Pós-Graduação em Ciência da Computação

Bernardo de Moraes Santana Júnior

Explorando frameworks multiplataforma para desenvolvimento Android: uma investigação sobre o consumo de recursos

Universidade Federal de Pernambuco
posgraduacao@cin.ufpe.br
http://cin.ufpe.br/~posgraduacao

Recife
2024

Bernardo de Moraes Santana Júnior

Explorando frameworks multiplataforma para desenvolvimento Android: uma investigação sobre o consumo de recursos

Dissertação de Mestrado apresentada ao Programa de Pós-graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco, como requisito parcial para obtenção do título de Mestre em Ciência da Computação.

Área de Concentração: Engenharia de Software
Orientador: Fernando José Castor de Lima Filho
Coorientador: Wellington de Oliveira Júnior

Recife
2024

.Catalogação de Publicação na Fonte. UFPE - Biblioteca Central

Júnior, Bernardo de Moraes Santana.
Explorando frameworks multiplataforma para desenvolvimento Android: uma investigação sobre o consumo de recursos / Bernardo de Moraes Santana Júnior. - Recife, 2024.
71 f.: il.

Dissertação (Mestrado) - Universidade Federal de Pernambuco, Centro de Informática, Programa de Pós-graduação em Ciência da Computação, 2024.
Orientação: Fernando José Castor de Lima Filho.
Coorientação: Wellington de Oliveira Júnior.

1. Android; 2. Automação; 3. Desenvolvimento Multiplataforma; 4. Ebserver. I. Filho, Fernando José Castor de Lima. II. Júnior, Wellington de Oliveira. III. Título.

UFPE-Biblioteca Central CDD 004

Bernardo de Moraes Santana Júnior

"**Explorando frameworks multiplataforma para desenvolvimento Android: uma investigação sobre o consumo de recursos**"

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de Pernambuco, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação.

Aprovado em: 27/03/2024.

Orientador: Fernando José Castor de Lima Filho

Coorientador: Wellington de Oliveira Júnior

BANCA EXAMINADORA

Prof. Dr. Leopoldo Motta Teixeira
Centro de Informática / UFPE

Prof. Dr. Fernando Antonio Mota Trinta
Universidade Federal do Ceará / UFC

Dedico este trabalho à minha família e minha namorada, que foram porto seguro perante as dificuldades durante este percurso.

AGRADECIMENTOS

Agradeço à minha família que me deu a base que me permitiu chegar onde cheguei, à minha namorada que me ajudou e incentivou para terminar a pesquisa, ao orientador e co-orientador pela paciência infinita para lidar comigo.

Eu acredito que às vezes, são as pessoas de quem ninguém espera nada que fazem as coisas que ninguém consegue imaginar (HODGES, 1983).

RESUMO

O desenvolvimento multiplataforma tem chamado a atenção de várias empresas do mercado devido à possibilidade de uma única base de código para várias plataformas distintas. Essa característica é vantajosa, pois tende a acelerar o processo de desenvolvimento de *software* e reduzir a quantidade de *bugs*. Porém, essa forma de desenvolvimento normalmente adiciona camadas de abstração ao código que podem impactar negativamente no desempenho da aplicação final. Este trabalho visa analisar o impacto causado pelo desenvolvimento com a abordagem multiplataforma em dispositivos Android, e também apresenta uma ferramenta para automação do processo de análise, chamada de Ebserver. Esta pesquisa focou nos *frameworks* de maior destaque da atualidade para analisar como aplicações móveis são afetadas por eles. Foram utilizados dois programas e 10 *benchmarks* para comparação de performance em relação ao desenvolvimento na abordagem nativa. Para facilitar o processo de análise, a construção do Ebserver se mostrou de grande utilidade para garantir uma maior confiabilidade dos dados gerados e velocidade no processo de coleta dos dados durante os testes do programas. Para análise dos frameworks, foi utilizado um conjunto de programas de *benchmarks* - sem interação com a interface gráfica -, além de uma aplicação focada em animação de imagens e outra aplicação de gerenciamento de contatos, sendo uma delas com atualização constantemente da GUI e a outra envolve interação com usuário. Para coleta de informações, como consumo de memória, CPU, energia e tempo de execução, foi utilizado o Android *Debug Bridge* (ADB). Dados encontrados mostram que, a depender da aplicação, os *frameworks* multiplataforma podem ser bastante competitivos a nível de desempenho em comparação com a abordagem nativa, em Java. Flutter e .Net Maui, no geral, foram os que apresentaram menor impacto em comparação aos demais *frameworks* multiplataforma, havendo cenários em que até mesmo teve desempenho melhor do que a abordagem nativa. React Native, enquanto tendo o pior desempenho em cenários de alto consumo de CPU, obteve o melhor desempenho nos testes mais focados em atualização frequente da interface gráfica. Os resultados mostram que a escolha de qual ferramenta utilizar vai depender do cenário e o processo de análise é de grande importância antes de se dedicar a um framework para desenvolvimento de aplicações de grande porte.

Palavras-chaves: Android. Automação. Desenvolvimento Multiplataforma. Ebserver.

ABSTRACT

Multiplatform development has caught the attention of several companies in the industry due to the need of a single code base for multiple platforms. This characteristic is advantageous, as it tends to accelerate the software development process and reduce the number of bugs. However, this form of development usually adds layers of abstraction to the code that can negatively impact the final application performance. This work aims to analyze the impact caused by development with the multiplatform approach on Android devices, and also presents a tool for automating the analysis process, called Ebserver. This research focused on today's most prominent frameworks to analyze how mobile applications are affected by them. Two programs and 10 benchmarks were used for comparing performance in relation to development with the native approach. To facilitate the analysis process, the construction of Ebserver proved to be of great utility to ensure more reliability of generated data and more speed in the process of data collection while testing. To analyze the frameworks, a set of benchmark programs was used - without interaction with the graphics interface - in addition to an application focused on image animation and another one for contact management, one that constantly updates the GUI and other that involves user interaction. For collecting information such as memory consumption, CPU, energy and execution time, the Android Debug Bridge (ADB) was used. Data found shows that, depending on the application, multiplatform frameworks can be quite competitive in terms of performance, compared to the native approach, in Java. Flutter and .Net Maui, overall, had the lowest impact compared to the others multiplatform frameworks, with scenarios in which it they performed even better than the native approach. React Native, while having the worst performance in high consumption scenarios of CPU, obtained the best performance in tests more focused on frequent updates of the graphical interface. The results show that the choice of which tool to use will depend on the scenario and the analysis process is of great importance before dedicating to a framework for developing complex applications.

Key-words: Android. Automation. Ebserver. Multiplatform Development.

LISTA DE FIGURAS

LISTA DE TABELAS

SUMÁRIO

1 INTRODUÇÃO

O conceito do termo "aparelho celular" vem sendo modificado com o decorrer do tempo desde 1973, ano em que o primeiro celular do mundo veio a público (MOTOROLA MOBILITY LLC, s.d.). Tendo surgido inicialmente como um mero aparelho portátil de comunicação por voz, a Figura 1 expõe uma breve linha do tempo de como esse dispositivo evoluiu nas suas capacidades. A primeira função extra dos celulares foi o Serviço de Mensagens Curtas (SMS, do inglês *Small Message Service*), permitindo a comunicação por texto. Entre os anos de 1994 e 1996 os botões físicos são, pela primeira vez, substituídos pelas telas sensíveis ao toque (telas *touch screen*); e o acesso à internet via navegador passa a ser uma realidade.

