AVALIAÇÃO DE DESEMPENHO ENTRE *FRAMEWORKS CROSS-PLATFORM* PARA APLICAÇÕES *MOBILE*

Tobias Rodrigues da Silva*
Silas Santiago Lopes Pereira**

RESUMO

O desenvolvimento de aplicações móveis com *frameworks cross-platform* tem se consolidado como alternativa para reduzir custos e acelerar a entrega de *software*, mas ainda levanta dúvidas quanto ao impacto dessas tecnologias no desempenho das aplicações. Este trabalho apresenta um estudo aplicado e quantitativo que compara o desempenho dos *frameworks Flutter* e *React Native* em dois estudos de caso equivalentes. No primeiro cenário, foi desenvolvido um aplicativo com carga computacional intensa, envolvendo a geração e ordenação de listas, a serialização JSON e cálculo de números primos. No segundo, implementou-se um aplicativo de captura de vídeo, de modo a avaliar o uso direto da câmera e a escrita de arquivos em tempo real. Os testes foram executados em um dispositivo físico *Android*, sob condições controladas, com medições realizadas pela ferramenta *Flashlight*, considerando métricas como uso de CPU, memória RAM, FPS, tempo em carga máxima, uso da *UI Thread* e tamanho do aplicativo. Os resultados indicam que o *Flutter* apresentou desempenho superior em ambos os cenários, com maior taxa de quadros por segundo, menor uso médio de CPU e consumo de memória significativamente menor no aplicativo multimídia, enquanto o *React Native* obteve vantagem no uso de *UI Thread* no primeiro cenário e no tamanho do pacote gerado. Esses estudos fornecem evidências empíricas que podem apoiar desenvolvedores na escolha do *framework cross-platform* mais adequado para aplicações que priorizam alta responsividade gráfica e eficiência no uso de recursos do dispositivo.

Palavras-chave: desenvolvimento móvel; *framework cross-platform*; *Flutter*; *React Native*; desempenho.

ABSTRACT

The development of mobile applications with cross-platform frameworks has become established as an alternative to reduce costs and accelerate software delivery, but it still raises questions about the impact of these technologies on application performance. This work presents an applied and

* Tobias Rodrigues da Silva é graduando em ciência da computação no Instituto Federal do Ceará (IFCE). Endereço eletrônico: tobias.rodrigues.silva.07@aluno.ifce.edu.br.
** Silas Santiago é professor do IFCE, doutor e mestre em ciências da computação na Universidade Estadual do Ceará (UECE) e bacharel em ciência da computação pela UECE. Endereço eletrônico: silas.santiago@ifce.edu.br.

quantitative study that compares the performance of the Flutter and React Native frameworks in two equivalent case studies. In the first scenario, an application with an intensive computation load was developed, involving the generation and sorting of lists, JSON serialization, and the calculation of prime numbers. In the second, a video capture application was implemented to evaluate the direct use of the camera and real-time file writing. The tests were executed on a physical Android device, under controlled conditions, with measurements collected using the Flashlight tool, considering metrics such as CPU usage, RAM consumption, FPS, time under maximum CPU load, UI thread usage, and application size. The results indicate that Flutter showed superior performance in both scenarios, with a higher frame rate, lower average CPU usage, and significantly lower memory consumption in the multimedia application, while React Native presented an advantage in UI thread usage in the first scenario and in the size of the generated package. These studies provide empirical evidence that can support developers in choosing the most suitable cross-platform framework for applications that prioritize high graphical responsiveness and efficient use of device resources.

Keywords: mobile development; framework cross-platform; Flutter; React Native; performance.

1 INTRODUÇÃO

O desenvolvimento de *software* para dispositivos móveis (*mobile*) tem passado por uma profunda transformação com o recente crescimento e consolidação de *frameworks cross-platform*. Esses *frameworks* vêm ganhando destaque por possibilitarem o desenvolvimento de aplicações para múltiplas plataformas a partir de uma única base de código, reduzindo custos, esforço de manutenção e tempo de entrega (SOUHA et al., 2024). No entanto, essa praticidade traz consigo uma série de desafios técnicos e decisórios, sobretudo no que se refere ao desempenho das aplicações e à escolha da melhor ferramenta para cada projeto.

Historicamente, o desenvolvimento de aplicativos móveis era realizado separadamente para sistemas operacionais móveis *Android* e iOS, utilizando tecnologias nativas específicas para cada plataforma (SOUHA et al., 2024). Esse modelo, embora eficiente em termos de desempenho, apresentava altos custos e grande complexidade de manutenção. Com o surgimento dos *frameworks cross-platform*, como *Flutter* e *React Native*, tornou-se possível criar aplicações eficientes para múltiplos sistemas operacionais a partir de uma única base de código, impulsionando mudanças significativas no panorama do desenvolvimento *mobile*(GOWRI et al., 2023).

A literatura científica ainda é limitada quando se trata de estudos comparativos aprofundados entre *frameworks cross-platform* e soluções nativas. Como apontado por Nawrocki et al. (2021), muitos estudos anteriores focaram em tecnologias ultrapassadas ou realizaram testes apenas com aplicações simples, sem considerar cenários reais de uso ou tarefas mais exigentes. Essa carência metodológica foi reforçada por Dorfer, Demetz e Huber (2020), ao mostrar que o uso de funcionalidades comuns, como acesso a câmera e chamadas a APIs externas, pode

gerar diferenças significativas no consumo de CPU, memória e bateria entre abordagens nativas e *cross-platform*.

Além da questão do desempenho, outro desafio recorrente no desenvolvimento *mobile* é a escolha do *framework* mais adequado para cada projeto. Essa decisão torna-se ainda mais complexa diante da variedade crescente de opções disponíveis, cada uma com características técnicas específicas, ecossistemas distintos e diferentes níveis de maturidade. Como destacam Souha et al. (2024), a seleção de um *framework* envolve múltiplos critérios: desde a linguagem de programação e preferências da equipe, até requisitos de desempenho, compatibilidade com IDEs[1] e suporte oferecido pelas comunidades.

Essa complexidade também aparece nos estudos de Wambua (2024), que analisaram mais de 150 mil perguntas no *Stack Overflow*[2] e identificaram que desenvolvedores *Flutter* frequentemente enfrentam dificuldades relacionadas a gerenciamento de estado, uso de *widgets*, navegação e integração com bibliotecas externas. Tais achados reforçam que, mesmo em *frameworks* modernos e amplamente documentados, ainda existem barreiras técnicas e uma curva de aprendizagem significativa, o que impacta diretamente a experiência de desenvolvimento e pode influenciar a qualidade final do *software*.

Diante dos desafios apresentados, torna-se evidente a necessidade de estudos comparativos mais abrangentes e fundamentados, capazes de oferecer aos desenvolvedores ferramentas objetivas para a escolha do *framework cross-platform* mais adequado. Essa decisão não deve considerar apenas métricas técnicas isoladas, mas também aspectos práticos da experiência de desenvolvimento e o comportamento real das aplicações em cenários comuns de uso.

A escolha do *Flutter* e do *React Native* como foco deste trabalho se justifica por estes serem, conforme apontado por Nawrocki et al. (2021) e Souha et al. (2024), os *frameworks* mais populares e amplamente utilizados no cenário atual de desenvolvimento *mobile*. Além disso, segundo Karami et al. (2023), ambos oferecem vantagens relevantes em termos de produtividade e facilidade de manutenção de código, sendo frequentemente preferidos por equipes de desenvolvimento que buscam agilidade na codificação.

Neste contexto, o presente trabalho propõe uma avaliação sistemática de desempenho entre os *frameworks Flutter* e *React Native*, baseada na construção de dois estudos de caso com funcionalidades reais e alinhadas a aplicações modernas. O primeiro cenário envolve um aplicativo projetado para executar tarefas de alta carga computacional, permitindo observar a eficiência interna de processamento de cada tecnologia. O segundo cenário simula situações práticas de uso de dispositivos móveis, como captura de vídeo e acesso a recursos de *hardware*, avaliando o comportamento dos *frameworks* em interações que dependem de APIs[3] nativas. A análise utiliza métricas quantitativas amplamente empregadas na literatura (DORFER; DEMETZ; HUBER, 2020; NAWROCKI et al., 2021), como uso de CPU, memória RAM, FPS, uso da *UI Thread*, tempo em carga máxima e tamanho final do aplicativo, de modo a possibilitar

[1] *Integrated Development Environment*
[2] <https://stackoverflow.com/questions>
[3] *Application Programming Interfaces*

uma comparação objetiva entre os ambientes de execução. As medições foram obtidas a partir da execução controlada de aplicações desenvolvidas especificamente para esta pesquisa, sob condições idênticas de *hardware* e configuração do ambiente. Com isso, busca-se garantir que as diferenças observadas reflitam, de fato, características inerentes aos *frameworks*, e não variáveis externas ao experimento.

Este artigo está estruturado em cinco seções principais. A Seção 2 aborda a fundamentação teórica, destacando os conceitos essenciais e características técnicas dos *frameworks Flutter* e *React Native*. A Seção 3 apresenta os trabalhos relacionados, com uma análise crítica de estudos prévios sobre desempenho, qualidade e adoção de *frameworks cross-platform*. Na Seção 4, mostra a metodologia adotada na avaliação dos *framewoks*. A Seção 5 trata dos resultados. Por fim, a Seção 6 apresenta a conclusão e as considerações finais, com reflexões sobre as contribuições do estudo e sugestões para trabalhos futuros.

2 FUNDAMENTAÇÃO TEÓRICA

2.1 Desenvolvimento de Aplicativos Móveis

O desenvolvimento de aplicações móveis tem acompanhado a rápida evolução de *hardware* e sistemas operacionais em *smartphones*. Inicialmente, a construção de soluções era conduzida de forma nativa, com linguagens específicas para cada plataforma, como *Java* e *objective-C* nos sistemas operacionais *Android*[4] e iOS[5], respectivamente. Tais *softwares* eram restritos às funcionalidades básicas, como chamadas telefônicas e mensagens de texto. Ao longo do tempo, novas linguagens surgiram para facilitar esse processo, como *Kotlin*, oficialmente adotada pelo Google em 2017, e *Swift*, lançada pela Apple em 2014 para substituir o *Objective-C* (SOUHA et al., 2024).

Com essa evolução, os dispositivos móveis modernos transformaram-se em plataformas de acesso a serviços complexos, redes sociais, jogos, ferramentas de produtividade e comércio eletrônico. Esse crescimento impulsionou a necessidade por aplicações móveis mais sofisticadas, eficientes e capazes de atender a uma demanda de usuários cada vez mais exigente(SOUHA et al., 2024). A busca por ciclos de desenvolvimento mais curtos e com menor custo impulsiona o surgimento de *frameworks* chamados de *cross-platform*. Esses *frameworks* possibilitam a construção de aplicativos para diferentes sistemas operacionais a partir de uma única base de código. Segundo Souha et al. (2024), esse movimento representou uma alteração significativa no desenvolvimento *mobile*, de modo que contribui para a padronização de interfaces e o reaproveitamento de código.

