UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ

RÔMULO OLIVEIRA SOUZA

FLUTTER *VERSUS* REACT NATIVE: UM ESTUDO DE CASO CONSIDERANDO CONSUMO DE RECURSOS, ASPECTOS DA INTERFACE GRÁFICA E TEMPOS DE INICIALIZAÇÃO E RESPOSTA

CORNÉLIO PROCÓPIO

2025

versão 11.0 (abr. 25)

RÔMULO OLIVEIRA SOUZA

FLUTTER *VERSUS* REACT NATIVE: UM ESTUDO DE CASO CONSIDERANDO CONSUMO DE RECURSOS, ASPECTOS DA INTERFACE GRÁFICA E TEMPOS DE INICIALIZAÇÃO E RESPOSTA

Flutter versus React Native: A case study considering resource consumption, graphical interface aspects and startup and response times

Trabalho de Conclusão de Curso de Graduação apresentado como requisito para obtenção do título de Bacharel em Engenharia De Computação do Curso de Bacharelado em Engenharia De Computação da Universidade Tecnológica Federal do Paraná.
Orientador(a): Prof. Dr. Henrique Yoshikazu Shishido

CORNÉLIO PROCÓPIO

2025

RÔMULO OLIVEIRA SOUZA

FLUTTER *VERSUS* REACT NATIVE: UM ESTUDO DE CASO CONSIDERANDO CONSUMO DE RECURSOS, ASPECTOS DA INTERFACE GRÁFICA E TEMPOS DE INICIALIZAÇÃO E RESPOSTA

Trabalho de Conclusão de Curso de Graduação apresentado como requisito para obtenção do título de Bacharel em Engenharia De Computação do Curso de Bacharelado em Engenharia De Computação da Universidade Tecnológica Federal do Paraná.

Data de aprovação: 12/novembro/2025

Alexandre Rômolo Moreira Feitosa
Doutorado
Universidade Tecnológica Federal do Paraná

Francisco Pereira Junior
Mestrado
Universidade Tecnológica Federal do Paraná

Henrique Yoshikazu Shishido
Doutorado
Universidade Tecnológica Federal do Paraná

CORNÉLIO PROCÓPIO

2025

AGRADECIMENTOS

Primeiramente, agradeço a Deus, por me guiar, me dar força e me sustentar em todos os momentos desta jornada.

Aos meus pais e à minha família, meu muito obrigado pelo apoio, amor e incentivo incondicional em todas as etapas da minha vida. Sem vocês, nada disso seria possível.

À Universidade Tecnológica Federal do Paraná (UTFPR), campus Cornélio Procópio, agradeço por toda a formação que me proporcionou, pelo aprendizado, pelas experiências e pelo ambiente que tanto contribuiu para o meu crescimento pessoal e profissional.

Ao meu orientador, Prof. Dr. Henrique Yoshikazu Shishido, sou muito grato pela disponibilidade, paciência e apoio na construção deste trabalho, sempre pronto para esclarecer dúvidas e orientar nos momentos necessários.

Aos meus amigos, que tornaram a graduação mais leve e divertida, agradeço pelos momentos de aprendizado, risadas, apoio e pelas experiências compartilhadas.

Agradeço também a todos que, de alguma forma, contribuíram direta ou indiretamente para que eu chegasse até aqui, seja compartilhando conhecimento, oferecendo apoio ou simplesmente estando presentes nesta jornada.

RESUMO

Este trabalho apresenta um estudo comparativo entre os *frameworks* Flutter e React Native, amplamente utilizados no desenvolvimento de aplicações móveis multiplataforma. O objetivo principal foi analisar e comparar o desempenho de ambas as tecnologias em diferentes métricas de desempenho, incluindo a taxa de uso do processador, fluidez da interface gráfica, consumo de memória, tempo de inicialização, tempo de renderização da interface gráfica, latência de entrada e tempo de resposta de requisições a uma *Application Programming Interface* (API) local. Para isso, desenvolveu-se um estudo de caso composto por sete casos de teste, cada um referente a uma métrica analisada, os quais consistiam em aplicações desenvolvidas de forma equivalente em ambos os *frameworks*, executadas no sistema operacional Android. A utilização das aplicações foi automatizada mediante *scripts* em PowerShell, responsáveis por executar comandos do *Android Debug Bridge* (ADB) relacionados a gestos. A coleta das métricas, por sua vez, foi realizada por meio de comandos do ADB executados de forma automatizada por códigos em Python, além do uso de ferramentas específicas de cada tecnologia, como o Flutter *DevTools*, e da instrumentação diretamente no código-fonte das aplicações. Posteriormente, os dados coletados foram analisados também utilizando a linguagem Python, permitindo a consolidação dos resultados e a geração dos gráficos utilizados na análise comparativa. Os resultados obtidos indicaram que o React Native apresentou melhor desempenho nas métricas de consumo de memória e tempo de inicialização, enquanto o Flutter obteve resultados superiores em fluidez da interface, tempo de renderização, latência de entrada e tempo de resposta de requisições a uma API. Em relação à taxa de uso do processador, ambas as plataformas demonstraram eficiência, com diferenças pequenas. Conclui-se, portanto, que cada tecnologia apresenta vantagens específicas, sendo a escolha entre elas dependente das necessidades e características de cada projeto.

Palavras-chave: Flutter; React Native; métricas de desempenho; análise comparativa; desenvolvimento móvel.

ABSTRACT

This work presents a comparative study between the Flutter and React Native frameworks, widely used in cross-platform mobile application development. The main objective was to analyze and compare the performance of both technologies across different performance metrics, including CPU usage, graphical interface fluidity, memory consumption, startup time, interface rendering time, input latency, and response time to requests made to a local Application Programming Interface (API). For this purpose, a case study was conducted, consisting of seven test cases, each related to one of the analyzed metrics, which involved applications developed equivalently in both frameworks and executed on the Android operating system. The use of the applications was automated through PowerShell scripts responsible for executing Android Debug Bridge (ADB) commands related to gestures. Metric collection was performed using ADB commands automated by Python scripts, in addition to framework-specific tools, such as Flutter DevTools, and instrumentation directly in the source code. Subsequently, the collected data were analyzed using Python, enabling the consolidation of results and the generation of graphs used in the comparative analysis. The results indicated that React Native achieved better performance in memory consumption and startup time, while Flutter outperformed in interface fluidity, rendering time, input latency, and response time to API requests. Regarding CPU usage, both platforms demonstrated efficient performance, with minimal differences. It is therefore concluded that each technology presents specific advantages, and the choice between them depends on the requirements and characteristics of each project.

Keywords: Flutter; React Native; performance metrics; comparative analysis; mobile development.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE QUADROS

LISTAGEM DE CÓDIGOS FONTE

SUMÁRIO

1 INTRODUÇÃO

O desenvolvimento de aplicativos móveis tem passado por transformações significativas, impulsionado pela crescente demanda por aplicativos eficientes e disponíveis em múltiplos sistemas operacionais (Jain, 2025). No Brasil, por exemplo, essa demanda é impulsionada pela massiva presença dos *smartphones*, que se consolidaram como o principal meio de acesso à internet. De acordo com uma pesquisa do Centro de Tecnologia de Informação Aplicada (FGVcia), coordenada por Meirelles (2025), o país possui 272 milhões de dispositivos móveis em uso, o que equivale a uma média de aproximadamente 1.3 *smartphone* por habitante. Diante desse cenário de ampla disseminação de aparelhos móveis, as empresas e desenvolvedores precisam considerar, entre outros fatores, qual abordagem adotar para o desenvolvimento de aplicativos: nativa, híbrida, *web app* ou então multiplataforma (também conhecida como *cross-platform*).

O desenvolvimento nativo envolve a criação de aplicativos específicos para cada sistema operacional, como iOS e Android, utilizando as linguagens e ferramentas próprias de cada plataforma. Esse modelo permite um desempenho otimizado e acesso completo aos recursos do dispositivo, proporcionando uma experiência de usuário superior. Em contrapartida, as demais abordagens utilizam uma única base de código que pode ser executada em diferentes sistemas operacionais, permitindo alcançar um público mais amplo com menor tempo e custo de desenvolvimento. No entanto, essas estratégias podem apresentar limitações quanto ao desempenho e ao acesso às funcionalidades específicas do dispositivo (Pinto; Coutinho, 2018).

Ao analisar a adesão ao desenvolvimento multiplataforma, observa-se que duas tecnologias se destacam entre os desenvolvedores: o Flutter e o React Native. Segundo os resultados de uma pesquisa anual do Stack Overflow (2024), que investiga os *frameworks*[1] mais populares entre desenvolvedores profissionais, essas duas ferramentas figuram como as mais utilizadas no âmbito do desenvolvimento de aplicativos móveis.

Neste contexto, este trabalho compara o Flutter com o React Native em termos de desempenho, a partir de diferentes cenários de teste desenvolvidos de forma equivalente em ambas as plataformas, fornecendo uma compreensão abrangente da eficiência de cada *framework*.

1.1 Problematização

Diante da crescente popularidade de *frameworks* multiplataforma para o desenvolvimento de aplicativos móveis, torna-se relevante compreender como diferentes tecnologias disponíveis nesse ecossistema se comportam em relação ao desempenho. Entre essas tecnologias, Flutter e React Native se destacam por sua ampla adoção e pela proposta de facilitar o desenvolvimento para múltiplos sistemas operacionais a partir de uma única base de código.

Embora apresentem o mesmo objetivo, essas ferramentas adotam arquiteturas e métodos de funcionamento distintos, o que pode gerar impactos no desempenho das aplicações (Souha *et al.*, 2024). Ainda que sejam amplamente utilizadas, é de grande importância

[1] Disponível em: https://aws.amazon.com/what-is/framework/

entender como cada uma se comporta em diferentes cenários e sob quais métricas de performance demonstram resultados satisfatórios.

1.2 Justificativa

O desempenho impacta diretamente na experiência do usuário, influenciando aspectos como fluidez e tempo de carregamento das interfaces gráficas, tempo de resposta e estabilidade da aplicação. Segundo Majumder (2025), a performance dos aplicativos exerce papel significativo nesse contexto, influenciando diretamente as taxas de retenção e satisfação dos usuários. Ainda segundo o autor, aplicativos que priorizam melhorias de performance podem apresentar até 30% maior taxa de retenção em comparação àqueles que negligenciam otimizações.

Portanto, realizar uma análise comparativa em termos de desempenho entre diferentes alternativas de desenvolvimento multiplataforma torna-se pertinente, uma vez que permite identificar as vantagens e as limitações de cada abordagem. Logo, investigar tecnicamente essas tecnologias é essencial para fundamentar escolhas mais assertivas, auxiliando desenvolvedores e organizações a selecionar o *framework* mais adequado às necessidades específicas de cada projeto.

1.3 Objetivos

Este trabalho compara e analisa métricas de desempenho entre os *frameworks* Flutter e React Native por meio da implementação prática de um estudo de caso.

1.3.1 Objetivos específicos

Os objetivos específicos deste trabalho são:

1. Planejar e validar os casos de teste a serem realizados;
2. Analisar o desempenho de cada tecnologia com base nas seguintes métricas: taxa de uso do processador; fluidez da interface gráfica; consumo de memória; tempo de inicialização; tempo de renderização da interface gráfica; latência de entrada e tempo de resposta de requisições feitas a uma *Application Programming Interface* (API)[2] local;
3. Implementar casos de teste específicos e equivalentes em ambos os *frameworks* para cada métrica a ser analisada;
4. Projetar mecanismos específicos para a coleta automatizada de algumas métricas de desempenho e, para outras, realizar a instrumentação diretamente no código-fonte da aplicação;

[2] Disponível em: https://aws.amazon.com/what-is/api/

5. Analisar os resultados obtidos a partir dos dados das métricas coletadas, identificando as vantagens e limitações de cada tecnologia.

1.4 Estrutura do trabalho

Nesta seção, apresenta-se a organização geral deste trabalho. Após o Capítulo 1 que trata da introdução, segue o Capítulo 2, que aborda o referencial teórico que fundamenta a pesquisa. Na sequência, o Capítulo 3 descreve três trabalhos correlatos selecionados para análise comparativa. O Capítulo 4 detalha os procedimentos metodológicos adotados para a obtenção dos resultados, os quais são discutidos no Capítulo 5. Por fim, o Capítulo 6 apresenta as considerações finais e as perspectivas futuras deste estudo.

2 REFERENCIAL TEÓRICO

Este capítulo tem como objetivo apresentar os principais fundamentos teóricos que embasam o desenvolvimento desse trabalho. São abordados os principais conceitos acerca das estratégias de desenvolvimento móvel, com ênfase no Flutter e no React Native. Além disso, o capítulo também contempla uma visão geral sobre o sistema operacional Android, bem como as métricas de desempenho consideradas.

2.1 Panorama das estratégias de desenvolvimento móvel

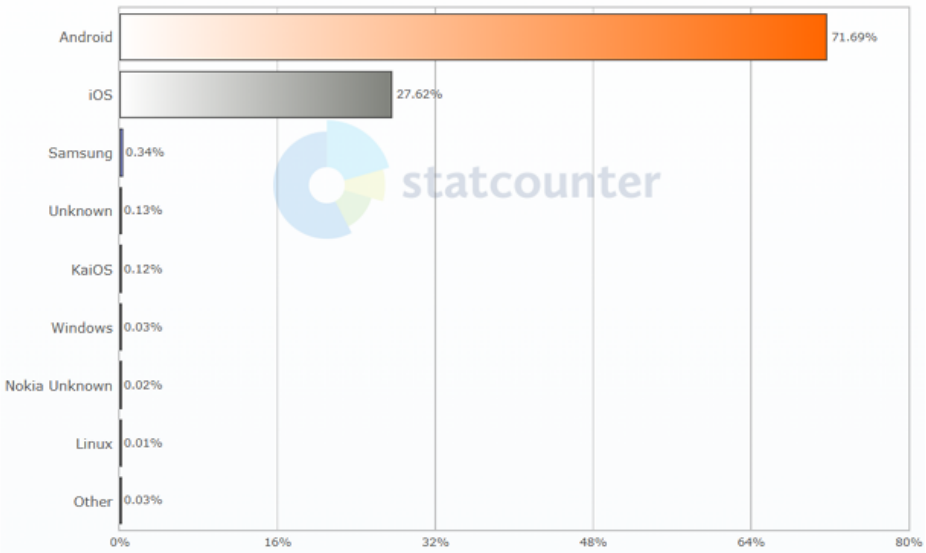
Nesta seção, serão apresentadas as principais abordagens existentes para o desenvolvimento de aplicativos móveis. Cada estratégia possui características específicas que influenciam no desempenho, na experiência do usuário e na compatibilidade com múltiplos sistemas operacionais.

2.1.1 Desenvolvimento nativo

Um aplicativo nativo é desenvolvido utilizando linguagens de programação específicas para o sistema operacional alvo. Essa abordagem permite que o aplicativo aproveite ao máximo os recursos e as funcionalidades do dispositivo, proporcionando um desempenho mais vantajoso (Poku-marboah, 2021).

Atualmente, os principais sistemas operacionais incorporados aos *smartphones* são o Android com uma taxa de adesão de 71,69% e o iOS com 27,62%, conforme dados dos últimos cinco anos do relatório da StatCounter (2025), o qual é mostrado na Figura 1.

Figura 1 – Participação de mercado de sistemas operacionais móveis mundialmente (2020 – 2025)



Fonte: (Statcounter, 2025).

Cada plataforma possui seu próprio *Software Development Kit* (SDK), bem como ferramentas e APIs que oferecem diversas funcionalidades. Para o Android, mantido pelo Google, é disponibilizado o Android SDK, o qual fornece recursos e bibliotecas específicas para o desenvolvimento de aplicativos nesse ambiente. De forma semelhante, o iOS SDK oferece recursos voltados para o ecossistema da Apple. A abordagem nativa possui algumas vantagens em relação às outras, pois disponibiliza APIs completas para acessar todos os recursos do dispositivo móvel, como câmera, sensores, rede, *Global Positioning System* (GPS), entre outros. Entretanto, essa metodologia também apresenta desafios significativos, como a necessidade de desenvolver separadamente para cada sistema operacional, o que pode resultar em maior custo, tempo e esforço de desenvolvimento (Zhou, 2024).

Para ilustrar de forma mais clara as particularidades de cada plataforma, o Quadro 1 apresenta as linguagens de programação utilizadas no desenvolvimento nativo, bem como o principal Ambiente de Desenvolvimento Integrado (IDE) associado a cada uma delas.

Quadro 1 – Linguagens de programação e IDEs utilizadas no desenvolvimento nativo

Sistema Operacional	Linguagem de programação	IDE oficial
Android	Java Kotlin	Android Studio
iOS	Swift Objective-C	Xcode

Fonte: (Zhou, 2024).

2.1.2 Desenvolvimento híbrido

De acordo com Koram e Garg (2023), a criação de aplicativos híbridos é semelhante ao padrão de desenvolvimento *web*, portanto utiliza tecnologias como *HyperText Markup Language* (HTML), *Cascading Style Sheets* (CSS) e JavaScript para a construção de interfaces. Essa abordagem permite que o aplicativo seja suportado por múltiplos sistemas operacionais a partir de uma única base de código.

Aplicações híbridas utilizam páginas *web* responsivas, ou seja, que se adaptam a diferentes tamanhos de tela, embarcadas em um aplicativo móvel. Para isso, essa metodologia combina tecnologias *web* com componentes nativos por meio do uso do *WebView*. O *WebView* é um componente que atua como um navegador embutido dentro do aplicativo, encapsulando o conteúdo *web* em um ambiente nativo, permitindo que interfaces desenvolvidas em HTML, CSS e JavaScript sejam exibidas como parte integrante do aplicativo (Enihe; Joshua, 2020).

Além disso, os aplicativos híbridos podem acessar recursos nativos do dispositivo, como câmera, GPS e sensores, por meio de APIs específicas fornecidas por *frameworks* como o Apache Cordova, que funcionam como pontes entre o código JavaScript e as funcionalidades do sistema operacional (Enihe; Joshua, 2020).

Segundo Souha *et al.* (2024), as vantagens dessa estratégia incluem a possibilidade de reuso do código para múltiplas plataformas, acesso ao hardware do dispositivo e distribuição da aplicação por meio de lojas de aplicativos, como a *Play Store* e *App Store*. Entretanto, apresenta limitações relevantes, como experiência de usuário menos otimizada em virtude da ausência de componentes nativos na interface, e um desempenho inferior em razão da sobrecarga causada pelo uso do *container web*.

2.1.3 Web App

Para Koram e Garg (2023), qualquer aplicativo acessado mediante uma conexão de rede, em vez de estar instalado na memória do dispositivo, é denominado “aplicativo baseado na *web*” (*web based application*). O navegador do dispositivo é utilizado para acessar esse tipo de aplicação, e de forma semelhante ao desenvolvimento híbrido, essa abordagem utiliza tecnologias padrões do desenvolvimento *web*, como HTML, CSS e JavaScript e se configura como um *website* responsivo, acessado por diferentes plataformas.

Apesar do benefício oferecido pela compatibilidade com diferentes dispositivos, aplicativos baseados na *web* possuem desvantagens consideráveis, tais como: incapacidade de uso sem conexão com a rede; acesso limitado às funcionalidades nativas do aparelho; maior tempo de renderização da página *web* em comparação a componentes nativos; e por fim, só são acessíveis mediante um endereço de *website*, não estando disponível na loja de aplicativos do *smartphone*, o que reduz a atratividade dessa solução (Gerges; Elgalb, 2024).

2.1.4 Progressive Web App (PWA)

Introduzido pelo Google em 2015, o termo *Progressive Web App* (PWA) refere-se a um aplicativo *web* mais robusto e com novos recursos suportados por navegadores modernos, como o uso de *service workers*, de maneira a proporcionar uma experiência mais próxima de aplicativos nativos (Tandel; Jamadar, 2018).

De maneira técnica, um PWA é um *website* responsivo servido pelo protocolo *Hypertext Transfer Protocol Secure* (HTTPS), constituído pelos seguintes elementos: um manifesto *web* (arquivo que fornece informações sobre o aplicativo) em formato de *JavaScript Object Notation* (JSON); um *Application Shell*; e um *script* em JavaScript executado em segundo plano denominado *service worker*, que propicia implementar recursos avançados como *cache* programático, experiência sem conexão com a rede e notificações *push* (Gerges; Elgalb, 2024; Majchrzak; Biørn-hansen; Grønli, 2018).

Ao contrário de um *Web App* convencional, que depende exclusivamente de uma conexão com a internet e requer o recarregamento completo do *website* a cada acesso, um PWA é capaz de operar sem conexão com a rede. Isso é possível pois, após o primeiro carregamento do *website*, os arquivos essenciais que compõem a estrutura mínima da interface gráfica e da lógica de navegação, conhecidos como *Application Shell*, são armazenados localmente pelo

service worker. Além disso, um PWA oferece a possibilidade de instalação a partir do navegador, sendo adicionado à tela inicial do dispositivo como um ícone, o que permite ao usuário acessá-lo diretamente (Majchrzak; Biørn-hansen; Grønli, 2018).

2.1.5 Desenvolvimento multiplataforma (*cross-platform*)

Por fim, o desenvolvimento multiplataforma viabiliza o desenvolvimento de aplicativos capazes de serem executados em diferentes sistemas operacionais a partir de um único código-fonte, economizando no tempo e no custo de desenvolvimento, além de manter um desempenho semelhante ao dos aplicativos nativos (Mushtaq; Azam; Anwar, 2024).

Apesar dos benefícios oferecidos por essa abordagem, ela exige lidar com uma camada de abstração fundamental para a sua capacidade multiplataforma, que, em determinados casos, produz gargalos de performance. Além disso, para acessar recursos exclusivos de cada sistema operacional, é necessário recorrer a *plug-ins* ou mecanismos que conectam o código escrito na linguagem do *framework* às APIs nativas do sistema operacional, o que acrescenta complexidade ao projeto (Mushtaq; Azam; Anwar, 2024).

De acordo com Fatkhulin *et al.* (2023) e Zou e Darus (2024), os *frameworks* mais populares que adotam essa metodologia são: Flutter, React Native, Ionic e Xamarin. O Quadro 2 apresenta algumas características desses *frameworks*.

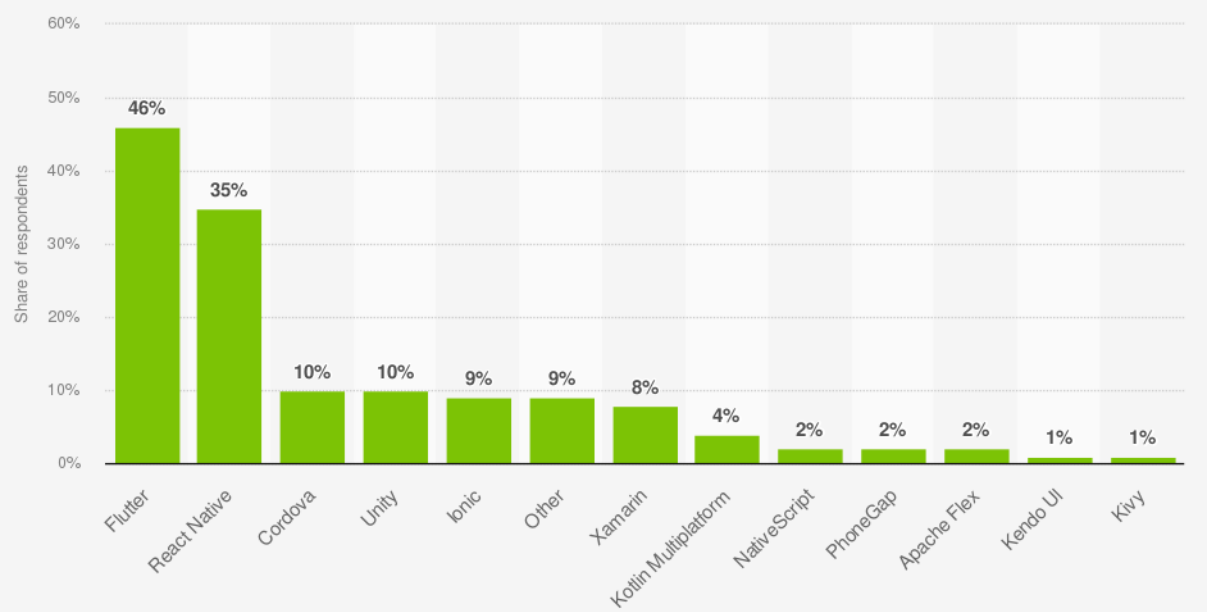
Quadro 2 – Características dos principais *frameworks* multiplataforma

Framework	**Tecnologias utilizadas**	**Modo de renderização**
Flutter	Dart	Utiliza seu próprio motor gráfico (*Impeller* ou *Skia*)
React Native	JavaScript + React	Utiliza componentes nativos da plataforma via JavaScript *Interface* (JSI)
Ionic	JavaScript, HTML, CSS + Angular, React, Vue	Utiliza o *WebView*
Xamarin	C# + .NET	Utiliza APIs nativas para renderização via Xamarin.iOS e Xamarin.Android

Fonte: Adaptado de Zou e Darus (2024).

