call the right handler, does a component render with given props, etc.). Aim for a good coverage of core logic and critical UI flows.
- Integration and E2E Testing: For deeper testing that involves the actual app running, consider end-to-end (E2E) testing. Detox is a popular E2E testing framework for React Native. With Expo, you can use Detox on apps built with the development build or standalone binaries (Expo's managed workflow requires using EAS to create a build that Detox can run). Detox allows you to script user interactions and assertions on the app (e.g., "login flow works correctly"). Setting up Detox with Expo involves using the bare workflow or a custom dev client, since Expo Go itself cannot be controlled by Detox. Alternatively, you can use Appium or other app automation frameworks if Detox setup is complex. Keep E2E tests for the most crucial user journeys, as they are heavier to run.
- Continuous Testing: Integrate tests into your CI pipeline. For instance, run `npm test` (Jest) on every pull request via GitHub Actions. You can also run Detox tests on CI using simulators, though this requires more setup (and macOS runners for iOS tests).
- Test Writing Best Practices: Focus on testing behavior over implementation. For components, simulate user interactions (e.g., RN Testing Library's `fireEvent.press(button)` to test a button). For logic, test edge cases (e.g., state reducers, utility functions). This will give you confidence to refactor or add features without breaking existing functionality[48].

- Continuous Integration / Deployment (CI/CD): Expo's EAS Build and EAS Submit simplify building and deploying your app:
- EAS Build: Instead of maintaining local Xcode/Android Studio environments for each developer or CI, use EAS Build to compile your app in the cloud. Define build profiles in `eas.json` for different targets (development, preview, production). Running `eas build --profile production` will package your app (APK/AAB or IPA) on Expo's servers, with all necessary certificates and provisioning handled (you can store signing credentials with Expo or supply your own). This ensures consistency – the build environment is standardized. EAS Build integrates with managed workflow seamlessly, and also supports bare projects. Use EAS Build in CI by triggering it with the CLI; it can return build status and you can script wait-for-completion or use webhooks. This way, every commit to main (for example) can kick off a build, ensuring you always have an updated build for testing or release.
- EAS Submit: After building, Expo can also handle app store submissions. `eas submit` will take a build artifact and upload it to Apple App Store Connect or Google Play Console. This can be integrated into CI for automated deploys (after a manual approval step, if desired). Automating submissions saves time especially if you do frequent releases.
- Updates (OTA): Use EAS Update (the successor to classic `expo publish`) for over-the-air updates when appropriate. This is a great tool for pushing bug fixes or minor improvements instantly to users without waiting for app store review[49]. Plan a workflow for updates: use EAS Update channels to separate release tracks (e.g., an "staging" channel for internal testing and "production" for users). However, put guardrails: OTA updates should not be used for any changes that require native module changes or permissions – those still need a new binary release. Also, test updates thoroughly (you can use `eas update --branch=staging` and the expo Updates API to pull specific branches in dev builds).

EAS Update, used wisely, can significantly reduce time-to-fix for critical issues, strengthening your release strategy[50].

- Analytics and Monitoring: To understand how your app is used and catch issues in production, integrate analytics and error reporting:

- Crash Reporting: Use a service like Sentry or BugSnag. Both have React Native SDKs that work with Expo (Sentry has an official Expo config plugin). These services automatically capture JS exceptions (and native crashes) along with stack traces and device info. Sentry, for instance, can be added via `@sentry/react-native` and their provided config plugin; it will log errors and even performance data. Sentry integrates with Expo in that you can view Sentry data on the Expo dashboard if configured[20]. With crash reports, you can prioritize and fix bugs that real users encounter. Be sure to upload source maps (EAS Build can automate this with the Sentry plugin) so that stack traces are symbolicated.
- User Analytics: To track user behavior (screen visits, actions, retention), integrate an analytics SDK. Popular options include Firebase Analytics, Amplitude, Mixpanel, or Segment. Expo projects can use these via config plugins. For example, `expo-firebase-analytics` provides a lightweight way to use Google Analytics for Firebase without ejecting. Whichever tool, define key metrics to track (e.g., daily active users, conversion funnels) and log events in the app accordingly. This data will help in making informed improvements and demonstrating app growth.
- EAS Insights: Expo offers EAS Insights which is a lightweight analytics specifically for updates and usage. By installing `expo-insights`, you can get data on active users, update adoption, OS distribution, etc., right in the Expo dashboard[51]. This can complement a full analytics tool by giving a quick health check of your deploys (e.g., how many users have downloaded the latest OTA update).
- Session Replay and More: Tools like LogRocket or Vexo provide session replay – literally recording user sessions (UI and interactions) so you can play them back

to see what went wrong in a bug scenario. Expo has guidance for using LogRocket with React Native[52]. These are more heavy-weight but can be invaluable for debugging complex user reports. Use them sparingly and be mindful of privacy (don't unintentionally record sensitive info).