O mercado *mobile* tornou-se um dos mais estratégicos da indústria de *software*. A ubiquidade dos *smartphones* e a dependência crescente dos usuários em relação aos aplicativos móveis transformaram esses sistemas em componentes fundamentais para comunicação, negócios e entretenimento. Como evidenciado por Gowri et al. (2023), o uso de *smartphones*

[4] <https://www.android.com/>
[5] <https://developer.apple.com/ios/>

está diretamente ligado à ascensão de aplicativos móveis em setores como saúde, comércio eletrônico, mobilidade urbana e educação. Essa realidade intensificou a necessidade de soluções que acelerem o desenvolvimento e reduzam os custos sem comprometer a experiência do usuário, contexto no qual os *frameworks cross-platform* ganham destaque.

Atualmente, existem diferentes modelos de desenvolvimento para aplicações móveis, que variam quanto à forma de execução e integração com os sistemas operacionais. O modelo nativo é aquele em que os aplicativos são desenvolvidos com linguagens e ferramentas específicas para cada plataforma, como *Java* ou *Kotlin* para *Android* (ANDROID, 2025), e *Swift* ou *Objective-C* para iOS (APPLE, 2025). Essa abordagem proporciona alto desempenho e acesso completo às APIs e componentes do sistema, sendo indicada para aplicações que exigem uso intensivo de recursos do dispositivo. No entanto, seu principal desafio é o custo e o tempo envolvidos, já que cada versão da aplicação precisa ser desenvolvida e mantida separadamente (NAWROCKI et al., 2021).

Em contrapartida, os *Web Apps* [6] são aplicações acessadas diretamente por navegadores móveis, construídas com tecnologias como HTML(HTML5, 2025), CSS(CSS3, 2025) e *JavaScript*(JS, 2025). Sua principal vantagem está na portabilidade e facilidade de manutenção, já que não requerem instalação nem dependem da loja de aplicativos. Contudo, possuem acesso restrito aos recursos do sistema, como sensores e APIs nativas, o que limita sua aplicação em projetos mais complexos (DORFER; DEMETZ; HUBER, 2020).

A abordagem híbrida, representada por *frameworks* como o *Apache Cordova*(CORDOVA, 2025), *Xamarin*(XAMARIN, 2025), *Ionic*(IONIC, 2025), entre outros, que busca um meio-termo ao permitir o desenvolvimento de aplicações utilizando tecnologias *web* empacotadas em um contêiner nativo. Dessa forma, é possível executar o mesmo código em diferentes plataformas com menor esforço de adaptação. Apesar disso, estudos como o de Souha et al. (2024) apontam que, em termos de desempenho, essa abordagem apresenta limitações, especialmente em aplicações que fazem uso mais intensivo de animações, sensores ou acesso em tempo real a recursos do dispositivo(GOWRI et al., 2023).

Já o modelo *cross-compiled*, também conhecido como *cross-platform*, vem ganhando grande popularidade com ferramentas como o *Flutter*(FLUTTER, 2025), desenvolvido pelo Google, e o *React Native*(NATIVE, 2025), mantido pela Meta. Ambas as tecnologias permitem o desenvolvimento de aplicações utilizando uma única base de código, em *Dart* no caso do *Flutter* e *JavaScript* no *React Native*, que é posteriormente compilada para código nativo. Segundo Karami et al. (2023), essa abordagem equilibra desempenho e produtividade, sendo eficaz para projetos que exigem suporte a múltiplas plataformas com custo reduzido e bom tempo de entrega.

Cada modelo de desenvolvimento apresenta benefícios e limitações que devem ser considerados de acordo com os objetivos e restrições de cada projeto. A Tabela 2 apresentada compara as principais abordagens de desenvolvimento de aplicações móveis nativo, *web app*, híbrido e *cross-platform*, com base em critérios técnicos e de produtividade. O desenvolvimento

[6] <https://developer.mozilla.org/pt-BR/docs/Web/Progressive_web_apps>

Tabela 1 – Comparativo dos modelos de desenvolvimento *mobile*

Critério	Nativo	Web App	Híbrido	Cross-Platform
Desempenho	Alto	Baixo	Moderado	Alto (com variações)
Acesso a APIs Nativas	Completo	Limitado	Parcial	Completo (com variações)
Curva de Aprendizado	Alta	Baixa	Moderada	Moderada
Reusabilidade de Código	Baixa	Alta	Alta	Alta
Custo de Desenvolvimento	Alto	Baixo	Médio	Médio
Manutenção	Complexa	Simples	Moderada	Moderada
Produtividade da Equipe	Baixa	Alta	Alta	Alta
Experiência do Usuário	Nativa/Fluida	Limitada	Razoável	Próxima do nativo

Fonte: Autoral.

nativo se destaca pelo alto desempenho e acesso completo às APIs do sistema, mas exige maior esforço de aprendizado, manutenção mais complexa e maior custo de desenvolvimento. Por outro lado, os *web apps* apresentam baixo desempenho e acesso limitado aos recursos do dispositivo, embora sejam fáceis de manter e baratos de implementar. As soluções híbridas oferecem reuso elevado de código e custo moderado, mas enfrentam limitações de performance e integração nativa. Já os *frameworks cross-platform*, como *Flutter* e *React Native*, alcançam desempenho geralmente alto, boa reutilização de código e acesso quase completo às APIs nativas, equilibrando produtividade e qualidade com curva de aprendizado e manutenção moderadas. Esses dados reforçam a importância de avaliar o contexto do projeto antes da escolha da abordagem mais adequada.

A literatura analisada reforça que, embora o desenvolvimento nativo gere ganhos de desempenho, soluções *cross-platform* modernas como *Flutter* e *React Native* têm evoluído significativamente. Estudos como os de Souha et al. (2024) e Karami et al. (2023) apontam que essas ferramentas oferecem *tradeoff* entre escalabilidade e produtividade, de modo que estas tecnologias são frequentemente adotadas por equipes que priorizam prazos curtos de entrega, manutenção centralizada e suporte a múltiplas plataformas.

2.2 *Framewoks Cross-Platform*

Os *frameworks mobile cross-platform* surgiram como resposta à necessidade crescente de construir aplicações móveis de forma mais rápida, eficiente e econômica. O modelo tradicional, baseado no desenvolvimento nativo para cada plataforma, embora oferecesse máximo desempenho, impunha altos custos de desenvolvimento e manutenção, além de exigir equipes com conhecimentos distintos para *Android* e iOS. Nesse cenário, os *frameworks cross-platform* propuseram uma alternativa mais prática: escrever uma única base de código capaz de gerar

aplicativos para diferentes sistemas operacionais móveis (NAWROCKI et al., 2021).

Segundo Souha et al. (2024), os primeiros *frameworks* desse tipo enfrentaram sérias limitações, especialmente relacionadas à performance e à integração com funcionalidades nativas, como sensores e APIs específicas do sistema. *Frameworks* como *PhoneGap* e *Cordova*, por exemplo, baseavam-se em contêineres *web* e não ofereciam uma experiência fluida ao usuário final. Com o tempo, ferramentas mais sofisticadas, como *Flutter* e *React Native*, passaram a ser desenvolvidas com foco em desempenho, usabilidade e experiência do desenvolvedor, resolvendo várias dessas limitações iniciais.

2.2.1 Flutter

O *Flutter* é um *framework open-source* desenvolvido pelo Google, lançado oficialmente em 2017, com o objetivo de proporcionar alto desempenho e uma experiência nativa em múltiplas plataformas a partir de um único código-fonte. Utiliza a linguagem *Dart*, também criada pelo Google, que é compilada para código nativo e executada diretamente no dispositivo, sem a necessidade de uma ponte com elementos da plataforma.

Um dos principais diferenciais do *Flutter* está em sua arquitetura baseada no *Skia*[7], um motor gráfico que renderiza todos os elementos da interface diretamente na tela do dispositivo, garantindo controle total sobre o layout e aparência da aplicação. Essa abordagem permite interfaces customizáveis, além de eliminar discrepâncias visuais entre plataformas. O recurso de *hot reload*, que atualiza a interface do aplicativo instantaneamente após alterações no código, também é amplamente destacado na literatura como uma das razões da popularidade do *Flutter* entre os desenvolvedores (SOUHA et al., 2024; KARAMI et al., 2023). Além disso, o *Flutter* conta com uma documentação extensa, suporte ativo da comunidade e forte integração com os serviços e ferramentas do Google, o que contribui para sua adoção crescente em aplicações comerciais e acadêmicas (GOWRI et al., 2023).

2.2.2 React Native

Criado pelo Facebook (atualmente Meta) em 2015, o *React Native* também busca permitir o desenvolvimento de aplicações multiplataforma com uma única base de código. Sua principal característica é o uso da linguagem *JavaScript* e da biblioteca *React*, já bastante conhecida por desenvolvedores *front-end*.

Diferentemente do *Flutter*, o *React Native* utiliza uma arquitetura baseada em ponte (*bridge*), que permite a comunicação entre os componentes J*avaScript* e os componentes nativos do sistema. Essa arquitetura garante a utilização de componentes visuais nativos, o que favorece a consistência visual com o sistema operacional. No entanto, a comunicação entre a camada *JavaScript* e os elementos nativos ocorre por meio de uma ponte assíncrona, o que pode introduzir gargalos de desempenho, especialmente em aplicações que exigem atualizações frequentes da

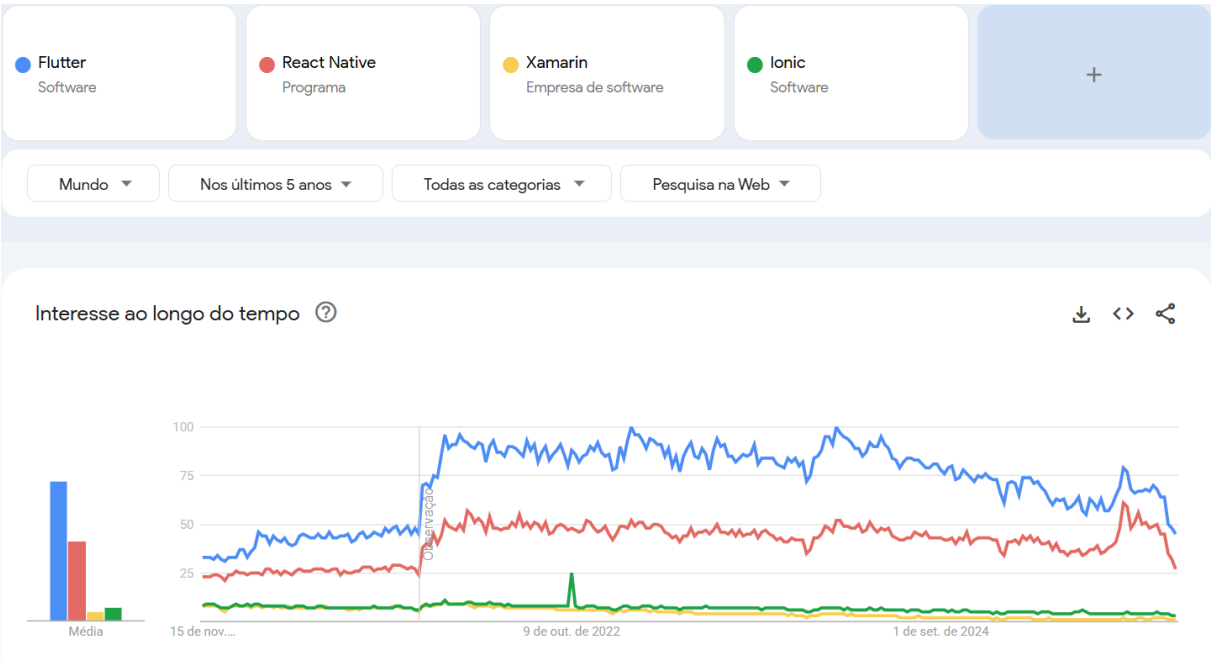
[7] <https://skia.org/>

interface ou uso intensivo de recursos nativos (NAWROCKI et al., 2021; DORFER; DEMETZ; HUBER, 2020).