A Figura 2 apresenta o percentual de utilização dos *frameworks* multiplataforma pelos desenvolvedores em 2023, evidenciando o Flutter e o React Native como as principais tecnologias desse ecossistema, o que também justifica a comparação entre ambas neste trabalho.

Figura 2 – Percentual de utilização de *frameworks* multiplataforma em 2023



Fonte: Statista (2023).

Para sintetizar as diferentes abordagens existentes no desenvolvimento de aplicativos móveis, apresenta-se no Quadro 3 um resumo das principais características de cada abordagem.

Quadro 3 – Síntese das principais abordagens de desenvolvimento móvel

Abordagem	Linguagens / Tecnologias	Características
Nativo	Java, Kotlin (Android); Swift, Objective-C (iOS)	Alto desempenho; acesso total aos recursos do dispositivo; código separado para cada sistema operacional
Híbrido	HTML, CSS, JavaScript (WebView, Cordova)	Código único; acesso parcial a recursos nativos; desempenho moderado
Web App	HTML, CSS, JavaScript	Compatível com vários dispositivos; acesso limitado a recursos nativos; desempenho inferior
PWA	HTML, CSS, JavaScript, Service Workers	Funciona off-line; instalável; acesso parcial a recursos nativos; desempenho semelhante ao *Web App*
Multiplataforma	Dart (Flutter); JavaScript (React Native, Ionic); C# (Xamarin)	Código único; bom desempenho; depende de camadas intermediárias; acesso à maioria dos recursos nativos via *plug-ins*

Fonte: Autoria própria (2025), com base em Zhou (2024), Enihe e Joshua (2020), Gerges e Elgalb (2024), Tandel e Jamadar (2018), Mushtaq, Azam e Anwar (2024).

Nas Seções 2.2 e 2.3 seguintes, são discutidas as características de funcionamento e de arquitetura dos *frameworks* multiplataforma que constituem o objeto de estudo deste trabalho.

2.2 Flutter

O Flutter é um *framework* de código aberto popular desenvolvido pelo Google para criação de aplicações compiladas nativamente para plataformas móveis, *desktop* e *web*. Ele utiliza a linguagem de programação Dart e possui uma arquitetura baseada em *widgets* (Subseção 2.2.4) modulares que funcionam como blocos de construção da aplicação, proporcionando o desenvolvimento de interfaces de usuário complexas de forma mais organizada e ágil (Zou; Darus, 2024).

As subseções a seguir explorarão os aspectos de funcionamento e de arquitetura dessa tecnologia.

2.2.1 Modelo de funcionamento

O funcionamento do Flutter fundamenta-se na execução de quatro *threads*: *Platform thread*, *User Interface* (UI) *thread*, *Raster thread*, *I/O thread* (Flutter, 2025e). A seguir, são detalhadas as funções de cada uma: (1) *Platform thread*: É a *thread* principal do sistema operacional em que a aplicação está sendo executada. É responsável por executar o código dos *plugins* nativos, permitindo que o Flutter acesse e manipule funcionalidades específicas da plataforma em questão; (2) *UI thread*: *Thread* onde todo o código Dart da aplicação é executado. Adicionalmente, essa *thread* gera uma árvore com comandos de desenho (*layer tree*), e posteriormente envia essa árvore para a *Raster thread* para renderização da interface; (3) *Raster thread*: Responsável por converter a *layer tree* em uma interface gráfica mediante o uso dos motores gráficos *Skia* ou *Impeller*. Para isso, ela faz o uso da *Graphic Processing Unit* (GPU), embora a *thread* seja executada na *Central Processing Unit* (CPU); (4) I/O *thread*: *Thread* encarregada de gerenciar as tarefas custosas de entrada/saída.

2.2.2 Dart

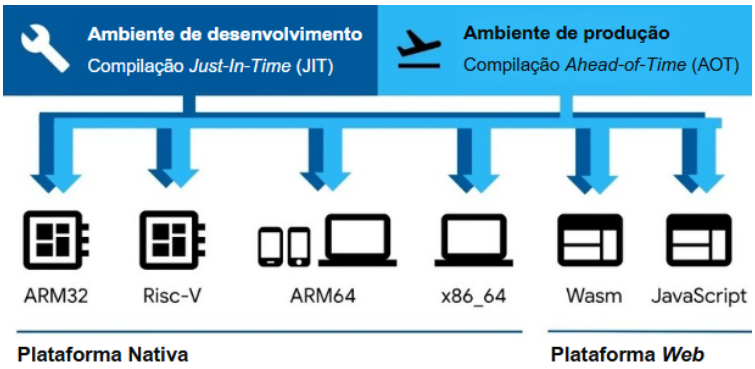
De acordo com Swathiga, Vinodhini e Sasikala (2021), Dart é uma linguagem de programação de propósito geral, criada pelo Google em 2011, e posteriormente padronizada pela *European Computer Manufacturers Association* (ECMA). Trata-se de uma linguagem otimizada para o desenvolvimento de aplicativos rápidos em qualquer plataforma.

Segundo Swathiga, Vinodhini e Sasikala (2021) e a documentação oficial do Dart (2025), a linguagem é orientada a objetos, possui tipagem estática, independência de plataforma pois possui sua própria Máquina Virtual (Dart VM) e disponibiliza um amplo conjunto de bibliotecas para diversas necessidades de programação. Em relação à compilação, o Dart oferece o modo *Just-In-Time* (JIT), utilizado durante o desenvolvimento para permitir o *hot reload*, recurso que atualiza instantaneamente as alterações no código sem reiniciar a aplicação, acelerando o ciclo de desenvolvimento. Já o modo *Ahead-Of-Time* (AOT) é empregado no ambiente de produção

e gera código de máquina nativo (ARM ou x64), proporcionando melhor desempenho em tempo de execução. Para a plataforma *web*, a linguagem também permite compilar o código para JavaScript ou WebAssembly, garantindo compatibilidade com navegadores.

A Figura 3 mostra as possibilidades de compilação do Dart para diferentes plataformas e arquiteturas alvo.

Figura 3 – Formas de compilação e plataformas alvo da linguagem Dart

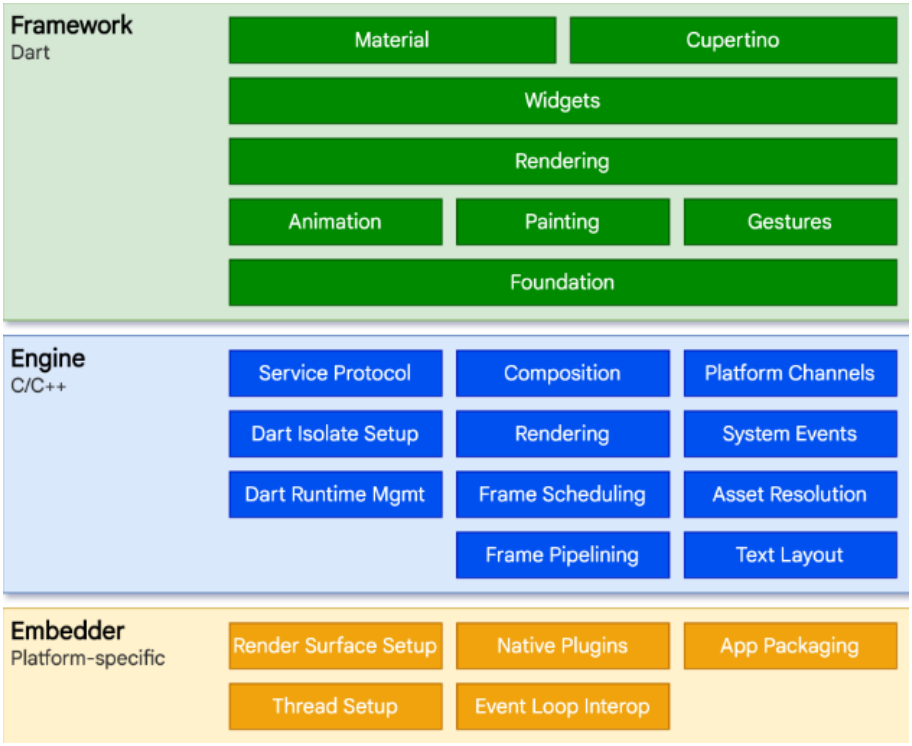


Fonte: Adaptado de Dart (2025).

2.2.3 Arquitetura

De acordo com a documentação oficial do Flutter (2025d), o *framework* é organizado em três camadas principais: *Embedder*, *Engine* e *Framework*. Cada uma desempenha funções distintas e complementares, e juntas garantem que o desenvolvimento com Flutter seja eficiente. A Figura 4 ilustra essa arquitetura, enfatizando que em cada camada, existem subcamadas que desempenham funcionalidades específicas.

Figura 4 – Arquitetura do Flutter



Fonte: Flutter (2025d).

A seguir, são detalhadas as principais funções de cada camada segundo a documentação oficial do Flutter (2025d): (1) *Embedder*: camada de incorporação responsável por integrar o Flutter ao sistema operacional do dispositivo, atuando como ponto de entrada da aplicação. Cada sistema operacional possui sua própria implementação, desenvolvida na linguagem nativa correspondente, além de ser responsável pela criação da janela do aplicativo, gerenciamento de eventos do sistema, fornecimento da integração com recursos nativos da plataforma e controle da superfície de renderização; (2) *Engine*: núcleo do Flutter, escrita em C/C++, encarregada de oferecer os recursos de baixo nível, como rasterização de cenas, renderização via *Impeller* ou *Skia*, layout de texto, entrada/saída de arquivos, rede e arquitetura de plug-ins; (3) *Framework*: camada escrita em Dart, na qual os desenvolvedores interagem, baseada em um modelo reativo a mudanças de estado. Ela abstrai a complexidade das camadas inferiores e fornece *widgets* reutilizáveis para a construção de interfaces, além de suporte a temas, navegação, gerenciamento de estado, animações e gestos.

2.2.4 *Widgets*

Segundo a documentação oficial do Flutter (2025d), *widgets* são componentes visuais da interface da aplicação, como, por exemplo, textos, botões, imagens, caixas de entrada de dados, estruturas de *layout*, entre outros. Os *widgets* são imutáveis, ou seja, uma vez que são criados, eles não podem ser modificados diretamente. Sempre que algo precisa mudar na interface, o Flutter cria um novo *widget* com as novas informações e reconstrói a interface.

Outra característica fundamental dos *widgets* é o princípio da composição, que define a construção da interface por meio do agrupamento de múltiplos *widgets* menores. Esse processo dá origem a uma estrutura hierárquica, conhecida como Árvore de *Widgets* (*Widget Tree*), na qual cada *widget* está aninhado dentro de um *widget* pai. Essa hierarquia se estende até o *widget* raiz da aplicação, como o *MaterialApp*, utilizado em aplicações com o estilo *Material Design* no Android, ou o *CupertinoApp*, adotado em aplicações com aparência nativa do iOS. Sendo assim, um aplicativo em Flutter, como um todo, pode ser entendido como uma estrutura hierárquica de widgets (Flutter, 2025d).

Além disso, os *widgets* podem ser divididos em duas classes principais: *widgets* com estado (*StatefulWidget*) e *widgets* sem estado (*StatelessWidget*). *Widgets* que não possuem estado mutável, ou seja, não possuem propriedades que se alteram durante o seu ciclo de vida, são classificados como *StatelessWidget*. Por outro lado, *widgets* cujas propriedades mudam conforme a interação do usuário, ou outros fatores (como animações ou recebimento de dados), são classificados como *StatefulWidget*. Nesses casos, embora o *widget* em si continue sendo uma estrutura imutável, suas informações mutáveis são armazenadas em uma classe separada, chamada *State*. Essa classe é responsável por gerenciar e atualizar o estado do *widget*, sempre que o estado é alterado, utiliza-se o método *setState*(), que notifica o Flutter para reconstruir a interface, refletindo as novas informações (Flutter, 2025d).

A fim de exemplificar a implementação de *widgets* com e sem estado, criou-se um repositório no GitHub[1] com um exemplo para cada tipo de *widget*.

2.2.5 Renderização

O mecanismo de renderização que o Flutter utiliza em dispositivos móveis, a partir de sua versão 3.27, é o motor gráfico *Impeller*, que substitui o motor anterior conhecido como *Skia*, ainda empregado em outras plataformas. O *Impeller* se destaca por proporcionar um desempenho mais consistente e previsível, uma vez que realiza a pré-compilação de todos os *shaders* (programas que instruem a GPU sobre como desenhar pixels e formas) e objetos de renderização ainda durante o processo de compilação do aplicativo, evitando compilações em tempo de execução. Além disso, o motor gera e gerencia explicitamente o *cache*[2] de recursos gráficos, como texturas e *buffers*[3] de dados gráficos, assegurando maior controle sobre o ciclo de renderização. Foi projetado também para aproveitar APIs gráficas modernas, como Vulkan e Metal, enquanto o *Skia* ainda mantém compatibilidade com APIs mais antigas (Flutter, 2025g).

O processo de renderização ocorre primeiramente pela construção da Árvore de *Widgets* (*Widget Tree*), posteriormente, ela é convertida em uma Árvore de Elementos (*Element Tree*) e, por fim, é transformada na Árvore de Renderização (*Render Tree*) (Flutter, 2025d).

[1] Disponível em: https://github.com/romulo-souza/TCCrepo/tree/main/codigosComponentes

[2] Disponível em: https://azure.microsoft.com/pt-br/resources/cloud-computing-dictionary/what-is-caching

[3] Disponível em: https://www.lenovo.com/pt/pt/glossary/input-buffer/

Conforme mencionado anteriormente na Subseção 2.2.4, a Árvore de *Widgets* é a estrutura declarativa que descreve a interface visual da aplicação. Esses *widgets* imutáveis não possuem lógica de *layout* ou pintura, eles são apenas a declaração do que se deve aparecer na tela (Flutter, 2025d).

Após o Flutter construir os *widgets*, ele os transforma em uma Árvore de Elementos, onde cada elemento representa uma instância correspondente de um *widget*, mantendo a hierarquia da árvore. A Árvore de Elementos desempenha um papel central no processo de reconciliação, permitindo que o Flutter atualize apenas as partes necessárias da interface do usuário, sem recriar toda a estrutura quando uma mudança ocorre. Os elementos mantêm uma referência constante ao seu *widget* associado, e nesse processo de reconciliação, o Flutter compara a nova Árvore de *Widgets* criada com a anterior e decide, de forma eficiente, quais elementos podem ser reaproveitados e quais precisam ser atualizados (Flutter, 2025d).

Finalmente, na última etapa de renderização, a Árvore de Renderização é criada. Essa árvore é composta por objetos denominados *RenderObject*, que se encarregam de lidar com o *layout* e pintura na tela. Primeiramente, o fluxo de renderização passa pelo processo de *layout*, onde cada *RenderObject* calcula seu tamanho e sua posição, propagando essas informações na árvore no sentido de cima para baixo (do pai para os filhos). Depois, vem a fase de pintura, onde cada *RenderObject* será desenhado através de comandos gráficos. Por exemplo, o *RenderParagraph* sabe como medir e desenhar texto, enquanto o *RenderFlex* organiza seus filhos em linhas ou colunas (Flutter, 2025d).

A fim de exemplificar esse processo, apresenta-se a Listagem 1 a seguir:

Listagem 1 – Exemplo de Widget

```dart
class WidgetExemplo extends StatelessWidget {
  const WidgetExemplo({super.key});

  @override
  Widget build(BuildContext context) {
    return Container(
      color: Colors.blue,
      child: Center(child: Text('A')),
    ); // Container
  }
}
```

Fonte: Autoria própria (2025).

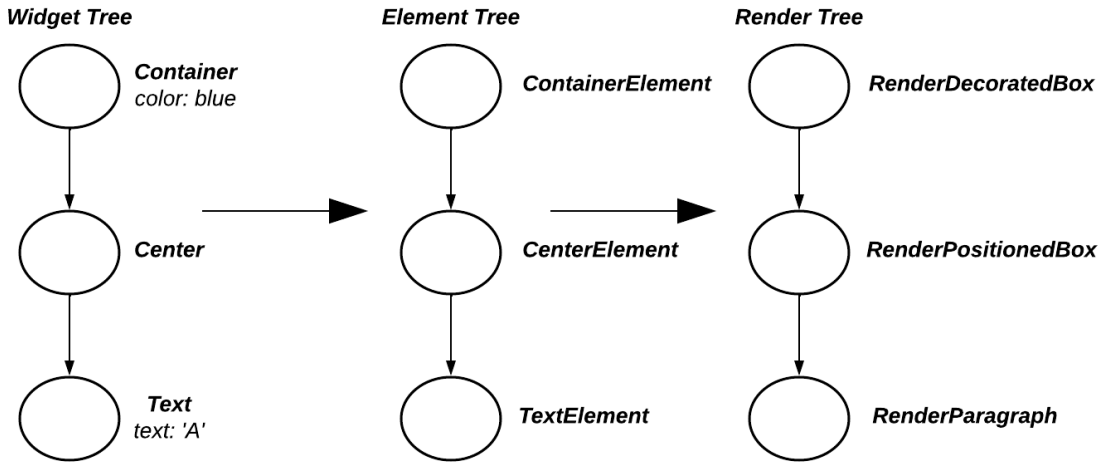
O *widget* desenvolvido para esse exemplo é composto por um texto centralizado e encapsulado por um bloco de cor azul. Ao analisar a estrutura hierárquica de “WidgetExemplo” que originará a Árvore de *Widgets*, observa-se que ele é composto por outros três *widgets*, sendo o *Container* seu filho direto, *Center* como filho do *Container*, e *Text* como filho do *Center*.

Posteriormente, a Árvore de *Widgets* é percorrida, e para cada *widget* presente, é executado o método *createElement*(), o qual gera elementos correspondentes, e assim a Árvore de Elementos é criada (Flutter, 2025i). Por fim, o Flutter percorre a Árvore de Elementos e, para

cada elemento associado a um *widget* de renderização, é chamado o método *createRenderObject*(), criando assim, os *RenderObjects* que compõem a Árvore de Renderização (Flutter, 2025j).

A Figura 5 ilustra cada etapa de renderização e as respectivas árvores geradas para esse exemplo.

Figura 5 – Processo de renderização do Flutter



Fonte: Adaptado de Flutter (2025d).

Em síntese, o Flutter combina eficiência e flexibilidade ao permitir o desenvolvimento multiplataforma a partir de uma única base de código, utilizando *widgets* modulares, reatividade a mudanças de estado e renderização otimizada pelo motor *Impeller*.

2.3 React Native

O React Native, criado pelo Facebook (atualmente Meta), é um *framework* que possibilita a criação de aplicações móveis multiplataforma. Utiliza principalmente a linguagem de programação JavaScript, mas também oferece suporte a TypeScript. O React Native possui uma abordagem baseada em componentes modulares que descrevem a interface gráfica da aplicação, isso possibilita a reutilização de códigos e facilita a manutenção (Zou; Darus, 2024).

As próximas seções abordarão as principais características que compõem o funcionamento e a arquitetura do *framework*.

2.3.1 Modelo de funcionamento

Assim como no Flutter, o React Native também tem seu funcionamento baseado em *threads*. Nesse caso, de acordo com a documentação oficial do React Native (2025a), acerca do modelo de execução em *threads*, as threads são: (1) *UI thread* (*Main thread*): É a *thread* principal do sistema operacional. É responsável por desenhar a interface gráfica nativa e lidar com toques, gestos e animações; (2) JavaScript *thread*: Essa *thread* executa todo o código JavaScript do aplicativo, incluindo a lógica do aplicativo e a manipulação de estado do componente;

(3) *Shadow thread*: *Thread* utilizada para calcular o *layout* da interface, utilizando o motor de *layout* denominado Yoga.

2.3.2 JavaScript

A linguagem JavaScript foi criada em 1995 por Brendan Eich, com a finalidade de tornar as páginas web interativas (Tong; Jikson; Gunawan, 2023). A especificação da linguagem é definida pelo padrão ECMAScript, que recebe atualizações anuais (Mozilla, 2025a).

De acordo com a documentação do JavaScript (2025a), a linguagem é interpretada, possui suporte à orientação a objetos, apresenta tipagem dinâmica, oferece recursos para programação assíncrona e dispõe de um amplo conjunto de bibliotecas.

Para a construção de interfaces de usuário, o React Native, assim como o React, utiliza o JavaScript XML (JSX), uma extensão de sintaxe para o JavaScript, que mistura código JavaScript com uma estrutura semelhante ao HTML. O JSX permite a declaração de componentes que descrevem partes da interface gráfica de um aplicativo (React, 2025b). A Listagem 2 mostra um exemplo de uso do JSX para a construção de uma interface simples em React Native, nesse caso, com um bloco envolvendo um texto.

Listagem 2 – Exemplo de código JSX

```
const App = () => {
  return(
    <View>
      <Text>Hello, World!</Text>
    </View>
  );
}
```

Fonte: Autoria própria (2025).

2.3.3 Arquitetura

A partir da versão 0.76, o React Native passou a utilizar, por padrão, uma arquitetura baseada na JavaScript *Interface* (JSI), que possibilita uma comunicação mais direta com os módulos nativos, substituindo a antiga arquitetura, mostrada na Figura 6, que consistia em um mecanismo de ponte (*Bridge*) assíncrona para realizar a comunicação entre o *framework* e recursos nativos do sistema operacional através de mensagens serializadas no formato JSON, processo que gerava sobrecarga no processamento das mensagens devido aos processos de serialização e desserialização das mensagens, o que torna o modelo menos eficiente que o atual (React Native, 2025a; React Native, 2018).

A JSI é o alicerce da nova arquitetura, a qual está ilustrada na Figura 7, e permite que o JavaScript mantenha referências diretas na memória a objetos em linguagem C++ e invoque seus métodos nativos de forma síncrona, sem os custos da serialização e desserialização de mensagens via ponte assíncrona da antiga arquitetura. Essa arquitetura é composta também

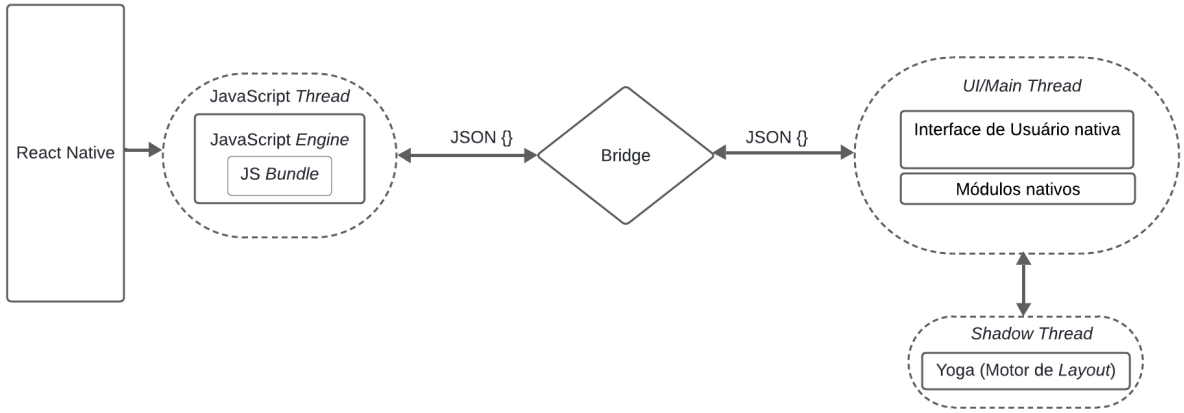
por outros três componentes importantes: o *Turbo Modules*, o *Fabric* e o *Codegen*. Os dois primeiros utilizam a JSI para comunicar-se com a parte nativa do sistema, e o último viabiliza essa comunicação (Korotych, 2024).