- Performance Monitoring: Beyond development profiling, in-production performance monitoring can catch issues like slow screens or frozen frames on certain devices. Sentry's performance monitoring can measure render times and see transactions for slow operations in the wild. If performance is a key focus, consider using those tools to gather data from real users (e.g., track app launch time, or screen load durations).

- Developer Experience Tools: A few other tools to mention:

- Linting and Formatting: Use ESLint (with an appropriate config, e.g., Airbnb or the Expo recommended config) to catch code smells and enforce style. Pair it with Prettier for consistent code formatting. Many Expo templates come with these configured. This keeps the code style uniform across contributors.
- Type Checking: If using TypeScript (highly recommended), leverage the compiler for catching mistakes. Run `tsc --noEmit` in CI to ensure no type errors slip through.
- VS Code Extensions: If using VS Code, the React Native Tools extension can run the app and enable debugging directly from the editor. Expo also has an official extension that shows an Expo projects view and logs. These can streamline the inner loop of coding and testing.

By utilizing these tools and integrating them into your workflow, you can significantly improve development speed and app quality. A typical optimal workflow in 2025 might look like: use Fast Refresh for quick UI iteration, debug with Flipper/DevTools for any complex issues, write and run Jest tests during development, use ESLint/TypeScript to maintain code quality, and have a CI pipeline that runs tests and builds the app via EAS, with Sentry monitoring live for any crashes once deployed. This toolchain, while it may

require effort to set up initially, pays off as the project grows and ensures you catch problems early rather than after release.

Expo-Specific Best Practices

Expo's managed workflow provides a lot of conveniences and powerful features. To get the most out of Expo (as of 2025), consider the following best practices unique to Expo:

- Prefer Managed Workflow (Avoid Ejecting) Whenever Possible: In the past, developers often ejected (switching to the bare workflow) for advanced use cases. Today, Expo's managed workflow can handle the vast majority of needs, thanks to Config Plugins and Development Builds. A config plugin is a small JS script that tweaks native config during build (e.g. adding a permission, integrating a library)[53][54]. The community has created plugins for most popular native libraries (camera, payments, maps, etc.), and you can write custom ones if needed. This means you "get to have your cake and eat it too" – use native libraries without fully ejecting. Only eject (i.e. go to bare workflow) if you absolutely must write custom native code that cannot be achieved with config plugins. In 2025, those cases are rare[55][56]. By staying managed, you keep the benefits of easy updates, Expo Go compatibility, and simpler build configuration. As one community expert put it: *"You only ever need to switch to the bare workflow if there's something you can't or don't want to do with config plugins. Most apps will be more than happy with the managed workflow."*[55] This holds true – features like push notifications, in-app purchases, authentication, etc., are all possible in managed apps now via Expo packages or plugins. Trust Expo's tooling to handle the native details while you focus on JavaScript[57].

- Leverage EAS Build and Submit: Adopt EAS Build for compiling your app instead of relying on `expo build` (the older service) or manual Xcode/Android builds. EAS Build is more flexible and production-ready. It supports managed and bare projects, allows custom native code, and you can customize the build process with eas.json. Best practice is to configure things like app version, gradle/ios build settings via expo config and eas.json rather than manual editing. Also, use EAS Submit to streamline deployments – it can automatically submit your builds

to app stores, which is especially handy if you do frequent releases (you can trigger a build and have it on TestFlight/Play Console in one command). This tight integration of Expo's build service into your workflow saves time and reduces human error (no more forgetting to increment build numbers or selecting the wrong provisioning profile – EAS handles that).

- Use Expo Updates (EAS Update) Wisely: Over-the-air updates are a standout Expo feature. With EAS Update, you can push JS/CSS asset updates to users instantly[49]. Best practices for using updates:

- Use them for minor fixes, text changes, or non-critical feature flags – things that *don't require new native code*. For example, if you found a typo or a small bug in a Redux logic after deploying to production, you can deploy an OTA update to fix it, and users will get it almost immediately (next time they open the app)[50]. This agility is incredibly valuable.
- Do not use updates to change anything that alters the app's native interface (like adding a new Expo plugin, changing app.json native config, adding a permission) – those require a new binary. If you did, the update would likely cause a crash or just not work on older binaries. To manage this, use runtimeVersion in your app config to ensure incompatible updates don't apply to older app binaries.
- Organize updates using channels (e.g. a "production" channel for live users, "staging" for internal). This way you can test an update with internal testers (on the staging channel) before promoting it to production.
- Monitor rollouts – Expo provides update event listeners and EAS dashboard info. You might roll out an update gradually (for instance, release to 10% of users, ensure stability, then 100%).
- Remember that if a user hasn't opened the app in a while, they might jump multiple updates – ensure backwards compatibility in your updates or use criticalVersion flags if needed to force a user to get a new binary after too many changes.