Segundo Karami et al. (2023), o *React Native* apresenta como pontos fortes a grande variedade de bibliotecas disponíveis, a integração fluida com projetos web baseados em *React* e a ampla base de usuários e contribuintes ativos. No entanto, a dependência da ponte entre camadas pode dificultar a manutenção de desempenho em aplicações mais exigentes, como também observado por Dorfer, Demetz e Huber (2020).

2.2.3 Flutter x React Native

Figura 1 – Procura dos últimos 5 anos dos *frameworks cross-platform*



Fonte: Google Trends

Flutter e *React Native* são hoje os *frameworks cross-platform* mais populares e utilizados no desenvolvimento mobile, como apontado por Souha et al. (2024) e corroborado pelos dados de repositórios públicos como GitHub e análises de tendências de busca, como o *Google Trends* como mostrado na Figura 1. Ambos os *frameworks* possuem vantagens e desvantagens que precisam ser avaliadas de acordo com as necessidades específicas de cada projeto.

Tabela 2 – Comparativo do *Flutter* e *React Native*

Critério	Flutter	React Native
Linguagem	Dart	JavaScript
Renderização	Skia Engine	Via ponte (bridge)
Hot Reload	Sim	Sim
Integração com nativo	Boas extensões e plugins próprios	Usa bibliotecas de terceiros e ponte nativa
Curva de aprendizado	Moderada	Moderada
Comunidade	Crescendo rapidamente	Estável e consolidada
Performance	Alta	Alta, com possíveis gargalos
Manutenção de código	Centralizada	Centralizada

Fonte: Autoral.

Como demonstrado por Nawrocki et al. (2021) e reforçado por Karami et al. (2023), o *Flutter* tende a oferecer desempenho superior em tarefas gráficas e controle visual da interface, enquanto o *React Native* proporciona maior familiaridade e facilidade de integração com projetos web existentes. A escolha entre um e outro depende diretamente do tipo de aplicação, dos requisitos de desempenho e do perfil da equipe de desenvolvimento.

A Tabela 3 apresenta uma comparação direta entre *Flutter* e *React Native*, com base em critérios como linguagem, arquitetura de renderização, integração com recursos nativos, performance e manutenção.

O *Flutter* utiliza *Dart* e o motor gráfico *Skia*, o que permite renderizar toda a interface de forma independente da plataforma, enquanto o *React Native* utiliza *JavaScript* e se comunica com os componentes nativos por meio de uma ponte (bridge), o que pode gerar gargalos em aplicações mais exigentes. Ambos oferecem *hot reload* e possuem manutenção de código centralizada, mas se diferenciam na integração nativa e curva de aprendizado. A comunidade do *Flutter* cresce rapidamente, enquanto a do *React Native* já é consolidada, o que também influencia na escolha conforme o contexto do projeto.

3 TRABALHOS RELACIONADOS

Nesta seção, são apresentados estudos que tratam da avaliação de *frameworks cross-platform* no desenvolvimento de aplicações móveis, com foco em desempenho, usabilidade e experiência do desenvolvedor. O objetivo é compreender como a literatura tem abordado o tema, identificar metodologias utilizadas, métricas analisadas e tecnologias comparadas. Esses trabalhos fornecem uma base importante para contextualizar a proposta desta pesquisa, além de permitir a identificação de lacunas que justificam a realização desta seção.

O trabalho de Nawrocki et al. (2021) propõem uma comparação entre aplicações desenvolvidas com *Flutter, React Native, Xamarin* e soluções nativas (*Android*/iOS), com foco em métricas técnicas de desempenho. Foram desenvolvidos dois tipos de aplicativos para testes: um aplicativo simples com funcionalidades básicas e outro mais complexo, com múltiplas páginas e tarefas em segundo plano. A análise considerou diversas métricas, como tempo de inicialização, tamanho da aplicação, uso de memória RAM, uso de CPU, facilidade de desenvolvimento e popularidade. Os resultados apontam que, embora as aplicações nativas apresentem melhor desempenho em termos de consumo de recursos e tempo de inicialização, o *Flutter* se destaca pela experiência de desenvolvimento, documentação e produtividade geral. *React Native* apresentou problemas com pacotes de terceiros e tempo de *build*, enquanto *Xamarin* teve o pior desempenho global. O artigo conclui que, mesmo com os avanços dos *frameworks cross-platform*, ainda há lacunas importantes em termos de performance e eficiência, especialmente em tarefas mais exigentes.

No estudo de Souha et al. (2024), os autores propõem uma análise comparativa detalhada entre diversos *frameworks* de desenvolvimento *mobile*, com foco em *Flutter, React Native, Xamarin* e *SwiftUI*. O diferencial do trabalho está na proposta de uma ferramenta de recomendação, que orienta desenvolvedores na escolha do *framework* mais adequado com base em critérios técnicos, preferências pessoais e contexto do projeto. A análise considera mais de 20 atributos comparativos, incluindo performance, reusabilidade de código, compatibilidade com IDEs, suporte a animações, facilidade de depuração, tempo de *build* e popularidade. Além disso, o estudo oferece uma estrutura de classificação dos *frameworks* (nativo, híbrido, interpretado e *cross-compiled*), o que ajuda a compreender as diferenças técnicas que impactam o desempenho e a experiência de desenvolvimento. A contribuição central do artigo é ampliar a visão do desenvolvedor, levando em conta não apenas métricas de performance, mas também critérios subjetivos e de produtividade, que muitas vezes influenciam a escolha da tecnologia.

O trabalho de Dorfer, Demetz e Huber (2020) se destaca por apresentar uma análise prática da abordagem nativo (*Android*) com o *React Native* no uso de recursos do dispositivo, com foco específico em consumo de CPU, memória RAM e bateria. Para isso, os autores desenvolveram um aplicativo de busca por restaurantes com geolocalização, consumo de API externa e exibição de listas e mapas, replicado em versões nativa (*Android*) e em *React Native*. Os resultados mostraram que o *React Native* apresentou, em média, um consumo de 4x mais CPU, uso de memória até 200MB superior, e consumo de bateria entre 6% e 8% maior do que o aplicativo nativo. O estudo evidencia que, embora o *React Native* proporcione maior produtividade e reuso de código, essas vantagens vêm acompanhadas de custos significativos de desempenho, especialmente em aplicações que utilizam sensores e recursos em tempo real.

O estudo de Gowri et al. (2023) apresenta uma análise dos principais *frameworks cross-platform* utilizados no desenvolvimento de aplicações móveis, com o objetivo de auxiliar desenvolvedores na escolha da ferramenta mais adequada para seus projetos. A pesquisa explora os aspectos técnicos e funcionais de plataformas como *Flutter, React Native, Xamarin* e *Ionic*, avaliando critérios como produtividade, curva de aprendizado, integração com APIs nativas,

manutenção de código e compatibilidade com múltiplos dispositivos. Um dos destaques do trabalho é a ênfase na relação entre desempenho e facilidade de desenvolvimento, apontando que, embora os *frameworks cross-platform* ofereçam vantagens em termos de velocidade de entrega e reuso de código, ainda enfrentam desafios relacionados ao uso intensivo de recursos e à estabilidade das aplicações em cenários complexos. O artigo também aborda a popularidade crescente de *Flutter* e *React Native*, reforçando sua adoção no mercado como alternativas viáveis ao desenvolvimento nativo.

Tabela 3 – Comparativo dos trabalhos relacionados

Autor	**Métricas Avaliadas**	**Tecnologias Comparadas**	**Ferramentas/ Metodologia**
Nawrocki et al. (2021)	CPU, memória RAM, tempo de inicialização, tamanho do aplicativo	Flutter, React Native, Xamarin, Nativo	Aplicativos simples e complexos, testes com ferramentas nativas de perfilamento
Souha et al. (2024)	Performance, facilidade de uso, reusabilidade de código	Flutter, React Native, Xamarin, SwiftUI	Análise comparativa baseadaem critérios técnicos + ferramenta de recomendação proposta
Dorfer et al. (2020)	CPU, memória RAM, consumo de bateria	React Native, Nativo	Aplicativo real com sensores e mapas, medição via Android Profiler e coleta em tempo real
Gowri et al. (2023)	Produtividade, curva de aprendizado, experiência do desenvolvedor, integração com APIs	Flutter, React Native, Xamarin, Ionic	Estudo técnico com foco em integração, usabilidade e análise de características funcionais
Trabalho Atual	Uso de CPU, memória RAM, FPS, tempo em carga máxima, uso da UI Thread e tamanho do aplicativo	Flutter e React Native	Desenvolvimento de 2 estudos de caso; ferramenta: Flashlight

Fonte: Autoral.

A Tabela 1 apresenta uma visão consolidada dos principais estudos utilizados como referência nesta pesquisa, destacando as métricas avaliadas, os *frameworks* comparados e as metodologias empregadas. A análise dos trabalhos relacionados evidencia uma lacuna importante na literatura: embora existam estudos focados em desempenho (como Nawrocki et al. (2021) e Dorfer, Demetz e Huber (2020)) e em experiência de desenvolvimento (Souha et al. (2024)), poucos trabalhos conseguem integrar de forma completa ambos os aspectos. Outra limitação observada é que muitos estudos utilizam aplicações simples ou cenários restritos de avaliação, o que dificulta a generalização dos resultados para aplicações mais complexas e próximas de situações reais.

O estudo de Nawrocki et al. (2021) concentrou-se na comparação de desempenho técnico básico entre diferentes *frameworks*, por meio de testes com aplicações simples e complexas, utilizando ferramentas nativas de medição. Apesar de oferecer uma boa base comparativa, não explora funcionalidades mais exigentes, como uso intensivo de sensores. Em contrapartida, o trabalho de Dorfer, Demetz e Huber (2020) avança nesse aspecto ao avaliar o impacto do uso de sensores e mapas, mostrando que o *React Native* apresentou consumo elevado de CPU, RAM e bateria em relação ao desenvolvimento nativo.

Já o estudo de Souha et al. (2024) amplia a análise ao propor uma ferramenta de recomendação baseada em mais de vinte critérios técnicos e subjetivos, considerando não apenas performance, mas também usabilidade e contexto de desenvolvimento. Complementando essas abordagens, Gowri et al. (2023) realiza uma análise centrada na produtividade e experiência do desenvolvedor, evidenciando o papel da curva de aprendizado, integração com APIs e estabilidade do sistema na escolha de um *framework*. O estudo atual, por sua vez, busca integrar os principais pontos dessas abordagens anteriores, utilizando uma metodologia que inclui o desenvolvimento de dois estudos de caso, aplicação de múltiplas métricas, e uso de ferramentas profissionais como *Flashlight* para mensuração dos resultados em cenários reais.