O *Turbo Modules* permite a comunicação com módulos nativos do sistema operacional. Esse mecanismo carrega os módulos nativos sob demanda, ou seja, o carregamento desses recursos nativos é feito somente quando necessário, o que reduz o tempo de inicialização de uma aplicação. Para isso, o módulo utiliza a JSI para uma comunicação mais direta e rápida com as funções nativas (Korotych, 2024).

O *Fabric*, por sua vez, é o sistema de renderização do React Native, projetado na linguagem de programação C++ (React Native, 2025c). Um dos pontos principais do *Fabric* é a capacidade de sincronizar a Árvore de Elementos do React com a estrutura da interface gráfica nativa do sistema (Android/iOS), o que, consequentemente, torna o processo de renderização mais eficiente (Korotych, 2024).

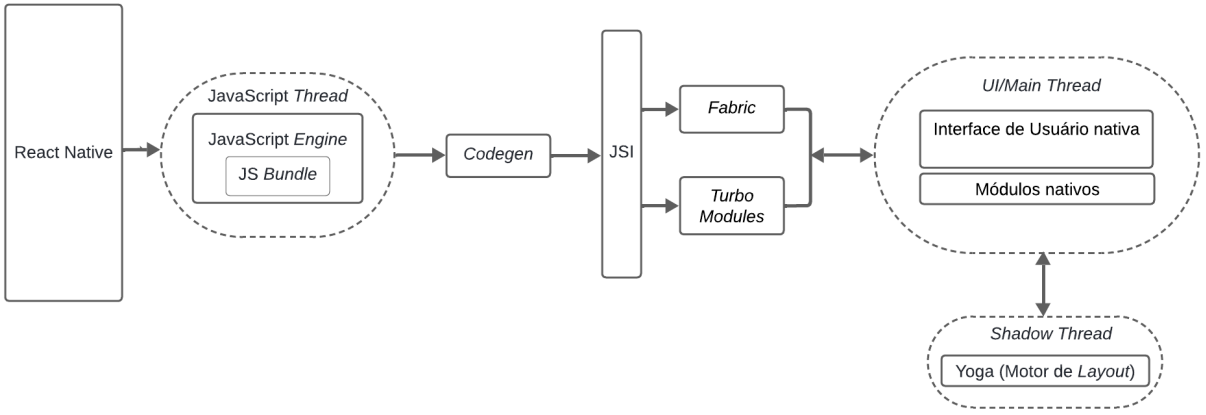
Por fim, para garantir uma comunicação segura e eficiente entre o código JavaScript e a JSI, o *framework* faz o uso do *Codegen*, responsável por gerar automaticamente o código necessário para a integração com a camada em C++ da JSI (React Native, 2025b).

Figura 6 – Diagrama de funcionamento da antiga arquitetura do React Native



Fonte: Adaptado de Korotych (2024).

Figura 7 – Diagrama de funcionamento da nova arquitetura do React Native



Fonte: Adaptado de Korotych (2024).

2.3.4 Componentes

Um componente em React Native é uma unidade fundamental para a construção da interface do usuário. Eles podem ser componentes nativos do React Native (*React Host Components*) ou componentes compostos (*React Composite Components*) (React Native, 2025h).

Componentes nativos são aqueles que já são integrados ao *framework* e disponibilizados para utilização, como por exemplo o *View*, o *Text*, o *Button*, entre outros. Por outro lado, componentes compostos seguem o conceito de composição, pois são componentes que são formados a partir de componentes nativos ou outros componentes compostos, que juntos, descrevem uma interface gráfica maior e seguem uma estrutura hierárquica (React Native, 2025h).

Os componentes podem ser divididos em componentes sem estado e com estado, a depender da forma como lidam com os seus dados internos. Componentes sem estado apenas exibem conteúdos que não sofrem modificações ao longo da execução. Em contrapartida, componentes com estado armazenam dados que podem mudar ao longo do tempo, permitindo que a interface reaja a essas alterações. Os estados são gerenciados em um componente mediante a utilização do *useState* (React Native, 2025g).

Assim como foi referenciado na subseção 2.2.4, os códigos de exemplo de implementação para cada tipo de componente estão disponíveis em um repositório no GitHub[4].

2.3.5 Renderização

De acordo com a documentação oficial do React Native (2025h), a respeito dos processos de renderização, o fluxo de renderização dos componentes passa por três fases: (1) *Render*: Nessa fase, o React executa a lógica da aplicação e cria a Árvore de Elementos do React (*React Element Tree*) em JavaScript. A partir dessa árvore, o *Fabric* gera a Árvore de Sombra do React (*React Shadow Tree*) em C++; (2) *Commit*: Após a criação da *React Shadow Tree*, o *Fabric* dispara um *commit*, que confirma a promoção dessa nova árvore como a próxima a ser montada na interface. Além disso, nessa fase também é efetuado o cálculo do *layout* da interface, pelo motor Yoga; (3) *Mount*: Na última fase, com os cálculos de *layout* já executados, o sistema de renderização *Fabric* transforma a *React Shadow Tree* na *Host View Tree*, que representa a interface gráfica nativa exibida no dispositivo. Nesta fase, cada nó da *React Shadow Tree* é transformado em um componente nativo de acordo com a plataforma utilizada e a montagem ocorre na *thread* principal (*UI thread*).

Com a finalidade de demonstrar as fases de renderização do React Native, utiliza-se novamente a Listagem 2, apresentada na Subseção 2.3.2. Nesse exemplo, “App” é um componente composto, pois é construído a partir de outros componentes nativos do *framework*, nesse caso, pelo *View* e pelo *Text*.

Na fase de *Render*, “App” é reduzido recursivamente até que existam apenas componentes nativos e um nó raiz. Esses componentes são então transformados em elementos (*React*

[4] Disponível em: https://github.com/romulo-souza/TCCrepo/tree/main/codigosComponentes

Elements), que constituem a *React Element Tree*. É importante salientar que o componente composto “App” não é incluído nessa árvore e nem nas árvores subsequentes, pois ele foi reduzido no início dessa etapa. Ainda nessa fase, de forma síncrona, o renderizador *Fabric* cria, para cada *React Host Component*, um *React Shadow Node* correspondente, mantendo a hierarquia. Então, para o *View* é criado o objeto *ViewShadowNode* e para o *Text* o objeto *TextShadowNode*, e assim a *React Shadow Tree* é formada. Por fim, todas as operações da fase *Render* são executadas na JavaScript *thread* (React Native, 2025h).

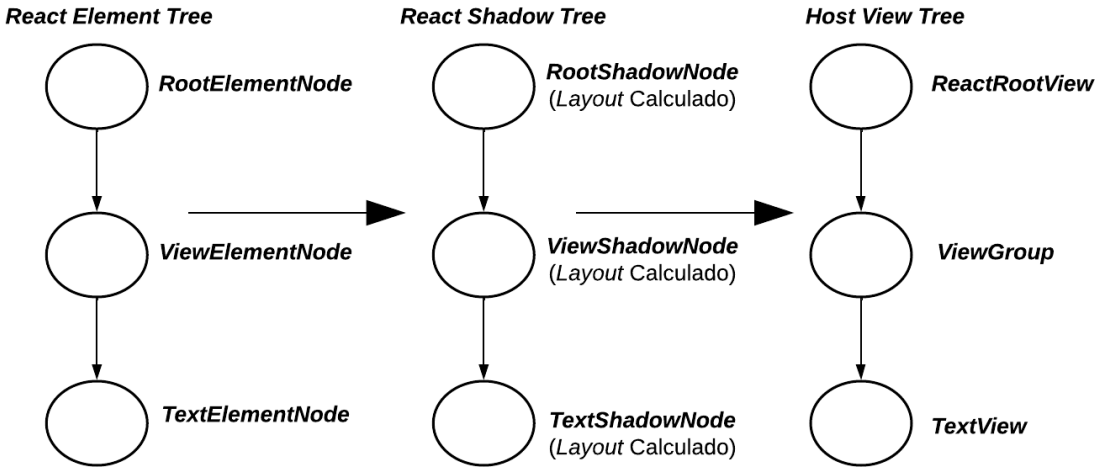
Um ponto importante da *React Shadow Tree* é que ela é uma estrutura imutável. Toda vez que um *React Shadow Node* precisa ser alterado, uma nova árvore é construída. Entretanto, para evitar que esse processo seja lento e custoso, o React Native oferece uma operação de clonagem otimizada, que copia apenas os nós que requerem atualização, já aplicando as mudanças necessárias durante a clonagem. Dessa forma, os nós que não mudaram são reutilizados, evitando a reconstrução completa da árvore (React Native, 2025h).

Em seguida, na fase de *Commit*, o cálculo de posição e tamanho é realizado para cada *React Shadow Node* da *React Shadow Tree*. Esse processo é feito pelo motor de *layout* Yoga. Para a execução desse cálculo, o Yoga necessita das definições de estilos aplicadas em cada *React Shadow Node* via código. Outra operação importante que ocorre nessa fase é a promoção da *React Shadow Tree* como árvore ativa e, consequentemente, a próxima a ser montada na tela. Essas operações ocorrem de forma assíncrona e são executadas na *Shadow thread* (React Native, 2025h).

Por último, na fase de *Mount*, a *React Shadow Tree*, que agora já contém todos os cálculos de *layout* processados, é convertida na *Host View Tree*, a qual é composta por *Host Views* (componentes nativos do Android ou do iOS) correspondentes. Então, para esse exemplo, no Android, é gerada uma instância de android.*view*.*ViewGroup* para o componente *View*, e android.*widget*.*TextView* para o componente *Text*. Similarmente, no iOS, é criada uma *UIView* e o texto é apresentado mediante o *NSLayoutManager*. Por fim, as Host Views são renderizadas no dispositivo. Essa etapa ocorre na *UI thread* (React Native, 2025h).

A Figura 8 demonstra todas as árvores geradas durante o processo de renderização para o exemplo considerado.

Figura 8 – Processo de renderização do React Native



Fonte: Adaptado de React Native (2025h).

Em resumo, o React Native permite a criação de aplicações móveis multiplataforma com uma única base de código, aproveitando componentes modulares, comunicação eficiente com módulos nativos via JSI e renderização otimizada pelo sistema *Fabric* e pelo motor de *layout* Yoga.

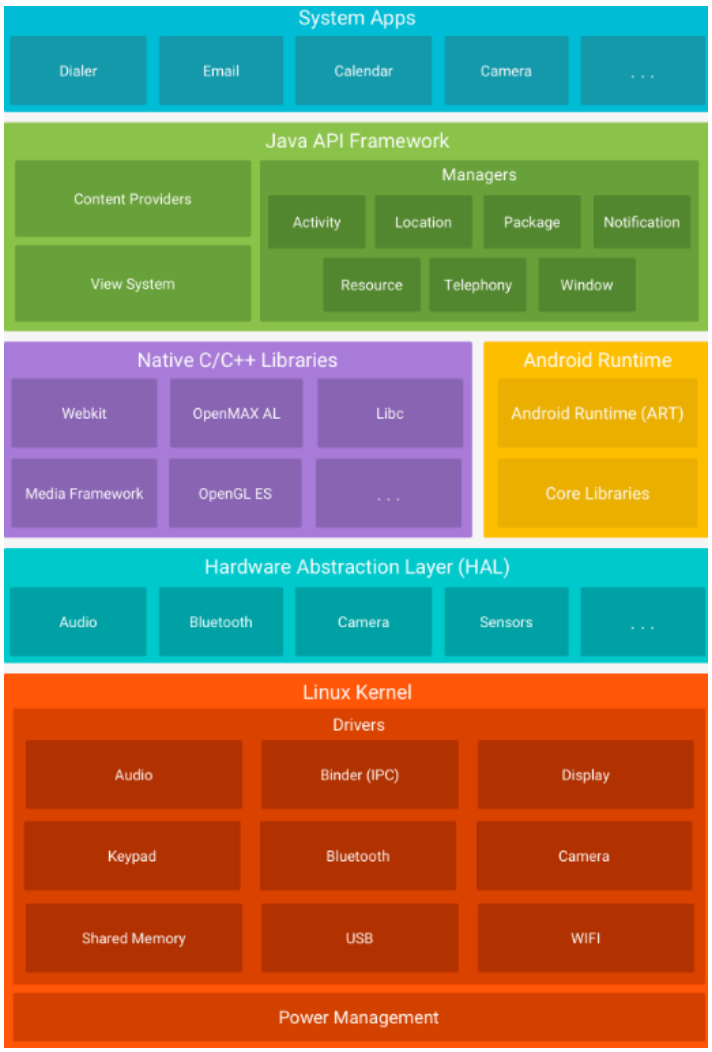
2.4 Android

Embora este estudo tenha como foco a avaliação do desempenho de *frameworks* multiplataforma, o desenvolvimento do estudo de caso foi realizado na plataforma Android, atualmente o sistema operacional mais utilizado no mundo, conforme mostrado na Figura 1 apresentada na Subseção 2.1.1. Assim, faz-se necessário apresentar as principais características desse sistema operacional, bem como a função de um recurso essencial oferecido pela plataforma, denominado *Android Debug Bridge* (ADB).

2.4.1 Visão geral da arquitetura do Android

Em concordância com a documentação oficial do Android (2024b), em relação a sua arquitetura, o Android é baseado no sistema operacional Linux, e sua arquitetura é composta essencialmente por seis componentes: kernel do Linux, camada de abstração de *hardware*, bibliotecas nativas escritas em C/C++, ambiente de execução Android, API Java e aplicativos do sistema. A Figura 9 ilustra esses componentes.

Figura 9 – Componentes da arquitetura do Android



Fonte: Android (2024b).

Na base da arquitetura do Android está o *kernel* (núcleo do sistema operacional) do Linux, o qual fornece funcionalidades essenciais como gerenciamento de memória, gerenciamento de processos, segurança, rede e *drivers* de *hardware* (Android, 2024b).

O próximo componente é o *Hardware Abstraction Layer* (HAL), sendo a camada de abstração de *hardware* responsável por prover uma interface que permite a comunicação entre o *hardware* do dispositivo (como câmera, bluetooth, sensores, entre outros) e a API Java utilizada pelos desenvolvedores de aplicações nativas (Android, 2024b).

No terceiro nível estão as bibliotecas nativas implementadas em C/C++ que são utilizadas por diversos componentes e serviços do sistema. Neste nível está presente também o *Android Runtime* (ART), que representa o ambiente de execução Android. O ART possui alguns recursos como: compilação antecipada (AOT), compilação em tempo de execução (JIT) e otimizações quanto ao gerenciamento de memória (Android, 2024b).

O próximo nível é o Java API *Framework*, que fornece o conjunto dos principais recursos da plataforma Android, disponíveis por meio de APIs escritas em Java. Essa camada segue uma estrutura modular, permitindo que os desenvolvedores criem aplicativos reutilizando os componentes principais do sistema (Android, 2024b).

Por fim, estão os *System Apps*, que são os aplicativos nativos do sistema, ou seja, aqueles que já vêm pré-instalados no dispositivo, como o aplicativo de mensagens, contatos, configurações do sistema, calendário, entre outros (Android, 2024b).

2.4.2 Android Debug Bridge (ADB)

O *Android Debug Bridge* (ADB) é uma ferramenta de linha de comando que torna possível a comunicação entre um dispositivo Android e uma máquina de desenvolvimento, como um computador. O ADB é utilizado para instalar e depurar aplicativos, além de oferecer acesso a um *shell* Unix para a execução de diversos comandos que possibilitam extrair muitas informações relevantes do sistema. Para ser possível a utilização desse recurso, é necessário estar com a opção de depuração *Universal Serial Bus* (USB) ativa nas opções do desenvolvedor, nas configurações do *smartphone* (Android, 2025).

De acordo com a documentação oficial do Android (2025), a respeito do *Android Debug Bridge*, o funcionamento dessa ferramenta é descrito como um programa cliente-servidor, sendo composto por três partes principais: (1) Cliente: É executado no computador do desenvolvedor e envia comandos ao dispositivo Android. Pode ser iniciado por meio do terminal com o comando "adb"; (2) Processo em segundo plano (*daemon*): Esse processo em segundo plano é executado no dispositivo Android e é responsável por executar os comandos recebidos; (3) Servidor: Tem como finalidade gerenciar a comunicação entre o cliente e o processo em segundo plano. Adicionalmente, este servidor funciona em segundo plano no computador do desenvolvedor.

2.5 Métricas de desempenho

Esta seção visa apresentar as métricas de desempenho consideradas neste trabalho.

2.5.1 Taxa de uso do processador

Conforme evidenciado na Subseção 2.4.1, o Android é baseado no Linux. Portanto, diversos comandos executados por meio do ADB compartilham princípios de funcionamento herdados desse sistema operacional.

Nesse cenário, segundo o manual do comando "top" no sistema Linux (2025), esse comando fornece um resumo de informações, em tempo real, de como o sistema está funcionando, apresentando uma lista de processos ou *threads* que estão sendo gerenciados pelo *kernel* do Linux.

Dentre essas informações exibidas pelo comando "top", tem-se o parâmetro "%CPU", o qual exibe a taxa de uso do processador para determinado processo. De forma técnica, esse parâmetro mostra quanto tempo de CPU um processo usou desde a última atualização do comando "top", que por padrão acontece a cada um segundo. Outra característica relevante desta

métrica é que, em processadores com múltiplos núcleos, o valor da utilização de CPU pode ultrapassar 100% em processos *multi-thread*, o que indica que vários núcleos estão sendo utilizados em paralelo, refletindo a carga total distribuída entre todos os núcleos disponíveis (man7, 2025).

2.5.2 Fluidez da interface gráfica

As aplicações feitas em React Native e em Flutter visam manter, no mínimo, a taxa de sessenta quadros (*frames*) renderizados por segundo, a fim de manter uma experiência fluida de utilização para os usuários. Para isso, é necessário que cada quadro seja renderizado em, no máximo, 16.67 milissegundos (ms) (Flutter, 2025e; React Native, 2025f).

Quadros que levam um tempo maior que 16.67 ms para serem renderizados são denominados *janky frames*. Um *janky frame* faz com que o quadro seja atrasado ou até descartado caso o atraso seja significativamente alto, e isso ocasiona pequenos travamentos e instabilidades nas animações da interface, comprometendo a experiência do usuário (Android, 2023; Flutter, 2025e).

No React Native, uma das formas de se obter a quantidade de *janky frames* em um aplicativo é através da ferramenta ADB (Android, 2025). Por outro lado, no Flutter, por utilizar seu próprio motor de renderização chamado *Impeller*, é possível obter essa métrica somente através da ferramenta para análise de desempenho que o *framework* oferece, o Flutter *DevTools* (Flutter, 2025c).

2.5.3 Consumo de memória

Averiguar a quantidade de memória que uma aplicação utiliza é crucial para encontrar problemas de desempenho, como lentidão e comportamentos inesperados, especialmente em dispositivos com pouca memória. No sistema operacional Android, dentre as métricas de memória existentes, uma boa métrica que indica efetivamente a quantidade de memória *Random Access Memory* (RAM) que determinado aplicativo está utilizando é o *Proportional Set Size* (PSS) (Android, 2025).

O PSS é a soma da memória privada de um processo mais a parte proporcional da memória compartilhada que ele utiliza com outros processos. Por exemplo, uma página de memória compartilhada entre dois processos contribui com metade do seu tamanho para o PSS de cada processo. Sendo assim, essa métrica evita a superestimação do consumo de memória RAM e oferece um valor mais preciso para análise desta métrica (Android, 2025).

Além disso, o valor do PSS pode ser obtido por meio da ferramenta ADB (Android, 2025).

2.5.4 Tempo de inicialização

Existem três tipos de inicialização de um aplicativo, o *cold start*, o *warm start* e o *hot start*, cada um é utilizado em um determinado contexto ao abrir uma aplicação (Android, 2024a).

O *cold start* refere-se a uma inicialização do aplicativo do zero, ou seja, não há nenhum processo relacionado a esse aplicativo em execução na memória do sistema naquele momento. Esse tipo de inicialização leva mais tempo e consome mais recursos que o *hot start* e o *warm start*, pois o sistema e a aplicação precisam realizar uma série de etapas do sistema e da aplicação para a conclusão dessa atividade (Android, 2024a).

De acordo com a documentação oficial do Android (2024a), o processo de *cold start* compreende etapas executadas tanto pelo sistema quanto pela aplicação. Inicialmente, o sistema realiza o carregamento do aplicativo e, logo após a sua inicialização, exibe temporariamente uma janela inicial em branco enquanto o processo do aplicativo é criado. Em seguida, a aplicação dá continuidade ao fluxo de inicialização com a criação do objeto principal do aplicativo, da UI *thread* e da *Main Activity*, que representa o ponto de entrada da aplicação e normalmente corresponde à primeira tela exibida ao usuário. Por fim, ocorre o carregamento dos componentes visuais, a aplicação das regras de *layout* e o desenho da interface gráfica na tela.

O *warm start*, por sua vez, pode ocorrer em duas situações: a primeira é quando o usuário sai do aplicativo com o botão “voltar” e, posteriormente, o abre novamente. Nesse caso, o processo relacionado ao aplicativo ainda pode estar na memória, porém, o sistema precisa recriar a *Main Activity*, pois ela foi encerrada no processo de saída da aplicação. A segunda situação ocorre quando o sistema, por algum motivo, remove o aplicativo da memória e, posteriormente, o usuário o inicializa novamente. Nesse caso, o processo e a *Main Activity* precisam ser recriados, mas o sistema pode se beneficiar de algumas recuperações de estado que o Android faz, o que torna essa inicialização mais rápida que um *cold start* (Android, 2024a).

Por fim, o *hot start* ocorre quando um aplicativo previamente aberto, que ainda permanece em segundo plano na memória, é retomado ao primeiro plano para ser utilizado novamente. Nesse tipo de inicialização, não é preciso recriar o processo nem a *Main Activity* relacionada ao aplicativo, apenas retomá-lo do ponto em que parou. Dessa forma, essa inicialização é a mais rápida dentre as três existentes (Android, 2024a).

Ademais, todas as métricas de inicialização podem ser obtidas através da ferramenta ADB (Android, 2024a).

2.5.5 Tempo de renderização da interface gráfica

Ambos os *frameworks* analisados neste trabalho oferecem meios para obter o tempo que uma interface gráfica demanda para ser renderizada.

No React Native, assim como no React, é possível obter o tempo de renderização da interface programaticamente. Isso é possível mediante o uso de um componente especial denominado *Profiler*. Esse componente possui duas propriedades, um identificador do tipo *string*

que identifica de forma única a parte da interface que está sendo analisada, e uma função de *callback* denominada *onRender* que é executada sempre que ocorre a renderização de um elemento dentro do *Profiler* (React, 2025a).

O parâmetro *actualDuration* da função *onRender* mostra o tempo, em milissegundos, que foi gasto para a renderização de determinada parte da interface gráfica da aplicação (React, 2025a).

Já para o Flutter, o método *addPostFrameCallback* permite que os desenvolvedores registrem uma função de *callback* que será invocada após o quadro atual ser completamente renderizado, ou seja, após a interface gráfica estar totalmente renderizada (Flutter, 2025h). Sendo assim, é possível instrumentar a medição desta métrica via código-fonte.

2.5.6 Latência de entrada

De acordo com Saarinen (2019), a latência de entrada é o intervalo de tempo entre o momento em que o usuário realiza uma interação na tela, como, por exemplo, pressionar um botão, e o instante em que a interface gráfica responde visualmente a essa ação. Ainda segundo o autor, uma forma de medir a latência de entrada em uma aplicação é por meio da instrumentação diretamente no código-fonte, capturando o tempo necessário para que uma determinada ação seja refletida na interface.

2.5.7 Tempo de resposta de requisições a uma API

Quando uma requisição é realizada a uma API que funciona sobre o protocolo *Hypertext Transfer Protocol* (HTTP), obtém-se uma resposta por parte do servidor a essa requisição, normalmente no formato JSON. O tempo de resposta de uma requisição refere-se ao intervalo de tempo decorrido entre o momento em que o cliente realiza a requisição e o momento em que o cliente recebe a resposta do servidor (Prayogi *et al.*, 2020).

