Flutter vs React Native vs Native: 2025 Comparison

Created	@May 19, 2025 11:19 PM
Tags	
Author	Aman
Category	Technical Blog
Description	*We benchmark Flutter, React Native, and Native in 2025 across FPS, memory, startup, and battery. See which framework delivers the performance edge for your next app.*
Excerpt	*Flutter, React Native, or Native? We benchmarked all three with identical apps on real devices in 2025. From startup times to memory usage, our scientific tests reveal who truly delivers smooth FPS, low jank, and efficient battery use. This blog unpacks the results, explains the causation behind each framework's performance, and helps you choose the right tool for your product strategy.*
Slug	flutter-vs-react-native-vs-native-performance-benchmark-2025
Keywords	• ***Primary**: Flutter vs React Native vs Native, mobile app performance benchmark 2025, cross-platform framework performance* • ***Secondary**: app startup time comparison, dropped frames Flutter React Native, iOS Android benchmark, Impeller vs Fabric vs SwiftUI, best framework for mobile app 2025*

> *"Flutter is fast." "React Native is easier." "Nothing beats Native."*
>
> We've all heard the claims. But when **performance is mission-critical**...
>
> **Who actually delivers?**

In this blog, we benchmark **Flutter**, **React Native**, and fully **Native development** (Android/iOS) using 2025-ready tooling, real-world test cases, and a scientific methodology. Our goal is not just to measure **performance**, **responsiveness**, and **user experience**, but also to **recommend the right tool for the right context**.

To make this comparison meaningful, we developed the same **Flashcard Generator App** using all three technologies—ensuring a consistent **UI**, **behavior**, and **user experience** across platforms. This approach allows us to provide **practical, data-driven insights** and guide you on **which framework to choose and when**, based on your needs and constraints.

Why Performance Still Matters (More Than Ever)

In 2025, users demand **zero lag**, **instant loads**, and **battery-friendly** behavior, no matter the device. Whether you're building the next big consumer app or optimizing a mission-critical enterprise tool:

> A slow app is a failed product.

The choice of development framework directly influences app performance and long-term scalability. A beautifully designed UI that stutters or drains power won't survive in production, no matter how good the codebase looks.

That's why we're putting these three frameworks through real benchmarks, not just theoretical comparisons.

The Core Contenders

Flutter (Google)

- **Language**: Dart
- **Target Platforms**: Android, iOS
- **UI Rendering**: Custom rendering engine (Impeller/Skia)

- **RenderThread** for issuing GPU commands
- **Choreographer** for frame scheduling at 60/120 FPS
- **Skia GPU pipeline** for drawing operations

SwiftUI operates with similar elegance, backed by a system-level rendering engine:

- **SwiftUI View hierarchy** is compiled into render instructions
- **Core Animation** manages layout, animation, and rasterization
- **Metal** powers smooth, high-FPS rendering via GPU acceleration

This architecture ensures **tight integration with the OS**, smooth animations, and precise control over memory, gestures, accessibility, and GPU usage. It's especially effective for building high-performance, responsive UIs that align seamlessly with platform guidelines and system behavior.

The result is a **highly optimized native render path**, purpose-built for mobile performance, with direct access to every pixel, sensor, and system capability the platform has to offer.

> But how much better is native in terms of real-world performance?

How We Designed the Benchmark?

To make this comparison meaningful, we did not rely on opinions or anecdotes. We created a **controlled benchmark methodology** where each framework was tested with **identical UI, behavior, workload, and sampling**, ensuring consistent user flows on both iOS and Android.

All measurements were taken on **real devices**, with multiple iterations under repeatable conditions. The benchmarks tracked **startup time, frame rendering stability, memory behavior, I/O operations, and dropped frames**. This approach ensured that we were not only capturing raw speed but also the **runtime behavior** that defines real-world user experience.

By grounding the study in a transparent and repeatable methodology, our goal is to move past assumptions and provide **practical, data-driven insights** into when to choose Flutter, React Native, or Native.

Our study combined two perspectives:

- The **Flashcard AI app** demonstrated developer experience, architectural choices, and integration complexity.
- A dedicated **List Rendering Benchmark** provided a controlled environment to measure runtime performance under identical workloads.

> This study focuses on runtime behavior rather than feature parity.

Disclaimer on Benchmarking Limitations

While every effort was made to control environmental factors during testing, such as battery level, background processes, thermal state, and device configuration. It's important to acknowledge that mobile application performance benchmarking can never be entirely free from exogenous variables. Factors like OS-level scheduling, hardware-specific optimizations, and runtime variability across platforms inevitably introduce some degree of noise. Our goal is not to claim absolute performance truths, but to present repeatable, well-controlled comparisons relative performance trends

Building the Flashcard AI App

To explore developer experience, we built the same **Flashcard Generator AI app** in Flutter, React Native, and Native (Swift for iOS and Kotlin for Android). The app connects with the **OpenAI API** to generate flashcard decks, organizes them in a clean UI, and follows a **clean code architecture** to keep logic, UI, and data layers separate.

This gave us a real-world way to test:

- **Architecture fit**: How each framework supports clean code practices and modularity

- **Key Strength**: Pixel-perfect, consistent UI across platforms

Flutter takes a fundamentally different approach from traditional native rendering. Instead of relying on platform-specific UI components, Flutter renders its UI entirely through its own graphics engine. With the introduction of Impeller, Flutter's new default renderer, it draws every frame directly to the screen using the GPU via Metal on iOS or Vulkan/OpenGL on Android.

Unlike traditional frameworks that rely heavily on the main thread for UI updates, Flutter's architecture separates concerns across multiple threads. Impeller enhances this model by providing pre-compiled shader pipelines and a more deterministic rendering path. This helps eliminate runtime shader jank and ensures consistent, high frame-rate animations.

The result is smooth and reliable performance across platforms, especially for apps with custom UIs, rich animations, or pixel-level precision, without depending on the underlying platform's widget set.

> In theory: Flutter should be blazing fast. But does it hold up under real-world stress? Can it maintain performance without dropping frames or consuming excess memory?

React Native (Meta)

- **Language**: JavaScript / TypeScript
- **Target Platforms**: Android, iOS
- **UI Rendering**: Native components bridged from JS
- **Key Strength**: Web-like developer experience, massive ecosystem

React Native renders UI by bridging React components written in JavaScript or TypeScript to **native platform views**. Instead of drawing pixels directly like Flutter, React Native delegates rendering to the native UI toolkits, **UIKit** on iOS and **ViewGroups** on Android—ensuring platform fidelity and seamless integration with native APIs.

Modern React Native, powered by the **Fabric renderer**, introduces a streamlined, synchronous rendering path that leverages the **JavaScript Interface (JSI)**. This architecture enables faster and more predictable UI updates by directly connecting JavaScript and native C++ objects without serialization. Layout calculation is handled by **Yoga**, a cross-platform Flexbox layout engine optimized for performance and consistency.

React Native components are represented internally as a tree of lightweight **Shadow Nodes**, which are reconciled and committed in batches to the native view hierarchy. This design enables precise control over rendering, layout, and updates.

Additionally, **TurboModules** allow native modules to be loaded on demand, reducing memory footprint and improving startup time. Combined with **Concurrent React**, Fabric enables advanced capabilities like prioritized rendering, smooth transitions, and fine-grained UI scheduling.

The result is a modern and efficient rendering engine that brings the flexibility of React together with high-performance native rendering, offering a robust foundation for scalable, interactive mobile applications.

> But is React Native fast enough to compete with Flutter and Native on today's devices?

Native Development (Android & iOS)

- **Languages**: Kotlin (Android), Swift (iOS)
- **UI Rendering**: Platform-native (Jetpack Compose, SwiftUI/UIKit)
- **Key Strength**: Maximum performance, full platform access

Native development leverages the **platform's built-in rendering engine** and system-optimized graphics stack to deliver high-performance UIs tailored for Android and iOS. On Android, **Jetpack Compose** renders using the Android UI toolkit, powered by RenderThread, Choreographer, and the **Skia** graphics engine. On iOS, **SwiftUI** (or UIKit) renders directly through **Core Animation and Metal**, the same stack Apple uses for all system UIs.

Both Jetpack Compose and SwiftUI adopt a **declarative UI paradigm**, where UIs are composed of reactive state-driven functions. Under the hood, they convert composable functions into **efficient view trees**, track state changes through recomposition or diffing, and submit batched updates directly to the native rendering pipeline.

Jetpack Compose handles layout and rendering through a series of coordinated threads:

- **UI Thread** for event handling and lifecycle

- **Integration experience**: How smooth it was to work with APIs like OpenAI for flashcard generation
- **UI development**: How quickly we could design and implement card-based layouts with responsive interactions
- **Developer workflow**: How the ecosystem, tooling, and language shaped speed and productivity

Working across all three frameworks highlighted important trade-offs. For example:

- Flutter made it straightforward to achieve pixel-perfect UIs with a single codebase.
- React Native felt familiar for teams with a web background, but required more care in performance-sensitive sections.
- Native offered the most power and system-level integration, but at the cost of more complexity and duplicated work across platforms.

This exercise surfaced **what it feels like to build and ship** in each ecosystem. But developer experience is only one side of the story.

To understand how these frameworks perform under pressure, we designed a **dedicated List Rendering Benchmark**. This allowed us to test runtime behavior under identical conditions, removing subjective impressions and focusing on measurable outcomes.