4 METODOLOGIA

Este é um estudo aplicado, quantitativo e de abordagem experimental. Trata-se de uma investigação aplicada porque busca resolver um problema prático de tomada de decisão quanto à escolha do *framework cross-platform* mais adequado para o desenvolvimento *mobile*, de modo a oferecer dados empíricos que auxiliem profissionais e equipes de *software* na escolha da tecnologia apropriada. Do ponto de vista metodológico, adota-se uma abordagem quantitativa, uma vez que o foco está na coleta, mensuração e análise de dados objetivos relacionados ao desempenho das aplicações. A escolha por essa abordagem experimental justifica-se pela escassez de estudos que ofereçam uma comparação sistemática entre *frameworks cross-platform*. Essa lacuna é destacada por Nawrocki et al. (2021), Dorfer, Demetz e Huber (2020), Karami et al. (2023), que reforçam a necessidade de experimentos práticos e mensuráveis para compreender o comportamento dessas tecnologias.

4.1 Aplicações

Para conduzir a investigação, foram desenvolvidos dois estudos de caso independentes, cada um implementado integralmente nas duas tecnologias analisadas: *Flutter* e *React Native*. A estratégia metodológica adotada consiste em simular cenários representativos do desenvolvimento *mobile* moderno, selecionando aplicações que pressionam diferentes subsistemas internos do dispositivo.

4.1.1 Estudo de caso 1

O primeiro aplicativo foi desenvolvido com o objetivo de avaliar a capacidade de processamento interno e manipulação de dados em cada *framework*, simulando um cenário de uso intensivo da CPU e da manipulação de dados em memória. Durante 30 segundos de execução contínua, o sistema realiza uma sequência de operações que envolve a criação e a manipulação de estruturas de dados, cálculos matemáticos e a ordenação de grandes conjuntos numéricos. A aplicação inicia gerando listas extensas, compostas por 20 mil objetos, que são convertidas para o formato JSON e imediatamente decodificadas. Esse procedimento exige uma quantidade significativa de operações de leitura, escrita e alocação de memória, permitindo avaliar o desempenho da linguagem e do motor de execução na serialização e desserialização de dados, processos fundamentais em aplicações que fazem comunicação constante com APIs, bancos de dados locais ou serviços externos.

Na etapa seguinte, o sistema gera vetores contendo dezenas de milhares de números inteiros aleatórios e realiza operações repetidas de ordenação. Essa atividade simula cenários reais de alto custo computacional, presentes em algoritmos de classificação, manipulação estatística e processamento de dados em larga escala. Por fim, o aplicativo executa uma rotina matemática contínua para identificar números primos dentro de um intervalo pré-determinado. Essa tarefa demanda uma grande quantidade de operações aritméticas sequenciais, permitindo avaliar a eficiência do *framework* no tratamento de *loops* e cálculos de complexidade elevada.

4.1.2 Estudo de caso 2

O segundo aplicativo foi projetado para avaliar o comportamento das aplicações desenvolvidas em *Flutter* e *React Native* em operações que exigem integração direta com o *hardware* do dispositivo, sobretudo a câmera e os mecanismos internos de captura e armazenamento de vídeo. Esse tipo de aplicação representa um cenário de uso extremamente comum em soluções modernas, como redes sociais, plataformas de produção audiovisual, sistemas de monitoramento e aplicativos que dependem de captura contínua de imagem e áudio.

Durante 30 segundos, o sistema ativa a câmera traseira do dispositivo, realiza a gravação do vídeo em tempo real, processa o fluxo multimídia de forma contínua e, ao final, salva automaticamente o arquivo gerado no armazenamento interno temporário. Esse estudo não avalia diretamente a capacidade de processamento, mas sim o desempenho da aplicação desenvolvida em cada *framework* ao interagir com recursos nativos, verificando a eficiência da ponte entre o código multiplataforma e os serviços internos do sistema operacional. Este experimento possibilita observar como *Flutter* e *React Native* se comportam ao lidar com tarefas multimídia, monitorar o uso da CPU e da memória durante a captura, verificar a estabilidade da taxa de quadros (FPS) e avaliar a eficiência na escrita do arquivo ao final do processo.

A combinação dos dois estudos de caso permite uma análise abrangente dos *frameworks*, contemplando tanto tarefas de cálculo intensivo quanto operações que dependem de integração com o *hardware*. Com isso, é possível comparar *Flutter* e *React Native* em cenários comple-

mentares, obtendo uma visão equilibrada e tecnicamente fundamentada do desempenho de cada tecnologia.

4.2 Métricas para Avaliação de Desempenho

A avaliação de desempenho em aplicações móveis é uma etapa essencial no ciclo de desenvolvimento, especialmente em contextos onde a experiência do usuário, a eficiência energética e o uso racional dos recursos do dispositivo são fatores críticos. Segundo Gowri et al. (2023), pequenas variações em tempo de resposta, uso de CPU ou consumo de memória podem impactar significativamente na percepção de qualidade e na retenção do usuário final.

Os artigos analisados (NAWROCKI et al., 2021),(DORFER; DEMETZ; HUBER, 2020), (KARAMI et al., 2023), (SOUHA et al., 2024) demonstram que não existe um padrão único para avaliação de desempenho em aplicações móveis, mas sim um conjunto de métricas quantitativas que, combinadas, oferecem uma visão mais precisa do comportamento das aplicações sob diferentes condições de uso. Neste trabalho, as métricas foram obtidas por meio da plataforma Flashlight (2025), que permite o monitoramento e a coleta automatizada de dados de desempenho em tempo real. As principais métricas selecionadas são apresentadas a seguir.

***Frames* por Segundo (FPS)**: A taxa de FPS indica a fluidez visual da aplicação, representando o número de quadros renderizados a cada segundo. Um FPS mais alto reflete uma experiência de uso mais suave e responsiva, especialmente em aplicações que envolvem renderização contínua (NAWROCKI et al., 2021).

Uso da CPU: O uso da CPU representa a porcentagem de ciclos de processamento utilizados pela aplicação durante a execução. Essa métrica é essencial para avaliar a eficiência computacional dos *frameworks*, indicando se há sobrecarga em operações de cálculo, renderização ou I/O (SOUHA et al., 2024).

Uso da *UI Thread*: O uso da *UI Thread* representa o percentual de utilização de CPU da *thread* principal responsável pela renderização da interface e pelo tratamento dos eventos de entrada do usuário. Essa métrica permite avaliar o quão sobrecarregada está a camada de apresentação.

Uso de Memória RAM: O consumo de RAM indica a quantidade de memória alocada pela aplicação durante o período de execução. Segundo Karami et al. (2023), o uso excessivo de memória pode reduzir a estabilidade do sistema, aumentar o risco de encerramento inesperado e afetar o tempo de resposta da aplicação.

Tempo sob Carga Máxima *(High CPU Time)*: Essa métrica expressa o intervalo de tempo em que a aplicação opera sob carga de CPU elevada, geralmente acima de 90% de utilização. Ela é fundamental para entender a estabilidade sob estresse, pois um tempo prolongado em carga máxima pode indicar gargalos de processamento ou má otimização de *threads*.

Tamanho final da aplicação: O tamanho final do aplicativo refere-se ao espaço ocupado pelo pacote gerado após a *build* (APK[8]). Essa métrica afeta diretamente o tempo de download e

[8] *Android Package Kit*

o espaço necessário no dispositivo do usuário, sendo especialmente importante em contextos com conectividade ou armazenamento limitados. Este valor é obtido diretamente a partir do tamanho do arquivo final gerado pelo processo de compilação nos ambientes de *build* dos próprios *frameworks*.

4.3 Ferramentas e Ambientes

Para garantir precisão e confiabilidade nos resultados obtidos, todos os testes de desempenho foram realizados em ambiente controlado, utilizando a ferramenta Flashlight (2025), uma solução de código aberto voltada à medição de métricas de performance em aplicações móveis.

Os experimentos foram executados em um mesmo dispositivo físico *Android* (Modelo: *Galaxy Tab S6 Lite*, armazenamento interno: 128G, memória RAM: 4G, velocidade do processador: 2.3GHz, 1.8GHz), em vez de emuladores, visto que apenas o *hardware* real reproduz fielmente as condições de execução e consumo de recursos, especialmente no uso de CPU, memória e bateria. Dessa forma, foi possível assegurar que o único fator variável entre os testes fosse o *framework* utilizado (*Flutter* ou *React Native*).

Os aplicativos foram desenvolvidos nos ambientes recomendados por cada tecnologia, mantendo fidelidade às melhores práticas da literatura e às diretrizes oficiais de cada *framework*:

- *Flutter*: foi utilizado o SDK oficial, com o Studio (2025) como IDE principal para edição e *Dart* como linguagem de programação.

- *React Native*: as aplicações equivalentes foram desenvolvidas com *JavaScript/TypeScript*, utilizando o *Expo* como ambiente de execução, empacotamento e *build*. O desenvolvimento foi realizado no Code (2025), adotado como IDE principal.

Ambos foram testados sob as seguintes condições padronizadas: modo avião ativado, brilho e volume fixos, nenhum aplicativo em segundo plano, testes repetidos três vezes por cenário, com cálculo de média aritmética dos resultados. Esses cuidados seguem recomendações metodológicas observadas em estudos de referência, como Nawrocki et al. (2021), Dorfer, Demetz e Huber (2020), Souha et al. (2024), que destacam a importância de controlar variáveis externas durante a coleta de métricas.

A ferramenta *Flashlight* desempenhou papel central neste estudo, sendo responsável por toda a coleta de métricas e pela visualização dos dados. De acordo com sua documentação oficial (FLASHLIGHT, 2025), trata-se de um sistema multiplataforma voltado especificamente para *benchmarking* de aplicações móveis, capaz de monitorar em tempo real indicadores como uso de CPU, consumo de memória RAM, taxa de quadros por segundo (FPS), energia e tempo de execução. A ferramenta opera via linha de comando e possui capacidade de exportar relatórios completos em formato JSON, facilitando análises comparativas e replicação de resultados.

Sua principal vantagem em relação a ferramentas tradicionais como *Android Profiler* e *Firebase Performance Monitoring*, é a capacidade de oferecer um ambiente unificado de

medição entre *frameworks* distintos, permitindo avaliações equivalentes sob as mesmas condições experimentais. Além disso, o *Flashlight* fornece métricas de alta granularidade, exige pouca configuração, não interfere significativamente no desempenho do aplicativo durante a coleta e permite execução simples e reproduzível dos experimentos. Essas características o tornam adequado para estudos comparativos, como defendido por Karami et al. (2023), Gowri et al. (2023), justificando sua adoção como ferramenta central na presente pesquisa.

4.4 Detalhes da Implementação

Para a condução dos experimentos e medições de desempenho, foram desenvolvidos dois aplicativos equivalentes em *Flutter* e *React Native*, ambos projetados para simular cargas reais de processamento e utilização de recursos de *hardware*, conforme descrito nas seções anteriores. O processo de desenvolvimento, teste e geração do APK, seguiu as boas práticas documentadas por cada *framework*, respeitando suas ferramentas nativas de *build* e execução.