No React Native, uma das possíveis maneiras para realizar requisições HTTP é através da *Fetch* API, que provê uma interface em JavaScript para realizar essas requisições e processar as respostas recebidas do servidor (React Native, 2025e; Mozilla, 2025b).

De forma semelhante, o Flutter oferece essa funcionalidade por meio de sua biblioteca denominada http, a qual adota uma abordagem própria para o envio e tratamento das requisições HTTP de forma eficiente (Flutter, 2025b).

3 TRABALHOS RELACIONADOS

Neste capítulo, apresentam-se três trabalhos relacionados que foram analisados, considerando o escopo do presente trabalho.

Kishore *et al.* (2022) conduziram um estudo comparativo entre Flutter e React Native nos quesitos desempenho e estabilidade a partir do desenvolvimento de um aplicativo. Nessa ocasião, a aplicação executou no sistema operacional Android e os autores consideraram as seguintes métricas de desempenho: espaço de armazenamento utilizado pelo aplicativo, consumo da memória *Random Access Memory* (RAM), tempo de inicialização do aplicativo, consumo de bateria, e comparações do *hot reload* presente nos dois *frameworks*.

Apesar dos autores apenas mencionarem que foi realizado um experimento prático em três dispositivos diferentes, não foi especificado como as métricas foram extraídas. No entanto, os resultados obtidos estão bem documentados.

Durante o experimento, foi constatado que a aplicação em React Native exigiu mais espaço de armazenamento, consumiu mais memória RAM e drenou mais bateria do que a aplicação desenvolvida em Flutter. Nesse caso, os resultados foram os mesmos nos três dispositivos testados. Entretanto, para a métrica de tempo de inicialização da aplicação, o React Native obteve vantagem sobre o Flutter, inicializando o aplicativo mais rápido em dois dos três dispositivos utilizados. Já para a métrica de comparação do *hot reload*, o Flutter obteve uma pequena vantagem, recarregando a interface mais rapidamente após mudanças no código. Por fim, os autores apresentam um ponto de vista relacionado à tecnologia que eles consideraram mais fácil para o desenvolvimento da aplicação, sendo esta o React Native.

Outro trabalho considerado foi o dos autores Markowski e Smołka (2023), que também realizaram uma análise comparativa do uso de recursos no Flutter e no React Native. Para esse estudo, desenvolveu-se em cada *framework*, um aplicativo simples com as mesmas funcionalidades, sendo executado no sistema operacional Android (versão 9) do *smartphone* Huawei P20 Lite. As métricas consideradas foram: consumo da memória RAM, uso da memória compartilhada e taxa de uso da CPU. As métricas desse estudo foram obtidas utilizando a ferramenta *Android Debug Bridge* (ADB) e a coleta foi automatizada através de *scripts* implementados em Bash. Nesse trabalho, o comando ADB utilizado para extrair as métricas foi o “adb shell top”.

Em relação à taxa de uso da CPU, concluiu-se que o aplicativo em Flutter exigiu menos do processador, atingindo em média 97% de uso, já o React Native utilizou em média 130%. Em relação ao uso de memória RAM e memória compartilhada, o React Native foi um pouco mais eficiente que o Flutter, consumindo aproximadamente 20 Megabytes (MB) a menos nos dois parâmetros.

Por fim, Fentaw (2020), em sua dissertação de mestrado, também compara os dois *frameworks* em questão de desempenho tanto no Android quanto no iOS. Neste trabalho, o foco foi analisar o consumo de CPU, fluidez da interface gráfica e utilização da memória RAM. Um aplicativo desenvolvido pelo autor de forma similar nas duas tecnologias foi utilizado como caso de teste.

Considerando a plataforma iOS, utilizou-se a ferramenta *Instruments* que é integrada ao Xcode IDE e possibilita visualizar as métricas de desempenho consideradas por Fentaw (2020). Os resultados obtidos nessa plataforma apontam que o React Native utilizou mais CPU que o Flutter em todos os testes realizados, entretanto, o Flutter consumiu mais memória. Em relação à fluidez da interface gráfica, nesse caso medida pela taxa de quadros renderizados por segundo, o Flutter obteve uma pequena vantagem sobre o React Native nos testes em geral.

De forma semelhante, no sistema Android, a coleta das métricas de desempenho, com exceção da taxa de quadros, foi realizada pelo *Android Studio Profiler*.

Em relação à utilização de CPU e memória no sistema operacional Android, em alguns testes o React Native consumiu mais, enquanto em outros o Flutter apresentou maior consumo. Desta forma, Fentaw (2020) concluiu que ambos os *frameworks* gerenciaram o uso de CPU e memória de maneira igualmente eficiente.

Para a métrica de fluidez da interface gráfica no Android, o autor utilizou a ferramenta *Profile GPU Rendering* (ou *Profile HWUI rendering*, a depender da versão do Android) que é habilitada no modo desenvolvedor do sistema. Essa ferramenta apresenta na tela do dispositivo um histograma em que cada barra vertical representa o tempo que um *frame* levou para ser renderizado em milissegundos (ms). Como referência, há uma linha verde horizontal que representa 16.67 ms, o tempo máximo que um *frame* pode levar para ser renderizado a fim de atingir a taxa de sessenta quadros renderizados por segundo, que representa bom desempenho. Portanto, as barras que ficarem abaixo dessa linha implicam em boa fluidez de interface, enquanto as que ultrapassarem a linha sugerem a ocorrência de *janky frames*, resultando em leves travamentos.

De acordo com Fentaw (2020), a ferramenta *Profile GPU Rendering* não estava disponível no Flutter, e o autor optou por não utilizar o Flutter *DevTools*, ferramenta específica do *framework* que possibilita obter informações de desempenho e realizar depuração, pois a ferramenta ainda estava em desenvolvimento ativo, portanto, o autor não considerou viável utilizá-la naquele momento. Isso impossibilitou uma comparação direta nessa métrica, no entanto, ele executou os testes no React Native e observou que algumas barras verticais ficaram acima da linha de referência, porém, a maioria ficou abaixo, indicando um bom desempenho geral para a fluidez da interface.

O Quadro 4 apresenta uma síntese comparativa entre os trabalhos relacionados considerados e o presente trabalho, com base nas métricas de desempenho avaliadas por cada um.

Quadro 4 – Comparação das métricas de desempenho consideradas nos trabalhos relacionados e no presente trabalho: (1) Kishore *et al.* (2022); (2) Markowski e Smołka (2023); (3) Fentaw (2020)

	Trabalhos			
Métricas avaliadas	1	2	3	Presente trabalho
Taxa de uso do processador		X	X	X
Fluidez da interface gráfica			X	X
Consumo de memória	X	X	X	X
Tempo de inicialização	X			X
Tempo de renderização da interface gráfica				X
Latência de entrada				X
Tempo de resposta de requisições a uma API				X
Espaço de armazenamento utilizado pelo aplicativo	X			
Tempo de *hot reload* após mudanças no código	X			
Consumo de bateria	X			

Fonte: Autoria própria (2025).

4 METODOLOGIA

A metodologia de pesquisa deste trabalho se caracteriza como uma pesquisa experimental e quantitativa. De acordo com Guerra (2023), a pesquisa experimental tem como característica principal a manipulação direta das variáveis relacionadas ao objeto de estudo, permitindo estabelecer de forma objetiva a relação causal entre variáveis independentes e os efeitos que produzem sobre as variáveis dependentes. No caso do presente trabalho, observa-se como variáveis independentes os *frameworks* Flutter e React Native, e como variáveis dependentes as métricas de desempenho a serem analisadas.

Constata-se, ainda, que este estudo possui uma abordagem quantitativa, pois envolve a coleta e a análise de dados numéricos mensuráveis, visando comparar o desempenho entre os dois *frameworks*.

Para atender aos objetivos definidos nesse trabalho, descritos na Seção 1.3, foi implementado um estudo de caso, o qual está detalhado nas seções subsequentes deste capítulo.

4.1 Ferramentas utilizadas

Para o desenvolvimento do trabalho, foram utilizadas as seguintes tecnologias:

1. *Frameworks*: Flutter (versão 3.29.3) e React Native (versão 0.81.1);
2. IDEs: O Visual Studio Code foi utilizado para a implementação dos aplicativos e *scripts*, enquanto o Google Colab foi empregado para a análise dos dados;
3. Computador: Um Notebook foi utilizado para realização deste trabalho. As principais especificações técnicas desse computador são apresentadas no Quadro 5.

Quadro 5 – Principais especificações do Notebook utilizado

Processador	Intel i5-7300HQ *Quad-Core* (4x2.5GHz)
Memória RAM	16 GB
Armazenamento	466 GB
Versão do Windows	10

Fonte: Autoria própria (2025).

4. Ferramentas para coleta das métricas de desempenho: *Android Debug Bridge (ADB)*, Flutter *DevTools* e códigos em linguagem Python para coleta automatizada e análise dos dados.
5. Ferramenta para simular a utilização da aplicação: *Scripts* em PowerShell compostos por comandos do ADB para simular interações com o aplicativo em um dispositivo Android.

6. Dispositivo Android: O dispositivo utilizado para a execução das aplicações em cada caso de teste foi o Samsung Galaxy A55. O Quadro 6 mostra as especificações técnicas mais relevantes desse *smartphone*.

Quadro 6 – Principais especificações do *smartphone* utilizado

Processador	Samsung Exynos 1480 *Octa-Core* (4x2.75GHz 4x2.0GHz)
Memória RAM	8 GB
Armazenamento	256 GB
Versão do Android	15

Fonte: Samsung (2024).

4.2 Estudo de caso

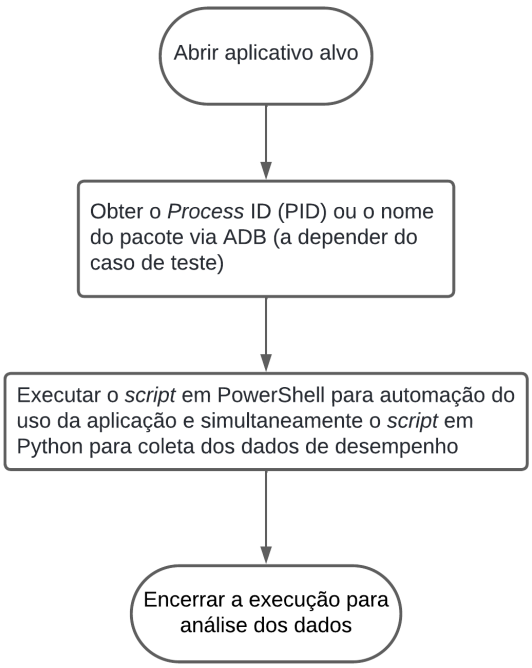
O estudo de caso deste trabalho é composto por sete casos de teste, cada um foi projetado para avaliar uma métrica de desempenho específica. Os casos de teste consistiram em aplicações simples, com interfaces e funcionamentos projetados para avaliar especificamente determinada métrica. Além disso, as aplicações foram implementadas de forma equivalente em ambos os *frameworks* considerados neste trabalho, a fim de proporcionar uma comparação justa e objetiva dos resultados obtidos para cada métrica.

É importante destacar que para garantir uma análise comparativa padronizada e justa entre os dois *frameworks*, a utilização das aplicações desenvolvidas foi automatizada para todos os casos de teste. Dessa forma, a velocidade das interações com os elementos da aplicação ocorreu de forma igual em ambos os aplicativos desenvolvidos, evitando distorções que o uso manual poderia ocasionar. Para isso, foram implementados *scripts* em PowerShell que executam comandos do ADB referentes a gestos, como toques, deslizes na tela, entre outros, os quais foram executados concomitantemente com os códigos responsáveis pela captura dos dados de desempenho que serão destacados nas próximas subseções. Além disso, cada aplicação foi executada de forma isolada, sem outras aplicações em segundo plano, a fim de evitar interferências externas. Por fim, todos os códigos desenvolvidos para a realização deste trabalho estão disponíveis em um repositório[1] público no GitHub.

A fim de detalhar o fluxo de execução dos casos de teste, tanto nos que tiveram a coleta das métricas automatizadas por *scripts* em Python (casos 1, 2, 3 e 4) quanto naqueles em que a coleta foi realizada por meio da instrumentação diretamente no código-fonte (casos 5, 6 e 7), apresentam-se, nas Figuras 10 e 11, os fluxogramas representando as etapas executadas.

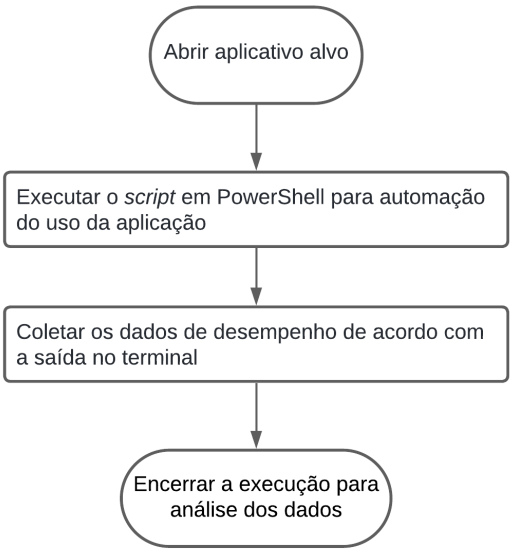
[1] Disponível em: https://github.com/romulo-souza/TCCrepo/tree/main

Figura 10 – Fluxograma para execução dos casos de teste cuja coleta das métricas foi automatizada



Fonte: Autoria própria (2025).

Figura 11 – Fluxograma para execução dos casos de teste cuja coleta das métricas foi instrumentada no código-fonte



Fonte: Autoria própria (2025).

Na Listagem 3, apresenta-se o comando ADB utilizado para obter o nome do pacote de uma aplicação, em conjunto com um filtro (*findstr*) que busca pelo nome da aplicação.

Listagem 3 – Comando do ADB com filtro para obter o pacote de um aplicativo

```
adb shell pm list packages | findstr <nome_da_aplicação>
```

Fonte: Adaptado de Android (2025).

O PID referente a uma aplicação foi obtido por meio do comando apresentado na Listagem 4.

Listagem 4 – Comando do ADB para obter o PID de um processo

```
adb shell pidof <nome_do_pacote>
```

Fonte: Adaptado de Android (2025).

As subseções seguintes detalham os processos de desenvolvimento dos casos de teste, bem como a obtenção e a análise dos dados referentes às métricas de desempenho consideradas.

4.2.1 Caso de teste 1: Taxa de uso do processador

Para este caso de teste, desenvolveu-se uma aplicação composta por elementos gráficos dispostos em uma lista vertical com *layout* de duas colunas e suporte a rolagem (*scroll*) vertical. Cada elemento da lista consistia em um *card*, que continha um texto fixo e dois tipos de elementos atualizáveis de forma aleatória: uma imagem e três ícones. Para avaliar o impacto da frequência de atualização desses elementos no consumo do processador devido às re-renderizações, foram conduzidos três experimentos independentes, utilizando os seguintes intervalos de tempo para a alteração dos elementos: a cada 400 ms, 600 ms e 800 ms.

Para a construção da interface em React Native, utilizou-se o componente *Card* da biblioteca *React Native Paper*, a qual segue os padrões do *Material Design* (sistema de *design* criado pelo Google) (React Native, 2025b; Google, 2025). Na aplicação em Flutter, utilizou-se o *widget* nativo referente ao *Card*, o qual já segue o padrão *Material Design* (Flutter, 2025a). O código-fonte desta aplicação em cada *framework* encontra-se disponível no repositório[2] dedicado a este trabalho.

Para a disposição das imagens nos *cards*, foram selecionadas imagens de diferentes resoluções, com o intuito de verificar o efeito que isso gera na renderização da aplicação e no consumo do processador. As resoluções escolhidas foram: 640x960, 1280x1920 e 1920x2880 pixels. Ao todo, a lista foi preenchida com trinta *cards*, sendo dez imagens para cada resolução. Ademais, o site utilizado para a obtenção das imagens sem direitos autorais foi o Pexels[3], sendo armazenadas localmente.

Já para os ícones, foram escolhidos sete ícones do conjunto de ícones do Google (*Material Icons*) para se alternarem entre si, sendo que no Flutter foram obtidos por meio de sua biblioteca nativa *material.dart*[4], e no React Native mediante a biblioteca *react-native-vector-icons*[5], utilizando o conjunto de ícones *Material Icons*.

[2] Disponível em: https://github.com/romulo-souza/TCCrepo/tree/main/apps/cpu
[3] Disponível em: https://www.pexels.com/pt-br/
[4] Disponível em: https://api.flutter.dev/flutter/material/Icons-class.html
[5] Disponível em: https://github.com/oblador/react-native-vector-icons

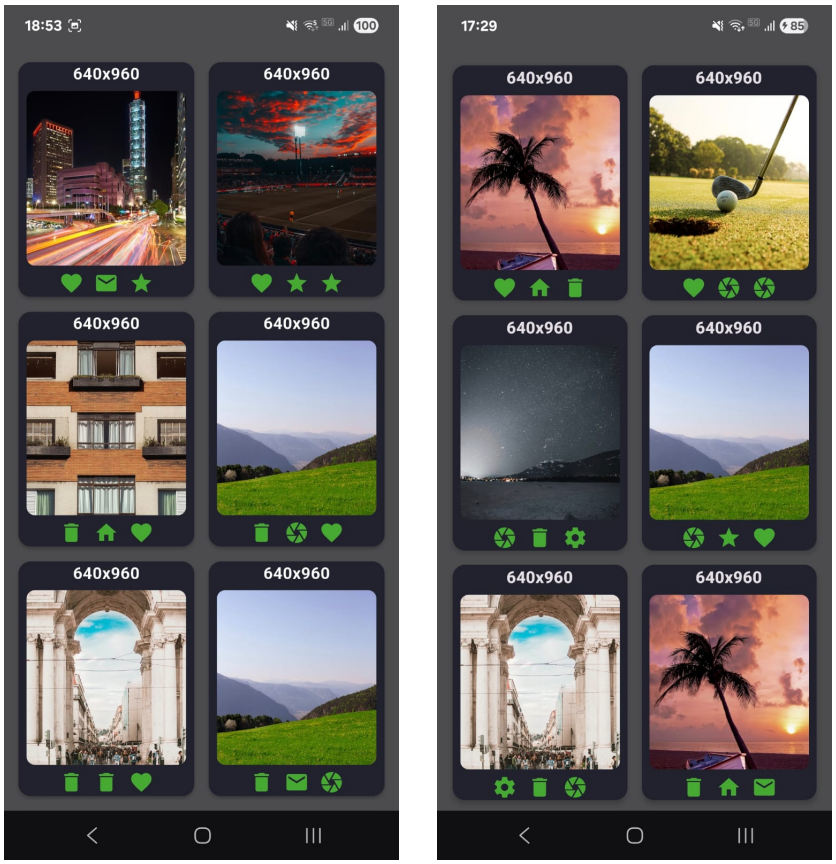
No React Native, a lista foi implementada utilizando o componente *FlatList*, enquanto que no Flutter utilizou-se o *widget GridView.builder*. Em ambos os *frameworks*, o desempenho do aplicativo é otimizado, uma vez que renderizam somente os itens visíveis na tela, evitando o processamento desnecessário de renderizar todos os elementos da lista de uma única vez (React Native, 2025d; Flutter, 2025f).

O processo da rápida alternância de estados visuais das imagens e ícones, combinado com a rolagem contínua da tela, intensifica o uso do processador, uma vez que exige que o sistema recalcule constantemente o *layout* dos componentes e gerencie a renderização. Isso eleva a carga de operações de cálculo e pintura na interface, aumentando os esforços gerados sobre as *threads* dos dois *frameworks*, conforme foram descritas nas Subseções 2.2.1 e 2.3.1.

A Figura 12 apresenta uma captura de tela das aplicações desenvolvidas em cada plataforma. Cabe destacar que, no momento das capturas de tela, as imagens exibidas nos *cards* variaram aleatoriamente entre os *frameworks*. Ainda que isso não altere o procedimento experimental, essa diferença visual pode introduzir pequenas variações no processo de renderização e, portanto, constitui uma limitação do experimento.

Figura 12 – Caso de teste 1: (a) Aplicação em Flutter, (b) Aplicação em React Native

(a) **(b)**



Fonte: Autoria própria (2025).

Para extração dessa métrica, o seguinte comando apresentado na Listagem 5 foi utilizado:

Listagem 5 – Comando do ADB para monitorar o uso de recursos de um processo específico

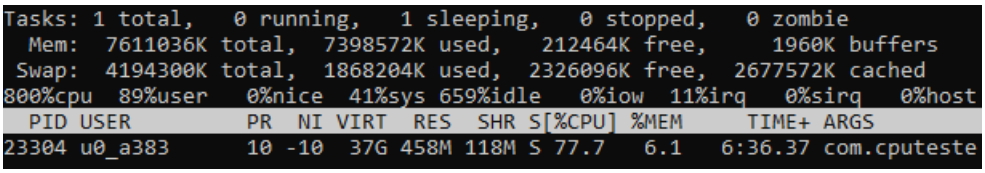
```
adb shell top -p <PID> -n 1
```

Fonte: Adaptado de Android (2025).

Este comando é executado apenas uma vez por conta do parâmetro “-n 1” e está relacionado a um processo em específico que se refere a uma aplicação, identificado pelo seu PID.

Um exemplo de saída do comando apresentado na Listagem 5 pode ser observado na Figura 13.

Figura 13 – Exemplo de saída para o comando da Listagem 5



Fonte: Autoria própria (2025).

Para simular o uso da aplicação deste caso de teste em cada plataforma, os seguintes códigos[6] em PowerShell foram executados: “`swipe_loop_Flutter.ps1`” e “`swipe_loop_ReactNative.ps1`”.

Para automatizar a coleta dos dados e realizar suas análises, foram desenvolvidos dois *scripts*[7] em linguagem Python. O *script* denominado “`scriptCapturaCpu.py`” faz uso do módulo *subprocess*, o qual permite executar comandos do sistema operacional, nesse caso comandos do ADB, e capturar suas saídas diretamente a partir do código-fonte em Python (Python, 2025). Esse código executa o comando apresentado na Listagem 5 e extrai especificamente o valor do campo “%CPU”, sendo executado a cada 0,5 segundos (devido ao tempo de execução do comando). Para cada cenário, considerando todos os períodos de atualização dos elementos utilizados, foram executados trinta experimentos, cada um com duração de trinta segundos de utilização da aplicação. Ao final, foi gerado um arquivo no formato *Comma Separated Values* (CSV) contendo todos os dados obtidos, composto pelos campos: experimento, tempo e consumo de CPU.

Desta forma, no segundo *script*, denominado “`analiseCpu.ipynb`”, utilizou-se a biblioteca Pandas[8] para realizar a análise dos dados presentes em cada arquivo CSV gerado. Nesse sentido, agruparam-se os valores de consumo de CPU dos experimentos por tempo, de forma a obter o consumo médio do processador em cada instante de tempo, ao longo do período de trinta segundos. A partir dessa série de valores médios, identificou-se o pico (maior valor médio observado), o mínimo (menor valor médio observado), a partir dos quais foi determinada a variação do consumo médio de CPU. Além disso, calculou-se a média geral da taxa de utilização da CPU considerando todos os registros coletados.

[6] Disponível em: https://github.com/romulo-souza/TCCrepo/tree/main/Scripts_PowerShell
[7] Disponível em: https://github.com/romulo-souza/TCCrepo/tree/main/Scripts_Python/Consumo_CPU
[8] Disponível em: https://pandas.pydata.org/

Ao final desse código, mediante o uso da biblioteca Plotly[9], representou-se graficamente o consumo médio de CPU em cada instante de tempo. Nesse gráfico, o eixo vertical representa a porcentagem de consumo do processador, enquanto o eixo horizontal exibe o tempo decorrido em segundos.

A fim de resumir as informações deste caso de teste, apresenta-se no Quadro 7 os principais aspectos considerados.

Quadro 7 – Principais aspectos para o Caso de teste 1

Aspecto	Descrição
Períodos de atualização do conteúdo do *card*	A cada 400 ms, 600 ms e 800 ms
Quantidade de *cards*	30
Resoluções das imagens	640×960, 1280×1920 e 1920×2880 px
Ícones utilizados	7 ícones do conjunto *Material Icons*
Implementação das listas	Flutter: *GridView.builder* React Native: *Flatlist*
Quantidade de experimentos	30
Tempo de execução de cada experimento	30 s
Tempo de aquisição dos dados pelo *script* em Python	A cada 0,5 s
Quantidade de dados coletados para cada cenário	1800

Fonte: Autoria própria (2025).