- Document and keep track of what updates are live in production, just like you would versioned releases.

When used appropriately, EAS Update can be a lifesaver (hot-fixing a critical bug same-day instead of waiting for App Store review)[50]. Just treat it with respect – test your updates thoroughly, and don't use it as a crutch to ship untested code (an update is effectively a prod deployment).

- Stay Updated with Expo SDK Releases: Expo moves fast, with a new SDK version roughly every quarter. It's a best practice to keep your app's SDK version relatively up-to-date (within a version or two of the latest). New SDK versions bring not only new features and support for new React Native versions, but often performance improvements and security patches. Expo provides an `expo upgrade` command to assist, and the release notes detail any breaking changes. Regularly plan time to upgrade – the longer you wait, the harder upgrades can become. By 2025, React Native's pace has stabilized a bit and Expo SDK upgrades are smoother, but you still don't want to fall too far behind (e.g., upgrading from SDK 45 to 55 in one go could be painful). Expo also deprecates older SDK support on their services after a while, so upgrading ensures you can keep using Expo Go and EAS for your app.

- Utilize Expo's Built-in Modules: Expo SDK includes many high-quality modules for common requirements – e.g., `expo-camera` for camera access, `expo-media-library` for gallery, `expo-sensors` for device sensors, etc. Before pulling in an unknown third-party library, check if Expo has it covered. Expo modules are maintained to work well with managed workflow and are often easier to set up (just `expo install`). They also often have web support out-of-the-box (for Expo for Web), which third-party libraries might not. By using the official modules, you reduce risk and ensure compatibility. If Expo doesn't have something and you use a third-party RN library, check for an Expo config plugin for it (most popular ones have one). This will save you from ejecting and allow that library to work seamlessly in your app.

- Continuous Native Generation (CNG) Approach: Embrace the concept that your android and ios folders in a managed workflow are generated artifacts. You generally should not edit them by hand – instead, configure via Expo config (app.json/app.config.js) or config plugins. This ensures your changes are repeatable and tracked in source. For example, if you need to add an entry to Info.plist, write a config plugin or use the app.json ios.infoPlist key to specify it, rather than ejecting and editing Info.plist. Expo is moving toward this "config as code" model, which makes your workflow more like web development (one source of truth, rather than messing with Xcode). This approach pays off especially in CI and multi-developer teams – no "it works on my machine" for native config. Also, when upgrading Expo SDK, regenerating the native projects is straightforward if you haven't manually tweaked them; your config remains in JS where it's easy to adjust.

- Expo Dev Client for Advanced Use: If you find yourself needing to include custom native modules during development (which Expo Go doesn't support), use Expo Development Builds (Dev Client). A dev build is essentially your own custom Expo Go that includes the native plugins you need. Best practice is to set up a dev build early if you know you'll use one, and use it just like Expo Go for development (it connects to expo start). This way, you're testing the actual native environment of your app throughout development. Expo provides eas build --profile development to create these easily. Using dev builds avoids surprises when you do a production build, because you've been running with the actual native modules all along.

- Expo Community and Documentation: Leverage the Expo community forums, Slack/Discord, and the official docs whenever you face an issue. Often, the fastest way to solve a problem is reading the Expo documentation guides (which are very extensive as of 2025) or searching the forums. Expo's team and community are active and chances are someone has solved a similar issue (whether it's how to integrate a certain library or how to optimize something for

Expo). Following the official Expo blog can also keep you informed about new features or deprecations (for example, when Expo moved from the classic build service to EAS, or the introduction of Expo Router). Staying informed will let you adopt best practices as they evolve.

In essence, the modern Expo workflow encourages you to *stay within Expo's managed ecosystem* for as long as possible, using its tooling (EAS, updates, config plugins) to cover needs that previously required ejecting. This yields huge benefits in development speed and reliability. By following Expo's recommended path – managed workflow with config plugins, EAS for build and OTA, and regular updates – you position your app for long-term success with minimal friction. Expo in 2025 is a production-ready choice for serious apps[58][59], and following these practices will help you leverage it to the fullest.