Com o passar dos anos, capturar e armazenar fotografias, e novas opções de conectividade como *Bluetooth*, Wi-Fi e GPS foram integradas aos celulares e passaram a fazer parte do conjunto cada vez maior de funcionalidades básicas desses produtos. Em consonância com essa evolução de utilidades, o termo "aparelho celular" também teve sua transição para "*smartphone*", significando, literalmente, celular inteligente. A mudança parece ter sido adequada dada a quantidade de tecnologia colocada num celular atualmente.

2007 foi o ano de grande revolução no mercado com o lançamento do primeiro celular da marca Apple, ele foi o precursor do estilo de *smartphones* disponíveis hoje em dia. Apple, Samsung e Xiaomi dominam o mercado atual e fecharam o último trimestre de 2023 com, respectivamente, 23%, 16% e 13% do *market share* global. Nesse mesmo período, 323,2 milhões de aparelhos foram vendidos no mundo, dos quais 167,7 milhões correspondem à soma dos dispositivos das marcas citadas (COUNTERPOINT TECHNOLOGY MARKET RESEARCH, 2024).

Em sua maioria, as marcas lançam diferentes modelos anualmente, seja para atender segmentos de mercado com foco em custo-benefício, ou para atender usuários modernos e sedentos pela tecnologia mais avançada e ponta de linha disponível. Enquanto essas marcas focam na construção de dispositivos físicos e customização dos sistemas operacionais (SO), empresas como Google e Apple são, respectivamente, responsáveis pela construção dos SOs Android e iOS, os dominantes nesse contexto (STATCOUNTER, 2024).

A existência dos SOs nos *smartphones* permite a instalação de diversos aplicativos pelo próprio usuário, e a complexidade tecnológica atual já se equipara a computadores. Para cada SO, a forma de criar aplicativos é diferente, e as empresas disponibilizam uma documentação detalhada de como desenvolver aplicações para seus sistemas operacionais. São chamados de *frameworks* nativos o conjunto de ferramentas disponibilizado pelas próprias provedoras dos respectivos sistemas. Enquanto o Android permite linguagens de programação como Java/Kotlin e C/C++, o iOS utiliza as linguagens Swift e Objective-C. Outra forma de desenvolver aplicativos para *smartphones* é através de *frameworks*

Figura 1 – A evolução do celular

Fonte: Adaptado de (MELHORPLANO.NET, s.d.)
Original disponível em: https://melhorplano.net/tecnologia/evolucao-do-celular
Acesso em 11 fev. 2024.

multiplataforma, disponibilizados, majoritariamente, por empresas sem nenhuma relação direta com a Apple ou a Google. Uma exceção é o Flutter, que pertence à Google.

Os *frameworks* multiplataforma permitem o uso de uma única linguagem de programação para construção de programas que podem ser executados tanto no Android quanto no iOS. Por vezes, também em plataformas como Windows ou Web. É uma grande vantagem para os desenvolvedores, pois a expertise de conhecimento torna-se mais genérica. Para as empresas de *software*, reduz a necessidade de diferentes perfis de profissionais e a quantidade de código necessária além de, possivelmente, reduzir o tempo para desenvolvimento. No entanto, também existem ônus relacionados a essa forma de construção de aplicativos.

1.1 DESCRIÇÃO DO PROBLEMA

Com o uso dos *frameworks* multiplataforma, a integração com os recursos dos celulares continua sendo feita na camada nativa. Uma camada de abstração adicional é utilizada, na qual a linguagem a ser utilizada nas diferentes plataformas é executada e então esse

ambiente se comunica com a camada nativa para acessá-la. Essa característica, em conjunto com algumas outras condições, podem resultar em problemas de desempenho dos aplicativos, assim como a usabilidade, que também pode ser prejudicada.

Atributos como utilização de memória, carga no CPU e consumo energético são alguns dos recursos impactados pela camada adicional de abstração imposta pelos *frameworks* multiplataforma. O nível de utilização destes recursos pode variar de acordo com diferentes configurações dos dispositivos; versão dos programas e características durante a execução, como outros aplicativos abertos simultaneamente e processos executando em *background*. Dessa forma, surge a necessidade de uma análise do gasto de recursos, nos cenários desejados, em comparação com os *frameworks* nativos para aclarar vantagens e desvantagens de cada opção.

Para uma análise adequada, é comum executar uma bateria de testes repetidas vezes, em busca de menor variância sobre os processos em execução automática no SO dos *smartphones*. Quando executados manualmente, os testes tornam-se exaustivos e propensos a erros humanos, o que gera a necessidade de uma ferramenta para automação das execuções dos testes e da coleta dos dados gerados.

Desenvolver uma aplicação com bom desempenho, que atende às necessidades do produto enquanto mantém o menor custo possível, é o objetivo de qualquer empresa. A escolha de boas ferramentas, algo não trivial, é crucial para alcançar esse objetivo. Algumas empresas relataram publicamente a dificuldade de adaptação a determinadas ferramentas, como o Airbnb, que adotou um *framework* multiplataforma para desenvolvimento iOS e Android em 2016, e em 2018 decidiu voltar com a abordagem nativa devido às dificuldades técnicas e organizacionais encontradas (AIRBNB, INC, s.d.b; AIRBNB, INC, s.d.c; AIRBNB, INC, s.d.a; AIRBNB, INC, s.d.d; AIRBNB, INC, s.d.e). Em contrapartida, a empresa Discord, de forma um tanto quanto inusitada, durante anos utilizou um *framework* multiplataforma apenas para desenvolvimento do seu aplicativo para iOS, mantendo a versão Android com a abordagem nativa. Em 2022, com a evolução do multiplataforma e com o aprendizado obtido após 6 anos de uso, a empresa resolveu aplicar o *framework* multiplataforma também para o desenvolvimento no Android (DISCORD INC, s.d.a; DISCORD INC, s.d.b).

A comparação entre *frameworks* pode abranger diversas variáveis, como consumo de diferentes recursos, retrocompatibilidade, usabilidade, tempo para aprendizado e desenvolvimento, entre outras. A velocidade com a qual a tecnologia evolui - impactando em novas versões de SO, *smartphones* cada vez mais complexos e poderosos computacionalmente, surgimento de novos *frameworks* para programação de aplicativos, e atualizações dos *frameworks* existentes - pode, inclusive, tornar as análises comparativas rapidamente defasadas. Portanto, a grande maioria dos trabalhos científicos da área, que possuem o intuito de análise comparativa, foca em um pequeno conjunto de *frameworks* e de informações a serem coletadas para comparação. É bastante comum o foco em consumo ener-

gético, como pode ser visto em (OLIVEIRA; OLIVEIRA; CASTOR, 2017), (CIMAN; GAGGI, 2017a), (CIMAN; GAGGI et al., 2014) e (HUBER; DÖLLER; FELDERER, 2023); enquanto outros trabalhos são mais genéricos, como (EL-KASSAS et al., 2017), que foca em explicar o funcionamento dos tipos de *frameworks* multiplataforma. Já (XANTHOPOULOS; XINOGALOS, 2013), dá mais atenção ao processo de desenvolvimento com esse tipo de tecnologia; (PALMIERI; SINGH; CICCHETTI, 2012) faz comparação de *frameworks*, mas não analisa como eles afetam na performance dos aplicativos finais. (NAWROCKI et al., 2021) tem uma proposta mais parecida com esta pesquisa, visando analisar a performance dos aplicativos com um certo grupo de *frameworks*. No entanto, este último trabalho realiza coleta manuais, o que poderia ser expandido com o uso de uma ferramenta de automação do processo de coleta, por exemplo.