4.4.1 Implementação em Flutter

O desenvolvimento dos aplicativos em *Flutter* foi realizado utilizando o *Android Studio* como IDE principal, integrando o SDK oficial e *Dart* como linguagem base. Essa escolha foi motivada pela integração direta entre o ambiente de desenvolvimento e o dispositivo, possibilitando compilação e monitoramento em tempo real dentro da mesma interface. Durante a implementação, foram seguidos os padrões recomendados na documentação oficial do *Flutter*: estruturação dos códigos em `lib/main.dart`; configuração do arquivo `pubspec.yaml` para geração das dependências externas quando necessário; utilização do comando `flutter run --release` para execução no dispositivo físico, garantindo medições sem interferência do modo de *debug*; geração do arquivo `.apk` por meio do comando `flutter build apk --release`, que cria o pacote executável na pasta `/build/app/outputs/flutter-apk/`.

As medições foram realizadas diretamente no dispositivo físico *Android*, conectado via ADB[9]. Foi utilizado o comando `flashlight measure`, que executou o *Flashlight* durante 44,5 segundos em cada teste. Esse processo garante fidelidade aos resultados, uma vez que a compilação em modo *release* remove sobrecargas do interpretador do *Dart*.

4.4.2 Implementação em React Native

Os aplicativos equivalentes em *React Native* foram desenvolvidos utilizando o *Visual Studio Code* (CODE, 2025) como ambiente de desenvolvimento e o Expo (2025) como plataforma de execução e *build*. O *Expo* foi escolhido por simplificar o processo de configuração e compilação, permitindo testar o aplicativo diretamente em um dispositivo *Android* através do aplicativo *Expo Go* e, posteriormente, realizar o *download* do arquivo `.apk`.

[9] *Android Debug Bridge*

A estrutura inicial do projeto foi criada com o comando: `npx create-expo-app@latest --template blank-typescript`. Em seguida, o arquivo principal `App.tsx` foi modificado com o código da aplicação e as dependências necessárias, enquanto o arquivo `app.json` foi ajustado para incluir as permissões e configurações específicas de cada teste.

Durante a fase de desenvolvimento, utilizou-se o comando: `npx expo start`, o qual gera um *QR Code* que, ao ser escaneado pelo aplicativo *Expo Go*, permite executar o aplicativo diretamente no dispositivo físico, possibilitando ajustes rápidos. Ao finalizar a implementação, o *Expo* foi responsável pela geração do pacote final em formato APK. Esse processo foi realizado por meio do comando: `eas build -p android --profile preview`, que realiza o empacotamento completo do aplicativo em modo *release*, gerando o executável otimizado para instalação direta via ADB com o comando `adb install nome_do_aplicativo.apk`. Após o download, o arquivo foi instalado manualmente no dispositivo, onde as medições de desempenho foram conduzidas com o uso da ferramenta *Flashlight* com o comando `flashlight measure`.

Assim como no *Flutter*, todos os testes foram realizados no mesmo dispositivo físico, sob condições controladas, e com tempo de execução idêntico de 44.500 ms. Essa padronização assegurou a comparabilidade direta entre os *frameworks*, isolando diferenças apenas nos aspectos técnicos de compilação e execução.

5 RESULTADOS

Este capítulo apresenta os resultados obtidos nos testes de desempenho realizados com os dois aplicativos experimentais, desenvolvidos em *Flutter* e *React Native*. As métricas foram coletadas através da ferramenta *Flashlight*, considerando: tempo médio de execução monitorado, fluidez da interface, uso médio do processamento, tempo em pico máximo de processamento e uso médio de memória. Cada teste foi executado três vezes por tecnologia, conforme metodologia descrita anteriormente.

5.1 Resultados do primeiro aplicativo

A primeira aplicação executa operações intensivas de CPU, incluindo geração e ordenação de listas, serialização JSON e cálculo de números primos. Esse tipo de carga representa cenários computacionalmente custosos, nos quais o comportamento da aplicação depende diretamente da eficiência do *framework* e da capacidade de gerenciar tarefas síncronas e assíncronas. (NAWROCKI et al., 2021; DORFER; DEMETZ; HUBER, 2020)

5.1.1 Análise dos resultados - Flutter

O aplicativo desenvolvido em *Flutter* apresentado na Tabela 4, mostra que a taxa média de quadros atingiu 41 FPS, evidenciando a eficiência do motor de renderização *Skia* e a compilação

nativa AOT[10], discutidas em Karami et al. (2023). O uso de CPU variou entre 72% e 76%, demonstrando menor esforço computacional para executar a mesma carga. O consumo médio de RAM foi de 325.8 MB, similar ao *React Native*. O tempo sob CPU máxima foi de 1 segundo,

Tabela 4 – Resultados dos teste em *Flutter*

Teste	Avg Runtime	Avg FPS	Avg CPU	High CPU	Avg RAM
T1	44 500 ms	40.9	75.9%	1 s	323.7 MB
T2	44 500 ms	41.4	72.8%	1 s	335.1 MB
T3	44 500 ms	41.0	75.6%	1 s	318.6 MB

Fonte: Autoral.

mostrando alta eficiência no gerenciamento da carga pesada. O tamanho final do APK foi de 69.4 MB, característica já destacada pela literatura devido ao empacotamento do *Engine* próprio do Flutter (SOUHA et al., 2024).

5.1.2 Análise dos resultados - React Native

Nos testes realizados com o *React Native* mostrado na Tabela 5, o aplicativo apresentou desempenho consistente, com taxa média próximas de 59 FPS ao longo das três execuções. O consumo de CPU permaneceu elevado, variando entre 88% e 89%, indicando que o *framework* exigiu maior esforço de processamento para manter a aplicação responsiva. O consumo de RAM

Tabela 5 – Resultados dos teste em *React Native*

Teste	Avg Runtime	Avg FPS	Avg CPU	High CPU	Avg RAM
T1	44 500 ms	59.8	88.6%	31.5 s	329.4 MB
T2	44 500 ms	59.8	88.0%	31.0 s	305.1 MB
T3	44 500 ms	59.7	88.3%	30.5 s	294.1 MB

Fonte: Autoral.

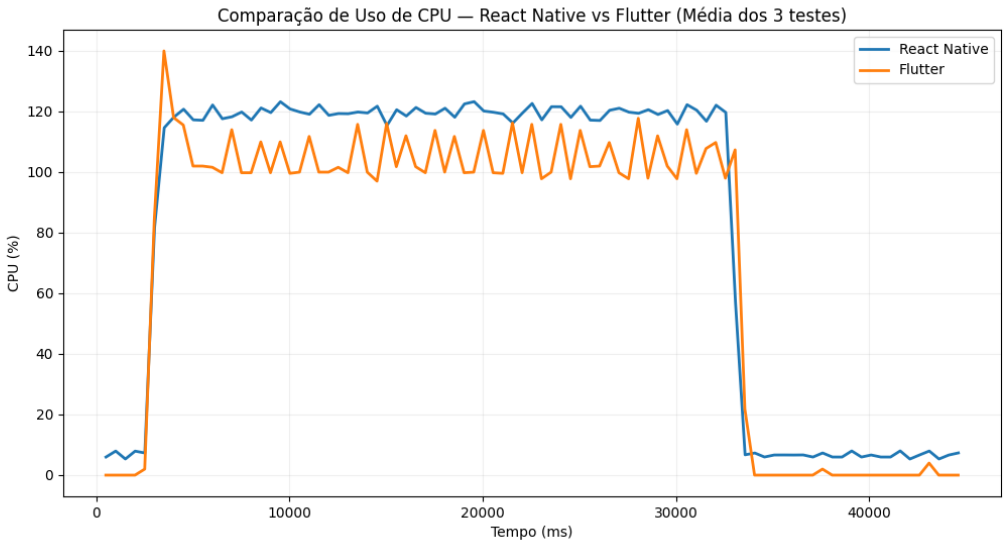
se manteve estável, com média de 309.5 MB. O tempo em alta carga (High CPU) aproximou-se de 31 segundos, demonstrando maior permanência em esforço máximo e o tamanho final do APK compilado foi de 53.9 MB.

5.1.3 Comparativo entre as tecnologias

Os resultados mostram que o *Flutter* apresentou uma taxa menor no FPS por causa de seu motor Skia que ficou sobrecarregado na UI Thread. Esse comportamento também se observa na Figura 2, que mostra um gráfico comparativo do uso da CPU. É mostrado que o *React Native* opera continuamente em faixas mais elevadas, situadas entre 115% e 125%, enquanto o *Flutter* mantém variação menor, concentrada entre 100% e 115%. O uso médio de CPU, portanto, foi 13,5% menor para o *Flutter*, demonstrando maior eficiência computacional e indicando menor *overhead* estrutural em *frameworks* compilados nativamente, como mostrado por Dorfer, Demetz e Huber (2020).

[10] *Ahead-of-Time*

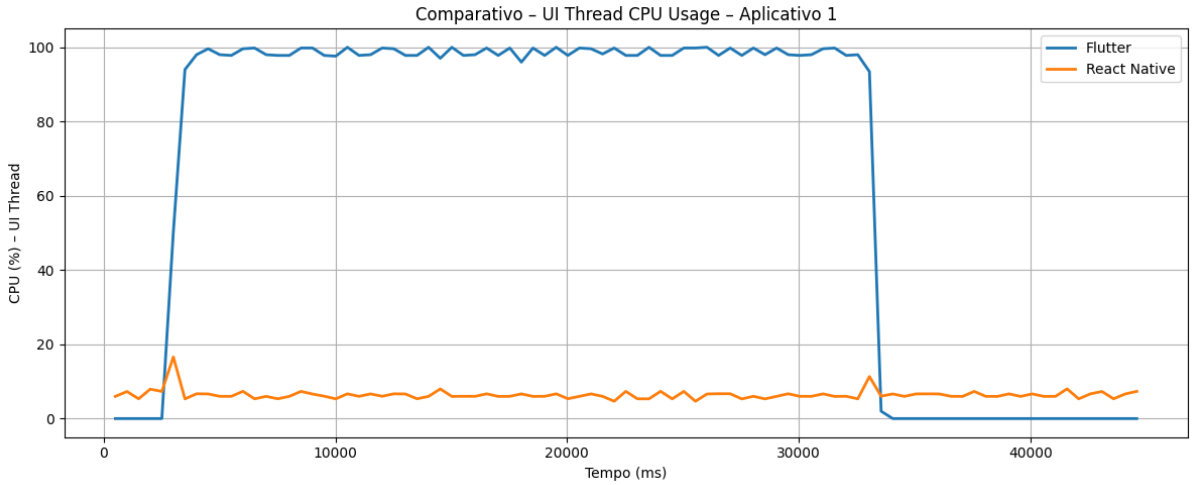
Figura 2 – Comparativo entre o consumo da CPU



Fonte: Autoral

A métrica de uso da *UI Thread* acrescenta uma nuance importante à interpretação dos resultados. Conforme ilustrado na Figura 3, o *Flutter* manteve a *UI Thread* próxima de 100% de utilização durante praticamente todo o período de teste, o que indica que a maior parte do trabalho de processamento ficou concentrada na *thread* responsável pela renderização da interface. Na prática, esse comportamento foi percebido como pequenos "engasgos" na evolução da tela, com atualizações ocorrendo em blocos em vez de forma totalmente contínua. No *React*

Figura 3 – Comparativo entre o uso da *UI Thread*



Fonte: Autoral

Native, por outro lado, embora o uso total de CPU tenha sido superior, a *UI Thread* permaneceu significativamente mais leve, com percentuais bem abaixo de 20% na maior parte do tempo. Isso resultou em uma evolução mais linear e suave da interface durante as operações intensivas, sugerindo melhor distribuição da carga entre a *UI Thread* e outras *threads* de trabalho. Gráfico

adicional de FPS, pode ser consultado no Anexo 1.