4.2.2 Caso de teste 2: Fluidez da interface gráfica

A aplicação desenvolvida para este caso de teste consistiu em uma interface gráfica organizada em um *layout* de três colunas, contendo dezoito animações simultâneas no formato *Lottie*, executadas de forma contínua em *loop*[10]. O objetivo foi gerar uma carga significativa de renderização gráfica a partir dessas animações, a fim de avaliar a fluidez da interface em cada *framework*.

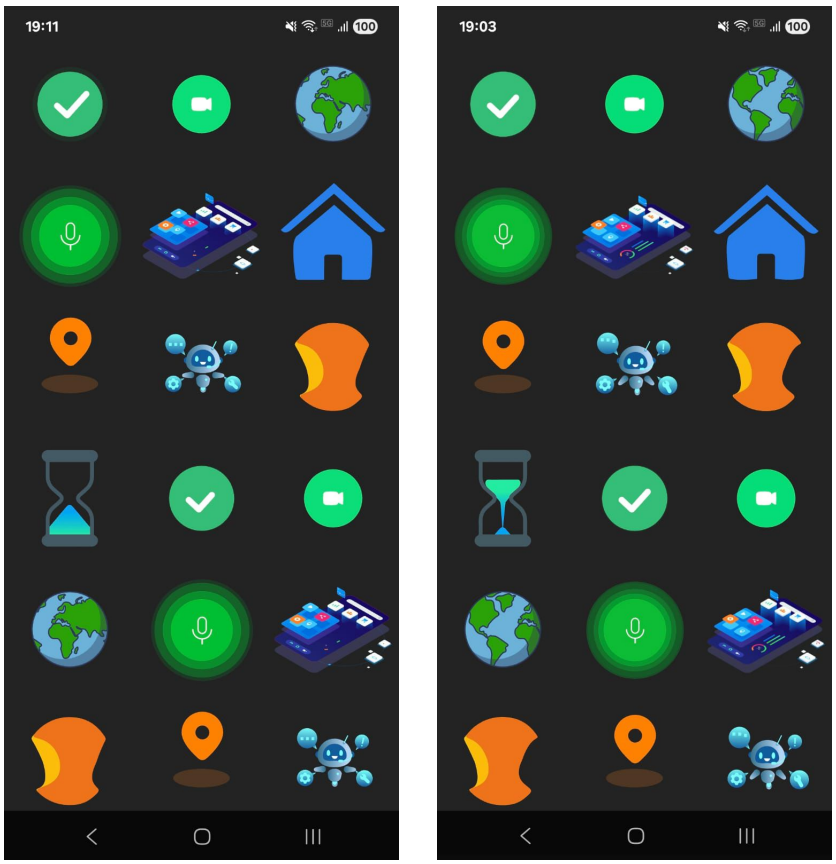
Arquivos *Lottie* são animações compactas, ocupando pouco espaço de armazenamento e de fácil integração, codificados em formato JSON, sendo disponibilizados em plataformas como o *LottieFiles* (LottieFiles, 2025). No Flutter o seu uso é possível mediante a biblioteca *lottie*[11], enquanto no React Native utiliza-se a biblioteca *lottie-react-native*[12]. O código-fonte completo desta aplicação em cada plataforma está disponível no repositório[13] deste trabalho.

A Figura 14 exibe a aplicação desenvolvida em cada *framework*.

[9] Disponível em: https://plotly.com/python/
[10] Disponível em: https://www.w3schools.com/programming/prog_loops.php
[11] Disponível em: https://pub.dev/packages/lottie
[12] Disponível em: https://github.com/lottie-react-native/lottie-react-native
[13] Disponível em: https://github.com/romulo-souza/TCCrepo/tree/main/apps/fluidezUI

Figura 14 – Caso de teste 2: (a) Aplicação em Flutter, (b) Aplicação em React Native

(a) **(b)**



Fonte: Autoria própria (2025).

A Listagem 6 exibe o comando utilizado para a aquisição de informações relacionadas ao desempenho gráfico de determinada aplicação, especificada pelo nome do seu pacote. Dentre essas informações, encontra-se a métrica alvo deste caso de teste: o *janky frames*. Este comando foi utilizado para a coleta desta métrica na aplicação desenvolvida em React Native.

Listagem 6 – Comando do ADB para obtenção de informações gráficas de um aplicativo

```
1 adb shell dumpsys gfxinfo <nome_do_pacote>
```

Fonte: Adaptado de Android (2025).

É importante salientar que os *janky frames* contabilizados pela ferramenta ADB são acumulativos durante a utilização de uma aplicação até que seu processo seja encerrado. Portanto, é importante que, antes de iniciar a coleta desta métrica, o seu valor atual seja restaurado para zero. Para isso, adicionou-se a palavra "*reset*" logo após o comando exibido na Listagem 6 antes de iniciar a aplicação.

Um exemplo do resultado obtido por este comando para a métrica alvo é mostrado na Figura 15.

Figura 15 – Exemplo de saída para o comando da Listagem 6

```
C:\Users\User>adb shell dumpsys gfxinfo com.instagram.android
Applications Graphics Acceleration Info:
Uptime: 200819110 Realtime: 259181337

** Graphics info for pid 12845 [com.instagram.android] **

Stats since: 200743122499374ns
Total frames rendered: 3277
Janky frames: 228 (6.96%)
```

Fonte: Autoria própria (2025).

No aplicativo em Flutter, conforme apresentado na Subseção 2.5.2, devido ao uso do seu próprio motor gráfico denominado *Impeller*, o comando ADB da Listagem 6 não tem efeito sobre a aplicação. Portanto, foi utilizado o Flutter *DevTools* para extração da quantidade de *janky frames* que o aplicativo apresentou durante o seu uso. Assim como a ferramenta ADB, o Flutter *DevTools* também acumula o valor desta métrica, logo, fez-se necessário restaurar esse valor previamente a cada processo de coleta.

Na aba “Performance”, o Flutter *DevTools* exibe um gráfico mostrando o tempo em que cada quadro foi renderizado, bem como a quantidade de quadros renderizados enquanto o aplicativo está sendo utilizado, destacando na cor laranja os *janky frames* detectados. Adicionalmente, a cada ocorrência de um *janky frame*, um contador em cor vermelha é exibido e incrementado nessa aba, o que possibilitou registrar a quantidade total de *janky frames* detectados durante o uso do aplicativo.

Um exemplo da utilização dessa ferramenta para esse fim é exibido na Figura 16.

Figura 16 – Utilização do Flutter *DevTools* para verificar a quantidade de *janky frames*

Fonte: Autoria própria (2025).

Por fim, foram executados trinta experimentos para esse caso de teste, sendo que cada experimento teve um período de trinta segundos de uso da aplicação. Desta forma, calculou-se a média aritmética, os valores máximo e mínimo, e o desvio padrão para esse parâmetro. Sendo que na aplicação em React Native, por possibilitar a coleta mediante o uso do comando da Listagem 6, um *script* em Python denominado “`scriptCapturaJankyFramesRN.py`”[14] foi desenvolvido para automatizar o seu uso, a coleta dos dados e gerar um arquivo CSV a ser ana-

[14] Disponível em: https://github.com/romulo-souza/TCCrepo/blob/main/Scripts_Python/Fluidez_UI/scriptCapturaJankyFramesRN.py

lisado. Já na aplicação em Flutter, utilizou-se apenas o *script* "`usoAppUI_Flutter.ps1`"[15] para simular a sua utilização durante o período definido, sendo a coleta dos dados feita de forma manual, com auxílio do Flutter *DevTools*, e armazenados também em um arquivo CSV. Finalmente, para a análise dos resultados em cada plataforma, utilizou-se um código[16] em Python que faz uso da biblioteca Pandas para manipulação dos dados e da biblioteca Plotly para a construção de um gráfico do tipo *boxplot*[17] que permite uma análise da distribuição dos dados.

O Quadro 8 mostra uma síntese dos principais aspectos para esse caso de teste.

Quadro 8 – Principais aspectos para o Caso de teste 2

Aspecto	Descrição
Quantidade de animações *Lottie*	18
Implementação das listas (sem *scroll*)	Flutter: *GridView.builder* React Native: *Flatlist*
Quantidade de experimentos	30
Tempo de execução de cada experimento	30 s
Captura dos dados	React Native: *script* em Python Flutter: Flutter *DevTools*

Fonte: Autoria própria (2025).

4.2.3 Caso de teste 3: Consumo de memória

Para este caso de teste, reaproveitou-se a aplicação desenvolvida no Caso de teste 1, descrita na Subseção 4.2.1, porém com imagens e ícones fixos, sendo uma imagem diferente para cada *card*, mantendo trinta itens na lista e as três resoluções diferentes para as imagens. O código desta aplicação em cada *framework* está disponível no repositório[18] deste trabalho.

A Figura 17 mostra uma captura de tela das aplicações desenvolvidas em cada tecnologia.

[15] Disponível em: https://github.com/romulo-souza/TCCrepo/blob/main/Scripts_PowerShell/usoAppUI_Flutter.ps1

[16] Disponível em: https://github.com/romulo-souza/TCCrepo/blob/main/Scripts_Python/Fluidez_UI/analiseFluidezUI.ipynb

[17] Disponível em: https://plotly.com/chart-studio-help/what-is-a-box-plot/

[18] Disponível em: https://github.com/romulo-souza/TCCrepo/tree/main/apps/memoria

Figura 17 – Caso de teste 3: (a) Aplicação em Flutter, (b) Aplicação em React Native

(a) **(b)**

Fonte: Autoria própria (2025).

Para obtenção dessa métrica, o comando presente na Listagem 7 foi executado.

Listagem 7 – Comando do ADB para obtenção de informações acerca do consumo de memória de um aplicativo

```
adb shell dumpsys meminfo {nome_do_pacote}
```

Fonte: Adaptado de Android (2025).

Um exemplo da seção de interesse da saída produzida pelo comando apresentado na Listagem 7 é ilustrado na Figura 18.

Figura 18 – Exemplo de saída para o comando da Listagem 7

```
App Summary
                 Pss(KB)                        Rss(KB)
                 ------                         ------
       Java Heap:    57136                      78400
     Native Heap:   132464                     137128
            Code:   114020                     191436
           Stack:     2848                       3260
        Graphics:    25980                      25980
   Private Other:    49392
          System:    86400
         Unknown:                               47052

        TOTAL PSS:   468240            TOTAL RSS:   483256       TOTAL SWAP PSS:    80000
```

Fonte: Autoria própria (2025).

Assim como foi feito no Caso de teste 1, o uso da aplicação foi automatizado mediante a execução dos mesmos códigos apresentados naquela ocasião. Além disso, a coleta dos dados referentes à memória utilizada por determinado aplicativo (representada pelo campo “TOTAL PSS”) e a análise dos dados também foram automatizadas por códigos[19] em Python, sendo o código “`scriptCapturaMemPSS.py`” responsável pela captura dos dados e o “`analiseMemoria.ipynb`” pela análise dos dados. Essas implementações seguem a mesma lógica aplicada no Caso de teste 1 e utilizam o mesmo módulo e bibliotecas daquela ocasião, com a aquisição dos dados ocorrendo a cada 0,5 segundos durante um período de uso da aplicação de trinta segundos, tendo sido realizados trinta experimentos.

Ao final, realizou-se a análise dos dados registrados, a partir da qual foi construído um gráfico que permitiu visualizar o consumo médio de memória RAM em cada instante de tempo, bem como identificar o pico e o mínimo desse consumo médio para cada *framework*, a partir dos quais foi determinada a variação do consumo médio de memória. Adicionalmente, calculou-se a média geral do consumo de memória com base em todos os dados coletados.

O Quadro 9 sintetiza os principais aspectos considerados para esse caso de teste.

Quadro 9 – Principais aspectos para o Caso de teste 3

Aspecto	Descrição
Aplicação utilizada	Mesma do Caso de teste 1, porém com o conteúdo do *card* fixo
Quantidade de *Cards*	30
Resoluções das imagens	640×960, 1280×1920 e 1920×2880 px
Ícones utilizados	3 ícones do conjunto *Material Icons*
Implementação das listas	Flutter: *GridView.builder* *React Native: Flatlist*
Quantidade de experimentos	30
Tempo de execução de cada experimento	30 s
Tempo de aquisição dos dados pelo *script* em Python	A cada 0,5 s
Quantidade de dados coletados	1800

Fonte: Autoria própria (2025).

4.2.4 Caso de teste 4: Tempo de inicialização

Neste caso de teste, a aplicação utilizada foi a mesma desenvolvida anteriormente no Caso de teste 2 descrito na Subseção 4.2.2.

Para obter o tempo de uma inicialização *cold start*, o comando da Listagem 8 foi executado.

[19] Disponível em: https://github.com/romulo-souza/TCCrepo/tree/main/Scripts_Python/Consumo_Mem

Listagem 8 – Comando do ADB para a obtenção do tempo de inicialização *cold start*

```
adb shell am start -S -W <nome_do_pacote>/<nome_da_atividade>
```

Fonte: Adaptado de Android (2024a).

Neste comando, o parâmetro "-S" força o encerramento do aplicativo caso ele já esteja aberto, garantindo uma inicialização do zero. O parâmetro "W", por sua vez, faz com que o sistema espere a *Main Activity* estar completamente carregada e pronta para uso.

A Figura 19 mostra um exemplo de execução desse comando. O campo de interesse é o "*TotalTime*", que representa o tempo total gasto por esse tipo de inicialização, em milissegundos. É importante destacar que o campo "*WaitTime*" possui um valor maior, pois inclui a latência de comunicação entre o dispositivo Android e o computador, realizada via USB.

Figura 19 – Exemplo de saída para o comando da Listagem 8

```
C:\Users\User>adb shell am start -S -W com.instagram.android/.activity.MainTabActivity
Stopping: com.instagram.android
Starting: Intent { act=android.intent.action.MAIN cat=[android.intent.category.LAUNCHER]
Status: ok
LaunchState: COLD
Activity: com.instagram.android/com.instagram.mainactivity.InstagramMainActivity
TotalTime: 967
WaitTime: 985
Complete
```

Fonte: Autoria própria (2025).

A Listagem 9 mostra o comando usado para a obtenção das inicializações *warm start* e *hot start*.

Listagem 9 – Comando do ADB para a obtenção dos tempos de inicialização *warm start* e *hot start*

```
adb shell am start -W <nome_do_pacote>/<nome_da_atividade>
```

Fonte: Adaptado de Android (2024a).

O tempo de inicialização em *warm start* foi registrado abrindo e, em seguida, fechando o aplicativo por meio do botão "voltar" do dispositivo. Após essa ação, executou-se o comando mostrado na Listagem 9. Já o tempo de inicialização em *hot start* foi mensurado mantendo o aplicativo em segundo plano e, posteriormente, executando o mesmo comando. Em ambos os casos, a métrica alvo corresponde ao campo "*TotalTime*".

As Figuras 20 e 21 a seguir ilustram um exemplo de *warm start* e *hot start*, respectivamente.

Figura 20 – Exemplo de saída para o comando da Listagem 9: *Warm start*

```
C:\Users\User>adb shell am start -W com.example.teste/.MainActivity
Starting: Intent { act=android.intent.action.MAIN cat=[android.intent.category.LAUNCHER]
Status: ok
LaunchState: WARM
Activity: com.example.teste/.MainActivity
TotalTime: 182
WaitTime: 184
Complete
```

Fonte: Autoria própria (2025).

Figura 21 – Exemplo de saída para o comando da Listagem 9: *Hot start*

```
C:\Users\User>adb shell am start -W com.example.teste/.MainActivity
Starting: Intent { act=android.intent.action.MAIN cat=[android.intent.category.LAUNCHER]
Warning: Activity not started, its current task has been brought to the front
Status: ok
LaunchState: HOT
Activity: com.example.teste/.MainActivity
TotalTime: 95
WaitTime: 101
Complete
```

Fonte: Autoria própria (2025).

Em ambos os *frameworks*, a obtenção dessa métrica e a análise dos dados foram automatizadas por dois *scripts*[20] em Python. O código "`scriptCapturaTempoIni.py`" foi responsável pela coleta dos valores referentes ao campo "*TotalTime*', sendo executados trinta experimentos para cada tipo de inicialização e gerado um arquivo CSV com os dados. O código "`analiseTempoIni.ipynb`", por sua vez, fez a análise dos dados com o uso da biblioteca Pandas, agrupando os valores de tempo por tipo de inicialização, o que permitiu o cálculo do tempo médio, dos valores máximo e mínimo, e do desvio padrão para cada ocasião. Além dessas medidas estatísticas, gerou-se gráficos do tipo *boxplot* para cada tipo de inicialização mediante o uso da biblioteca Plotly.

Para sumarizar os principais elementos desse caso de teste, apresenta-se no Quadro os principais aspectos observados.

Quadro 10 – Principais aspectos para o Caso de teste 4

Aspecto	Descrição
Aplicação utilizada	Mesma do Caso de teste 2
Quantidade de experimentos	30
Captura dos dados	*Script* em Python

Fonte: Autoria própria (2025).

4.2.5 Caso de teste 5: Tempo de renderização da interface gráfica

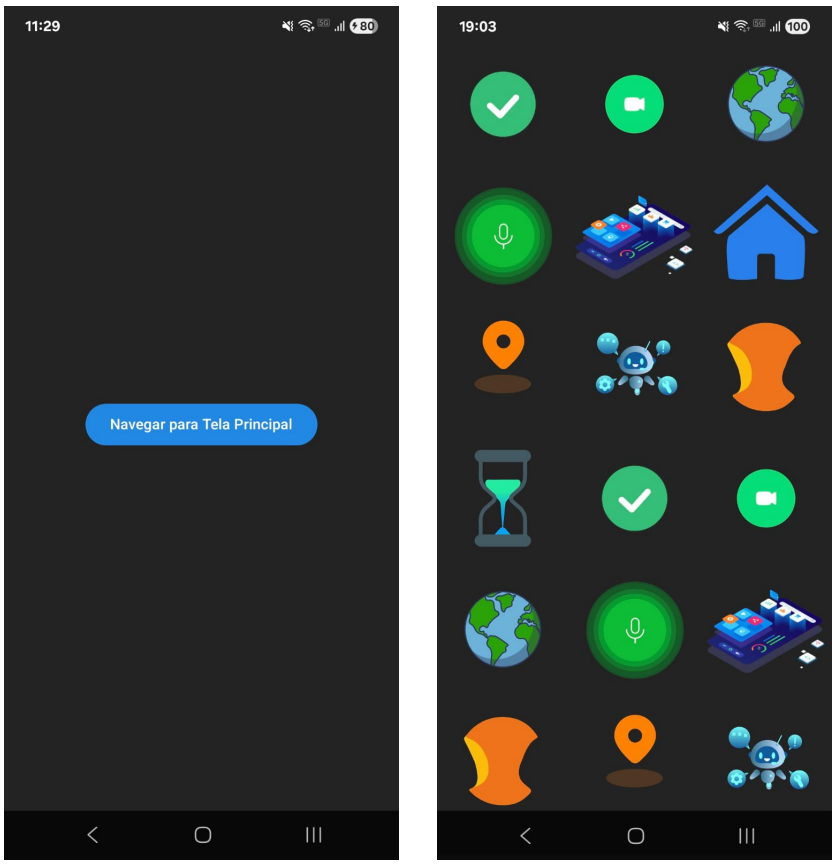
Para o desenvolvimento desse caso de teste, também reaproveitou-se a interface gráfica construída no Caso de teste 2 (Subseção 4.2.2), devido à presença de múltiplos elementos a serem renderizados simultaneamente.

A aferição do tempo de renderização da interface desenvolvida foi instrumentada via código-fonte em ambos os *frameworks*, cujas implementações encontram-se disponíveis no repositório[21] deste trabalho. Para isso, desenvolveu-se uma interface inicial responsável por navegar até a interface principal, na qual o tempo de renderização foi mensurado, conforme demonstra a Figura 22.

[20] Disponível em: https://github.com/romulo-souza/TCCrepo/tree/main/Scripts_Python/Tempo_Ini
[21] Disponível em: https://github.com/romulo-souza/TCCrepo/tree/main/apps/tempoRenderUI

Figura 22 – Caso de teste 5: (a) Interface inicial, (b) Interface principal

(a) **(b)**



Fonte: Autoria própria (2025).

Conforme apresentado na Subseção 2.5.5, as duas plataformas dispõem de meios para se obter programaticamente o tempo necessário para a renderização completa de uma interface gráfica.

Na aplicação desenvolvida em React Native, todos os elementos da interface principal foram envolvidos pelo componente especial *Profiler*, e através da função de *callback* (*onRender*) presente nesse componente, foi possível registrar o tempo de renderização, em milissegundos, da fase de montagem da interface gráfica.

Já para a interface desenvolvida em Flutter, um cronômetro de alta precisão foi iniciado dentro do método *initState*, que é o método chamado quando o *widget* é criado e inserido na Árvore de *Widgets*. Em seguida, utilizou-se o método *addPostFrameCallback*, que executa uma função de *callback* após a interface ser totalmente renderizada. No escopo dessa função, o cronômetro foi pausado, possibilitando a obtenção do tempo de renderização da interface gráfica.

Em ambos os *frameworks*, essa métrica foi coletada trinta vezes utilizando um *script*[22] que simula a navegação entre as interfaces. A obtenção dos dados nesse caso foi instrumentada diretamente no código-fonte, com os valores registrados em um arquivo CSV. Posteriormente, o

[22] Disponível em: https://github.com/romulo-souza/TCCrepo/blob/main/Scripts_PowerShell/appRender.ps1

cálculo do tempo médio de renderização, dos valores máximo e mínimo, do desvio padrão e a representação visual do conjunto de dados pelo gráfico do tipo *boxplot* foram realizados por um código[23] em Python, utilizando as bibliotecas Pandas e Plotly.

O Quadro 11 resume as principais informações consideradas nesse caso de teste.

Quadro 11 – Principais aspectos para o Caso de teste 5

Aspecto	Descrição
Aplicação utilizada	Aproveitou-se a interface gráfica feita no Caso de teste 2
Quantidade de experimentos	30
Captura dos dados	Instrumentada diretamente no código-fonte React Native: uso do componente especial Profiler Flutter: utilização do método addPostFrameCallback e um cronômetro de alta precisão

Fonte: Autoria própria (2025).

4.2.6 Caso de teste 6: Latência de entrada

Para este caso de teste, desenvolveu-se uma aplicação que consiste em um contador, na qual um botão altera o estado de um elemento de texto na interface (valor do contador). A cada interação com o botão, o valor exibido é incrementado, permitindo medir a latência de entrada, ou seja, o tempo decorrido entre a ação do usuário (toque) e a atualização visual do contador na tela. O código referente à implementação deste caso de teste para cada *framework* está disponível no repositório[24] deste trabalho.

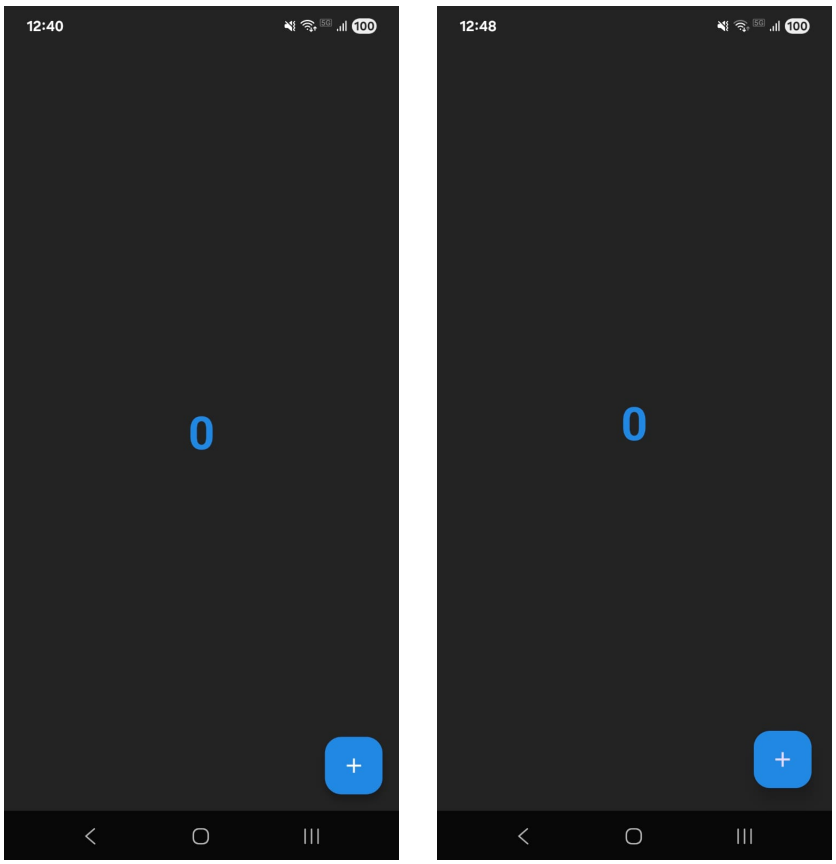
A Figura 23 exibe uma captura de tela das aplicações implementadas em cada plataforma.