Outra forma de comparar ferramentas é através de aplicativos executados em produção, ou seja, no ambiente final no qual os programas são implantados para uso dos usuários reais. Por um lado, essa abordagem pode fornecer uma grande quantidade de dados para análise, mas por outro, ela abre margem para aplicativos de baixa qualidade e performance chegarem às mãos do usuário, o que pode reduzir a confiança deles em relação ao produto e/ou à marca, e até mesmo descredibilizar completamente uma empresa. Dessa forma, é mais interessante que haja um processo de teste em ambientes internos, que consigam replicar cenários parecidos com o real, mas que restrinjam o acesso aos usuários. Visto que aplicações complexas podem resultar em uma inúmera quantidade de cenários possíveis e uma grande quantidade de dados gerados, é imprescindível a automação dos testes, uma forma automatizada de análise e coleta de informações.

1.2 ABORDAGEM PROPOSTA

Este trabalho tem dois objetivos principais:

Objetivo 1: Investigar qual o custo do consumo de recursos do sistema imposto por *frameworks* de desenvolvimento de aplicativos móveis.

Objetivo 2: Desenvolver uma ferramenta capaz de automatizar o processo de coleta dos dados durante uma pesquisa e que seja simples de utilizar e personalizar.

Para isso, será analisado o comportamento dos distintos tipos de *frameworks* apresentados para desenvolvimento de aplicativos móveis e, não obstante, será apresentada uma ferramenta construída para otimizar a análise - Ebserver. A ferramenta foca em automatizar testes de consumo de recursos por aplicações no SO Android, é de simples utilização e fácil customização e já foi utilizada em outros trabalhos publicados (OLIVEIRA

et al., 2023c; FERREIRA et al., 2023; MATOS et al., 2022; OLIVEIRA et al., 2023a; OLIVEIRA; OLIVEIRA; CASTOR, 2017; OLIVEIRA et al., 2021; ZIMMERLE et al., 2019).

Nesta pesquisa é realizada a comparação entre 4 *frameworks* multiplataforma, mais o desenvolvimento nativo - Java - para desenvolvimento Android, visando a análise de performance destes. É feita uma análise mais detalhada da arquitetura e funcionamento de cada *framework* e, a partir disso, o estudo baseia-se no uso de 2 aplicações e 10 *benchmarks* replicados em cada uma dos *frameworks* para execução de testes. Os testes visam a captura de métricas de consumo de recursos, como energia, memória, CPU, entre outros, em cenários distintos. Com a crescente quantidade de testes, surgiu a oportunidade e necessidade de automação. Com isso, foi criada a ferramenta Ebserver, já inclusive também utilizada em outros grupos de pesquisa, conforme mencionado anteriormente.

1.3 ESTRUTURA DO DOCUMENTO

Este trabalho divide-se em 5 capítulos: o Capítulo 2 foca em descrever o referencial teórico sobre cada um dos *frameworks* e formas de conexão com dispositivos Android, além de trabalhos relacionados; o Capítulo 3 explica a abordagem proposta de automação de coleta de dados através da construção do Ebserver; o Capítulo 4 descreve a metodologia de pesquisa, as aplicações construídas, experimentos executados e resultados dos *frameworks*; por fim, o Capítulo 5 apresenta discussões sobre os resultados, conclusões da pesquisa e trabalhos futuros.

2 FUNDAMENTOS

Este capítulo disserta sobre algumas das tecnologias existentes atreladas ao desenvolvimento de aplicações Android e as ferramentas de coleta de dados dessas aplicações. Na Seção 2.1 são apresentados os *frameworks* para desenvolvimento Android, incluindo o nativo e os multiplataforma; em seguida, a Seção 2.2 descreve as principais ferramentas para obtenção de dados de uma aplicação e, por fim, a Seção 2.3 fala sobre os trabalhos que buscaram resolver problemas parecidos ou relacionados com os que este trabalho discute.

2.1 FRAMEWORKS

O sistema operacional Android fornece um conjunto de APIs para desenvolvimento escritas na linguagem Java. Essas APIs fornecem acesso a todos os recursos do sistema, uma grande variedade de componentes visuais e demais funcionalidades necessárias para a grande maioria das aplicações modernas. Esse conjunto de ferramentas é também conhecido como o ***framework* nativo**.

Desenvolvimento nativo é o termo utilizado na criação de uma aplicação com tecnologias fornecidas pelo próprio sistema, sem a utilização de código terceiro. É comumente a primeira forma a se pensar quando necessita-se criar uma aplicação em alguma plataforma. Fatores como documentação detalhada, comunidade ativa, estabilidade, boa performance e melhor usabilidade tornam essa forma de desenvolvimento bastante atrativa.

No meio tempo, novas tecnologias nasceram com o objetivo de atacar necessidades não sanadas pelos *frameworks* nativos das diferentes plataformas. Quando se faz necessário desenvolver sistemas operacionais distintos, diferentes tecnologias com diferentes processos de desenvolvimento, manutenção, e configuração são necessários; o que normalmente também exige profissionais com diferentes competências, focados em cada uma das plataformas. Nesse contexto, os *frameworks* multiplataformas nasceram para centralizar as tecnologias para desenvolvimento em uma camada intermediária, tal que haja uma única base de código a ser utilizadas nos diferentes SOs almejados.

Outro problema comum no desenvolvimento de aplicações é a necessidade de rápida alteração dos programas para adequação nas mudanças que impactam o produto. Sendo assim, desenvolver para mais de uma plataforma de maneira nativa resulta em desenvolver várias bases de código, testar e realizar o processo de atualização das versões na loja de aplicativos de cada plataforma. Pela existência de vários processos separados, etapas são repetidas para cada uma das plataformas e os códigos resultantes podem ter divergências ou até mesmo falhas, e falhas distintas em cada uma. Dessa forma, a existência de uma base única de código reduz a redundância do processo, e, consequentemente, reduz o risco de erros.

De acordo com (PINTO; COUTINHO, 2018), os tipos de aplicativos móveis dividem-se, além dos nativos, em híbridos e web. Os dois últimos são os que são construídos com uma base única de código, mas que são capazes de serem executados em SOs diferentes. Os Híbridos são aqueles que utilizam uma camada intermediária, normalmente construída com tecnologias web (e.g., Ionic, PhoneGap). Essa camada centraliza a construção das telas, o controle de todas as funções da aplicação e faz chamadas à camada nativa para utilizar os recursos do sistema. Essa abordagem introduz limitações em usabilidade e performance, pois a forma em que os componentes visuais são construídos tende a diferir dos componentes nativos, tornando a usabilidade menos intuitiva para um usuário da plataforma. Já em relação à performance, normalmente há uma redução causada pela adição da camada intermediária.

Por outro lado, os aplicativos baseados em web têm o intuito de reduzir a distância da camada de abstração até a camada nativa. Para isso, eles compilam parte, ou todo, o código gerado em bytecode da plataforma. A diferença entre as duas abordagens está na proximidade adotada por cada uma com as tecnologias nativas. Para o desenvolvimento desses tipos de aplicativos, não nativos, existem os chamados *frameworks* multiplataforma (também conhecidos como *cross-platform*).

Exitem diversas opções de *frameworks* multiplataforma no mercado. De acordo com uma pesquisa realizada pelo Stackoverflow (STACK OVERFLOW, 2023) em 2023 para os profissionais de programação, Flutter, React Native, Ionic e .Net MAUI apresentam maior destaque em popularidade.