A Tabela 6 evidencia diferenças claras entre as duas tecnologias quando submetidas ao mesmo volume de processamento. O tempo em carga máxima foi o ponto de maior contraste: enquanto o *Flutter* permaneceu apenas 1 segundo em pico de processamento, o *React Native* sustentou 31 segundos, evidenciando menor capacidade de otimização interna. Esse comportamento se alinha ao observado no gráfico de CPU, em que o *React Native* mantém níveis elevados por longos intervalos, ao passo que o *Flutter* estabiliza rapidamente. Esses resultados seguem tendências já descritas em estudos experimentais recentes: Nawrocki et al. (2021) aponta que o *React Native* sofre impacto maior em tarefas síncronas intensivas devido à ponte *JavaScript*, enquanto o *Flutter* mantém execução mais estável.

Tabela 6 – Comparativo do primeiro aplicativo

Métrica	React Native	Flutter	Diferença
Avg FPS	59.8	41.1	18.7 FPS
Avg CPU	88.3 %	74.8 %	13.5%
Avg RAM	309.5 MB	325.8 MB	16.3MB
High CPU	31 s	1 s	30s

Fonte: Autoral.

Assim, no contexto do primeiro aplicativo, focado em carga computacional e operações intensivas, o *Flutter* demonstrou desempenho amplamente superior, especialmente na eficiência de CPU, tamanho do APK e tempo sob esforço máximo, ainda que o *React Native* apresente um pacote final menor e melhor distribuição de carga na *UI Thread*. Essa combinação torna o *Flutter* mais indicado para aplicações que envolvem cálculos pesados, interações contínuas, ao mesmo tempo em que evidencia que o *React Native* pode oferecer uma experiência de interface regular em cenários semelhantes.

5.2 Resultados do segundo aplicativo

O segundo aplicativo foi pensado em avaliar o comportamento dos *frameworks* em um cenário de uso direto de *hardware*, envolvendo captura de vídeo, operação contínua da câmera e escrita de arquivos. Esse tipo de aplicação exige um fluxo constante de dados entre sensores, *threads* nativas e rotinas de renderização, ao mesmo tempo em que demanda estabilidade no uso de CPU, boa fluidez gráfica (FPS) e comportamento consistente de memória. Assim como no primeiro estudo, os testes foram executados durante 44.500 ms e repetidos três vezes.

5.2.1 Análise dos resultados - Flutter

De acordo com a Tabela 7, os resultados mostram desempenho estável durante toda a gravação de vídeo. A média de FPS se manteve em 59.8FPS nos três testes, muito próxima ao limite de 60FPS imposto pelo *hardware*, o que indica excelente fluidez e sincronização com o pipeline da câmera. O uso médio de CPU oscilou entre 48% e 49.8%, revelando um processamento eficiente e sem sobrecarga excessiva. A memória RAM apresentou valores

entre 186 MB e 196 MB, mostrando baixo consumo mesmo sob atividade intensa de captura e escrita de vídeo. A ausência de picos de CPU demonstra que o *framework* conseguiu manter desempenho constante, sem gargalos perceptíveis. Esses resultados reforçam o comportamento já documentado pela literatura, que atribui ao *Flutter* boa performance em rotinas multimídia devido ao motor nativo e ao controle direto da renderização.

Tabela 7 – Resultados dos teste em *Flutter*

Teste	Avg Runtime	Avg FPS	Avg CPU	High CPU	Avg RAM
T1	44 500 ms	59.8	48%	-	196 MB
T2	44 500 ms	59.8	49.8%	-	187.5 MB
T3	44 500 ms	59.8	49.3%	-	186.2 MB

Fonte: Autoral.

5.2.2 Análise dos resultados - React Native

O *React Native* apresentou desempenho estável, registrando valores entre 51.7 e 52.2FPS como visto na Tabela 8. Apesar de ser uma taxa aceitável para gravação de vídeo, permanece abaixo do limite do *hardware* e indica maior oscilação durante a coleta. O uso médio de CPU variou entre 50% e 51.2%, o que pode ser associado à ponte *JavaScript* e à necessidade de comunicação constante com módulos nativos. O consumo de RAM foi variando entre 336 MB e 348 MB. Isso evidencia maior alocação interna de *buffers* e sobrecarga durante a transmissão de dados da câmera para o ambiente *JavaScript*. Assim como no *Flutter*, não foram identificados picos de CPU (High CPU), o que indica estabilidade.

Tabela 8 – Resultados dos teste em *React Native*

Teste	Avg Runtime	Avg FPS	Avg CPU	High CPU	Avg RAM
T1	44 500 ms	51.7	51.2%	-	336.9 MB
T2	44 500 ms	52.2	50.8%	-	342.5 MB
T3	44 500 ms	51.7	50.1%	-	348.8 MB

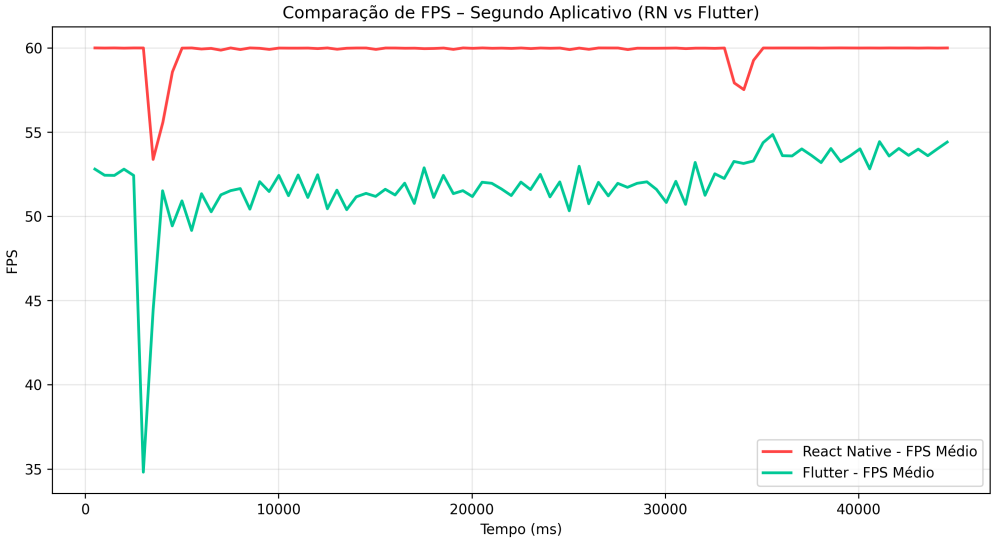
Fonte: Autoral.

5.2.3 Comparativo entre as tecnologias

Os resultados obtidos no segundo aplicativo demonstram novamente uma vantagem consistente do *Flutter* em diversos aspectos de desempenho. Como mostrado na Figura 5, referente ao comparativo de FPS, o *Flutter* manteve uma média estável de 59.8FPS, enquanto o *React Native* apresentou 51.9FPS. A diferença de 7.9FPS é particularmente relevante em cenários que envolvem captura de vídeo, gravação contínua ou qualquer aplicação sensível à fluidez gráfica, evidenciando maior estabilidade do pipeline de renderização do *Flutter*.

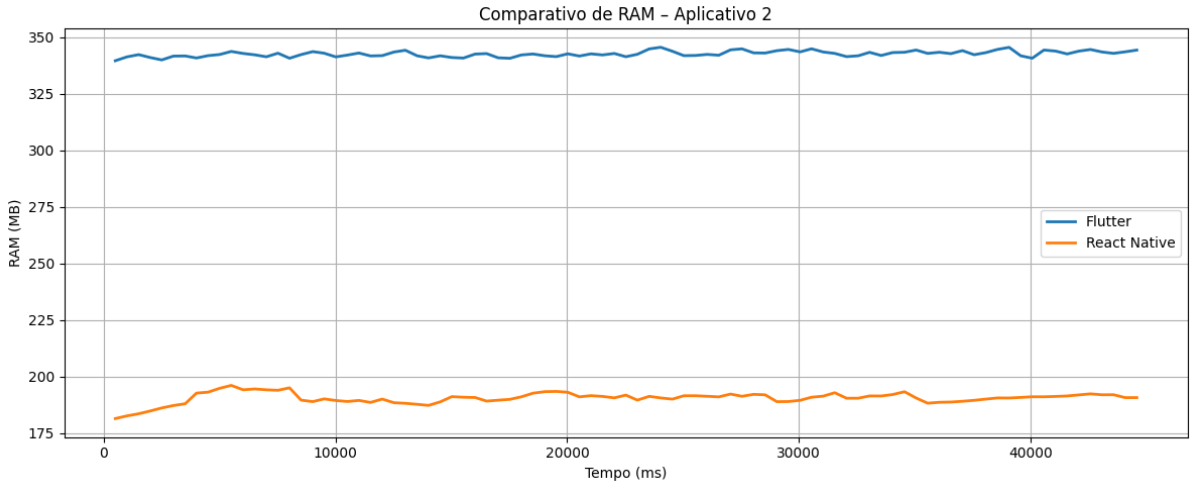
A análise da Figura 6 referente ao uso de memória RAM no segundo aplicativo revela o contraste mais acentuado entre as tecnologias. O *Flutter* manteve consumo aproximado de 190MB ao longo de toda a execução, com oscilações discretas. Já o *React Native* operou acima

Figura 4 – Comparativo entre a taxa de *Frames* por Segundo



Fonte: Autoral

Figura 5 – Comparativo entre o consumo da Memória RAM



Fonte: Autoral

de 330MB durante praticamente todo o teste, chegando a ultrapassar 348MB em períodos específicos. A diferença média de 152.8MB, apresentada na Tabela 9, indica um custo estrutural significativamente maior no *React Native*, que pode ser explicado pela sobrecarga da ponte *JavaScript–Nativo* no processamento de quadros e no gerenciamento de *buffers* de vídeo. Esse

Tabela 9 – Comparativo do segundo aplicativo

Métrica	**React Native**	**Flutter**	**Diferença**
Avg FPS	51.9	59.8	7.9 FPS
Avg CPU	50.7%	49%	1.7%
Avg RAM	342.7 MB	189.9 MB	152.8 MB
High CPU	-	-	-

Fonte: Autoral.

comportamento é consistente com a literatura, que aponta que aplicações com demandas frequentes de entrada/saída e sincronização com *hardware* tendem a expor gargalos mais facilmente em arquiteturas baseadas em *bridges* (NAWROCKI et al., 2021; DORFER; DEMETZ; HUBER, 2020).

No uso de CPU, o *Flutter* também apresentou leve vantagem, registrando consumo médio 1.7% menor que o *React Native*. Embora modesta, essa diferença reflete maior eficiência interna no processamento de quadros e na integração direta com APIs nativas de câmera. Em ambas as tecnologias, não houve registros de High CPU significativos, o que sugere que o fluxo de captura de vídeo operou dentro de limites estáveis sem picos altos de processamento.