Technical Benchmark Details

To measure **runtime performance objectively**, we created a dedicated **List Rendering Benchmark Screen** in each framework: Flutter, React Native, Swift (iOS), and Kotlin (Android).

This benchmark was carefully designed to be **consistent and reproducible** across all implementations:

- Rendered a fixed set of **100 list items** per run
- Automated the scrolling process for uniformity
- Repeated the benchmark **three times** per framework and platform
- Applied **identical layouts and scroll distances** in every case
- Measured **startup latency, rendering smoothness, and memory usage**
- Followed each framework's best practices while keeping implementations **functionally equivalent**

By isolating list rendering, we removed higher-level app variability and focused solely on **rendering efficiency**, giving us a clear picture of how each framework behaves under the same workload.

Devices and Environment

All benchmarks were executed on **physical devices** under controlled conditions to ensure consistency:

Platform	iOS	Android
Model	iPhone 16 Plus	Samsung Galaxy Z Fold 6
Chipset	Apple A18	Snapdragon 8 Gen 3 for Galaxy (Adreno 750 GPU)
Architecture	ARMv9.2-A	Armv9
GPU Interface	Metal	Vulkan 1.3
RAM Capacity	8 GB	12 GB
Memory Management	ARC	ART
Display Technology	Super Retina XDR (OLED)	Dynamic AMOLED 2X
Display Refresh Rate	60 Hz	120 Hz
Display Resolution	2796 × 1290 pixels at ~460 ppi	2376 × 968 pixels at ~407–410 ppi
Battery SoC (during benchmark)	80 ± 2 %	80 ± 2 %
Temperature	28-32 °C	28–32 °C
Device Config (during benchmark)	- Low Power mode: Off - All background apps closed - Display Brightness: 40%	- Display Brightness: 40% (Auto-brightness Off) - 120 Hz enabled

Platform	iOS	Android
	- Auto brightness: Off - True tone: Off - Night shift: Off - Reduced Motion: On - Airplane mode: On - Wifi: On - Network bandwidth: 200 ± 50 mbps (upload and download) - Background App refresh: Off - Bluetooth: Off - Airdrop: Off - Notifications: DND	- Airplane mode: On - Wi-Fi: On (200 ± 50 Mbps up/down) - Bluetooth: Off - Background processes minimized / background restrictions on test apps - Notifications: DND

To maintain fairness across runs, the environment was carefully controlled:

- Battery level kept **above 80 percent**
- **Minimal background processes** active
- Stable **thermal state** to avoid throttling
- **Minimal network** activity

> To keep things fair, we tested Android on the Galaxy Z Fold 6 with its silky 120 Hz Dynamic AMOLED 2X display: the best hardware we could get our hands on, so the device itself wasn't the bottleneck.
>
> iOS, on the other hand, ran on the iPhone 16 Plus, which still tops out at 60 Hz. For all the number crunching, we **normalized performance against each panel's refresh target,** so the comparison stays clean.
>
> *That said,*
> *Dear Apple, maybe it's time every $1,000+ iPhone gets 120 Hz. ProMotion shouldn't require a "Pro" suffix, smoothness is a human right, not a product tier.*

Parameters Measured

Our benchmark captured both **runtime performance** and **build/runtime resource characteristics**.

Runtime Metrics

- **Startup Time (TTFF)** – Time to first frame rendered after app launch, with standard deviation across runs.
- **Frame Rendering** – Average frame time, 95th percentile frame time (P95), actual FPS (clamped and unclamped), dropped frames percentage, and janky frames percentage.
- **Frame Stability** – Coefficient of Variation (CoV) of frame times, reflecting smoothness.
- **Total Frames** – Number of frames produced during the scroll benchmark.
- **Scroll Performance** – Distance scrolled (px) and average scroll duration (ms).

Memory Metrics

- **Memory Delta (MB)** – Increase in memory usage during the benchmark.
- **Memory per Item (KB)** – Memory cost per rendered list item.
- **Memory Stability** – Standard deviation of memory usage across iterations.

Build & Binary Metrics *(collected manually, outside runtime execution)*

- **Binary Size** – Final build size in MB for both iOS and Android.
- **Build Time** – Average time to produce a build, measured in seconds.

- **Build Memory** – Peak memory consumption during the build process.

Formulas Used

To ensure consistency across frameworks, the following formulas were applied when calculating benchmark metrics:

- **Startup Time (TTFF)**

$$TTFF = T_{first\_frame} - T_{app\_launch}$$

  Time difference between app launch and first rendered frame.

- **Frames Per Second (FPS)**

$$FPS = \frac{1000}{AvgFrameTime\ (ms)}$$

  Converts average frame time into effective frame rate.

- **95th Percentile Frame Time (P95)**

  Extracted as the frame time below which 95 percent of samples fall, to capture tail latency.

- **Dropped Frames %**

$$DroppedFrames\% = \frac{MissedFrames}{TotalFrames} \times 100$$

- **Janky Frames %**

$$JankyFrames\% = \frac{Frames\ exceeding\ threshold}{TotalFrames} \times 100$$

  Where threshold = 16.67 ms (for 60 Hz) or 8.33 ms (for 120 Hz).

- **Memory Delta (MB)**

$$\Delta Memory = PeakMemory - BaselineMemory$$

- **Memory per Item (KB)**

$$Memory_{per\_item} = \frac{\Delta Memory}{Items} \times 1024$$

- **Coefficient of Variation (CoV)**

$$CoV = \frac{\sigma}{\mu} \times 100$$

$$\sigma = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(x_i - \mu)^2}$$

$$\mu : \text{ Mean (average) frame time}$$

$$\sigma : \text{Mean (average) frame time}$$

  Ratio of standard deviation (σ) to mean frame time (μ), indicating frame stability.

Measurement Tools

- **iOS Native (Swift + Obj-C)**
  - **Frame timing:** `CADisplayLink` deltas from the display callback.
  - **Refresh info:** `CADisplayLink.preferredFrameRateRange` (iOS 15+) for min/max/preferred Hz.
  - **Memory:** Mach task APIs via `task_info(..., MACH_TASK_BASIC_INFO / TASK_VM_INFO)` to get RSS and related fields.
- **Android Native (Kotlin/Compose)**

- **Frame timing:** `android.view.Choreographer` frame callbacks to collect per-frame durations.
- **Memory:** `Debug.getMemoryInfo()` for PSS plus `ActivityManager.getMemoryInfo()` for system snapshot.

- **Flutter (Dart)**
  - **Frame timing:** `SchedulerBinding.instance.addTimingsCallback` capturing FrameTiming records.
  - **Memory:** `ProcessInfo.currentRss` (Dart `dart:io`) for process RSS.
- **React Native (JS + NativeModules)**
  - **Bridge modules:** `NativeModules.FrameProfiler` (frames) and `NativeModules.MemoryProfiler` (memory) call into the iOS/Android native code.
  - **Note:** We are **not** using JSI hooks, Fabric renderer internals, or Yoga layout metrics for profiling. Metrics flow over the classic RN bridge from those native modules.

Benchmark Results (2025): Flutter vs React Native vs Native

With the setup locked, we move from lab notes to lap times. All charts below clearly distinguishes each display's refresh target (16.7 ms @60 Hz, 8.3 ms @120 Hz) to reflect true, user-visible smoothness.