Example – Managed vs Bare (Using Config Plugins): Suppose you need to add a complex native library (like a custom authentication SDK). Instead of ejecting, first check if a config plugin exists. If not, you can often write one. For instance, to add some custom info to the iOS Info.plist and Android manifest, you can create a plugin in `plugins/yourPlugin.js` that uses Expo's config mod API to modify those files. Then add it in app.json under `expo.plugins`. When you run a build (or `expo prebuild`), Expo will apply those changes for you. This way, you maintain a managed project. The mantra: *"There is in fact no reason to eject anymore… before config plugins, you needed to eject as soon as you had a requirement that wasn't handled by a library. But now… even if a library has no plugin, making a custom one is not that hard."*[60][57]. Following this approach keeps your workflow smooth and your focus on React Native code rather than native build configs.

Common Anti-Patterns to Avoid

Throughout the above sections we've touched on many anti-patterns. Let's summarize a few common mistakes in React Native/Expo development and how to avoid them:

- Doing Too Much on the Main Thread: (Performance) Avoid long-running loops, expensive calculations, or heavy memory operations in the normal React render/update cycle. Anti-pattern: parsing a huge JSON or resizing an image on the fly in a component render. Solution: move it out – parse data in background (if from network, it won't block UI anyway), use something like the InteractionManager to defer work until after animations, or offload to native modules. Use profiling to catch these – if an animation janks, check what else is happening.

- Excessive Re-renders due to State Mismanagement: (Performance/Scalability) Anti-pattern: having a single global state that causes most of the app to re-render when anything changes, or passing down state through many layers causing each to update. This often happens if you put too much into React Context or Redux without structuring it. Solution: split state into smaller contexts or slices, use memoization (e.g., Redux selectors or `React.memo`) to ensure components only update when needed. Another anti-pattern is mutating state directly (which can lead to no re-render or a full list refresh) – always update state immutably. For example, do `setItems(prev => [...prev, newItem])` instead of `items.push()` + `setItems(items)`[61], so that React knows the array changed and can update efficiently rather than potentially ignoring or re-rendering everything incorrectly.

- Massive Components or God Classes: (Maintainability) Anti-pattern: one screen component does everything – data fetching, state management, complex UI, navigation logic – hundreds of lines long. This is hard to maintain or reuse. Solution: break it down. Use container/presentational split, move logic to hooks or helper functions, and keep your render functions focused on UI. If you notice

repeated patterns or very long render methods with many conditional branches, that's a smell that you should create subcomponents.

- Inconsistent Project Structure/Naming: (Maintainability) Anti-pattern: mixing naming styles (some files PascalCase, some snake_case, etc.), scattering files without order (e.g., some screens in a `screens` folder, others not), or having deeply nested relative imports (`../../../utils.js`). Solution: decide on a structure early (it can evolve, but have a plan) and stick to conventions. Use tools like a module resolver to avoid absurdly long relative import paths (you can configure `babel-plugin-module-resolver` to allow `import { X } from "@utils"` instead of relative paths)[62][63]. Run lint rules that catch naming inconsistencies or enforce a certain grouping. This consistency pays dividends as the app grows.

- Hardcoding Configuration and Secrets: (Security/Maintenance) Anti-pattern: having values in code that should be configured per environment, or secrets that should not be in code at all. For example, a URL or feature flag that differs between dev and prod should not be a literal in the code – it should come from an environment variable or config file. Secrets we discussed – they absolutely shouldn't be there. Solution: use expo's `app.config.js` or `.env` files for configuration. Expo allows conditional config based on environment (as shown in the dev.to example of dynamic config files)[64][65]. This makes it easy to scale to multiple environments (staging, production) without risky manual toggles in code.

- Not Accounting for Different Platforms: (Maintenance) Anti-pattern: assuming the app behaves the same on iOS, Android (and web, if using). There are subtle differences – e.g., Android back button handling, permissions flows, styling quirks. A common anti-pattern is testing only on iOS and shipping – only to realize an Android-specific bug later (or vice versa). Solution: test your app on all target platforms regularly. Use platform-specific code sparingly (Expo lets you use `Platform.OS` checks or even `.ios.js`/`.android.js` file extensions for divergence

when needed). If you find yourself writing lots of platform-specific code, consider if a library can abstract it or if you should factor it differently.