2.1.1 Framework Nativo

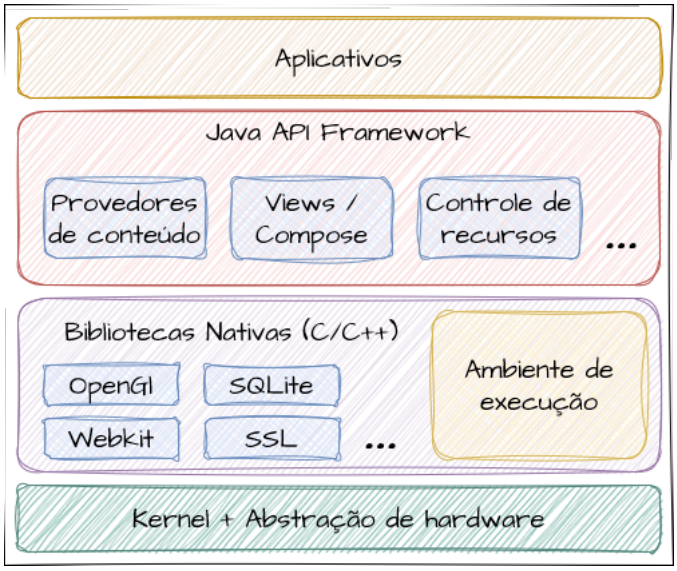
A plataforma Android disponibiliza duas formas para desenvolvimento de aplicações. Uma delas é o conjunto de bibliotecas nativas que inclui código nas linguagens de programação C e C++; e a outra é composta por um conjuto de APIs escritas em Java. Enquanto a primeira é somente indicada para reutilização de bibliotecas já existentes e para aplicações de alto desempenho, a segunda é onde a maior quantidade de ferramentas se concentra.

O termo "desenvolvimento nativo", na plataforma Android, normalmente refere-se ao conjunto de APIs escritas em Java. Nessa camada, são disponibilizadas ferramentas para construção de elementos visuais; acesso a todos os recursos do sistema operacional e comunicação entre diferentes aplicações. Enquanto as APIs em Java estão disponíveis no *Software Development Kit* (SDK) (GOOGLE LLC, s.d.b), as bibliotecas em C/C++ são disponibilizadas pelo *Native Develoment Kit* (NDK) (GOOGLE LLC, s.d.c).

Devido à existência do conjunto de APIs, também é possível utlizar outras linguagens de programação para construção de aplicativos móveis além de Java. Em 2017, a Google anuncionou o início de suporte oficial à linguagem Kotlin (MOSKALA; WOJDA, 2017). Atualmente, já há uma forte recomendação na utilização de Kotlin para Android devido à interoperabilidade de Kotlin com Java e a vantagens oferecidas como Null Safety (KO-

TLIN FOUNDATION, s.d.) e redução de linhas de código - em comparação com Java. Artigos como (PETERS; SCOCCIA; MALAVOLTA, 2021a) avaliam o impacto da migração entre as linguagens, e Kotlin já é utilizada por mais de 60% dos desenvolvedores da plataforma[1]. Vale mencionar que é possível também utilizar JavaScript através da integração de *WebViews* fornecida pela pelo *framework* e C/C++ utilizando as bibliotecas disponibilizadas pelo NDK.

Figura 2 – Arquitetura da plataforma Android simplificada



Conforme visto na Figura 2, a plataforma Android se estrutura da seguinte forma: é utilizado o Kernel Linux como base, que é responsável pelo funcionamento básico do sistema e fornece uma abstração do *hardware* para as demais camadas. Logo acima, bibliotecas nativas implementam funcionalidades centrais do sistema como banco de dados, *WebKit*, SSL, OpenGl e desenho de componentes visuais. O ambiente de execução ART (*Android Runtime*) é uma máquina virtual para execução das aplicações. Cada aplicação é executada em um processo diferente e conta com sua própria instancia do ART, que executa arquivos no formato *Dalvik Executables* (DEX), que são bytecode Java otimizados para a plataforma, gerados por alguma ferramenta de *build* como R8.

Com esse framework é possível:

- Construção de componentes visuais;
- Comunicação entre aplicações;
- Acesso aos sensores;
- Gerenciamento de arquivos locais como imagens, animações, fontes, entre outros;
- Controle de fluxo de telas, navegação e ciclo de vida da aplicação;

[1] https://developer.android.com/kotlin?hl

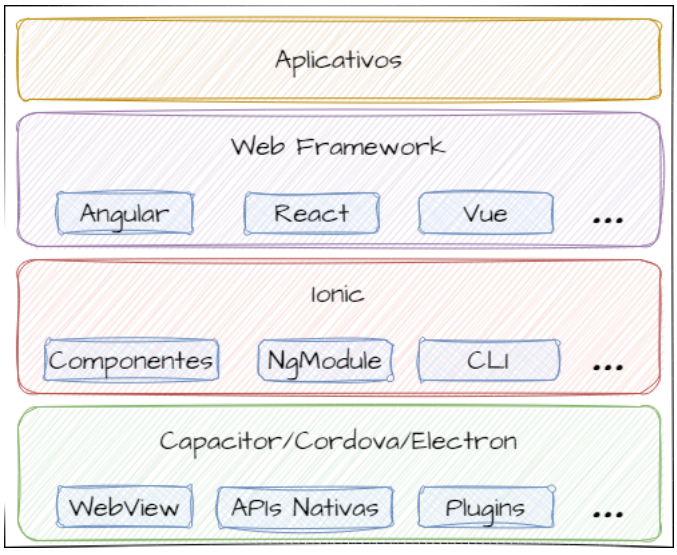
- Gerenciamento de notificações.

2.1.2 Ionic

Ionic é um *framework* híbrido em código aberto para desenvolvimento de aplicações utilizando tecnologias *web*. Todo controle do fluxo da aplicação e a construção dos componentes acontece através de *WebViews*, que são versões reduzidas e mais leves de navegadores, capazes de executar e exibir páginas *web*. Lançado em 2013, o Ionic busca fornecer para desenvolvedores *web* a possibilidade de construir aplicações móveis com bom desempenho utilizando as tecnologias já conhecidas desses profissionais. O *framework* disponibiliza o conjunto de componentes *web* produzidos para simular as aplicações nativas nas plataformas suportadas, oferecendo componentes, controle de fluxo de tela e interface de interação com o usuário.

O Ionic foi construído para ser extensível, integrando com diferentes ferramentas que fazem a ponte com as plataformas e dão ao *framework* sua maior força. Com ele, desenvolvedores *web* conseguem construir aplicações para quase todas as plataformas do mercado (Android, IOS, MacOS, Windows). Para desenvolvimento Android, o Capacitor (IONIC, s.d.) é a ferramenta indicada na sua documentação oficial, precedida pela antiga indicação do Cordova (THE APACHE SOFTWARE FOUNDATION, s.d.). Essas ferramentas oferecem o ambiente de execução JavaScript necessário para a execução do Ionic.

Figura 3 – Arquitetura simplificada do Ionic



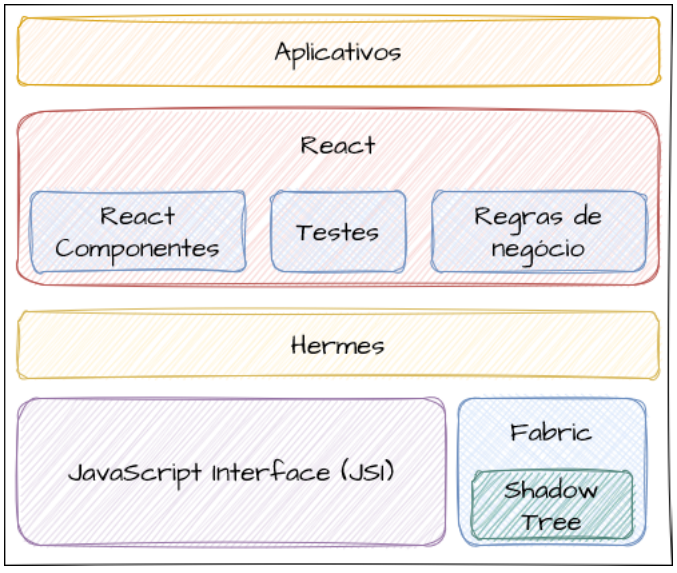
Para o seu funcionamento, o Capacitor utiliza a implementação nativa de *WebView* como motor de renderização do código JavaScript, de forma que uma aplicação completamente construída usando o *framework* é composta por uma tela com o componente nativo de *WebView*. O Capacitor oferece o controle dos recursos do dispositivo por intermédio de *plugins*, que são códigos escritos nas linguagens da plataforma e disponibilizados para a camada JavaScript. Também é oferecida a compilação da aplicação para a plataforma

desejada e a funcionalidades úteis como "Hot Reload"durante o desenvolvimento. A Figura ilustra as camadas da arquitetura do Ionic.