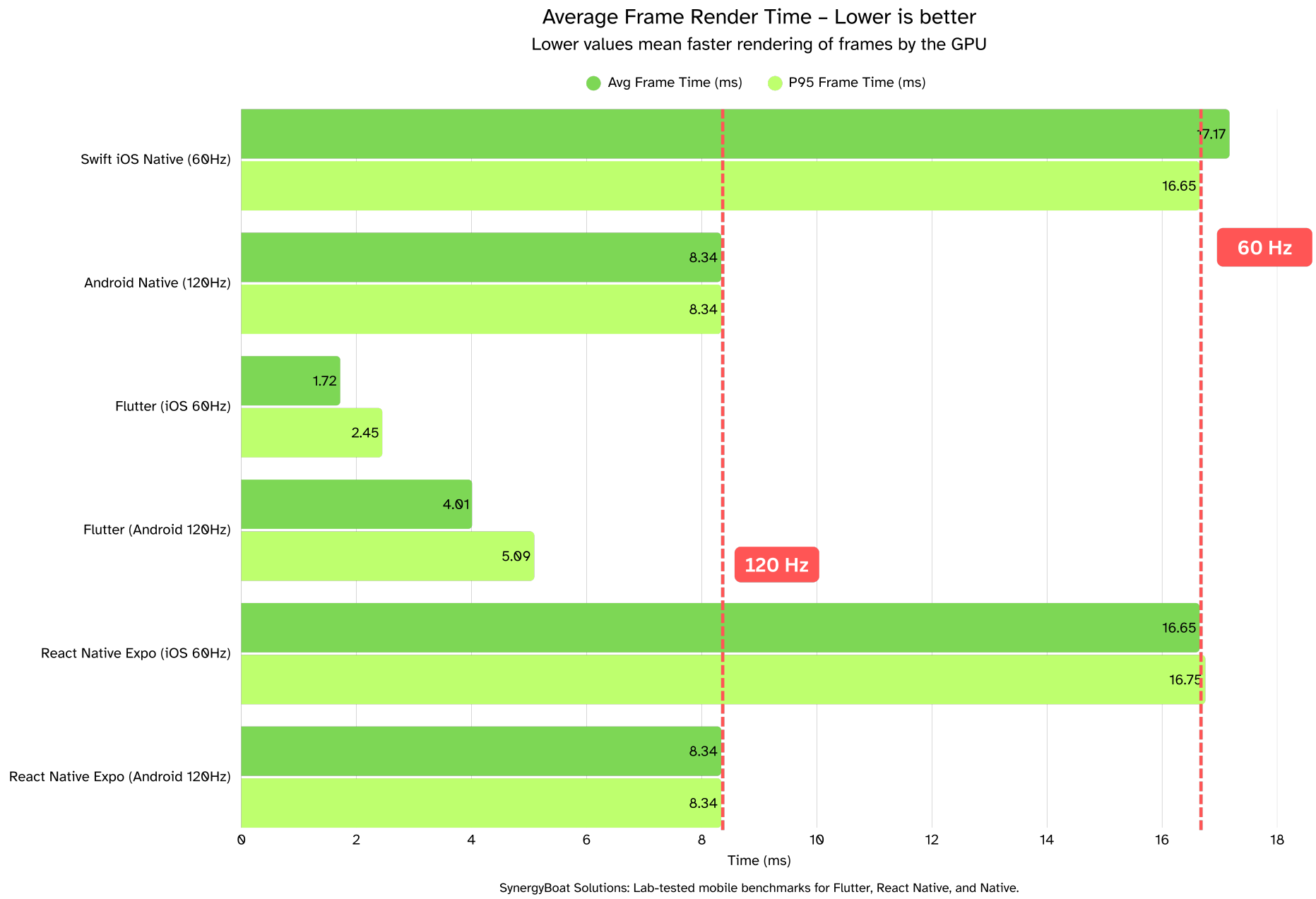
Average Frame Render Time

Average Frame Render Time is the mean time your app takes to produce each frame during the test run. Think of it as the "cruising speed" of your UI. Lower is better, and it directly maps to perceived smoothness. It's a solid **health check** for overall rendering efficiency and a good way to spot broad regressions (layout work, heavy decoding, overdraw).

However, averages can be charmingly deceitful: they hide stutters and hitching that users actually feel. Use it alongside **tail metrics** (**p95**), **DroppedFrame%**, and **Jank%** to see the whole picture, and always read it against the **frame budget** (≈16.7 ms at 60 Hz, ≈8.3 ms at 120 Hz).

Practical rule of thumb: Keep your average comfortably below the budget (e.g., ≤14 ms at 60 Hz, ≤7 ms at 120 Hz) to preserve headroom for GC, I/O, and animation spikes, because smooth isn't the average, it's the experience.

> **V-Sync Note (Render Time & FPS Capping):** Mobile UIs are synchronized to the display's v-sync, so frames can only appear at fixed intervals i.e. ~16.7 ms at 60 Hz and ~8.3 ms at 120 Hz. If a frame finishes early, it simply waits for the next v-sync. Users won't see "extra" frames. In **Flutter**, `FrameTiming` exposes engine-side **build** and **raster** durations that aren't v-sync-capped, so you can see true rendering effort. In **iOS/Android native** and **React Native** (with `CADisplayLink` / `Choreographer`), our timestamps are **v-sync-aligned**, so an on-time frame may still report near the budget even if it completed earlier.
>
> Practical reading: Judge **Average Frame Render Time** against the **frame budget**, and pay special attention to bars that **cross the red line**. Those are missed v-sync deadlines (jank). Sub-budget bars indicate headroom and stability, not higher visible FPS (no medals for "200 FPS on a 60 Hz screen").



Insights:

- **iOS (60 Hz, budget ≈ 16.7 ms)**
  - **Flutter: (Avg 1.72 ms, p95 2.45 ms)**
    Huge headroom; tails are comfortably sub-budget.
  - **React Native: (Avg 16.65 ms, p95 16.75 ms)**
    Right at the budget, stable but little margin for spikes (GC/decodes).
  - **Swift (Native): (Avg 17.17 ms, p95 16.65 ms)**
    Average is over budget while p95 is just under it, implying a **small set of very slow frames** skewing the mean.
- **Android (120 Hz, budget ≈ 8.3 ms)**
  - **Flutter: (Avg 4.01 ms, p95 5.09 ms)**
    Comfortably sub-budget (≈ 41-61% of the budget). Strong headroom and stable tails. Low jank risk at 120 Hz.
  - **React Native: (Avg 8.34 ms, p95 8.34 ms)**
    Right at the frame budget (v-sync limited). Smooth at steady state, but **no safety margin** for GC/decodes or image bursts.
  - **Android Native (Kotlin): (Avg 8.34 ms, p95 8.34 ms)**
    Also budget-bound. Smooth now, but any extra UI-thread work can tip frames over 8.3 ms.

How to read this: Averages are v-sync–aware on native/RN (they cluster near the frame period), while Flutter's engine timings reflect actual render work, so sub-budget numbers indicate **headroom**, not higher visible FPS.

The strategic takeaway: **Flutter shows the largest performance reserve on both platforms**; RN and Native look fine at steady state but need their **tail metrics (p95/p99, jank/big-jank)** to prove stability under stress

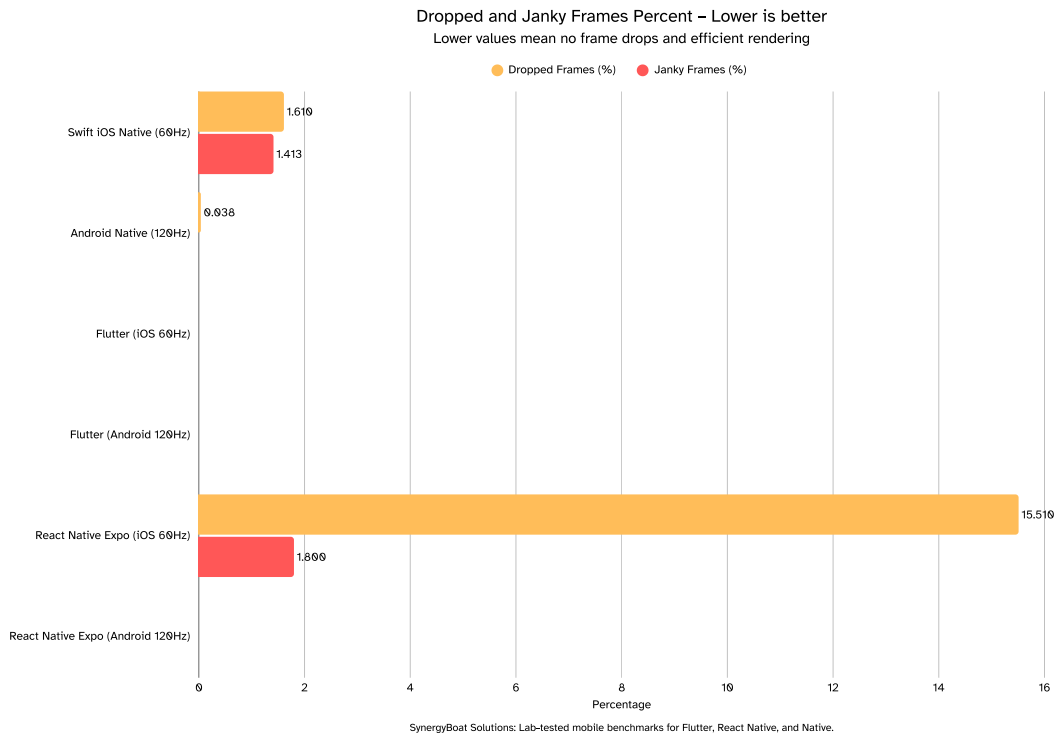
Dropped and Janky Frames

Dropped Frames Percent is the share of frames that miss their v-sync deadline entirely and are not presented on time. **Janky Frames Percent** is the share of frames that exceed the frame budget but still show up on the next v-sync. Lower is

better for both. These metrics reveal the stutters users actually feel. Use them to catch GC spikes, image decode stalls, layout bursts, or bridge contention that a good average can hide.

However, single percentages can be charmingly deceitful. Read them alongside **p95**, **Average Frame Render Time**, and your **frame budget** (≈16.7 ms at 60 Hz, ≈8.3 ms at 120 Hz)

Practical rule of thumb: Keep **Jank%** under **1 to 2 percent** and **DroppedFrames%** well under **1 percent** for sustained scrolls. Anything above that is noticeable in production.



Insights:

- **iOS (60 Hz, budget ≈ 16.7 ms)**
  - **Flutter: (Dropped 0%, Jank 0%)**

    No missed slots and no over-budget frames. Excellent headroom.
  - **React Native: (Dropped 15.51%, Jank 1.8%)**

    High DroppedFrames% indicates frequent missed v-sync under scroll.
  - **Swift (Native): (Dropped 1.61%, Jank 1.413%)**

    Small but noticeable tails. Review occasional layout or decode spikes.
- **Android (120 Hz, budget ≈ 8.3 ms)**
  - **Flutter: (Dropped 0%, Jank 0%)**

    Clean run with strong timing margin at 120 Hz.
  - **React Native: (Dropped 0%, Jank 0%)**

    At-budget averages with no measured misses in this scenario.
  - **Android Native (Kotlin): (Dropped 0.038%, Jank 0%)**

    Essentially zero drops. Any separation will show up in p95 on longer or heavier runs.