Na métrica de uso da *UI Thread*, observa-se um comportamento distinto do primeiro cenário. No aplicativo de captura de vídeo, nenhuma das tecnologias manteve a *UI Thread* sobrecarregada. Tanto o *Flutter* quanto o *React Native* apresentaram percentuais reduzidos e relativamente estáveis durante os 30 segundos de gravação, com variações mais perceptíveis apenas nos instantes de início e término da captura. Isso indica que boa parte do trabalho pesado é delegada a *threads* de fundo e rotinas nativas da câmera, reduzindo o impacto direto sobre a *thread* responsável pela interface. Dessa forma, não há vantagem clara de uma tecnologia sobre a outra nessa métrica específica, e a experiência visual observada na prática permaneceu fluida em ambos os casos.

Os demais gráficos de comparação entre cada métrica estão reunidos no Anexo 1, oferecendo uma visão detalhada de comportamento das duas tecnologias sobrepostas. Esses dados reforçam a tendência observada nas métricas médias: maior fluidez, menor consumo de recursos e maior estabilidade geral no *Flutter*, enquanto o *React Native* apresentou maior variação interna e maior dependência do seu ecossistema de ponte *JavaScript*.

6 CONCLUSÃO

Os resultados obtidos ao longo deste estudo evidenciam diferenças significativas entre os *frameworks Flutter* e *React Native* quando submetidos a cenários reais de execução. No primeiro aplicativo, voltado para operações de alta carga computacional, o *Flutter* demonstrou desempenho superior em praticamente todas as métricas avaliadas, apresentando maior fluidez, menor uso médio de CPU e tempo sobre carga máxima. O *React Native*, por sua vez, exigiu mais recursos para realizar as mesmas operações, reflexo de sua arquitetura baseada em ponte *JavaScript*. Entretanto, na métrica de uso da *UI Thread*, o cenário foi mais equilibrado. O *React Native* manteve essa *thread* menos sobrecarregada, resultando em uma evolução mais linear das interações na interface, enquanto o *Flutter* operou com a *UI Thread* próxima do limite durante grande parte da execução.

No segundo aplicativo, voltado ao uso de câmera e manipulação contínua de vídeo, o *Flutter* novamente se destacou, alcançando FPS mais elevados, menor uso de CPU e uma diferença expressiva no consumo de memória RAM, reforçando sua eficiência em tarefas

multimídia e de integração direta com *hardware*. Também não foram observadas diferenças relevantes entre as tecnologias no uso de *UI Thread*.

Com base nessas análises, conclui-se que o *Flutter* apresenta vantagens notáveis em performance e estabilidade quando comparado ao *React Native*, especialmente em aplicações que demandam alto processamento ou interação frequente com recursos nativos do dispositivo. Embora o *React Native* mantenha relevância no ecossistema de desenvolvimento devido à sua flexibilidade e vasta comunidade, os resultados deste estudo sugerem que o *Flutter* é, no contexto testado, a alternativa mais eficiente para cenários de maior complexidade computacional e multimídia, ainda que o *React Native* apresente pontos fortes específicos, como o comportamento da *UI Thread* no primeiro cenário e o menor tamanho do pacote final.

7 TRABALHOS FUTUROS

Para ampliar o escopo desta pesquisa, recomenda-se a realização de estudos que incluam outros *frameworks cross-platform*, como *Ionic* e *Xamarin* possibilitando uma análise comparativa mais ampla e atualizada do ecossistema de desenvolvimento móvel. Também se mostra relevante a condução de experimentos que comparem diretamente soluções multiplataforma e aplicações desenvolvidas de forma nativa (*Kotlin/Java* no *Android* e *Swift* no iOS), permitindo mensurar com maior precisão os impactos da abstração sobre o uso de recursos computacionais, energia e responsividade. Além disso, a realização dos testes em dispositivos distintos, com variações de memória, processador e versões de sistema operacional, pode contribuir para identificar eventuais limitações específicas ou comportamentos divergentes entre *hardwares*.

Outro caminho promissor é expandir o conjunto de cenários avaliados para além das operações computacionais e do uso de câmera. Aplicações que utilizam sensores contínuos (GPS, acelerômetro e giroscópio), renderização 2D/3D, jogos, transmissões ao vivo, manipulação de banco de dados local em larga escala e chamadas frequentes a APIs externas poderiam fornecer uma visão mais completa sobre a performance dos *frameworks* em condições mais variadas. Adicionalmente, incorporar métricas de temperatura do dispositivo e consumo energético aprofundaria a análise, especialmente para aplicações que exigem longos períodos de execução. Tais extensões permitiriam estabelecer um panorama mais robusto e abrangente, oferecendo bases mais sólidas para orientar a escolha tecnológica por parte de desenvolvedores e equipes de engenharia de *software*.

REFERÊNCIAS

ANDROID. **Android Developer**. 2025. Android Developer. Disponível em: <https://developer.android.com/?hl=pt-br>.

APPLE. **Apple Developer**. 2025. Apple Developer. Disponível em: <https://developer.apple.com/>.

CODE, V. S. **Visual Studio Code**. 2025. Visual Studio Code. Disponível em: <https://code.visualstudio.com/>.

CORDOVA. **Cordova Documentation**. 2025. Cordova Documentation. Disponível em: <https://cordova.apache.org/>.

CSS3. **CSS Documentation**. 2025. CSS Documentation. Disponível em: <https://developer.mozilla.org/en-US/docs/Web/CSS>.

DORFER, T.; DEMETZ, L.; HUBER, S. Impact of mobile cross-platform development on cpu, memory and battery of mobile devices when using common mobile app features. **Procedia Computer Science**, Elsevier, v. 175, p. 189–196, 2020.

EXPO. **Expo**. 2025. Expo. Disponível em: <https://expo.dev/>.

FLASHLIGHT. **Flashlight**. 2025. Flashlight. Disponível em: <https://flashlight.dev/>.

FLUTTER. **Flutter Documentation**. 2025. Flutter Documentation. Disponível em: <https://flutter.dev/>.

GOWRI, S. et al. Intelligent analysis on frameworks for mobile app development. In: IEEE. **2023 5th International Conference on Smart Systems and Inventive Technology (ICSSIT)**. [S.l.], 2023. p. 1506–1512.

HTML5. **HTML Documentation**. 2025. HTML Documentation. Disponível em: <https://html.spec.whatwg.org/>.

IONIC. **Ionic Documentation**. 2025. Ionic Documentation. Disponível em: <https://ionicframework.com/>.

JS. **JavaScript Documentation**. 2025. JavaScript Documentation. Disponível em: <https://developer.mozilla.org/pt-BR/docs/Web/JavaScript>.

KARAMI, P. et al. On the impact of development frameworks on mobile apps. In: IEEE. **2023 30th Asia-Pacific Software Engineering Conference (APSEC)**. [S.l.], 2023. p. 131–140.

NATIVE, R. **React Native Documentation**. 2025. React Native Documentation. Disponível em: <https://reactnative.dev/>.

NAWROCKI, P. et al. A comparison of native and cross-platform frameworks for mobile applications. **Computer**, IEEE, v. 54, n. 3, p. 18–27, 2021.

SOUHA, A. et al. Comparative analysis of mobile application frameworks: A developer's guide for choosing the right tool. **Procedia Computer Science**, Elsevier, v. 236, p. 597–604, 2024.

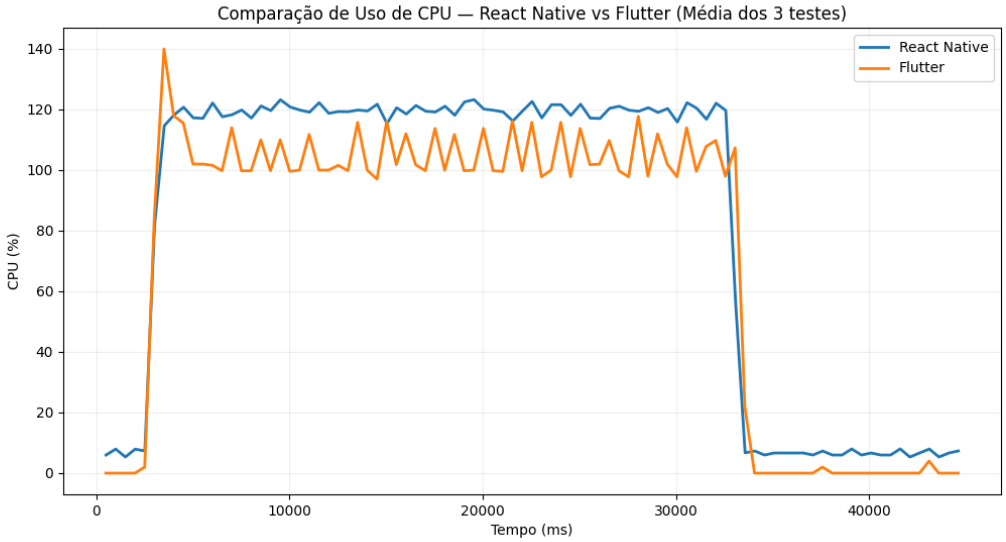
STUDIO, A. **Android Studio**. 2025. Android Studio Download. Disponível em: <https://bit.ly/androidstudiotcc>.

WAMBUA, A. W. What do flutter developers ask about? an empirical study on stack overflow posts. **Journal of Software Engineering Research and Development**, v. 12, n. 1, p. 7–1, 2024.

XAMARIN. **Xamarin Documentation**. 2025. Xamarin Documentation. Disponível em: <https://dotnet.microsoft.com/pt-br/apps/xamarin>.