2.1.3 React Native

React Native é um *framework* em código aberto para desenvolvimento de aplicações Android e iOS que utiliza o *framework web* React(META OPEN SOURCE, s.d.b). Foi desenvolvido pela empresa Meta e liberado ao público em 2015. O React Native permite o desenvolvimento de aplicações utilizando tecnologias *Web* enquanto possui um sistema de construção de interfaces que utiliza componentes nativos de cada plataforma. Os elementos visuais são descritos utilizando a estrutura do React e são mapeados para os componentes disponibilizados pelas APIs nativas. Esse conjunto de componentes já fornecido pelo *framework* são chamados de *Core Components* (META OPEN SOURCE, s.d.a). O objetivo desse mapeamento de componentes é o garantir uma melhor usabilidade e aumento de performance na exibição.

Figura 4 – Arquitetura React Native simplificada



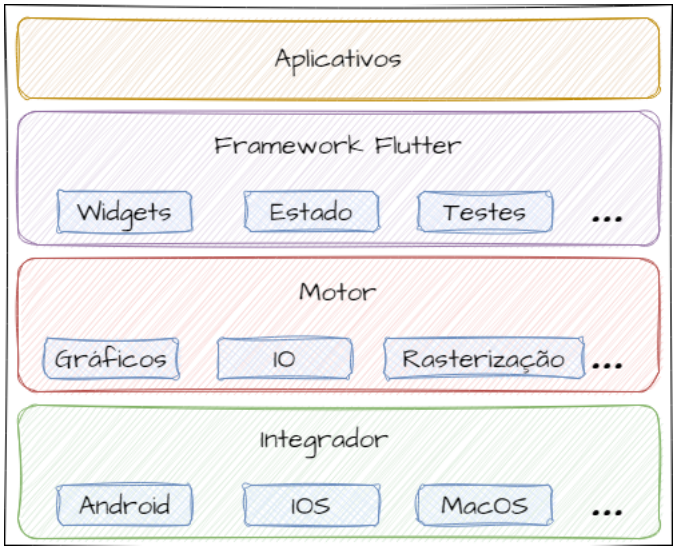
Na Figura 4 é possível visualizar a estrutura do *framework*. Para a execução da aplicação e a capacidade de utilizar componentes nativos da plataforma, o React Native utiliza um ambiente de execução diferente de *WebViews*. Hermes (FACEBOOK, INC, s.d.) é a solução padrão e não requer nenhuma configuração adicional para ser utilizada. Uma alternativa fornecida ao Hermes, é o JavaScriptCore (APPLE INC, s.d.). Para controle e atualização dos componentes visuais, o React Native utiliza o conceito de "*Shadow Tree*"que é uma estrutura de árvore usada para mapeamento dos componentes React para componentes nativos, utilizado pelo renderizador dos elementos visuais do *framework* - o Fabric (META PLATFORMS, INC, s.d.). As duas camadas utilizam a JavaScript Interface (JSI) para comunicação da camada JavaScript com a camada nativa e vice-versa.

2.1.4 Flutter

Flutter é um *framework* em código aberto lançado em 2017, desenvolvido pela Google e utiliza a linguagem Dart para a construção de aplicações. É capaz de gerar código para diferentes plataformas e dispositivos: Android, iOS, Windows, Linux, MacOS, Web e sistemas embarcados. Tem o objetivo de entregar um bom desempenho dada sua compilação para a arquitetura de cada plataforma, produtivo e flexível, dando a possibilidade de criação de componente sem utilizar códigos nativos das plataformas.

Durante o desenvolvimento, o Flutter executa um ambiente virtual de execução do código Dart para evitar o processo de compilação toda vez que alguma alteração no código for feita. Após o desenvolvimento, o Flutter compila o código diretamente para código de máquina, podendo ser instruções ARM, Intelx64 ou JavaScript no caso de *Web*.

Figura 5 – Arquitetura Flutter simplificada



O código escrito em Dart compõe a primeira camada de sua arquitetura, conforme visto na Figura 5. Nessa interface, é possível escrever instruções para composição de elementos visuais e regras de comportamento da aplicação. Ela conversa com o motor do Flutter, um conjunto de bibliotecas de baixo nível escritas em C ou C++ que são agnósticas a plataformas e fornecem o ferramental básico para a construção de aplicações, como a criação de pixeis a serem exibidos em tela, entrada e saída de dados da aplicação, acesso a arquivos locais, entre outras funcionalidades básicas. O motor comunica-se, então, com a camada de integração que disponibiliza o funcionamento básico do *framework*. Essa camada fornece implementações baixo nível, necessárias para execução, compilação e integração do Flutter com as funcionalidades de cada plataforma. Ela é escrita na linguagem da plataforma a ser compilada e permite que aplicações completas ou partes de uma aplicação sejam construídas.

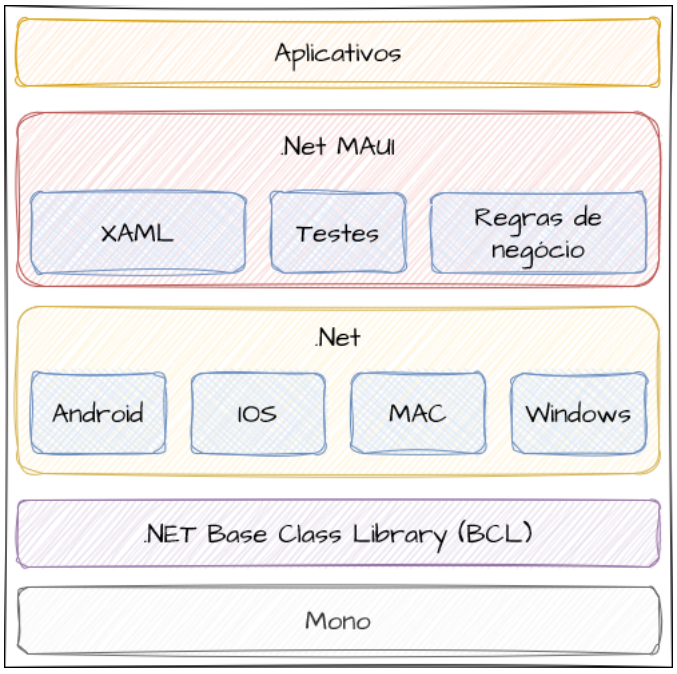
Flutter utiliza o conceito de *widget* para a construção de interfaces visuais reativas, utilizando o padrão declarativo para a construções dos componentes. Com inspiração na

biblioteca React (META OPEN SOURCE, s.d.b) da linguagem JavaScript, os elementos de *layout* são desenhados pelo próprio Flutter sem a utilização dos componentes disponibilizados pelo *framework* nativo.

2.1.5 .Net MAUI

.Net Multi-platform App UI, ou simplesmente MAUI, é o novo *framework* em código aberto para desenvolvimento de aplicações Android, MacOS, iOS e Windows. Lançado em 2022 pela Microsoft, MAUI surge para substituir outro *framework* desenvolvido pela mesma empresa, o Xamarin (MICROSOFT, s.d.b). Através de código C# e arquivos XAML, é possível construir *layouts* e controles da aplicação e do dispositivo. O *framework* oferece a funcionalidade de atualizações de aquivos XAML em tempo de execução, sem a necessidade de gerar novas versões para testar ajustes.