How to read this: DroppedFrames% captures missed presentations. Jank% captures frames that exceeded the budget. Zeroes mean the pipeline met its deadlines. Non-zero values point to contention on the UI, JS, or decode paths. Interpret these alongside **p95** and **Average Frame Render Time** to separate steady-state smoothness from stress behavior.

The strategic takeaway: **Flutter** shows zero jank and zero drops on both platforms in this scenario, which aligns with its headroom in average and p95 timing. **React Native** is clean on Android but shows elevated **DroppedFrames%** on iOS, which warrants optimization. **Native** is close to ideal on Android and slightly noisy on iOS.

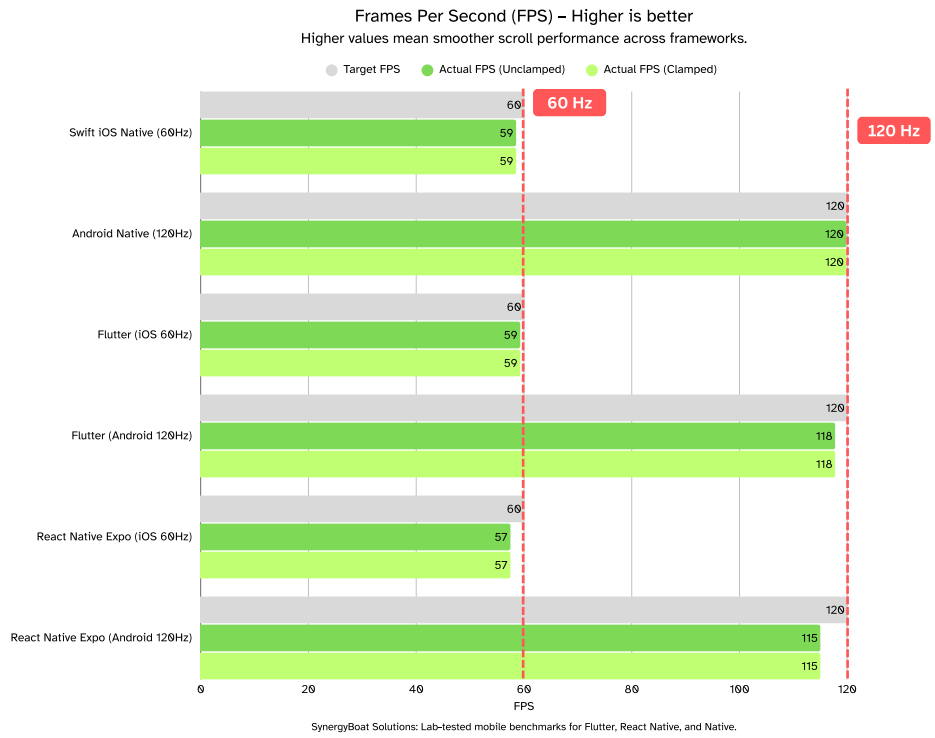
Frames Per Second

FPS is the number of frames the app actually presents per second during the test run. It is the presentation rate users see. Higher is better, up to the device’s refresh ceiling. It is a solid health check for steady-state smoothness and a quick way to spot underutilization of the frame budget.

However, it hides stutters and hitching that appear as tail events. Read FPS together with **p95**, **Average Frame Render Time**, **DroppedFrames%**, and **Jank%**, and always compare against the **panel ceiling** (60 or 120 Hz).

Practical rule of thumb: Aim for **≥ 98%** of the panel refresh under sustained scroll when tails are also clean.

> **V-Sync Note (Clamped vs Unclamped):** Mobile displays present frames in v-sync slots only. Because our runs stayed within the budget most of the time, **Unclamped** and **Clamped** FPS are identical here. Sub-budget frame times do not raise visible FPS beyond the panel limit. Use FPS as a presentation sanity check, and use frame time and p95 to understand real headroom.



Insights:

- **iOS (60 Hz, ceiling 60)**
  - **Flutter: (Actual 59.31, Clamped 59.31)**

    ~**98.85%** of ceiling. Aligns with zero drops and zero jank. Excellent headroom.
  - **React Native: (Actual 57.49, Clamped 57.49)**

    ~**95.82%** of ceiling. Matches the elevated DroppedFrames% earlier. Investigate JS thread pressure and image decode.
  - **Swift (Native): (Actual 58.54, Clamped 58.54)**

    ~**97.57%** of ceiling. Small shortfall consistent with light tails.
- **Android (120 Hz, ceiling 120)**
  - **Flutter: (Actual 117.75, Clamped 117.75)**

    ~**98.13%** of ceiling. Clean presentation with strong timing margin.
  - **React Native: (Actual 115, Clamped 115)**

    ~**95.83%** of ceiling. Smooth to the eye in this run, but below ideal. Check p95 and jank on longer traces.
  - **Android Native (Kotlin): (Actual 119.84, Clamped 119.84)**

    ~**99.87%** of ceiling. Essentially perfect presentation.

How to read this: FPS is the visible outcome of meeting v-sync deadlines. Equality of Unclamped and Clamped confirms v-sync capping. Use FPS to confirm presentation quality, then rely on **p95** and **Average Frame Render Time** to diagnose headroom and the real causes of any drops.

The strategic takeaway: On iOS, **Flutter** tracks closest to 60 while **React Native** trails and **Swift** is close behind. On Android, **Native** is nearly perfect, **Flutter** is very close, and **React Native** lags slightly.

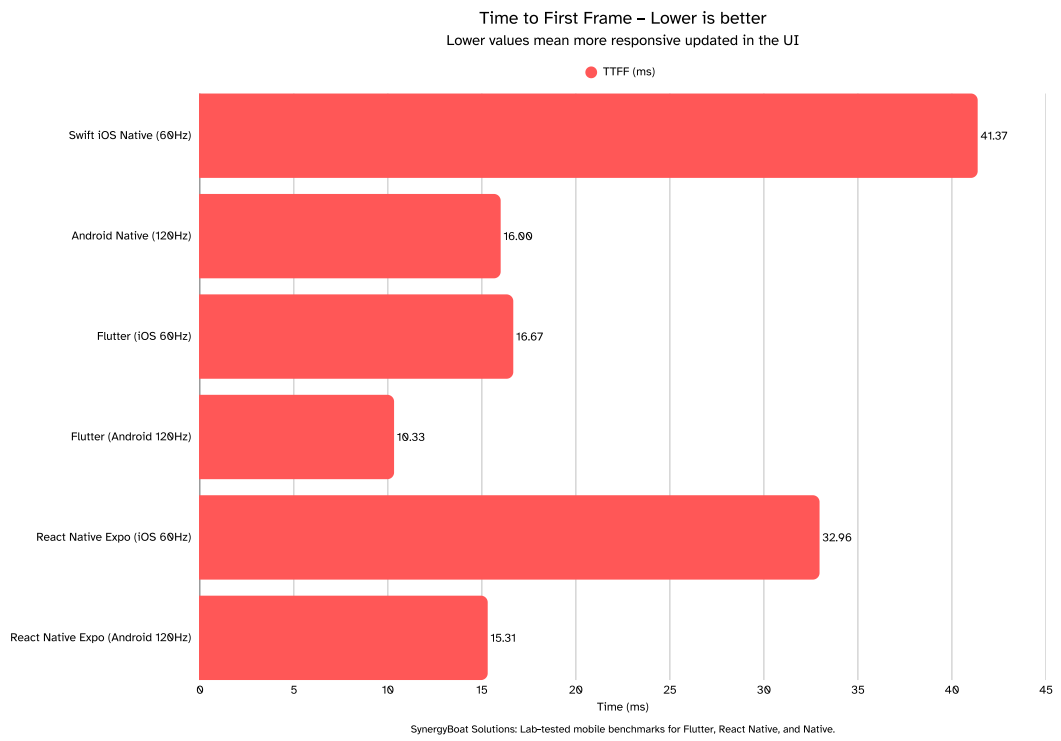
Time to First Frame (TTFF)

Time to First Frame (TTFF) is the time from launch to the first presented frame. Lower is better. It sets the very first impression of snappiness and is a solid health check for startup work like runtime initialization, dependency injection, and initial layout.

However, single TTFF values can be deceitful. Look at **StdDev** to understand stability across runs and read **TTFF** alongside **Average Frame Render Time**, **p95**, **DroppedFrames%**, and **Jank%**.

Practical rule of thumb: Aim for TTFF that feels instant to the eye. Sub-50 ms reads as immediate. Keep **StdDev** tight so users do not occasionally see a slower first paint.

> **Launch Semantics Note (Cold vs Warm):** TTFF depends on what is initialized before the first frame. Warm starts, cached assets, and prewarmed engines reduce TTFF. Cold starts, heavy module initialization, and synchronous I/O increase it.



Insights:

- **iOS (60 Hz)**
  - **Flutter: (TTFF 16.67 ms, StdDev 1.25 ms)**

    Very fast and very consistent. Startup pipeline is lean before first paint.
  - **React Native: (TTFF 32.96 ms, StdDev 0.16 ms)**

    Slower than Flutter but extremely stable. Good predictability for first paint.
  - **Swift (Native): (TTFF 41.37 ms, StdDev 54.75 ms)**

    Average is acceptable, variance is not. Occasional slow paths skew the experience.
- **Android (120 Hz)**
  - **Flutter: (TTFF 10.33 ms, StdDev 4.5 ms)**

    Fastest first paint with some variability. Still reads instant. Profile the few slower runs.
  - **React Native: (TTFF 15.31 ms, StdDev 0.46 ms)**

    Slightly slower than Flutter but very consistent. Predictable first paint.
  - **Android Native (Kotlin): (TTFF 16 ms, StdDev 0 ms)**

    Essentially quantized and rock-steady. Startup path is clean and repeatable.