[23] Disponível em: https://github.com/romulo-souza/TCCrepo/blob/main/Scripts_Python/Tempo_Render/analiseTempoRender.ipynb

[24] Disponível em: https://github.com/romulo-souza/TCCrepo/tree/main/apps/tempoResp

Figura 23 – Caso de teste 6: (a) Aplicação em Flutter, (b) Aplicação em React Native

(a) **(b)**



Fonte: Autoria própria (2025).

Essa métrica foi mensurada via código-fonte nos dois *frameworks*, com um cronômetro de alta precisão sendo iniciado assim que a interação com o botão é realizada, e pausado após o estado do elemento analisado ser alterado. Com isso, o tempo decorrido entre o momento da interação com o botão e a mudança refletida na interface gráfica foi obtido em milissegundos.

Por fim, por meio do *script* "`tempoLatencia.ps1`"[25], foram realizadas trinta interações consecutivas com o botão, e os valores de tempo obtidos foram armazenados em um arquivo CSV. Já o *script* "`analiseLatencia.ipynb`"[26], com o uso das bibliotecas Pandas e Plotly, possibilitou analisar os dados por meio do gráfico *boxplot* e obter o tempo médio, os valores máximo e mínimo, e o desvio padrão para a latência de entrada em cada plataforma.

A fim de sintetizar os principais aspectos desse caso de teste, apresenta-se no Quadro 12 a configuração experimental utilizada.

[25] Disponível em: https://github.com/romulo-souza/TCCrepo/blob/main/Scripts_PowerShell/tempoLatencia.ps1

[26] Disponível em: https://github.com/romulo-souza/TCCrepo/blob/main/Scripts_Python/Tempo_Latencia/analiseLatencia.ipynb

Quadro 12 – Principais aspectos para o Caso de teste 6

Aspecto	Descrição
Quantidade de experimentos	30
Captura dos dados	Instrumentada diretamente no código-fonte mediante o uso de um cronômetro de alta precisão

Fonte: Autoria própria (2025).

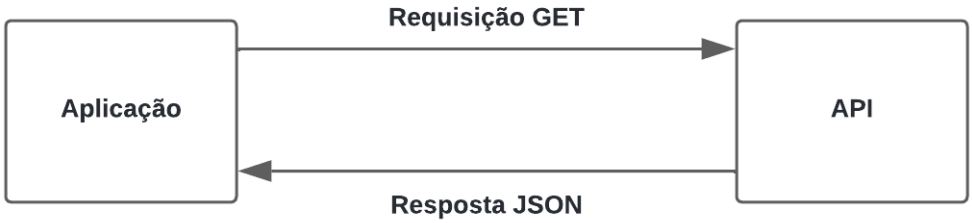
4.2.7 Caso de teste 7: Tempo de resposta de requisições a uma API local

Para este caso de teste, a fim de minimizar interferências externas nas medições de tempo decorrentes da conexão com a rede, foi desenvolvida uma API[27] local, executada no próprio computador utilizado para os testes. A API foi implementada em Node.js[28], utilizando o *framework* Express[29], o que facilitou a criação da rota e o tratamento das requisições.

A API possui uma rota responsável por retornar um objeto em formato JSON, cujo conteúdo é composto por uma *string* de tamanho definido. Essa *string* foi construída por meio da repetição de caracteres simples, de modo que cada caractere corresponde a um byte, considerando o padrão de codificação *Unicode Transformation Format - 8 bits* (UTF-8)[30]. Dessa forma, foi possível determinar o tamanho aproximado do JSON de resposta mediante o uso de *strings*. O valor desejado para o tamanho foi definido como parâmetro na rota, variando entre 1 KB, 10 KB e 100 KB.

Portanto, a aplicação desenvolvida em cada plataforma fez requisições HTTP do tipo "*GET*" para essa rota, sendo que no React Native a requisição foi feita com o uso da *Fetch* API e no Flutter com o uso da biblioteca http, conforme descrito na Subseção 2.5.7. A Figura 24 exibe o fluxo de funcionamento da API, enquanto a Figura 25 exibe a interface da aplicação implementada em cada tecnologia.

Figura 24 – Fluxo de funcionamento da API

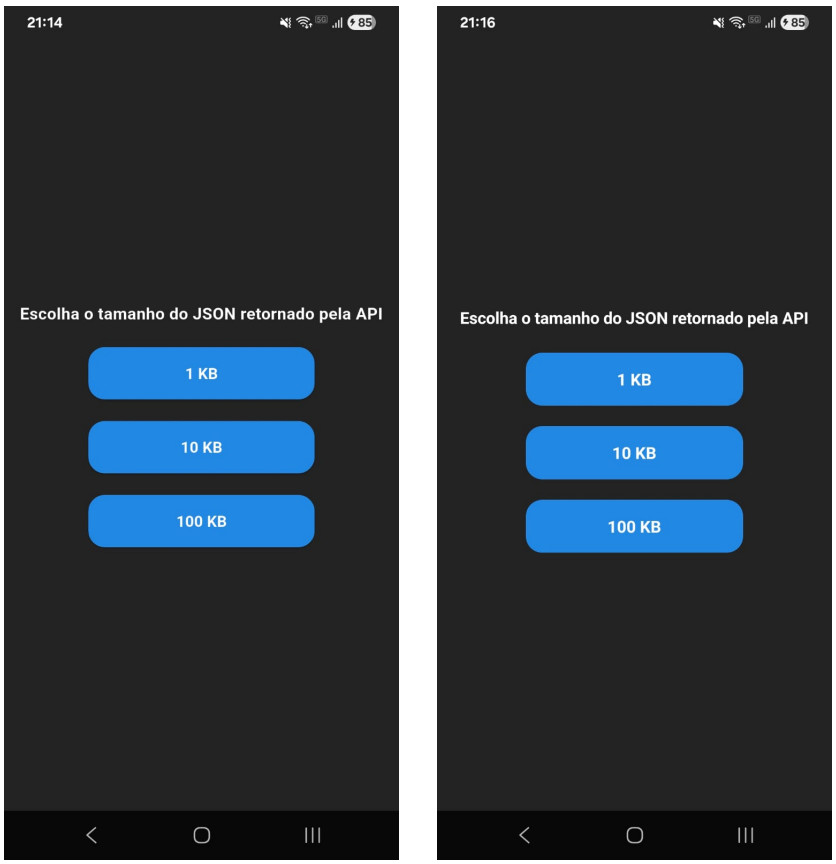


Fonte: Autoria própria (2025).

[27] Disponível em: https://github.com/romulo-souza/TCCrepo/tree/main/API
[28] Disponível em: https://nodejs.org/pt
[29] Disponível em: https://expressjs.com/
[30] Disponível em: https://developer.mozilla.org/pt-BR/docs/Glossary/UTF-8

Figura 25 – Caso de teste 7: (a) Aplicação em Flutter, (b) Aplicação em React Native

(a) **(b)**



Fonte: Autoria própria (2025).

Os tempos de resposta para essas requisições foram obtidos diretamente via código, utilizando um cronômetro de alta precisão iniciado imediatamente antes do envio da requisição e pausado assim que a resposta da API foi recebida. Por fim, essa métrica foi registrada por trinta vezes para cada tamanho de JSON, com o uso da aplicação automatizado por um *script*[31] em PowerShell, os dados armazenados em um arquivo CSV, e a análise dos dados realizada por um código[32] em Python utilizando as bibliotecas Pandas e Plotly, o que possibilitou determinar o tempo médio de resposta da API, os valores máximo e mínimo, o desvio padrão e os gráficos do tipo *boxplot* para cada caso, tanto no Flutter quanto no React Native.

O Quadro 13 mostra os principais aspectos levados em consideração para esse caso de teste.

[31] Disponível em: https://github.com/romulo-souza/TCCrepo/blob/main/Scripts_PowerShell/usoAPI.ps1

[32] Disponível em: https://github.com/romulo-souza/TCCrepo/blob/main/Scripts_Python/Tempo_API/analiseTempoApi.ipynb

Quadro 13 – Principais aspectos para o Caso de teste 7

Aspecto	Descrição
Implementação da API	Realizada em Node.js utilizando o *framework* Express
Tamanhos de JSON retornados	1 KB, 10 KB e 100 KB
Quantidade de experimentos	30
Método para realização das requisições	React Native: através da *Fetch* API do JavaScript Flutter: através de sua biblioteca HTTP
Captura dos dados	Instrumentada diretamente no código-fonte mediante o uso de um cronômetro de alta precisão

Fonte: Autoria própria (2025).

5 RESULTADOS

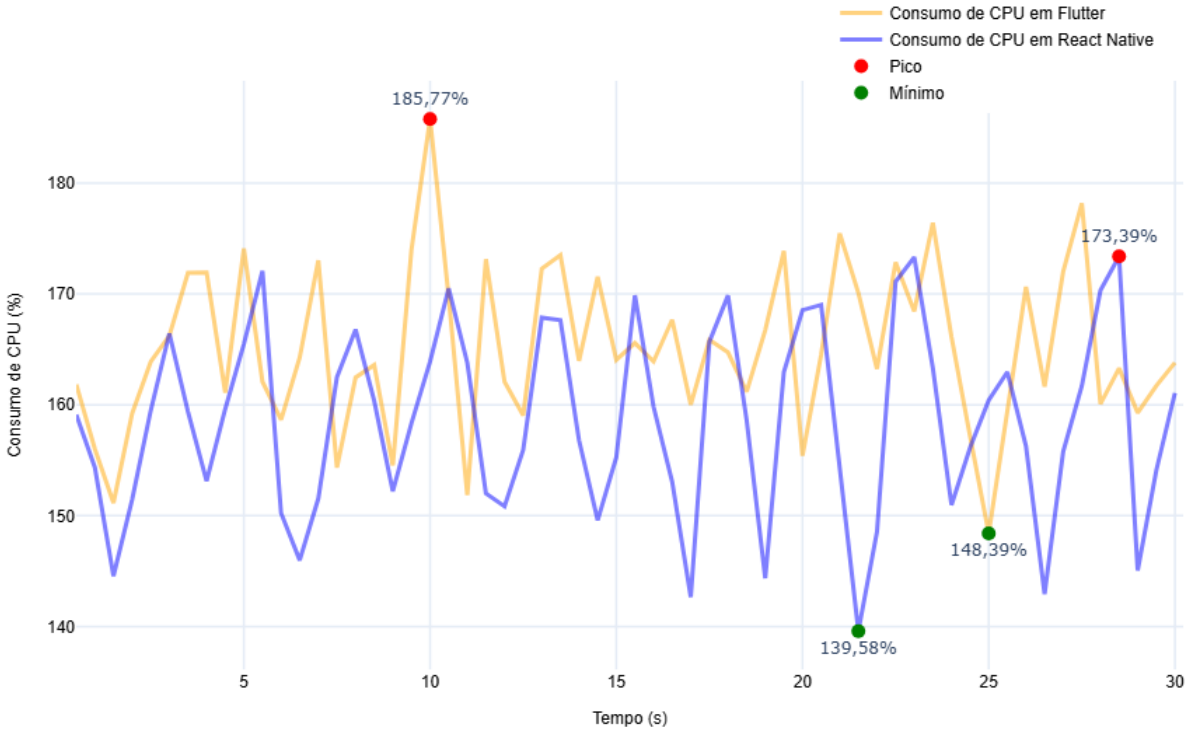
O presente capítulo tem como objetivo apresentar os resultados obtidos para cada *framework* analisado, a partir da execução dos procedimentos correspondentes a cada caso de teste, conforme descrito no capítulo anterior. Além disso, é apresentada uma análise comparativa entre as plataformas, com o intuito de avaliar o desempenho e identificar possíveis fatores que justifiquem os resultados observados.

5.1 Caso de teste 1: Taxa de uso do processador

Conforme elucidado na Subseção 4.2.1, esse experimento foi conduzido considerando os três diferentes períodos de atualização dos elementos na interface gráfica. Durante o experimento, foram coletados dados de consumo de CPU ao longo de um período de trinta segundos, com aquisições a cada 0.5 segundos, sendo trinta experimentos realizados. Ao todo, foram obtidos 1.800 valores para cada período de atualização utilizado. Esses dados podem ser encontrados em arquivos[1] no formato CSV no repositório deste trabalho.

A partir desses conjuntos de dados, foi elaborado, para cada caso, um gráfico composto pelas curvas correspondentes a cada *framework*, que representam o consumo médio de CPU em cada instante de tempo, destacando também os valores de pico e mínimo associados a esse consumo médio, conforme mostram as Figuras 26, 27 e 28.

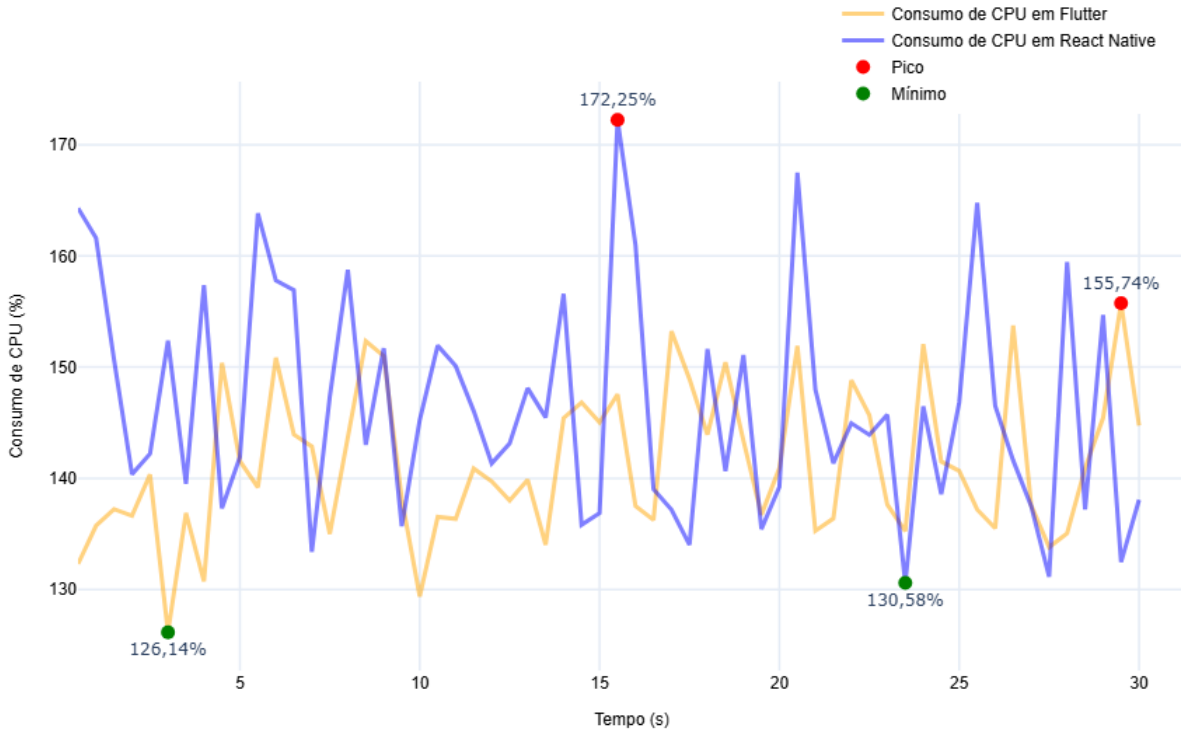
Figura 26 – Consumo médio de CPU: período de atualização dos elementos a cada 400 ms



Fonte: Autoria própria (2025).

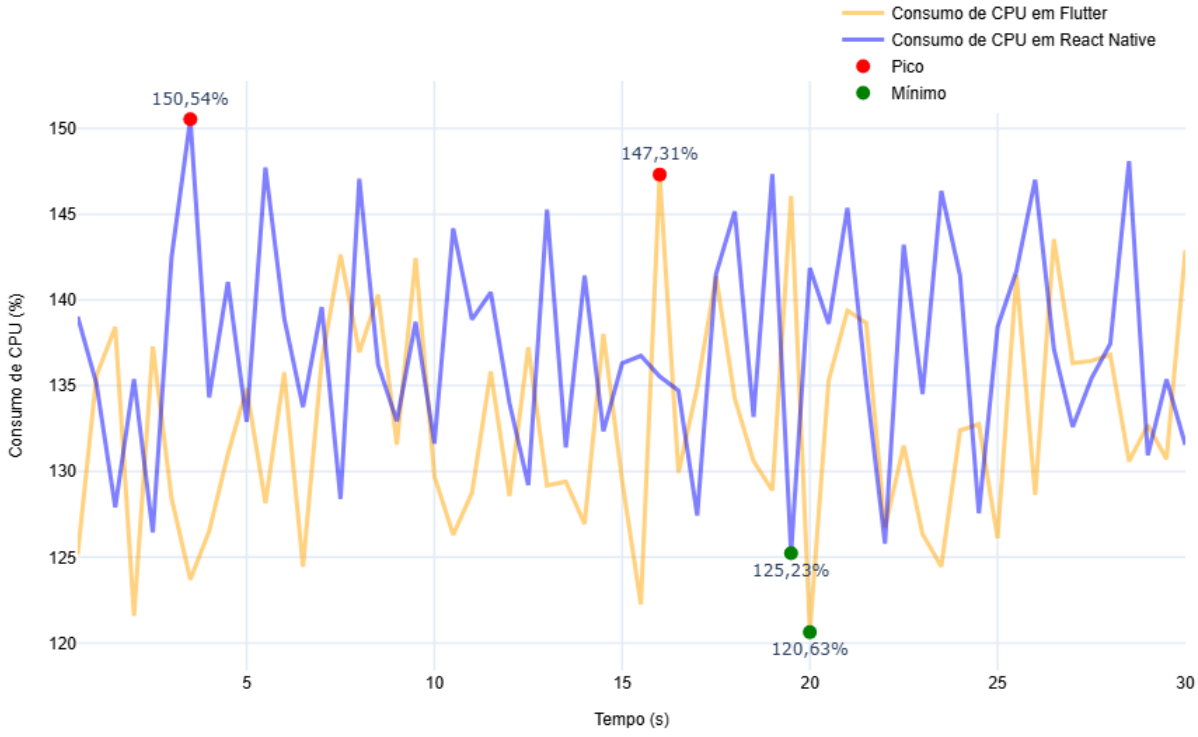
[1] Disponível em: https://github.com/romulo-souza/TCCrepo/tree/main/Scripts_Python/Consumo_CPU/data

Figura 27 – Consumo médio de CPU: período de atualização dos elementos a cada 600 ms



Fonte: Autoria própria (2025).

Figura 28 – Consumo médio de CPU: período de atualização dos elementos a cada 800 ms



Fonte: Autoria própria (2025).

Adicionalmente, com base nos gráficos gerados, apresentados nas Figuras 26, 27 e 28, calculou-se a variação do consumo médio de CPU em pontos percentuais (p.p.) para cada cenário, definida como a diferença entre os valores de pico e mínimo destacados nas curvas. A partir de todos os dados coletados, determinou-se também a média geral do consumo de CPU

em cada plataforma para cada período de atualização dos elementos. A Tabela 1 apresenta a variação do consumo médio em p.p. para cada período de atualização, enquanto a Tabela 2 mostra os valores médios obtidos ao considerar todos os dados registrados.

Tabela 1 – Variação do consumo médio de CPU para cada *framework*

Período de atualização dos elementos (ms)	Variação do consumo médio de CPU em Flutter (p.p.)	Variação do consumo médio de CPU em React Native (p.p.)
400	37,38	33,81
600	29,60	41,67
800	26,68	25,31

Fonte: Autoria própria (2025).

Tabela 2 – Média geral do consumo de CPU para cada *framework*

Período de atualização dos elementos (ms)	Média geral do consumo de CPU em Flutter (%)	Média geral do consumo de CPU em React Native (%)
400	165,15	158,55
600	141,34	146,59
800	132,84	137,27

Fonte: Autoria própria (2025).

A partir dos resultados obtidos para este caso de teste, observou-se que, independentemente do *framework* analisado, à medida que o período de atualização dos elementos aumenta, o processador é menos exigido e a variação do consumo é menor, indicando maior estabilidade. Esse comportamento já era esperado, uma vez que uma menor frequência de alternância dos elementos reduz a sobrecarga no recálculo do *layout* e no processo de re-renderização realizada pelas *threads* que compõem o modelo de funcionamento de cada plataforma, conforme apresentado nas Subseções 2.2.1 e 2.3.1.

Considerando a variação do consumo médio e a média geral do consumo de CPU, apresentadas nas Tabelas 1 e 2, respectivamente, constatou-se que para o período de 400 ms, a aplicação em React Native consumiu aproximadamente 4,0% a menos de CPU do que o Flutter e apresentou 9,55% maior estabilidade no consumo médio, considerando as oscilações entre os valores de pico e mínimo. Para o período de 600 ms, a aplicação em Flutter foi mais eficiente, apresentando um consumo de CPU cerca de 3,58% a menos do que o React Native, e 28,96% maior estabilidade no consumo médio. Por fim, para o período de 800 ms, o Flutter manteve uma pequena vantagem na taxa de uso do processador, consumindo 3,22% menos CPU, embora o React Native tenha apresentado 5,13% maior estabilidade em relação ao Flutter. Sendo assim, concluiu-se que ambos os *frameworks* gerenciaram de forma eficiente o consumo do processador durante o uso da aplicação projetada para este caso de teste.

Nota-se que o React Native apresentou uma pequena vantagem no cenário de maior frequência de atualização dos elementos, possivelmente devido à sua nova arquitetura baseada no JSI, que, conforme descrito na Subseção 2.3.3, permite uma comunicação mais eficiente e direta entre o código JavaScript e as camadas nativas do sistema operacional, reduzindo a

latência no processo de recálculo do *layout* e re-renderização dos elementos em ciclos curtos. Esse ganho provavelmente se deve à maior eficiência do seu sistema de renderização, o *Fabric*, que sincroniza a Árvore de Elementos do React com a estrutura da interface gráfica nativa do Android.

Por outro lado, o Flutter mostrou-se mais eficiente em situações de menor frequência de atualização dos elementos, resultado que pode ser associado ao seu motor de renderização próprio, o *Impeller*, o qual, conforme citado na Subseção 2.2.5, elimina a dependência de comunicação com as interfaces nativas e realiza a renderização diretamente na GPU, o que possivelmente reduz a sobrecarga de processamento em cenários de menor demanda de recálculos do *layout* e re-renderizações simultâneas.

5.2 Caso de teste 2: Fluidez da interface gráfica

Conforme descrito na metodologia para este caso de teste, apresentado na Subseção 4.2.2, foram conduzidos trinta experimentos para obtenção da quantidade de *janky frames* em cada *framework*. Todos os dados coletados encontram-se disponíveis em arquivos[2] no formato CSV no repositório do trabalho.

A partir dos valores coletados, foram obtidos a média, o desvio padrão e os valores mínimo e máximo para os *janky frames* de cada *framework*, como ilustra a Tabela 3.