Figura 6 – Arquitetura .Net MAUI simplificada



O controle em todas as plataformas é abstraído pela .Net Base Class Library (BCL), que tem a função a abstrair a camada nativa a fim de simplificar o acesso aos recursos do dispositivo e componentes necessários para a construção de aplicações. Para a execução da BCL, a *Runtime Mono* é utilizada no Android, MacOS e iOS. No Windows, utiliza-se o .Net CoreCLR. Para cada plataforma existe uma implementação do .Net correspondente que fornece as especificidades de cada um delas. O MAUI é responsável pela disponibilização de componentes visuais e ferramentas de controle desses componentes. Esses componentes fazem um mapeamento com um componente nativo, tal que o MAUI utiliza essas implementações de forma que o código gerado funcione em todas. Na Figura é exposta a arquitetura simplificada do MAUI.

Para execução dos aplicativos em diferentes plataformas, o MAUI adota diferentes abordagens. Enquanto para o Android o código C# é compilado para uma linguagem intermediária e na inicialização do *app* é compilado em tempo de execução, no iOS o código é pré compilado para instruções de máquina (ARM).

2.2 COLETA DE DADOS DE TEMPO DE EXECUÇÃO DE DISPOSITIVOS ANDROID

Coletar dados durante a execução de uma bateria de testes é uma etapa essencial. Considerando os dispositivos móveis, existem basicamente duas maneiras possíveis de realizar esse processo, através de *hardware* ou através de *software*. Apesar de alguns tipos de dados estarem disponíveis em ambos os tipos de coletas, também há os que são restritos a uma forma ou à outra. Os dados de consumo energético, por exemplo, são comumente mais precisos se coletados por meio da instrumentação física dos dispositivos. Vale mencionar que a instrumentação para realizar essa coleta pode acarretar em danos nos dispositivos, pois tende a ser uma abordagem mais invasiva no aparelho. Em contrapartida, informações como uso de sensores e memória só são possíveis de serem coletadas através da camada de *software*. Para esse tipo de coleta é possível utilizar uma aplicação terceira que é executada em segundo plano durante os testes; ou podem ser usados utilitários nativos da plataforma, como os disponibilizados no Android *Debugging Bridge* (ADB) (GOOGLE LLC, s.d.a).

2.2.1 Android *Debugging Bridge*

Android *Debugging Bridge* é uma ferramenta de linha de comando que permite a comunicação de uma máquina com dispositivos Android. Ela é disponibilizada no pacote "platform-tools", presente no SDK da plataforma, e oferece uma variedade de ações que podem ser executadas no *smartphone*. Com essa ferramenta é possível executar comandos como: instalação e desinstalação de aplicações, visualização de registros de uso - também conhecidos como *logs* -, acesso a shell Unix para execução de outros comandos no dispositivo, entre outros.

O ADB funciona através de uma comunicação cliente-servidor de 3 componentes. O cliente executa comandos localizados na máquina de desenvolvimento e esses comandos são recebidos no servidor, que é localizado na mesma máquina. O servidor, então, é responsável pelo envio dos comandos para o terceiro componente, o dispositivo Android, que tem, por padrão, um processo em plano de fundo para interpretação e execução de comandos recebidos do ADB. A comunicação com o ADB utiliza o protocolo TCP entre as 3 partes, e faz conexão com os dispositivos Android a partir da porta 5555, até 5585.

2.2.1.1 Conexão

Para conectar uma máquina de desenvolvimento com um aparelho Android, a máquina requer a instalação do ADB, e o dispositivo móvel deve ter a opção adequada de depuração liberada no menu de desenvolvedor. Ao plugar o dispositivo com a máquina, via cabo USB, o ADB conectará automaticamente. Essa é uma forma simples e aplicável para muitos casos, porém pode se tornar inviável quando há a necessidade de testes com múltiplos dispositivos simultâneos, ou quando a medição de determinadas variáveis pode sofrer alterações devido ao tipo de conexão. Um exemplo está no fato de que a conexão USB transfere carga para o dispositivo, o que pode afetar no aferimento do consumo de energia. Nesses casos, a conexão via Wifi torna-se mais adequada.

Na necessidade de conexão via rede Wifi, o computador e o *smartphone* precisam estar na mesma rede, de forma que o endereço IP de um possa estar visível para o outro. Para estabelecer a conexão existem duas formas principais, utilizando comandos no terminal; ou através da interface padrão para desenvolvimento. Com o Ambiente de Desenvolvimento Integrado (IDE, do inglês *Integrated Development Environment*) instalado, é possível realizar a conexão utilizando QR code ou código de pareamento. Em ambos os cenários a depuração via Wifi deve estar liberada no dispositivo móvel.

A opção de conexão utilizando a IDE está disponível para dispositivos Android com versão 11 (API 30) ou superior. Basta selecionar a opção "*Pair Devices Using WiFi*" disponível na IDE, dentro do dispositivo, dentro do menu de depuração Wifi e escolher a forma de conexão. Já através de conexão USB, alguns comandos precisam ser executados, o primeiro comando é o `adb tcpip 5555`, para criar a conexão com o dispositivo, seguido do comando `adb connect` *ip_do_dispositvo* `5555`, para a conexão propriamente dita. Alguns comandos relacionados à conexão e que ajudam na resolução de problemas relacionados são:

- `adb devices` lista todos os dispositivos conectados ao ADB. É um bom comando para ser executado assim que o dispositivo é conectado no computador, pelo cabo USB, para conferência da conexão bem sucedida ou falha. O parâmetro `-l` pode ajudar a diferenciar os dispositivos em caso de muitas opções conectadas.
- `adb tcpip` *porta_para_conexão* serve para criar a conexão e fazer o dispositivo escutar comandos do servidor ADB pela porta porta escolhida.
- `adb connect` *ip_do_dispositvo porta_para_conexão* conecta o dispositivo com o ADB utilizando a rede Wifi.
- `adb start-server` inicializa o processo do servidor ADB.
- `adb kill-server` finaliza o processo do servidor ADB.

2.2.1.2 Shell Unix

O ADB disponibiliza um shell Unix através do comando `adb shell`, que fornece controle sobre o sistema operacional, e também informações sobre o funcionamento desse. Essa é uma ferramenta muito importante para depuração de problemas no sistema e para análise de recursos do sistema em estudos. Como alguns comandos costumam ser usados em muitos trabalhos e, particularmente, nos trabalhos que usaram o Ebserver, eles já foram implementados como padrão nessa ferramenta, para uso sem alterações necessárias no código.

Normalmente, durante coleta de dados em execução de testes, é necessária a informação dos recursos do sistema que foram utilizados pelos programas testados. Consumo energético, uso de processador e uso de memória são alguns dos recursos comuns a serem analisados e, portanto, essas análises foram adicionadas ao Ebserver por padrão. Para obtenção desses dados, o Ebserver utiliza o comando `adb shell dumpsys`, disponibilizados pelo shell Unix. Esse comando retorna uma grande variedade de informações, que o torna muito genérico e difícil de trabalhar diretamente. Por isso, existem algumas opções de parâmetros que podem ser usados para filtrar, ou formatar os dados, de acordo com a necessidade dos testes.