- Neglecting Cleanup: (Performance/Memory) Anti-pattern: setting up listeners, timers, or API calls without cleaning them up properly. For example, adding an event listener in `useEffect` and not removing it in the cleanup, or not clearing intervals/timeouts on unmount. This can cause memory leaks or unwanted behavior (e.g. updating state after a component is unmounted). Solution: always pair your side effects with cleanups. If you use `addEventListener`, remove it in `return () => removeEventListener`. If using `setInterval`, clear it in cleanup. React Navigation also requires cleanup if you use listeners on navigation events. In Expo, if you use something like `Notifications.addNotificationReceivedListener`, keep the subscription object and remove it on unmount. The Sentry performance article specifically flags common memory leaks in RN – make sure to address those patterns (like not clearing timers)[66][67].

- Ignoring Warnings and Errors: (General) Anti-pattern: ignoring the yellow box (warnings) or red box (errors) in development. These often point to potential issues – e.g., deprecated libraries, performance problems (like an uncontrolled promise). Solution: treat warnings as tasks to address. Expo and RN will warn you about things like using old API or requiring a module that's not installed properly. Clean them up as you go, so you don't accumulate technical debt. The same goes for TypeScript errors (don't just use `// @ts-ignore` everywhere!). A clean console means fewer surprises in production.

By avoiding these anti-patterns and following the best practices outlined in this guide, you can develop a React Native app with Expo that is performant, maintainable, secure, and scalable. Expo's evolving ecosystem in 2025 empowers developers to build high-quality apps efficiently – combining the ease-of-use of managed workflow with the flexibility once only found in bare React Native projects. Keep learning and stay engaged with the community, and you'll be well-equipped to tackle challenges as they arise.

Sources:

- Sentry Blog – *React Native performance tactics: Modern strategies and tools*[68][21][23][24]
- Expo Documentation – Environment Variables and Security[69][70]
- Expo Documentation – Monorepos and Hermes Engine[41][71]
- Reddit Discussion – Expo Managed vs Bare Workflow[55]
- React Native Official Docs – Security Guidelines[34][37]
- Expo Documentation – Monitoring and Analytics (EAS Insights, Sentry, etc.)[51][72]
- Reddit Discussion – Naming Conventions in React Native[29]

---

[1] [3] [4] [6] [8] [10] [11] [12] [13] [14] [15] [16] [17] [18] [19] [20] [21] [22] [23] [24] [25] [61] [66] [67] [68] React Native performance tactics: Modern strategies and tools | Product Blog • Sentry

https://blog.sentry.io/react-native-performance-strategies-tools/

[2] Async routes - Expo Documentation

https://docs.expo.dev/router/web/async-routes/

[5] How can I improve React Native app performance on Android?

https://www.reddit.com/r/reactnative/comments/1kz260d/how_can_i_improve_react_native_app_performance_on/

[7] [9] Maximizing Performance in React Native (+ Expo)

https://koptional.com/resource/optimizing-react-native-expo/

[26] [27] Best Practices for Folder Structures in React Native Projects | by Abhishek kumaar | Stackademic

https://blog.stackademic.com/best-practices-for-folder-structures-in-react-native-projects-9c78c3866dd4?gi=02819331cbd5

[28] Expo on X: " How to organize Expo app folder structure for clarity ...

https://x.com/expo/status/1970480194063671589

[29] [30] [31] File name convention and folder structure for react native with expo : r/reactnative

https://www.reddit.com/r/reactnative/comments/1h0tq9g/file_name_convention_and_folder_structure_for/

[32] [34] [35] [36] [37] [38] [70] Security · React Native

https://reactnative.dev/docs/security

[33] [69] Environment variables in Expo - Expo Documentation

https://docs.expo.dev/guides/environment-variables/

[39] [40] [62] [63] 25 React Native Best Practices for High Performance Apps 2025

https://www.esparkinfo.com/blog/react-native-best-practices

[41] [42] [43] [44] Work with monorepos - Expo Documentation

https://docs.expo.dev/guides/monorepos/

[45] [46] [47] Unit testing with Jest - Expo Documentation

https://docs.expo.dev/develop/unit-testing/

[48] Testing React Native Applications: Best practices and Frameworks

https://medium.com/pen-with-paper/testing-react-native-applications-best-practices-and-frameworks-d0c44dca78e7

[49] [50] [53] [54] [58] [59] [64] [65] It's 2025. You Should Probably Be Using Expo for React Native. - DEV Community

https://dev.to/devi_green_00f82b6d705/its-2025-you-should-probably-be-using-expo-for-react-native-407a

[51] [52] [72] Monitoring services - Expo Documentation

https://docs.expo.dev/monitoring/services/

[55] [56] [57] [60] Expo managed vs bare workflow : r/reactnative

https://www.reddit.com/r/reactnative/comments/1ainaia/expo_managed_vs_bare_workflow/

[71] Using Hermes Engine - Expo Documentation

https://docs.expo.dev/guides/using-hermes/