How to read this: TTFF is your first impression metric. Use **StdDev** to judge if that impression is consistent. If TTFF looks great but StdDev is high, users will sometimes feel a slow start even if the average looks fine.

The strategic takeaway: All three stacks deliver sub-50 ms first paints in this scenario. **Flutter** is the quickest to first frame on both platforms, **React Native** trades a few milliseconds for excellent consistency, and **Native** is almost perfectly steady on Android but noisy on iOS.

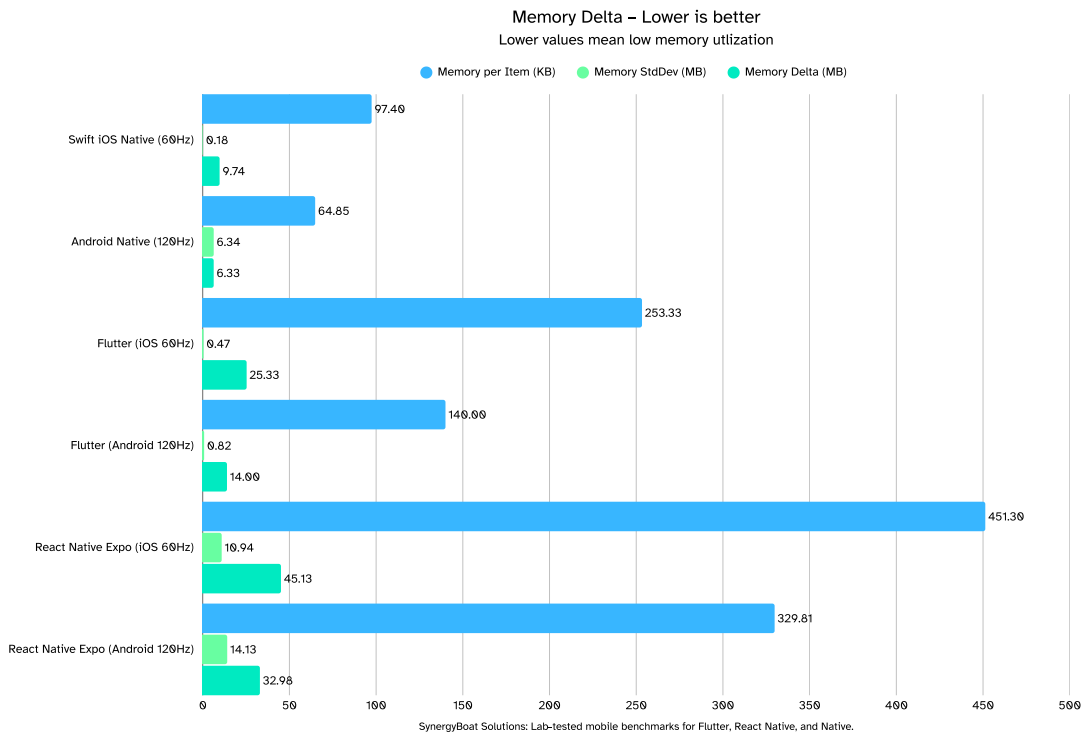
Memory Delta

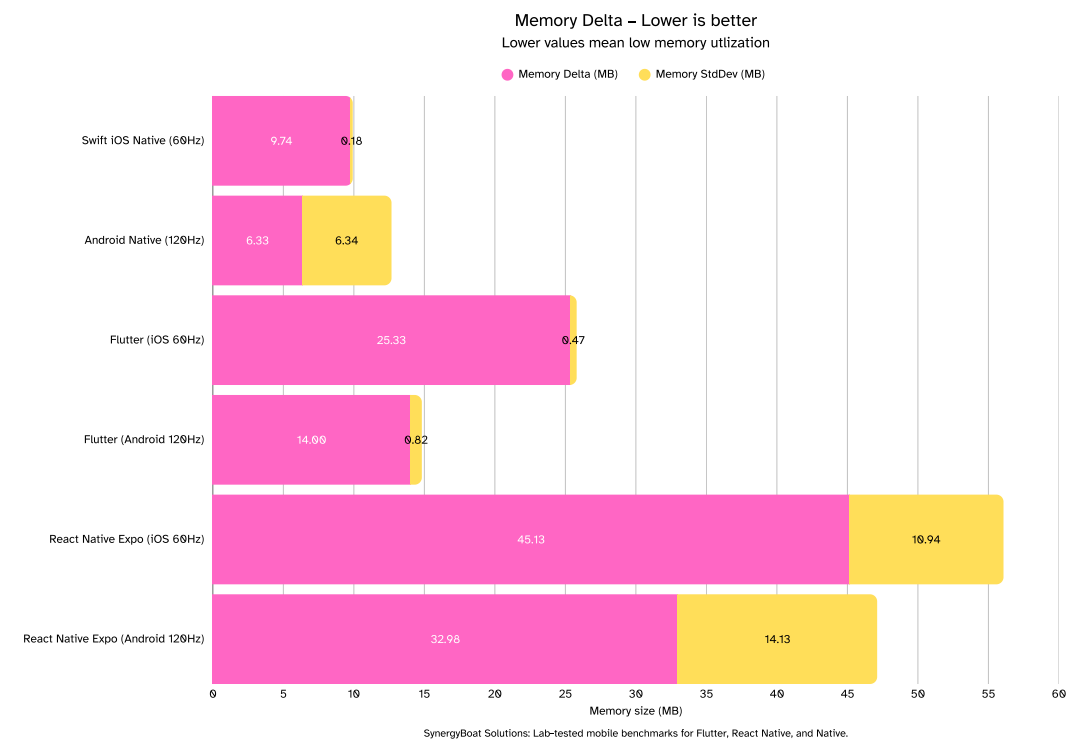
Memory Delta is the increase in process memory during the benchmarked action. Lower is better. It reflects allocations created by rendering, data structures, images, and caches that come alive while the list scrolls.

However, single deltas can be misleading. Always read **StdDev** to judge stability across runs, and compare **within each OS** because iOS and Android report memory differently.

Practical rule of thumb: Keep **ΔMemory** in the low teens for simple scrolls and keep **StdDev** small so growth is predictable.

> **Memory Metric Note (RSS vs PSS):** iOS reports **RSS** and Android reports **PSS**. These are not directly comparable. Use **within-platform deltas** for fairness and call out the metric in tables so no one mixes apples with Androids.





Insights:

- **iOS (60 Hz)**
  - **Flutter: (ΔMemory 25.33 MB, StdDev 0.47 MB)**

    Moderate growth with very tight variance. Headroom is good and behavior is consistent.
  - **React Native: (ΔMemory 45.13 MB, StdDev 10.94 MB)**

    Highest growth and high variability.
  - **Swift (Native): (ΔMemory 9.74 MB, StdDev 0.18 MB)**

    Small and stable footprint. Allocations look disciplined.

- **Android (120 Hz)**
  - **Flutter: (ΔMemory 14.00 MB, StdDev 0.82 MB)**

    Low growth with tight variance. Healthy profile for a scrolling workload.
  - **React Native: (ΔMemory 32.98 MB, StdDev 14.13 MB)**

    Elevated growth with large variance. Bursty allocations suggest cache churn or GC-sensitive paths.
  - **Android Native (Kotlin): (ΔMemory 6.33 MB, StdDev 6.34 MB)**

    Low average with high variance. Occasional spikes offset otherwise small growth.

How to read this: ΔMemory shows how heavy a screen becomes as it works. **StdDev** separates steady growth from spiky behavior. Interpret results alongside **Average Frame Render Time**, **p95**, **DroppedFrames%**, and **Jank%** to see whether memory pressure aligns with visual hitches.

The strategic takeaway: **Native** holds the leanest memory profile on both platforms. **Flutter** stays moderate and consistent, which pairs well with its timing headroom. **React Native** shows higher growth and variability, especially on iOS, which calls for targeted optimization.