Para entender o consumo energético de processos no dispositivos, o ADB conta com o comando `adb shell dumpsys baterystats`. Esse comando retorna informações como o consumo energético do dispositivo durante uma determinada janela de tempo, e o tempo em que o determinado processo utilizou o processador. Como o sistema está constantemente coletando essas informações em segundo plano, é necessário reiniciar essa coleta removendo os dados anteriores aos experimentos executados, para garantir um estado inicial do teste limpo. Para reiniciar essa janela de tempo, é necessário utilizar o comando `adb shell dumpsys batterystats –reset`. O *batteryStats* guarda informações detalhadas sobre o consumo energético de cada aplicativo instalado, serviços e demais componentes do sistema. Mostra, também, eventos relacionados à bateria, como descarga, carga, modo de economia, entre outras informações. *Battery Historian* é uma ferramenta capaz de interpretar os dados gerados e que mostra as informações de maneira visual.

Consumo de memória é outro atributo bastante comum em análise de aplicações. `adb shell dumpsys meminfo` é um bom comando comando para entender um corte de como a memória está sendo utilizada por processo, esse comando fornece informações como uso total de memória, estatísticas de memória VSS (*Virtual Set Size*), RSS (*Resident Set Size*), PSS (*Proportional Set Size*), entre outras informações. Todavia, esse comando carece de informações ao longo do tempo de execução do teste. Para esse fim, existe o comando `adb shell dumpsys procstats` [2], que fornece informação de valores mínimos, médios e máximos de consumo, e de outras métricas de memória, pelo período de tempo requisitado. Já que o comando captura dados ao longo de tempo, é necessário

[2] https://android-developers.googleblog.com/2014/01/process-stats-understanding-how-your.html

definir corretamente essa janela de tempo, com parâmetros como `–hours 1`, que utiliza informações da última hora de leitura e `–reset`, que reinicia a leitura.

Fluidez de uma aplicação é sempre um tópico importante na escolha de uma ferramenta para construção de elementos visuais, o `adb shell dumpsys gfxinfo` fornece informações detalhadas do desempenho gráfico de um aplicativo Android. Dados como quantidade de quadros gerados, percentis de tempo de renderização, Janky Frames - quadros que foram não foram gerados suavemente, podendo impactar negativamente na usabilidade -, e histograma do tempo de renderização do quadros são úteis para testar diferentes abordagens e medir os ganhos ou perdas com cada uma. Para utilizar o utilitário é necessário reiniciar a janela de leitura da ferramenta `adb shell dumpsys gfxinfo reset` com o comando e evitar sujeira nos dados gerados, após a execução da aplicação testada o comando deve ser chamado novamente sem o parâmetro de *reset*.

2.2.2 Android Studio Profilers

O Android Studio é o Ambiente de Desenvolvimento Integrado oficial para desenvolvimento e manutenção de aplicações Android, desenvolvido pela Google em conjunto com s IntelliJ. Anunciado oficialmente em 2013 (GOOGLE LLC, s.d.d) e baseado em outra IDE, o IDEA, o Android Studio é, desde então, a escolha padrão quando se fala em desenvolvimento na plataforma. No ano de 2023, tanto a IntelliJ IDEA quanto o Android Studio estiveram entre as seis IDEs mais utilizadas entre os profissionais que responderam a pesquisa do StackOverflow (STACK OVERFLOW, 2023).

Para execução de funções básicas como compilação, depuração e emulação de dispositivo, o Android Studio utiliza o Android SDK. Ele também fornece integração com ferramentas terceiras como o Gradle (GRADLE INC, s.d.), para gerenciamento de dependências, e Git (TORVALDS, s.d.), para controle de versão. Enquanto oferece a prévia dos componentes visuais durante o desenvolvimento, também permite alteração no comportamento da IDE através de *plugins*, além de um conjunto de ferramentas para inspecionar o comportamento e a performance de aplicações, como LogCat (GOOGLE LLC, s.d.f), Macro Benchmark (GOOGLE LLC, s.d.g) e Micro Benchmark (GOOGLE LLC, s.d.h).

Outras ferramentas, focadas em analisar o desempenho das aplicações, são os *profilers* (GOOGLE LLC, s.d.j). *Profilers* são úteis para entender problemas que possam acontecer com uma aplicação, que afete o desempenho dela mesma, ou de outras aplicações. Um detalhe importante a notar na utilização dos *profilers*, é que alguns deles não podem ser utilizados quando a aplicação foi compilada para versão de produção (*release*) por questão de segurança. A aplicação nessa versão garante uma maior otimização feita pelo compilador da plataforma, porém a sua otimização limita a capacidade das ferramentas.

A Tabela 1 expõe os *profilers* disponíveis no Android Studio e um resumo de suas funcionalidades, bem como uma avaliação sobre o funcionamento do determinado *profiler* em versões de produção. De maneira geral, essas ferramentas são úteis para resolver

Tabela 1 – Android profilers

Profilers	Capacidades	Funciona em Release
Memory	Contém uma quantidade granular de alocações de memória e uso de *heaps*; informações como referências ao Java *Native Interface* (JNI), que é a camada de comunicação entre a API Java e as bibliotecas escritas de C/C++ do Android, e informações sobre os diferentes *heaps* usados pelo sistema. O Android Studio oferece uma interface para interação com as informações dispostas, funções como gravação das alocações em tempo real, gráficos, controle do *Garbage Collector*, entre outras funções úteis.	Funciona parcialmente. Possui linha do tempo de eventos, mas informações de *head* e gravação de alocações em tempo real não funcionam.
CPU	Mostra uso de CPU a nível de métodos e *Threads*. É possível entender quais chamadas estão sendo feitas para as diferentes camadas do sistema em um período de tempo, mostra também as atividades do sistema em tempo real enquanto a aplicação transita entre diferentes estados do ciclo de vida.	Funciona parcialmente; a linha de tempo de eventos não funciona, e não é possível usar a API para iniciar a gravação dentro do código.
Network	Mostra todas as requisições e respostas realizadas pela aplicação em uma linha de tempo. Informações como tamanho das mensagens, tempo de resposta, *status*, cabeçalho e corpo da requisição estão disponíveis. Também é possível analisar em quais *threads* cada comunicação aconteceu.	Não funciona.
Energy	Apresenta um valor aproximado do consumo energético da aplicação em uma linha de tempo, e os eventos da linha de tempo que aconteceram durante cada momento. Oferece também uma visão mais detalhada separando o consumo por CPU, rede e localização. Selecionando uma região, a ferramenta disponibiliza informações de Alarmes, *Wake locks* e *Jobs* do sistema.	Não funciona.
Power	Diferentemente do *Energy profiler*, o *Power profiler* separa o consumo dos diferentes recursos do sistema em raias. É possível separar o consumo nos diferentes tipos de núcleo de CPU, câmera, GPS, display, rede, entre outros. Informações da bateria e do consumo de cada momento também são apresentados de forma mais detalhada.	Funciona completamente.

problemas de consumo indevido dos recursos dos dispositivos. Portanto, não são úteis para um processo de análise automatizado de aplicações.

2.2.3 Trepn Profiler

Desenvolvido pela Qualcomm, o Trepn Profiler (QUALCOMM TECHNOLOGIES, INC., s.d.a) é um aplicativo de monitoramento e diagnóstico para dispositivos móveis, bastante completa para análise de uso de recursos. A ferramenta é capaz de coletar dados de uso de CPU, GPU, Wi-Fi e consumo de energia de forma muito útil, com *overlays* para acompanhamento em tempo real dos recursos, e gráficos para análise ao longo do tempo. Ela é feita para ser executada em plano de fundo no dispositivo enquanto outras aplicações são

testadas. O Trepn Profiler ganhou certa atenção pelo fato de sua empresa mantenedora ser uma das fabricantes de chip para dispositivos móveis, e por também funcionar em *smartphones* cujos chips são oriundos de outros fabricantes. Atualmente, essa ferramenta foi descontinuada, visto que a Qualcomm não disponibiliza mais o seu download, nem no site da empresa nem na loja de aplicativos do Android, a alternativa oferecida pela Qualcomm é o Snapdragon Profiler (QUALCOMM TECHNOLOGIES, INC., s.d.b), que é uma aplicação *desktop* - Windows, Linux ou MacOs.