Tabela 3 – Média, desvio padrão, mínimo e máximo de *janky frames* por *framework*

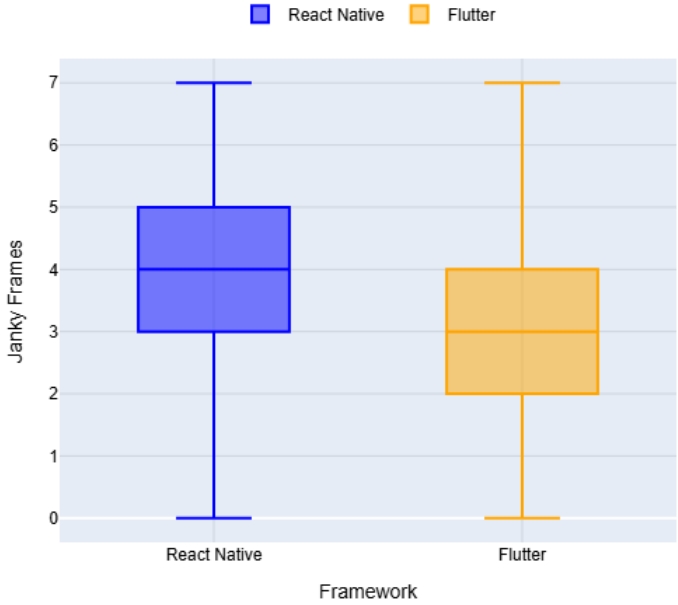
Framework	**Média de *janky frames***	**Desvio padrão**	**Mínimo - Máximo**
Flutter	2,93	1,91	0 - 7
React Native	3,57	1,63	0 - 7

Fonte: Autoria própria (2025).

Adicionalmente, foi gerado um *boxplot* para os dados de ambas as plataformas, permitindo comparar a distribuição da quantidade de *janky frames*, conforme apresenta a Figura 29.

[2] Disponível em: https://github.com/romulo-souza/TCCrepo/tree/main/Scripts_Python/Fluidez_UI/data

Figura 29 – *Boxplot* da quantidade de *janky frames* por *framework*



Fonte: Autoria própria (2025).

Ao analisar os valores da Tabela 3, nota-se que o Flutter apresentou uma média de quantidade de *janky frames* 17,92% menor que a observada no React Native, o que indica maior fluidez da interface gráfica para a aplicação implementada em Flutter. Em relação ao desvio padrão, o React Native teve um valor ligeiramente menor, demonstrando menos dispersão dos valores em relação à média, o que sugere um comportamento mais consistente entre as execuções dos experimentos. Quanto aos valores de máximo e mínimo, ambos os *frameworks* apresentaram os mesmos resultados, atingindo em alguns experimentos o melhor cenário possível, isto é, sem a ocorrência de *janky frames*.

Em relação à distribuição dos valores coletados, o gráfico do tipo *boxplot*, mostrado na Figura 29, possui uma caixa dividida em três quartis, com a linha central representando a mediana e as extremidades correspondendo aos limites inferior e superior, permitindo visualizar a dispersão dos dados e identificar possíveis valores discrepantes (*outliers*). Desse modo, tendo como referência a mediana, observa-se que os quartis do *boxplot* do Flutter estão abaixo dos do React Native, evidenciando que a quantidade de quadros atrasados registrada nessa plataforma foi menor e, consequentemente, apresentou melhor desempenho em termos de fluidez da interface.

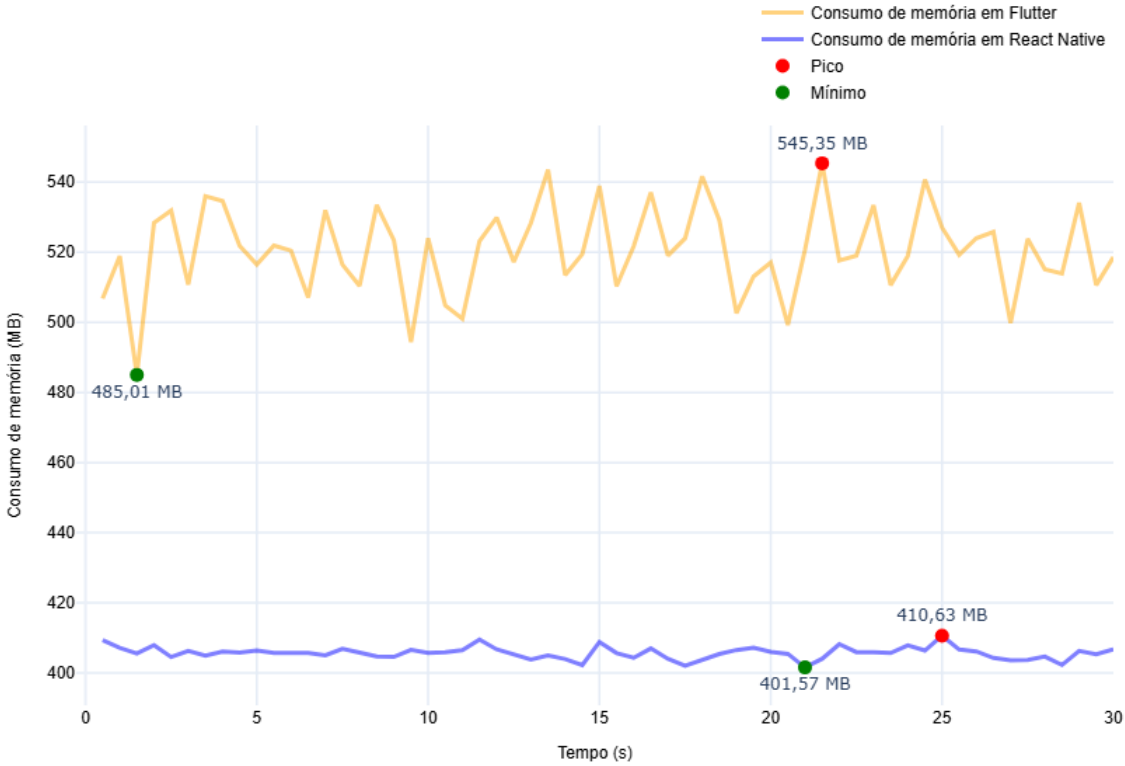
A vantagem observada no Flutter para esse caso de teste pode ser atribuída novamente ao seu motor gráfico próprio, o *Impeller*, que realiza a renderização de seus *widgets* diretamente na GPU, sem depender de componentes gráficos nativos, o que reduz a latência e, por consequência, a quantidade de quadros atrasados. Por outro lado, no React Native a renderização depende da sincronização constante entre o código JavaScript e os componentes gráficos nativos, realizada principalmente pelos componentes JSI e *Fabric* da nova arquitetura. Apesar de mais otimizados em relação à antiga arquitetura, como já mencionado no caso de teste anterior, esses módulos não possuem a otimização apresentada pelo motor gráfico do Flutter, o que resultou em uma maior quantidade média de quadros atrasados.

5.3 Caso de teste 3: Consumo de memória

Após a execução dos procedimentos descritos na Subseção 4.2.3, 1800 valores do consumo de memória RAM da aplicação foram registrados para cada *framework*, os quais estão disponíveis em arquivos[3] do tipo CSV no repositório do trabalho.

A partir desses valores, gerou-se, para cada plataforma, a curva referente ao consumo médio de memória RAM em cada instante de tempo, durante os trinta segundos de utilização da aplicação, conforme exibe a Figura 30.

Figura 30 – Consumo médio de memória RAM



Fonte: Autoria própria (2025).

A partir dos valores de pico e mínimo das curvas apresentadas no gráfico da Figura 30, determinou-se a variação do consumo médio de memória RAM em MB para cada plataforma, conforme mostra a Tabela 4. Além da análise visual do consumo médio ao longo do tempo, foi calculada também a média geral do consumo de memória em cada *framework*, com base em todos os dados coletados, como apresenta a Tabela 5.

Com base nos resultados apresentados para este caso de teste, observou-se que o React Native foi mais eficiente que o Flutter nas questões de estabilidade e consumo de memória. Para a estabilidade, considerando a variação apresentada na Tabela 4, o React Native apresentou aproximadamente 85% maior estabilidade de consumo durante o período de tempo considerado. Já em relação à média geral de consumo, a aplicação em React Native utilizou

[3] Disponível em: https://github.com/romulo-souza/TCCrepo/tree/main/Scripts_Python/Consumo_Mem/data

Tabela 4 – Variação do consumo médio de memória RAM para cada *framework*

Framework	Variação do consumo médio de memória (MB)
Flutter	60,34
React Native	9,06

Fonte: Autoria própria (2025).

Tabela 5 – Média geral do consumo de memória para cada *framework*

Framework	Média geral do consumo de memória (MB)
Flutter	520,44
React Native	405,70

Fonte: Autoria própria (2025).

22% menos memória, consumindo, em média, 114,74 MB a menos do que a aplicação em Flutter.

Tal vantagem apresentada pelo React Native, assim como no caso de teste referente ao consumo de processador, pode ser explicada pela sua nova arquitetura baseada no JSI, detalhada na Subseção 2.3.3, que elimina a questão da comunicação assíncrona com o sistema nativo, envolvendo serialização e desserialização de mensagens em formato JSON, como era no modelo antigo, baseado em *Bridge*, e realiza uma comunicação mais direta, rápida e eficiente com as camadas nativas, possivelmente reduzindo a sobrecarga na memória e proporcionando maior estabilidade.

Outro ponto relevante a ser analisado está relacionado às diferenças no modelo de renderização adotado por cada *framework*. Como já citado nos casos de teste anteriores, o Flutter utiliza um motor gráfico próprio, sem a necessidade de recorrer aos componentes gráficos nativos do sistema operacional. Embora esse modelo proporcione alto desempenho visual, ele tende a consumir mais memória RAM do que o React Native, uma vez que o próprio motor precisa pré-alocar e gerenciar internamente elementos gráficos, incluindo *shaders*, texturas e *buffers* de dados gráficos, mantendo-os em *cache* para acesso rápido durante a execução. Conforme detalhado na Subseção 2.2.5, o *Impeller* realiza a pré-compilação de todos os *shaders* durante o tempo de compilação e gerencia explicitamente o *cache* desses recursos. Em contrapartida, o React Native mantém a renderização por meio dos componentes nativos do Android, delegando ao sistema operacional o gerenciamento de recursos gráficos mediante o seu sistema de renderização denominado *Fabric*, o que possivelmente contribui para menor consumo de memória.

5.4 Caso de teste 4: Tempo de inicialização

Após a execução dos procedimentos descritos na Subseção 4.2.4, que gerou um arquivo[4] CSV com todos os dados registrados, calculou-se a média, o desvio padrão e os valores mínimo e máximo em relação aos tempos de inicialização, como mostra a Tabela 6.

Tabela 6 – Média, desvio padrão, mínimo e máximo do tempo de inicialização em cada *framework*

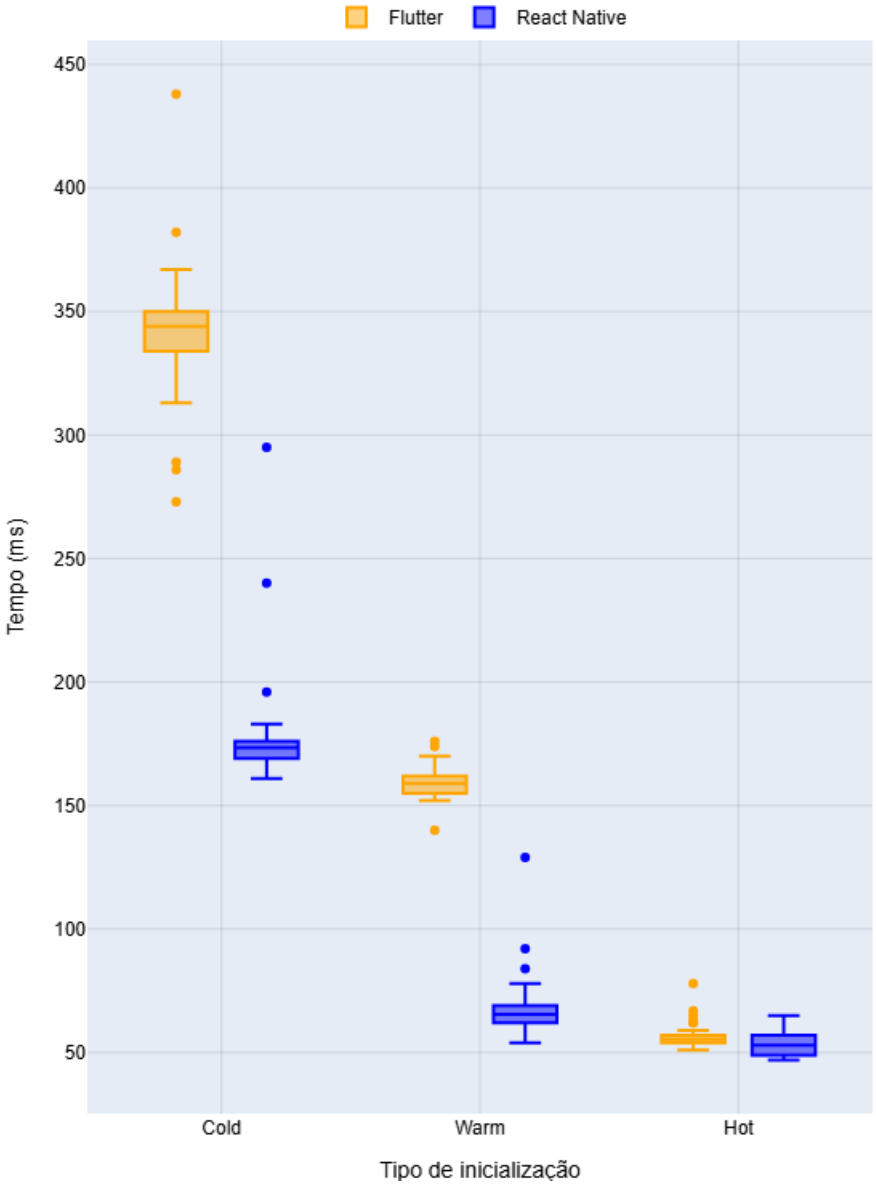
Framework	**Tipo**	**Tempo médio (ms)**	**Desvio padrão (ms)**	**Mínimo - Máximo (ms)**
Flutter	*Cold*	342,57	29,54	273 - 438
React Native	*Cold*	179,50	25,75	161 - 295
Flutter	*Warm*	159,37	7,28	140 - 176
React Native	*Warm*	68,17	14,15	54 - 129
Flutter	*Hot*	56,90	5,65	51 - 78
React Native	*Hot*	53,57	4,67	47 - 65

Fonte: Autoria própria (2025).

A fim de visualizar a distribuição dos valores de tempo de inicialização conforme o tipo de medição, foram gerados gráficos do tipo *boxplot* para cada categoria de inicialização, considerando ambos os *frameworks*, como mostra a Figura 31.

[4] Disponível em: https://github.com/romulo-souza/TCCrepo/tree/main/Scripts_Python/Tempo_Ini/data

Figura 31 – *Boxplots* dos tempos de inicialização por tipo e *framework*



Fonte: Autoria própria (2025).

Ao analisar a Tabela 6 e a Figura 31, percebe-se que o React Native teve vantagem em todas as categorias de inicialização, com diferenças mais acentuadas durante o *cold start* e o *warm start*.

Considerando os valores de tempo médio, para a inicialização do tipo *cold start*, o React Native foi 47,6% mais rápido do que o Flutter. Já para a inicialização do tipo *warm start*, observou-se a maior diferença de desempenho, com React Native sendo 57,22% mais rápido. Por fim, para o *hot start*, a diferença foi a menor obtida, com a aplicação em React Native sendo inicializada 5,85% mais rapidamente. Ademais, ao verificar os valores de desvio padrão para cada caso, nota-se que em ambas as plataformas eles estão próximos em todos os casos, com exceção para o tipo *warm start*, onde o React Native apresentou maior dispersão dos valores coletados em relação à média em comparação com o Flutter. Por fim, em relação aos valores

de mínimo e máximo, observou-se que o React Native possui os menores valores em todos os casos, o que indica maior eficiência na inicialização.

Conforme foi explicado na Subseção 2.5.4, o *cold start* é o processo de inicialização mais custoso entre os três, seguidos do *warm start* e do *hot start*. A vantagem significativa do React Native sobre o Flutter nos casos mais custosos possivelmente se deve à presença do seu componente denominado *Turbo Modules*, presente na nova arquitetura, que, como elucidado na Subseção 2.3.3, realiza a comunicação com módulos nativos do sistema operacional, carregando esses módulos sob demanda, isto é, carrega os recursos nativos somente quando necessário. Esta abordagem provoca redução no tempo de inicialização das aplicações, especialmente nos casos em que é necessária a obtenção desses módulos, como ocorre na inicialização *cold start* e parcialmente na *warm start*. Já para a inicialização menos custosa, o *hot start*, a diferença foi pequena, o que sugere que ambos os *frameworks* apresentam eficiência semelhante na retomada de uma aplicação previamente carregada na memória RAM.

5.5 Caso de teste 5: Tempo de renderização da interface gráfica

Uma vez executadas todas as etapas descritas na Subseção 4.2.5, foram gerados arquivos[5] em CSV com os dados coletados referentes aos tempos de renderização da interface gráfica implementada em cada tecnologia. A partir dos dados, determinou-se a média, o desvio padrão e os valores mínimo e máximo do tempo de renderização para cada *framework*, como ilustra a Tabela 7.

Tabela 7 – Média, desvio padrão, mínimo e máximo do tempo de renderização em cada *framework*

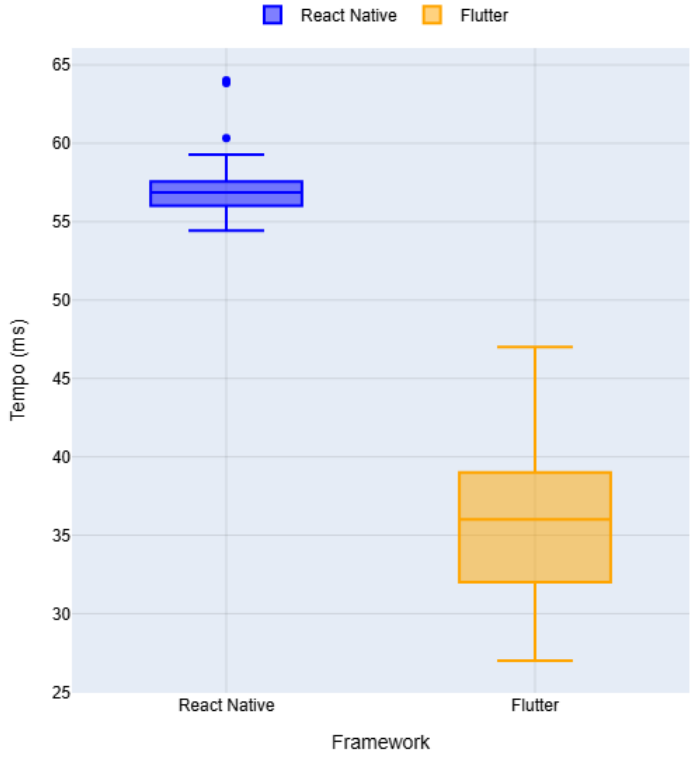
Framework	**Tempo médio (ms)**	**Desvio padrão (ms)**	**Mínimo - Máximo (ms)**
Flutter	35,90	5,18	27 - 47
React Native	57,27	2,28	54,43 - 64

Fonte: Autoria própria (2025).

Por fim, para melhor visualização da distribuição dos dados, um gráfico contendo o *boxplot* desses valores em cada plataforma foi elaborado, como mostra a Figura 32.

[5] Disponível em: https://github.com/romulo-souza/TCCrepo/tree/main/Scripts_Python/Tempo_Render/data

Figura 32 – *Boxplot* dos tempos de renderização por *framework*



Fonte: Autoria própria (2025).

Pela análise dos valores apresentados na Tabela 7, verificou-se que, em média, a aplicação desenvolvida em Flutter renderizou a interface 37,3% mais rápido do que a feita em React Native. O desvio padrão em Flutter foi um pouco superior ao observado no React Native, indicando maior dispersão dos valores em relação à média, entretanto, esses valores ainda permanecem menores do que os observados no React Native, como se pode constatar ao analisar os valores mínimo e máximo de cada plataforma, bem como pelos gráficos do tipo *boxplot* exibidos na Figura 32.

O fato de o Flutter ter se sobressaído em relação ao React Native neste caso de teste provavelmente está relacionado ao seu motor gráfico próprio, que, como citado nos casos de teste anteriores, otimiza a renderização diretamente na GPU, sem a necessidade de comunicação com os componentes visuais nativos do Android. Já o React Native depende dessa comunicação, o que pode gerar maior atraso até a completa construção dos elementos de interface gráfica na tela.

5.6 Caso de teste 6: Latência de entrada

Após realizar todos os passos mencionados na Subseção 4.2.6, foram obtidos arquivos[6] em CSV com os dados relacionados à latência de entrada, a partir dos quais determinou-se

[6] Disponível em: https://github.com/romulo-souza/TCCrepo/tree/main/Scripts_Python/Tempo_Latencia/data

a média, o desvio padrão e os valores mínimo e máximo da latência de entrada para cada *framework*, como mostra a Tabela 8.

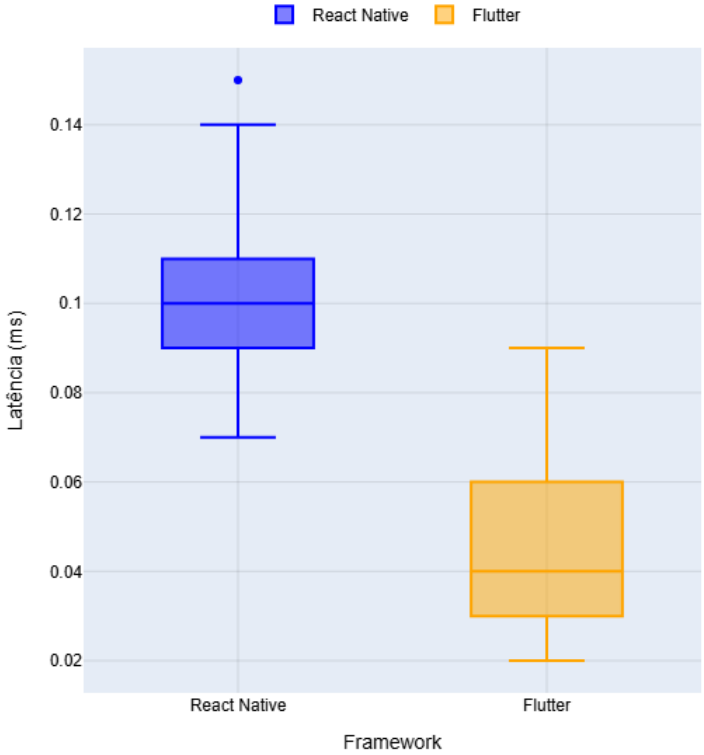
Tabela 8 – Média, desvio padrão, mínimo e máximo da latência de entrada em cada *framework*

Framework	**Latência média (ms)**	**Desvio padrão (ms)**	**Mínimo - Máximo (ms)**
Flutter	0,04	0,02	0,02 - 0,09
React Native	0,10	0,02	0,07 - 0,15

Fonte: Autoria própria (2025).

Adicionalmente, gráficos do tipo *boxplot* representando a distribuição dos valores coletados para cada plataforma foram construídos, como apresenta a Figura 33.

Figura 33 – *Boxplot* das latências de entrada por *framework*



Fonte: Autoria própria (2025).

Considerando os valores apresentados na Tabela 8, verifica-se que a aplicação desenvolvida em Flutter apresentou, em média, uma latência de entrada 60% inferior à observada no React Native, indicando maior eficiência na resposta às interações do usuário. Em relação ao desvio padrão, em ambas as plataformas o valor foi igual e bem pequeno. Por fim, ao analisar os valores mínimo e máximo em conjunto com a distribuição dos dados nos *boxplots* da Figura 33, nota-se que o Flutter apresenta desempenho superior, com valores de latência predominantemente inferiores aos registrados no React Native.

Como este caso de teste envolve a atualização do estado da interface, ou seja, a re-renderização do valor do contador após o acionamento do botão, isso está diretamente relacionado aos mecanismos de renderização que cada *framework* utiliza. Portanto, é possível supor que a menor latência observada no Flutter esteja relacionada à renderização direta de seus

widgets pelo motor *Impeller*, que, conforme discutido nos casos de teste anteriores, elimina a necessidade de camadas intermediárias entre o código e os componentes gráficos nativos, resultando em respostas mais rápidas às interações do usuário.

5.7 Caso de teste 7: Tempo de resposta de requisições a uma API local

Após a coleta dos dados[7] referentes a este caso de teste, foi possível realizar as análises. A Tabela 9 apresenta a média, o desvio padrão e os valores mínimo e máximo do tempo de resposta das requisições à API em ambas as plataformas, considerando os diferentes tamanhos de JSON retornados.