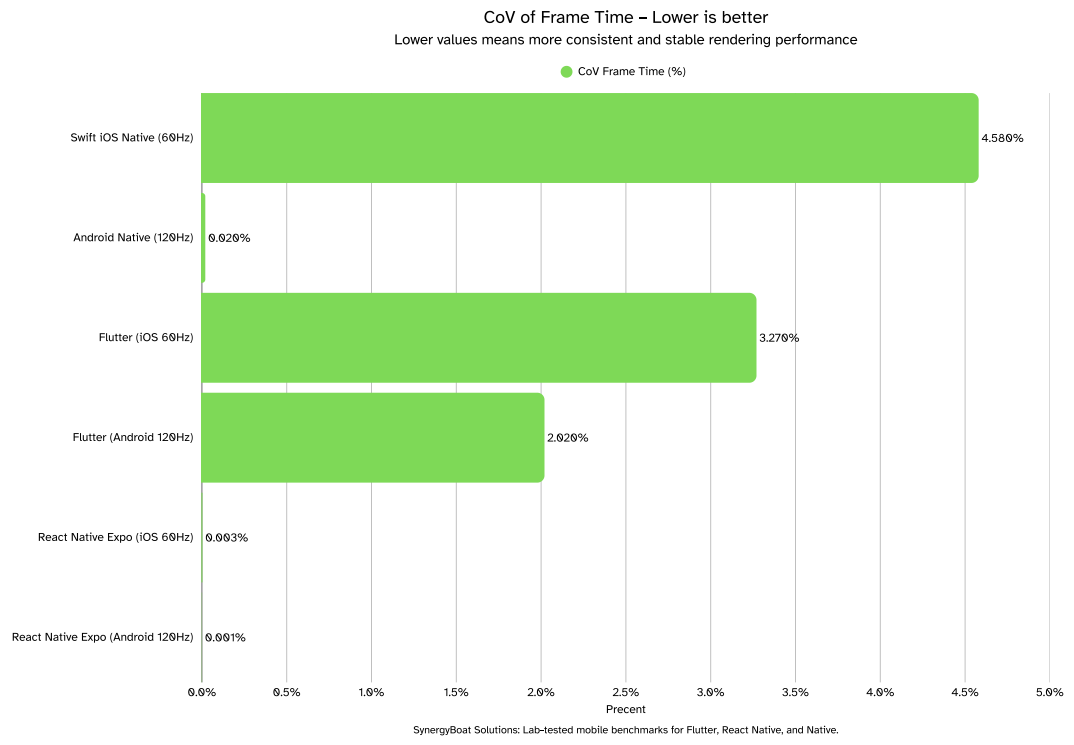
Coefficient of Variation (CoV) of Frame Time

Coefficient of Variation (CoV) of Frame Time is the ratio of the standard deviation to the mean frame time, expressed in percent. Lower is better. It captures frame-to-frame stability, which maps to perceived smoothness. It is a solid health check for pacing consistency and a good way to spot micro-hitches even when averages look fine.

Read them alongside **p95**, **Average Frame Render Time**, **DroppedFrames%**, and **Jank%**, and always compare against the **frame budget** (≈16.7 ms at 60 Hz, ≈8.3 ms at 120 Hz).

Practical rule of thumb: Keep **CoV** in the low single digits for sustained scrolls. If CoV rises above 5 percent on 60 Hz or 3 percent on 120 Hz, expect visible inconsistency.

> **Variability Note (Quantization and sample size):** V-sync quantization can compress dispersion when frames cluster near the budget, and small sample sizes can make CoV look artificially tiny. Practical reading: treat CoV as a stability lens, not a pass score. Confirm with **p95** and **JankFrames%** and **DroppedFrames%**.



Insights:

- **iOS (60 Hz)**
  - **Flutter: (CoV 3.27%)**

    Low variability. Pairs well with sub-budget averages and clean tails.
  - **React Native: (CoV 0.003%)**

Ultra-low dispersion, yet remember earlier DroppedFrames% on iOS. Stable pacing does not guarantee meeting the budget. Verify with **p95** and **drops**.

- **Swift (Native): (CoV 4.58%)**

  Moderate variability. Occasional pacing wobble is consistent with light tails.

- **Android (120 Hz)**
  - **Flutter: (CoV 2.02%)**

    Low and steady at 120 Hz. Consistent motion with headroom.

  - **React Native: (CoV 0.001%)**

    Essentially zero dispersion in this run. Stability is excellent. Confirm headroom with **p95**.

  - **Android Native (Kotlin): (CoV 0.02%)**

    Near-perfect stability. Any separation will show in tail metrics on heavier runs.

How to read this: CoV tells you how even the frame cadence is. It does not tell you whether frames meet the budget. A pipeline can have tiny CoV and still miss v-sync consistently. Interpret CoV together with **p95**, **Average Frame Render Time**, **DroppedFrames%**, and **Jank%**.

The strategic takeaway: All stacks exhibit low variability, especially on Android. **React Native** shows ultra-low CoV on both platforms, but iOS still needs attention given earlier drops. **Flutter** maintains low CoV with strong headroom, and **Native** is stable with minor wobble on iOS. Use CoV to validate pacing, then let tail metrics decide readiness.

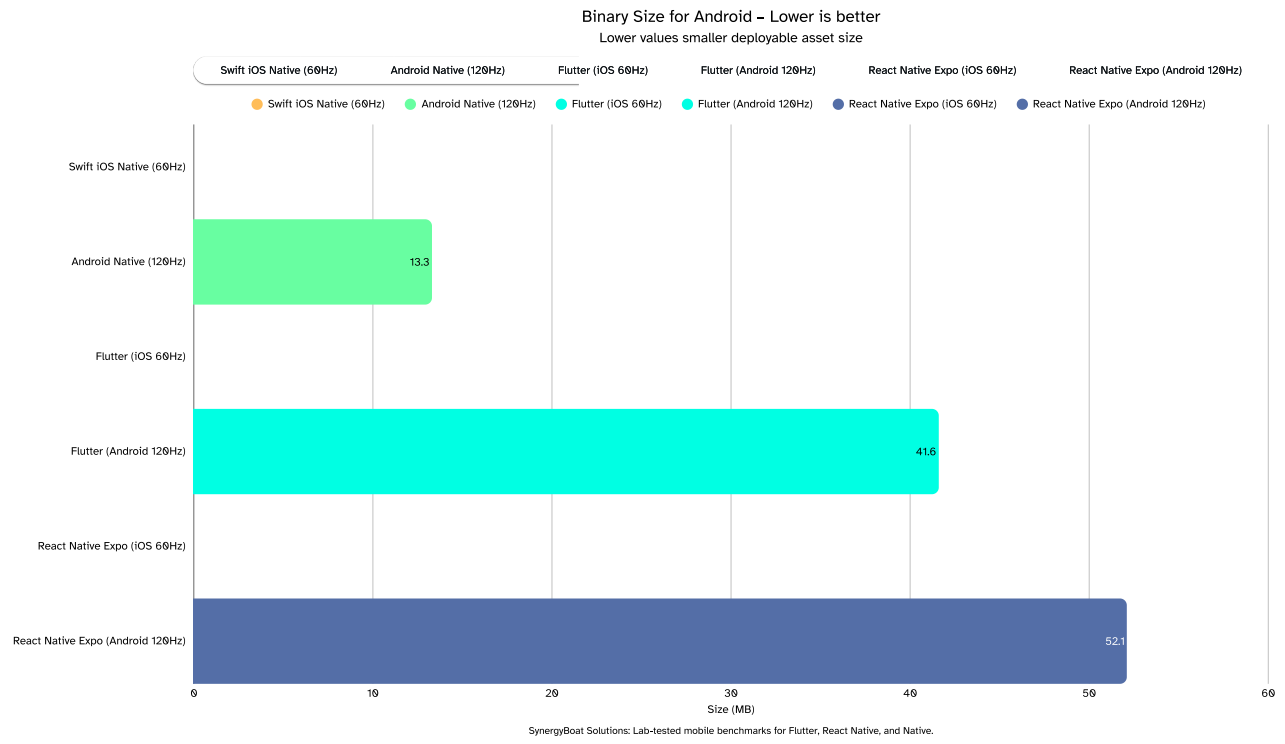
Binary Size

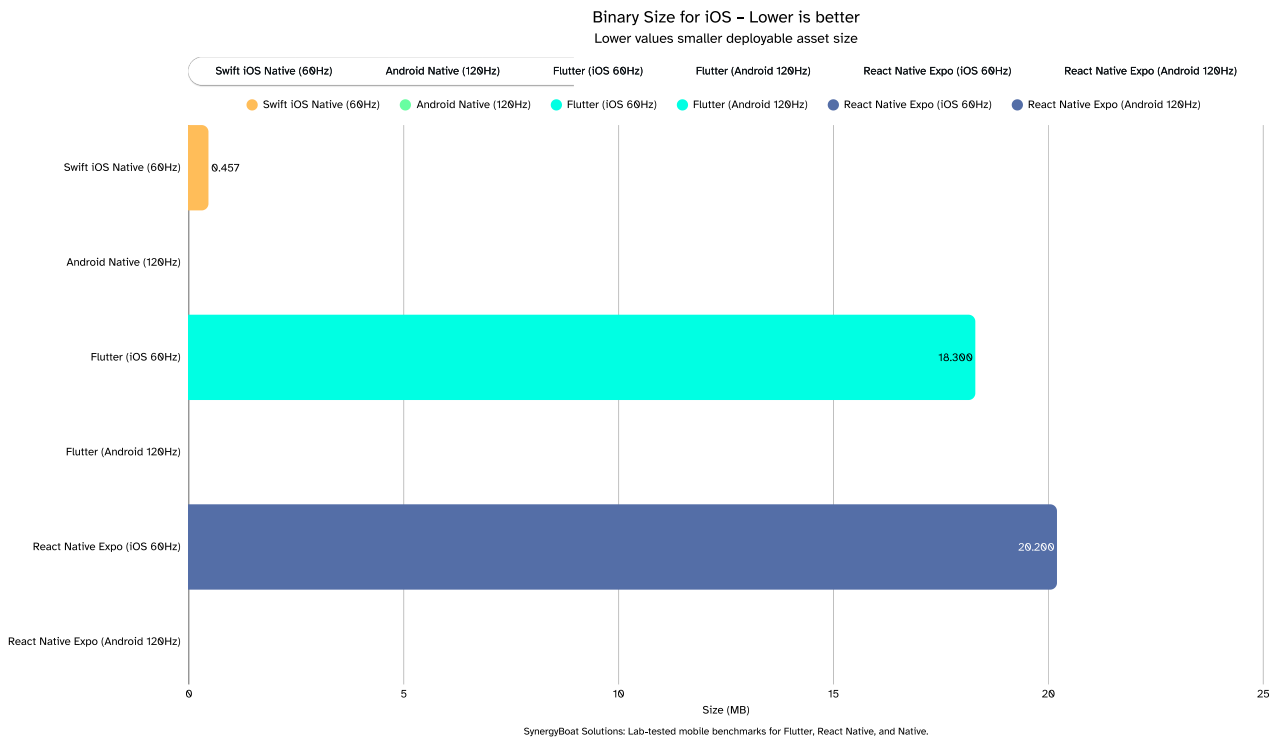
Binary Size is the size of the built app artifacts generated from the project. Lower is better. It affects store download, install time, update bandwidth, and cold start I/O due to package decompression and library loading.