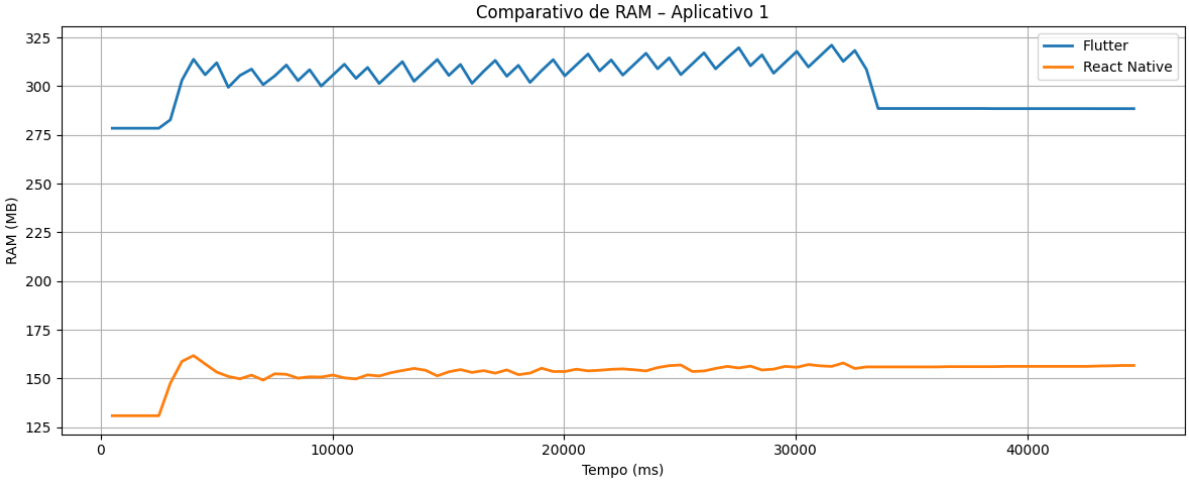
ANEXO 1

Figura 6 – Comparativo entre o consumo da CPU - App 01



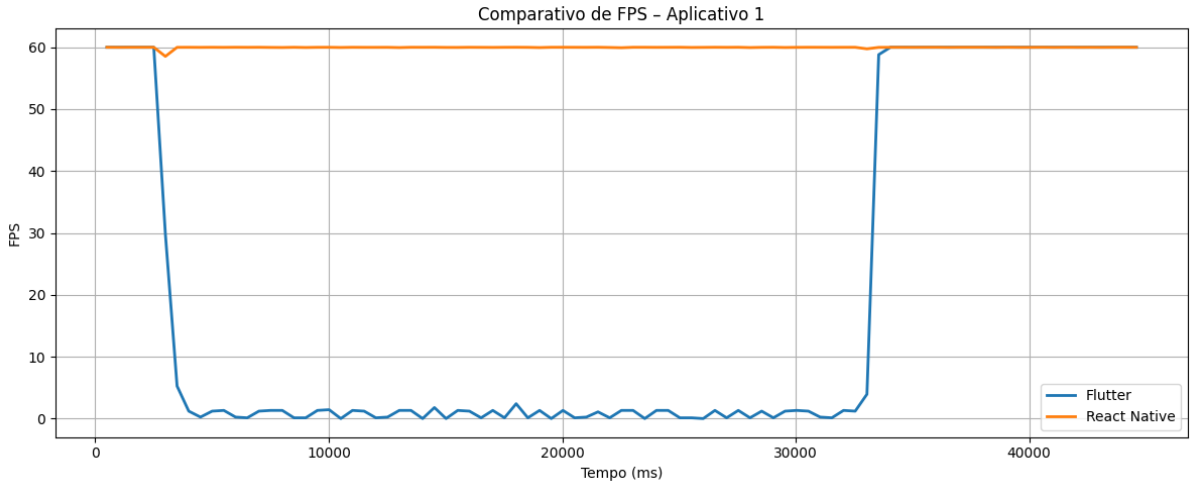
Fonte: Autoral

Figura 7 – Comparativo entre o consumo da memória RAM - App 01



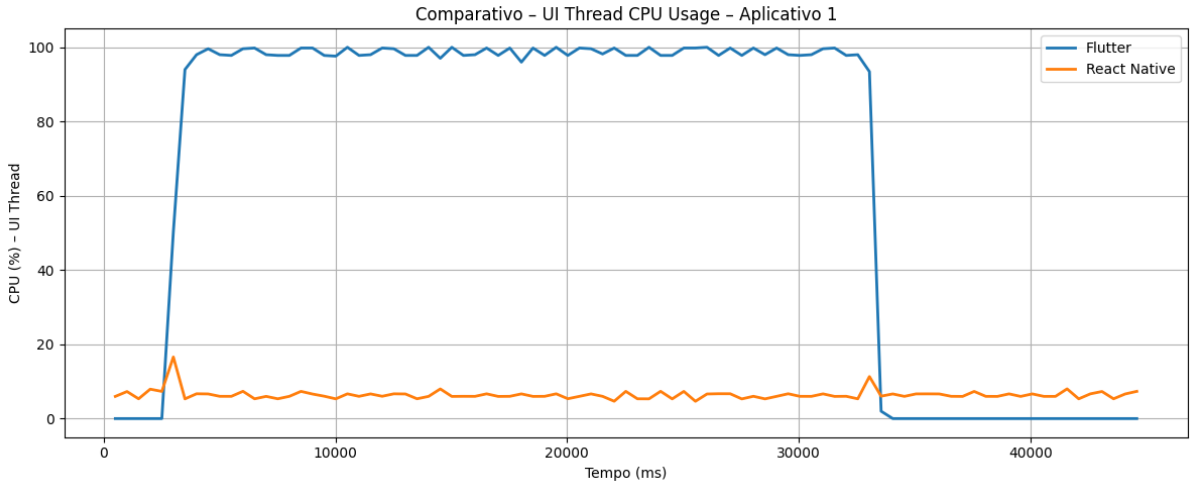
Fonte: Autoral

Figura 8 – Comparativo entre a taxa de FPS - App 01



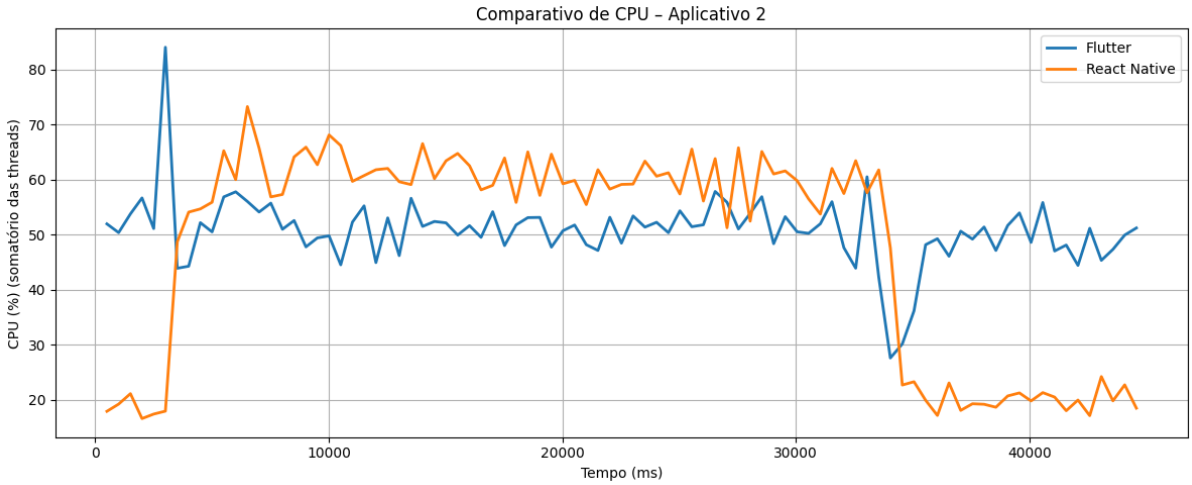
Fonte: Autoral

Figura 9 – Comparativo entre o uso UI Thread CPU - App 01



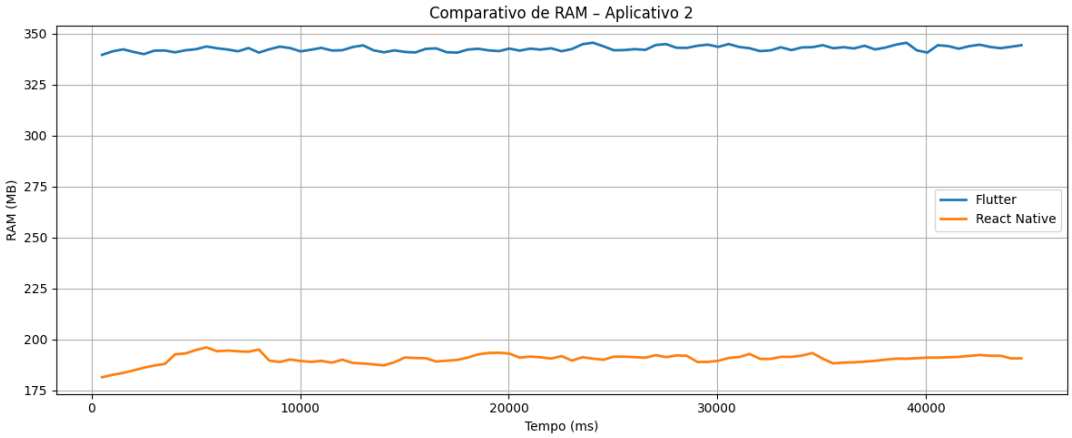
Fonte: Autoral

Figura 10 – Comparativo entre o consumo da CPU - App 02



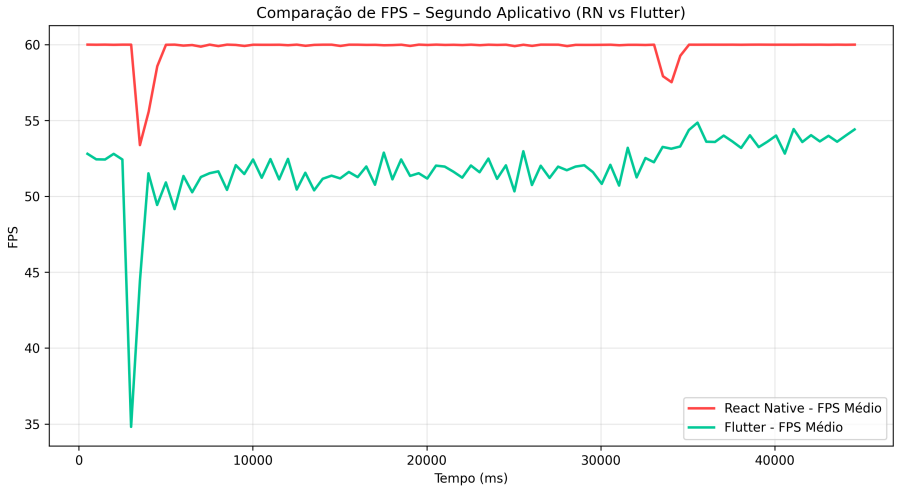
Fonte: Autoral

Figura 11 – Comparativo entre o consumo da memória RAM - App 02



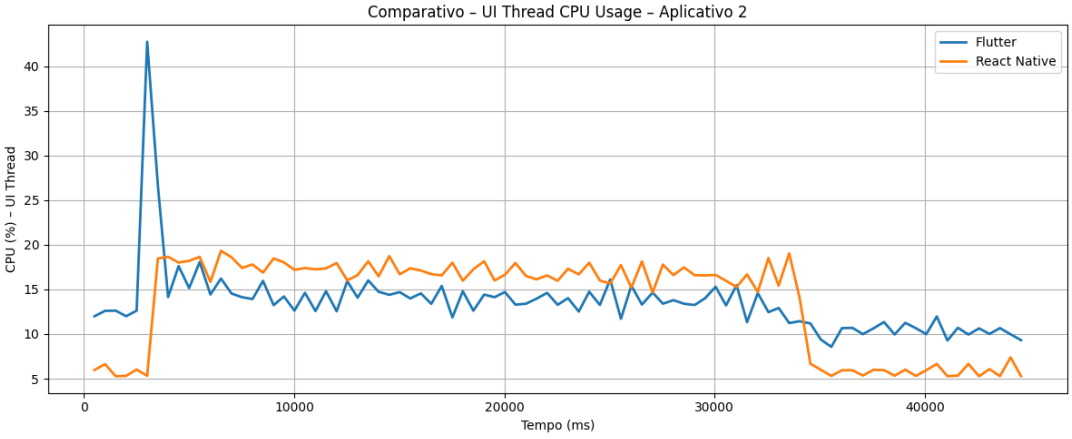
Fonte: Autoral

Figura 12 – Comparativo entre a taxa de FPS - App 02



Fonte: Autoral

Figura 13 – Comparativo entre o uso UI Thread CPU - App 02



Fonte: Autoral