Tabela 9 – Média, desvio padrão, mínimo e máximo do tempo de resposta da API em cada *framework*

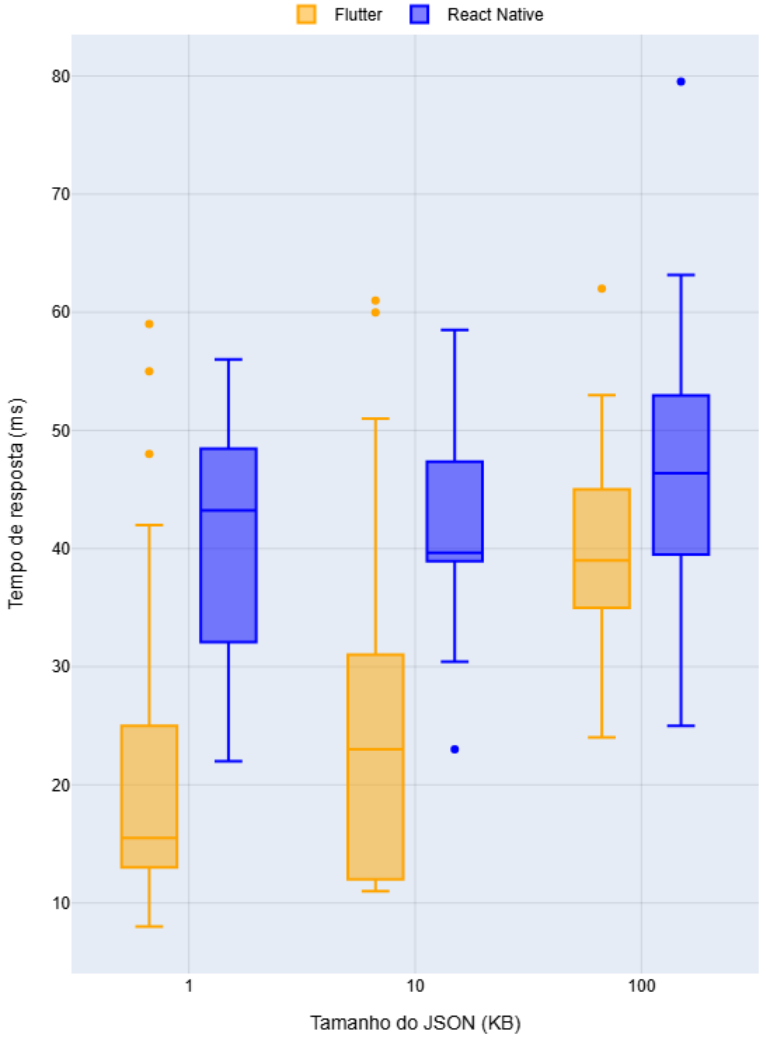
Framework	**Tamanho do JSON (KB)**	**Tempo médio de resposta (ms)**	**Desvio padrão (ms)**	**Mínimo - Máximo (ms)**
Flutter	1	21,63	13,77	8 - 59
React Native	1	40,76	11,15	22 - 56
Flutter	10	24,77	14,20	11 - 61
React Native	10	41,35	7,32	23 - 58,50
Flutter	100	39,93	8,76	24 - 62
React Native	100	45,18	11,51	25 - 79,53

Fonte: Autoria própria (2025).

Para facilitar a visualização da distribuição dos dados relacionados ao tempo de resposta da API em cada *framework*, foram criados gráficos do tipo *boxplot* para cada tamanho de JSON de resposta, como ilustra a Figura 34.

[7] Disponível em: https://github.com/romulo-souza/TCCrepo/tree/main/Scripts_Python/Tempo_API/data

Figura 34 – *Boxplots* dos tempos de resposta da API por tamanho de JSON e *framework*



Fonte: Autoria própria (2025).

A partir dos valores mostrados na Tabela 9 e da distribuição dos dados apresentada na Figura 34, observa-se que o Flutter apresentou melhor desempenho em relação ao React Native, com menores médias e uma distribuição com valores menores para os tempos de resposta da API, especialmente nos casos em que os tamanhos dos JSON retornados eram de 1 KB e 10 KB. Em relação ao desvio padrão obtido considerando cada tamanho de JSON de resposta, nota-se que em ambos os *frameworks* os valores ficaram próximos, exceto no caso em que o tamanho era de 10 KB, no qual o React Native apresentou um valor notavelmente menor, indicando menos variação em relação à média dos tempos de resposta.

Para o tamanho de JSON de 1 KB, a requisição do tipo *GET* feita pelo pacote http do Flutter obteve um tempo médio de resposta 46,93% mais rápido do que a feita pela *Fetch* API do React Native. Já para o tamanho de 10 KB, a requisição feita pelo aplicativo em Flutter foi, em média, 40,1% mais rápida. Por fim, para o tamanho de 100 KB, a diferença foi a menor obtida, com o Flutter sendo, em média, 11,62% mais rápido.

A diferença de desempenho observada pode estar relacionada aos mecanismos utilizados por cada *framework* para realizar as requisições HTTP. Como evidenciado na Subseção 2.5.7, no Flutter, o pacote HTTP adota uma abordagem própria para o envio e tratamento das

requisições, oferecendo uma implementação otimizada e integrada ao ambiente do aplicativo. Já no React Native, a *Fetch* API segue o padrão da *Web*, realizando as requisições a partir de chamadas gerenciadas pelo ambiente de execução JavaScript. Essa diferença na forma de comunicação e processamento das respostas pode ter contribuído para os tempos de resposta menores observados na aplicação em Flutter. Todavia, uma análise mais aprofundada sobre o funcionamento interno desses mecanismos seria necessária para determinar com maior precisão qual deles é mais eficiente e em quais contextos essa vantagem se manifesta.

6 CONSIDERAÇÕES FINAIS

Com base no embasamento teórico desenvolvido ao longo deste trabalho, aliado aos procedimentos metodológicos aplicados e aos resultados obtidos, foi possível apresentar uma visão abrangente sobre as vantagens e limitações dos *frameworks* Flutter e React Native em termos de desempenho. A comparação prática entre as duas tecnologias, por meio de sete casos de teste desenvolvidos de forma equivalente nas duas plataformas, permitiu identificar os contextos em que cada uma delas demonstra maior eficiência, possibilitando uma análise teoricamente fundamentada quanto à aplicabilidade dessas tecnologias em diversos cenários de desenvolvimento móvel.

De maneira geral, observou-se que o React Native apresentou desempenho superior nas métricas de consumo de memória e tempo de inicialização da aplicação, resultados que podem ser atribuídos à sua nova arquitetura baseada na JSI, a qual possibilita uma comunicação mais direta e eficiente com os recursos nativos do sistema operacional. Em contrapartida, o Flutter destacou-se nos aspectos relacionados à interface gráfica, apresentando melhor desempenho em fluidez, tempo de renderização e latência de entrada. Essa vantagem pode ser explicada pelo uso do seu motor de renderização próprio, o *Impeller*, que independe da comunicação com a camada nativa do sistema, otimizando o processo de renderização diretamente na GPU. Além desses aspectos, o Flutter também obteve resultados superiores quanto ao tempo de resposta de requisições feitas a uma API local. Por fim, em relação à taxa de uso do processador, ambos os *frameworks* gerenciaram de forma eficiente o consumo de CPU, com o React Native sendo um pouco mais eficiente no caso de maior exigência do processador, enquanto o Flutter demonstrou desempenho levemente superior nos demais cenários analisados.

Desta forma, os resultados obtidos nesse cenário de estudo demonstram que não há uma plataforma que se sobressaia de forma absoluta em todos os critérios avaliados, mas sim que a escolha da plataforma a ser utilizada depende das necessidades do projeto. Por exemplo, em cenários nos quais o consumo de memória RAM é um fator crítico, o React Native pode se mostrar mais adequado, permitindo a execução da aplicação em dispositivos com *hardware* mais limitado. Por outro lado, se o foco principal do projeto for a fluidez da interface gráfica e o desempenho em renderização, o Flutter tende a alcançar desempenho superior nessas métricas.

É importante ressaltar que este trabalho apresenta algumas limitações que devem ser consideradas na interpretação dos resultados. Todos os experimentos foram realizados exclusivamente no sistema operacional Android (versão 15), utilizando o *smartphone* Galaxy A55 da Samsung, o que restringe a generalização das conclusões para outros dispositivos Android e, especialmente, para plataformas diferentes, como o iOS. Além disso, a coleta das métricas de desempenho dos Casos de teste 5, 6 e 7 foi realizada por meio da instrumentação direta no código-fonte das aplicações, sem o uso de ferramentas externas de monitoramento, o que pode introduzir pequenas variações nos resultados obtidos devido ao consumo adicional de recursos (*overhead*) gerado pela instrumentação.

Por fim, como proposta para trabalhos futuros, sugere-se realizar uma comparação entre a nova e a antiga arquitetura do React Native, a fim de identificar e mensurar os ganhos de desempenho proporcionados pela adoção da nova estrutura. Além disso, um estudo futuro relevante seria investigar de forma mais aprofundada os mecanismos de funcionamento das bibliotecas responsáveis pela comunicação com APIs, a fim de identificar os fatores que contribuem para as diferenças observadas nos tempos de resposta entre os *frameworks*.

REFERÊNCIAS

ANDROID. **Android Debug Bridge (adb)**, . 2025. Disponível em: https://developer.android.com/tools/adb. Acesso em: 16 maio 2025.

ANDROID. **App startup time**, . 2024. Disponível em: https://developer.android.com/topic/performance/vitals/launch-time. Acesso em: 20 maio 2025.

ANDROID. **Platform architecture**, . 2024. Disponível em: https://developer.android.com/guide/platform. Acesso em: 15 maio 2025.

ANDROID. **UI jank detection**, . 2023. Disponível em: https://developer.android.com/studio/profile/jank-detection. Acesso em: 19 maio 2025.

ANDROID. **dumpsys**, . 2025. Disponível em: https://developer.android.com/tools/dumpsys. Acesso em: 19 maio 2025.

DART. **Dart overview**, . 2025. Disponível em: https://dart.dev/overview. Acesso em: 23 abr. 2025.

ENIHE, R.; JOSHUA, J. Hybrid mobile application development: A better alternative to native. **Global Scientific Journal**, v. 8, n. 5, p. 1373–1389, maio 2020. ISSN 2320-9186. Disponível em: https://www.globalscientificjournal.com/researchpaper/HYBRID_MOBILE_APPLICATION_DEVELOPMENT_A_BETTER_ALTERNATIVE_TO_NATIVE.pdf. Acesso em: 15 abr. 2025.

FATKHULIN, T. *et al.* Analysis of software tools allowing the development of cross-platform applications for mobile devices. *In*: 2023 SYSTEMS OF SIGNALS GENERATING AND PROCESSING IN THE FIELD OF ON BOARD COMMUNICATIONS. 2023, Moscow. **Anais [...]** Moscow: IEEE, 2023. p. 1–5.

FENTAW, A. E. **Cross platform mobile application development: a comparison study of React Native Vs Flutter**. 2020. Dissertação (Mestrado) — Faculty of Information Technology, University of Jyväskylä 2020.

FLUTTER. **Card class**, . 2025. Disponível em: https://api.flutter.dev/flutter/material/Card-class.html. Acesso em: 18 set. 2025.

FLUTTER. **Fetch data from the internet**, . 2025. Disponível em: https://docs.flutter.dev/cookbook/networking/fetch-data. Acesso em: 22 maio 2025.

FLUTTER. **Flutter and Dart DevTools**, . 2025. Disponível em: https://docs.flutter.dev/tools/devtools. Acesso em: 19 maio 2025.

FLUTTER. **Flutter architectural overview**, . 2025. Disponível em: https://docs.flutter.dev/resources/architectural-overview. Acesso em: 25 abr. 2025.

FLUTTER. **Flutter performance profiling**, . 2025. Disponível em: https://docs.flutter.dev/perf/ui-performance. Acesso em: 05 maio 2025.

FLUTTER. **GridView.builder constructor**, . 2025. Disponível em: https://api.flutter.dev/flutter/widgets/GridView/GridView.builder.html. Acesso em: 05 set. 2025.

FLUTTER. **Impeller rendering engine**, . 2025. Disponível em: https://docs.flutter.dev/perf/impeller. Acesso em: 30 abr. 2025.

FLUTTER. **addPostFrameCallback method**, . 2025. Disponível em: https://api.flutter.dev/flutter/scheduler/SchedulerBinding/addPostFrameCallback.html. Acesso em: 20 maio 2025.

FLUTTER. **createElement abstract method**, . 2025. Disponível em: https://api.flutter.dev/flutter/widgets/Widget/createElement.html#:~:text=createElement%20abstract%20method,-%40protected&text=In%20particular%20a%20given%20widget,will%20be%20inflated%20multiple%20times. Acesso em: 02 maio 2025.

FLUTTER. **createRenderObject abstract method**, . 2025. Disponível em: https://api.flutter.dev/flutter/widgets/RenderObjectWidget/createRenderObject.html. Acesso em: 02 maio 2025.

GERGES, M.; ELGALB, A. Comprehensive comparative analysis of mobile apps development approaches. **Journal of Artificial Intelligence General science (JAIGS) ISSN:3006-4023**, v. 6, n. 1, p. 430–437, Dec. 2024. Disponível em: https://newjaigs.com/index.php/JAIGS/article/view/269. Acesso em: 05 nov. 2025.

GOOGLE. **Material Design**, . 2025. Disponível em: https://m3.material.io/. Acesso em: 18 set. 2025.

GUERRA, A. d. L. e. R. Metodologia da pesquisa científica e acadêmica. **Revista OWL (OWL Journal) - REVISTA INTERDISCIPLINAR DE ENSINO E EDUCAÇÃO**, v. 1, n. 2, p. 149–159, ago. 2023. Disponível em: https://revistaowl.com.br/index.php/owl/article/view/48. Acesso em: 15 maio 2025.

JAIN, A. Scalable frameworks for cross-platform mobile app development. **Journal of Emerging Technologies and Innovative Research (JETIR)**, „, 2025. Disponível em: https://www.researchgate.net/profile/Abhishek-Jain-142/publication/393356799_Issue_6_wwwjetirorg_ISSN-2349-5162/links/68665e37e9b6c13c89e663b6/Issue-6-wwwjetirorg-ISSN-2349-5162.pdf. Acesso em: 04 nov. 2025.

KISHORE, K. *et al.* Performance and stability comparison of react and flutter: Cross-platform application development. *In*: 2022 INTERNATIONAL CONFERENCE ON CYBER RESILIENCE (ICCR). 2022. **Anais [...]** [*S.l.*: *s.n.*], 2022. p. 1–4.

KORAM, N.; GARG, R. Review on mobile app development: Tools and techniques. *In*: 2023 IEEE WORLD CONFERENCE ON APPLIED INTELLIGENCE AND COMPUTING (AIC). 2023, Sonbhadra. **Anais [...]** Sonbhadra: IEEE, 2023. p. 260–266.

KOROTYCH, K. Comparative analysis of components of the old and new architecture of the react native framework and their use for the development of mobile applications of e-commerce systems. **UNIVERSUM**, , n. 10, p. 53–61, 2024. Disponível em: https://archive.liga.science/index.php/universum/article/view/1168/1180. Acesso em: 06 maio 2025.

LottieFiles. **What is a Lottie?**, . 2025. Disponível em: https://lottiefiles.com/what-is-lottie. Acesso em: 30 maio 2025.

MAJCHRZAK, T. A.; BIøRN-HANSEN, A.; GRøNLI, T.-M. Progressive web apps: the definite approach to cross-platform development? *In*: PROCEEDINGS OF THE 51ST HAWAII INTERNATIONAL CONFERENCE ON SYSTEM SCIENCES (HICSS). 2018. **Anais [...]** [*S.l.*: *s.n.*], 2018.

MAJUMDER, A. S. **The Influence of UX Design on User Retention and Conversion Rates in Mobile Apps**, . 2025. Disponível em: https://www.researchgate.net/publication/388353843_The_Influence_of_UX_Design_on_User_Retention_and_Conversion_Rates_in_Mobile_Apps. Acesso em: 10 abr. 2025.

MAN7. **top(1) — Linux manual page**, . 2025. Disponível em: https://man7.org/linux/man-pages/man1/top.1.html. Acesso em: 18 maio 2025.

MARKOWSKI, M.; SMOŁKA, J. A comparative analysis of the flutter and react native frameworks. **Journal of Computer Sciences Institute**, v. 29„ p. 346–351, dez. 2023. Disponível em: https://ph.pollub.pl/index.php/jcsi/article/view/3794. Acesso em: 13 maio 2025.

MEIRELLES, F. S. Pesquisa do uso de ti-tecnologia de informação. **FGVcia**, „, jun. 2025. Disponível em: https://eaesp.fgv.br/producao-intelectual/pesquisa-anual-uso-ti. Acesso em: 04 nov. 2025.

MOZILLA. **JavaScript - MDN Web Docs**, . 2025. Disponível em: https://developer.mozilla.org/en-US/docs/Web/JavaScript. Acesso em: 03 maio 2025.

MOZILLA. **Using the Fetch API**, . 2025. Disponível em: https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API/Using_Fetch. Acesso em: 22 maio 2025.

MUSHTAQ, F.; AZAM, F.; ANWAR, M. W. Performance comparison of single code base development tools: Flutter, react native, and xamarin. *In*: 2024 14TH INTERNATIONAL CONFERENCE ON SOFTWARE TECHNOLOGY AND ENGINEERING (ICSTE). 2024, Macau. **Anais [...]** Macau: [*s.n.*], 2024. p. 17–23.

OVERFLOW, S. Stack Overflow **2024 Developer Survey: Technology**. [*S.l.*], 2024. Disponível em: https://survey.stackoverflow.co/2024/technology#most-popular-technologies-misc-tech-prof. Acesso em: 10 abr. 2025.

PINTO, C. M.; COUTINHO, C. From native to cross-platform hybrid development. *In*: 2018 INTERNATIONAL CONFERENCE ON INTELLIGENT SYSTEMS (IS). 2018, Funchal. **Anais [...]** Funchal: IEEE, 2018. p. 669–676.

POKU-MARBOAH, O. **Mobile Application Development Methods: Comparing Native and Non-Native Applications**, . 2021. Disponível em: https://www.theseus.fi/bitstream/handle/10024/505896/Poku-Marboah_Oheneba.pdf?sequence=2&isAllowed=y. Acesso em: 05 nov. 2025.

PRAYOGI, A. A. *et al.* Design and implementation of rest api for academic information system. **IOP Conference Series: Materials Science and Engineering**, IOP Publishing v. 875„, 2020. Disponível em: https://iopscience.iop.org/article/10.1088/1757-899X/875/1/012047. Acesso em: 21 maio 2025.

PYTHON. **subprocess — Subprocess management**, . 2025. Disponível em: https://docs.python.org/3/library/subprocess.html. Acesso em: 29 maio 2025.

REACT. **<Profiler>**, . 2025. Disponível em: https://react.dev/reference/react/Profiler. Acesso em: 20 maio 2025.

REACT. **Writing Markup with JSX**, . 2025. Disponível em: https://react.dev/learn/writing-markup-with-jsx. Acesso em: 03 maio 2025.

React Native. **About the New Architecture**, . 2025. Disponível em: https://reactnative.dev/architecture/landing-page. Acesso em: 05 maio 2025.

React Native. **Card**, . 2025. Disponível em: https://callstack.github.io/react-native-paper/docs/components/Card/. Acesso em: 18 set. 2025.

React Native. **Fabric**, . 2025. Disponível em: https://reactnative.dev/architecture/fabric-renderer. Acesso em: 07 maio 2025.

React Native. **FlatList**, . 2025. Disponível em: https://reactnative.dev/docs/flatlist. Acesso em: 05 set. 2025.

React Native. **Networking**, . 2025. Disponível em: https://reactnative.dev/docs/network. Acesso em: 22 maio 2025.

React Native. **Performance Overview**, . 2025. Disponível em: https://reactnative.dev/docs/performance. Acesso em: 19 maio 2025.

React Native. **React Fundamentals**, . 2025. Disponível em: https://reactnative.dev/docs/intro-react?language=javascript. Acesso em: 09 maio 2025.

React Native. **Render, Commit, and Mount**, . 2025. Disponível em: https://reactnative.dev/architecture/render-pipeline. Acesso em: 08 maio 2025.

React Native. **State of React Native 2018**, . 2018. Disponível em: https://reactnative.dev/blog/2018/06/14/state-of-react-native-2018#architecture. Acesso em: 08 out. 2025.

React Native. **Threading Model**, . 2025. Disponível em: https://reactnative.dev/architecture/threading-model. Acesso em: 05 maio 2025.

React Native. **What is Codegen?**, . 2025. Disponível em: https://reactnative.dev/docs/the-new-architecture/what-is-codegen. Acesso em: 07 maio 2025.

SAARINEN, J. **Evaluating cross-platform mobile app performance with video-based measurements**. 2019. Dissertação (Mestrado) — Faculty of Information Technology and Communication Sciences, Tampere University 2019. Disponível em: https://trepo.tuni.fi/bitstream/handle/10024/105726/1557996673.pdf. Acesso em: 20 maio 2025.

SAMSUNG. **Galaxy A55**, . 2024. Disponível em: https://www.samsung.com/pt/smartphones/galaxy-a/galaxy-a55-5g-awesome-iceblue-256gb-sm-a556blbceub/. Acesso em: 27 maio 2025.

SOUHA, A. *et al.* Comparative analysis of mobile application frameworks: A developer's guide for choosing the right tool. **Procedia Computer Science**, v. 236„ p. 597–604, 2024. ISSN 1877-0509. International Symposium on Green Technologies and Applications (ISGTA'2023). Disponível em: https://www.sciencedirect.com/science/article/pii/S1877050924010871. Acesso em: 15 abr. 2025.

STATCOUNTER. StatCounter **Mobile Operating System Market Share Worldwide**. [*S.l.*], 2025. Disponível em: https://gs.statcounter.com/os-market-share/mobile/worldwide/#monthly-202001-202503-bar. Acesso em: 14 abr. 2025.

STATISTA. Statista **Cross-platform mobile frameworks used by software developers worldwide from 2019 to 2023**. [*S.l.*], 2023. Disponível em: https://www.statista.com/statistics/869224/worldwide-software-developer-working-hours/. Acesso em: 05 nov. 2025.

SWATHIGA, U.; VINODHINI, P.; SASIKALA, V. An interpretation of dart programming language. **DRSR Journal**, v. 11, n. 3, p. 144–149, 2021. Disponível em: https://www.researchgate.net/publication/358661479_AN_INTERPRETATION_OF_DART_PROGRAMMING_LANGUAGE. Acesso em: 23 abr. 2025.

TANDEL, S.; JAMADAR, A. Impact of progressive web apps on web app development. **International Journal of Innovative Research in Science, Engineering and Technology**, v. 7, n. 9, p. 9439–9444, 2018. Disponível em: https://www.researchgate.net/profile/Sayali-Tandel-2/publication/330834334_Impact_of_Progressive_Web_Apps_on_Web_App_Development/links/5c5605d3a6fdccd6b5dde018/Impact-of-Progressive-Web-Apps-on-Web-App-Development.pdf. Acesso em: 16 abr. 2025.

TONG, J.; JIKSON, R. R.; GUNAWAN, A. A. S. Comparative performance analysis of javascript frontend web frameworks. *In*: 2023 3RD INTERNATIONAL CONFERENCE ON ELECTRONIC

AND ELECTRICAL ENGINEERING AND INTELLIGENT SYSTEM (ICE3IS). 2023, Yogyakarta. **Anais [...]** Yogyakarta: IEEE, 2023. p. 81–86.

ZHOU, C. **Challenges and solutions in cross-platform mobile development: a qualitative study of Flutter and React Native**. 2024. Dissertação (Mestrado) — Aalto University School of Science 2024. Acesso em: 05 nov. 2025.

ZOU, D.; DARUS, M. Y. A comparative analysis of cross-platform mobile development frameworks. *In*: 2024 IEEE 6TH SYMPOSIUM ON COMPUTERS & INFORMATICS (ISCI). 2024, Kuala Lumpur. **Anais [...]** Kuala Lumpur: IEEE, 2024. p. 84–90.