However, single numbers won't show the full picture. iOS and Android handle frameworks and splitting differently, so absolute sizes are not directly comparable across OS.

Practical rule of thumb: Keep minimal screens in low double digits on Android and sub-20 MB on iOS for cross-platform stacks. Track deltas over time. Sudden jumps usually come from large assets, extra native modules, or disabled shrinkers.

> **Build Artifact Note:** iOS often reports the `.ipa` or uncompressed `.app` without dSYMs. Android may be an **APK** or an **AAB** that Play splits per ABI and density. React Native with **Expo** includes runtime components that increase baseline size. Flutter bundles the engine and ICU data, which raises baselines versus pure native.





Insights:

- **iOS**
  - **Flutter: (18.3 MB)**

    Typical Flutter baseline for a simple screen. Includes engine and ICU.
  - **React Native (Expo): (20.2 MB)**

    Highest on iOS. Expo adds convenience plus a larger runtime.
  - **Swift (Native): (0.457 MB)**

    Extremely small stub that relies on system frameworks. Real apps rise with assets, Swift libs, and features.
- **Android**
  - **Flutter: (41.6 MB)**

    Mid-range baseline for a release artifact. Expect smaller effective download from AAB splits.
  - **React Native (Expo): (52.1 MB)**

    Largest here. Expo runtime and bundled JS raise size.
  - **Android Native (Kotlin): (13.3 MB)**

    Smallest on Android. Size will scale mainly with assets and any NDK components.

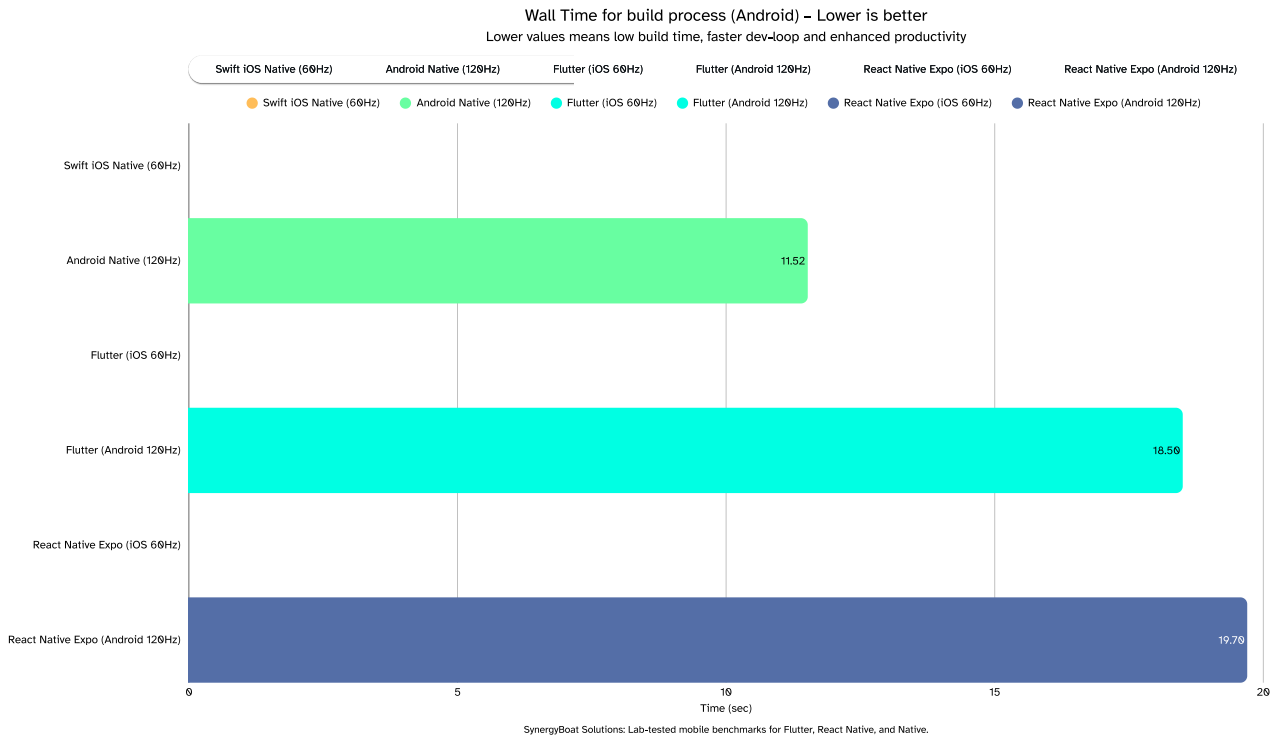
How to read this: iOS native looks tiny because the OS ships most frameworks. Cross-platform stacks bundle a runtime, so baselines are higher. On Android, final user download from the store is often smaller than the raw APK due to AAB splitting. Always pair binary size with **TTFF** and **ΔMemory** to avoid trading size for runtime cost.

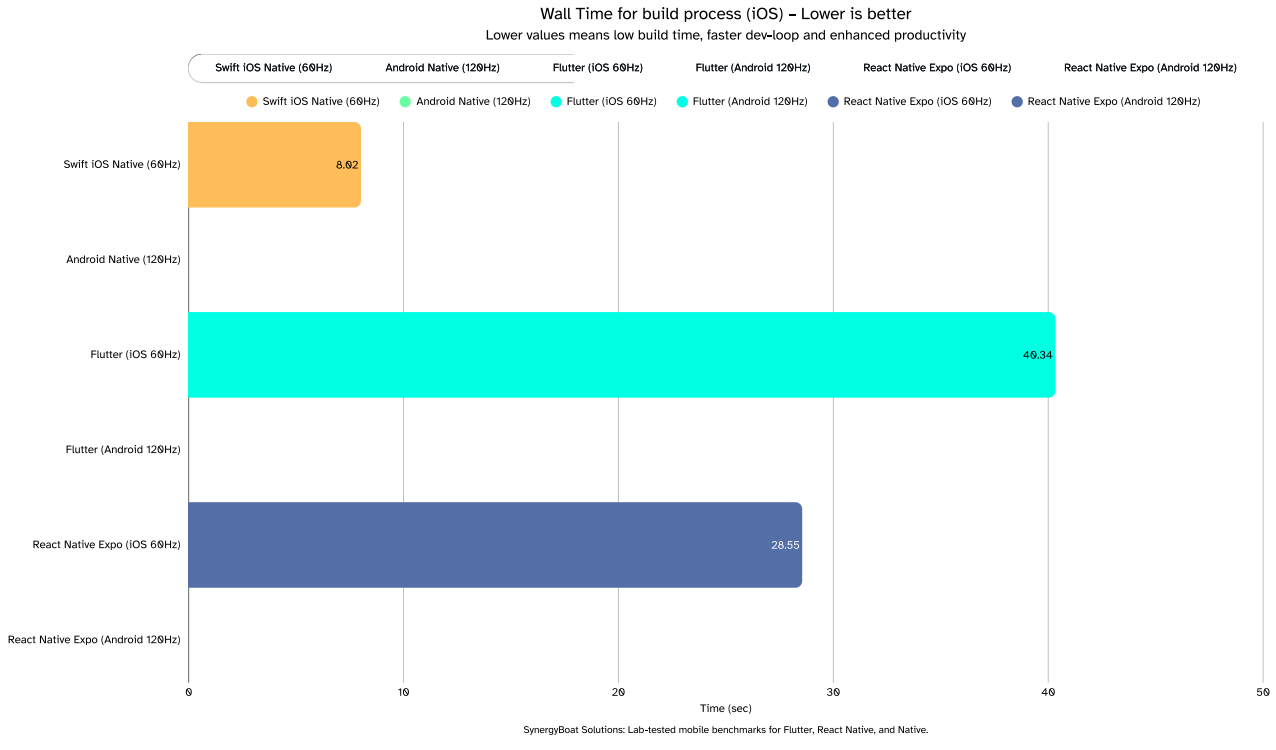
The strategic takeaway: **Native** wins raw footprint. **Flutter** stays moderate and predictable across both the operating systems. **React Native (Expo)** is the heaviest baseline. It buys speed of development but needs proactive trimming of modules, assets, and packaging to keep store size competitive.

Build Time (inner dev-loop)

Build Time (inner dev-loop) is the wall-clock time from a code change to the next runnable build on device or simulator after a complete stop. Lower is better. It directly impacts developer velocity, iteration cadence, and the number of safe experiments you can run in a day.

Always consider variability across runs and remember that tooling state, caches, and signing can skew results. Compare like for like: same machine, same project state, same build mode.





Insights:

- **iOS**
  - **Flutter: (40.34 sec ± 0.17)**

    Slowest iOS loop but very consistent. Engine and asset steps dominate.
  - **React Native (Expo): (28.55 sec ± 2.74)**

    Mid-pack with higher variance. Metro bundling and signing steps add overhead.
  - **Swift (Native): (8.02 sec ± 1.05)**

    Fastest loop. Small variance suggests stable incremental builds.
- **Android**
  - **Flutter: (18.50 sec)**

    Faster than RN in this run.
  - **React Native (Expo): (19.70 sec)**

    Slightly slower than Flutter.
  - **Android Native (Kotlin): (11.52 sec ± 0.99)**

    Fastest loop on Android with tight variance. Good baseline for critical projects.

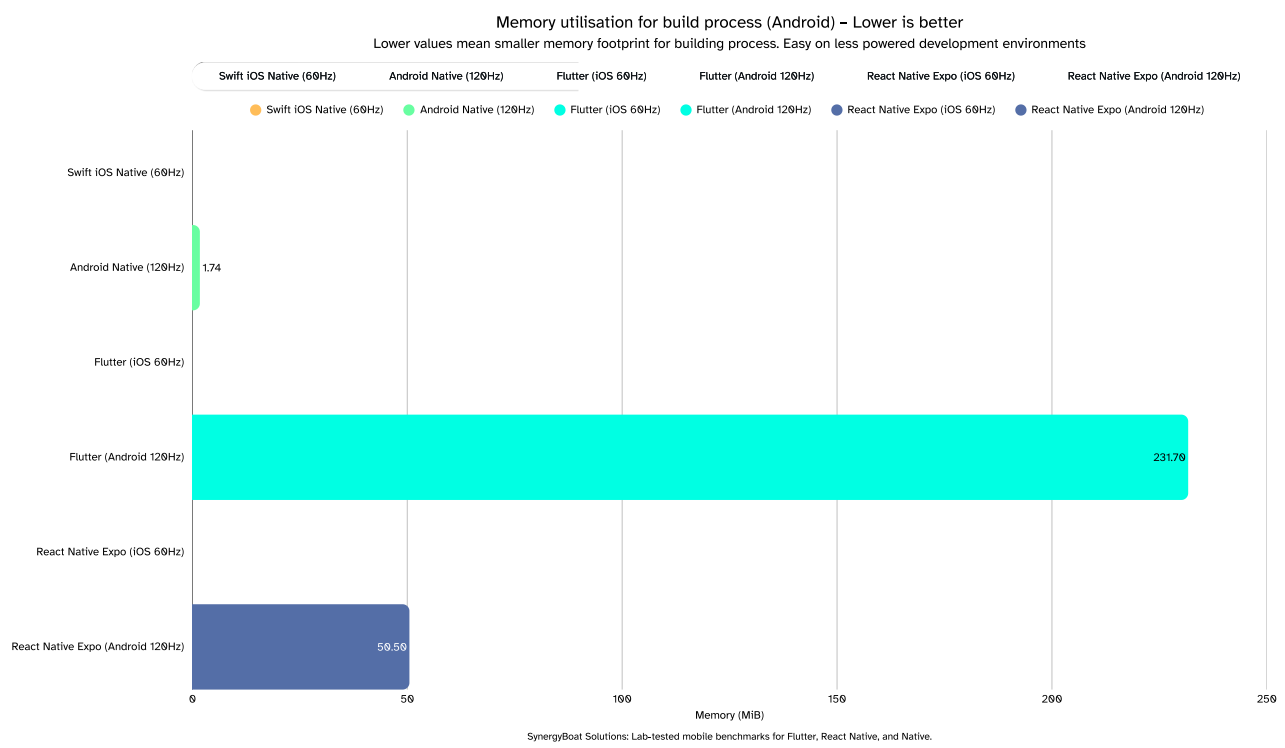
How to read this: Dev-loop time measures developer experience, not user experience. A 10 to 20 second gap translates into dozens of extra iterations per day.

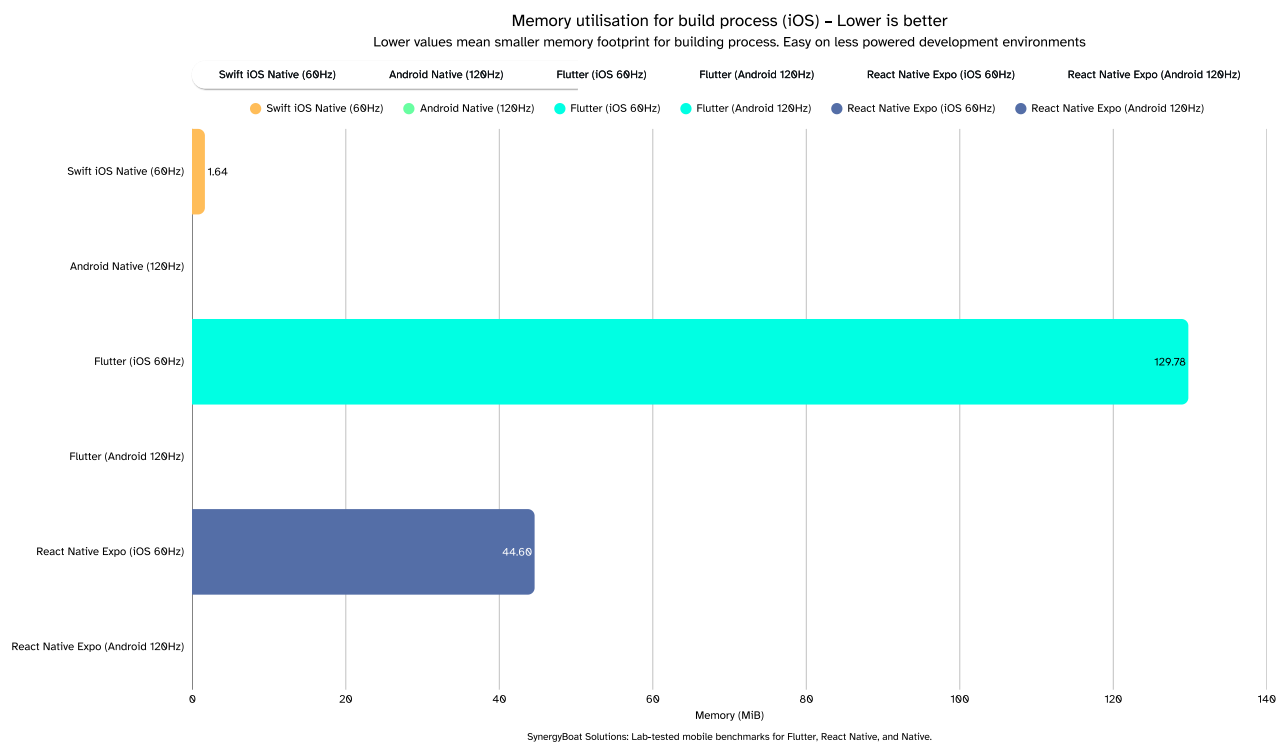
The strategic takeaway: Native delivers the quickest feedback on both platforms. React Native and Flutter are workable but require active hygiene to keep loops tight. If build speed is a critical constraint for your team, prioritize project modularization, cache hygiene, and minimizing heavy plugins or packages.

Memory Utilisation for build (inner dev-loop)

Memory Utilisation for build (inner dev-loop) is the peak memory your build and tooling stack consume during a build cycle after a complete stop. Lower is better. It determines how many simulators, emulators, IDEs, and background tasks your machine can juggle without slowing down. It is a solid health check for developer experience, CI concurrency, and machine sanity.

Always look at **StdDev** for stability across runs and remember this is **toolchain memory**, not app runtime memory. Read it alongside **Build Dev-loop Time**, since time and memory pressure often travel together.





Insights:

- **iOS**
  - **Flutter: (129.78 MiB ± 2.06)**

    Moderate to high footprint but very consistent. Engine and toolchain work dominate.
  - **React Native (Expo): (44.6 MiB)**

    Lighter than Flutter, heavier than pure native. Metro and Node processes add a steady baseline.
  - **Swift (Native): (1.64 MiB)**

    Tiny footprint. Incremental native builds place minimal pressure on RAM.

- **Android**
  - **Flutter: (231.7 MiB)**

    Largest here. Gradle + Dart toolchain and asset steps inflate peaks.
  - **React Native (Expo): (50.5 MiB)**

    Mid-range. Metro and Gradle contribute.
  - **Android Native (Kotlin): (1.74 MiB)**

    Very small footprint. Leaves headroom for parallel emulators or heavier IDE tasks.

How to read this: Dev-loop memory is about **developer velocity and capacity**, not user-facing performance. Pair it with **Build Dev-loop Time** to understand the true cost of an iteration. Rising peaks reduce how many devices and tools you can run side by side before the machine starts to stutter.

The strategic takeaway: **Native** imposes the smallest RAM footprint on both platforms, which scales well for large teams and dense CI. **Flutter** trades a heavier dev-loop memory cost for consistent behavior and strong runtime headroom. **React Native (Expo)** sits in the middle and benefits from proactive pruning of modules and careful Metro configuration. Keep this metric in the green if your team values fast parallel iteration.