2.2.4 Coletas com instrumentação física

Para alguns contextos de execução de testes, a coleta física resulta em dados mais autênticos do que os modelos utilizados pelas ferramentas de *software*. Nesse cenário, existem ferramentas que, a partir de uma instrumentação física do dispositivo, coletam as informações necessárias. O caso mais comum dessa forma de coleta é para analisar o perfil de consumo energético do teste executado. Essa abordagem tem a vantagem de ser mais precisa, pois o controle de energia que é disponibilizada para o dispositivo é feito pela ferramenta, para isso é feita uma instrumentação física no dispositivo, esse processo normalmente consiste na remoção da bateria e a solda de fios nos pontos de contatos que eram usados pela bateria, assim o *Power Meter* consegue fornecer energia para o dispositivo de maneira controlada. Essa característica mais invasiva pode ser perigosa, podendo dificultar o processo de testes ou até danificar o dispositivo testado, um outro problema é o custo que normalmente é elevado.

Monsoon (MONSOON SOLUTIONS, INC., s.d.) é uma combinação de um dispositivo de medição de corrente com uma fonte de energia. Oferece uma interface mais madura de utilização através do *software* proprietário PowerTool. Para utilização dessa ferramenta, são oferecidas APIs escritas em Python, e PowerTool Automation API para configuração da interface e alterações no comportamento do *software*. Monsoon disponibiliza diferentes formas de fornecimento de energia e uma porta USB, que além de servir para comunicação do dispositivo com um computador, também pode ser utilizada para fornecer energia. É uma ferramenta completamentar, focada na análise de consumo energético. Por isso, não oferece nenhum suporte para outras métricas ou execução de experimentos.

2.3 TRABALHOS RELACIONADOS

Testar uma aplicação normalmente envolve testar muitos cenários. Celulares hoje em dia são equipamentos complexos com muitas configurações e muitas versões de *software*. Há uma grande variedade de tipos de testes sendo executados na academia, que variam entre análise de segurança, performance, consumo energético, entre outros.

Hasan (HASAN et al., 2016) comparou o consumo energético de Coleções da linguagem Java. Cinco listagens, seis dicionários e seis conjuntos foram testados individualmente, e

em aplicações reais, para entender o consumo e qual seria o impacto em cada estrutura de dados. Resultados mostraram um aumento potencial de 300% na escolha inadequada de Coleção, e redução de até 38% no consumo energético em escolhas otimizadas. Para isso, a ferramenta GreenMiner (HINDLE et al., 2014) foi utilizada, que se trata de uma combinação entre um Raspberry Pi[3] e um Arduino[4] para realizar testes nos dispositivos. Enquanto o Raspberry executa comandos no dispositivo, o Arduino coleta informações de energia. A instrumentação física envolveu a remoção da bateria e a solda de cabos no lugar de contato da bateria. O Raspberry se conecta com o Arduino por pinos de comunicação, e com o celular pela porta USB. Já o Arduino, conecta-se com o sensor de Tensão e Corrente INA219. Dados de coleta são acrescidos com informações obtidas pelo Raspberry e enviados a um serviço *web* que disponibiliza uma visualização detalhada das informações através de gráficos, matriz de similaridade e listagem.

O trabalho (LINARES-VÁSQUEZ et al., 2014) analisou 55 aplicativos diferentes para entender o custo de chamadas às APIs do Android e identificar as chamadas que são mais custosas para a bateria. Para a coleta dos dados de energia, o Monsoon foi utilizado. O trabalho analisou um total de 807 métodos e percebeu um consumo mais elevado em 131 deles. Foi investigado também quais padrões de múltiplas chamadas APIs apresentam maior consumo - 2 ou 3 chamadas.

Equipamentos físicos para medir o consumo de energia são normalmente mais confiáveis, porém. como mostrado por (NUCCI et al., 2017b), os erros gerados na utilização de ferramentas de *software* podem ser diminutos e, portanto, confiáveis.

Para tentar entender como diferentes tipos de programas se portam, ProfileDroid (WEI et al., 2012) propôs um sistema para caracterização de aplicações utilizando informações em várias camadas. Análises estática, de interação com a tela e do sistema operacional, foram estudadas em aplicações gratuitas e pagas, disponíveis na loja da plataforma, e notou-se um perfil de uso de recursos discrepante, no qual um consumo elevado de rede foi notado em aplicações gratuitas. Esse trabalho utilizou uma ferramenta para copiar as ações no dispositivo com o intuito de automatizar o processo, e comandos via *ADB shell* para extrair informações dos programas e do sistema.

Palomba (PALOMBA et al., 2019) realizou um estudo de consumo de energia utilizando análise estática de código como métrica em aplicações nativas. Em um total de 60 aplicações, a refatoração do trecho com os padrões de maiores custo energético chegou a consumir até 87 vezes menos energia. Para a análise do consumo de energia, foi utilizada uma ferramenta de coleta baseada em *software*, PETrA. *Power Estimation Tool for* Android (PETrA) (NUCCI et al., 2017a) é uma ferramenta de automação de execução de testes, com interface gráfica para obtenção de dados de consumo energético de aplicações. MoneyRunner (GOOGLE LLC, s.d.i) ou Monkey (GOOGLE LLC, s.d.l) são utilizados para

[3] Raspberry Pi - <https://www.raspberrypi.com/>
[4] Arduino - <https://www.arduino.cc/>

interagir com a interface do dispositivo, em conjunto com um arquivo que representa o perfil de consumo de energia[5]. Com as devidas configuração feitas, o PETrA executa a bateria de teste e apresenta a lista de comandos utilizados junto com o consumo de cada um.

Quanto a ferramentas de automação de testes, PyAnaDroid (RUA; SARAIVA, 2023a) é focada em custo energético de aplicações escritas em Java ou Kotlin. É feita uma instrumentação da aplicação e a ferramenta coleta os dados durante a execução. PyAnaDroid é escrita em Python, com várias funções atreladas ao recurso alvo, como a detecção de possíveis pontos de consumo elevado de energia, e padrões no código, por meio de analise estática e execução de testes com interação. Enquanto se mostra como uma boa ferramenta para coleta de consumo energético, limita-se a esse recurso e não compara outras características do sistema.

Uma ferramenta mais genérica é apresentada por Ivano (MALAVOLTA et al., 2020), que é capaz de coletar uma grande variedade de informações do dispositivo e executa o processo de automação por comandos à interface gráfica salvos em um arquivo. A ferramenta tem uma grande variedade de funcionalidade, mas falta um modo de execução de experimentos cujo tempo limite de execução não é conhecido para testes de *benchmarks*.

E-MANAFA (RUA; SARAIVA, 2023b) é uma ferramenta para análise do consumo de recursos em uma aplicação com foco no consumo de energia. Está disponibilizada como uma biblioteca na linguagem Python ou linha de comando, que, com pouca configuração, consegue coletar informações de eventos do sistema que aconteceram durante a execução do teste. Para execução dos testes, a ferramenta disponibiliza integração com Monkey (GOOGLE LLC, s.d.l) ou DroidBot (LI et al., 2017).

Android Runner (MALAVOLTA et al., 2020) é uma ferramenta, escrita na linguagem Python, para automação de testes em dispositivos Android. É capaz de executar uma variedade de ferramentas terceiras para coleta de informações de forma simples, como Monsoon (MONSOON SOLUTIONS, INC., s.d.), Trepn Profiler (QUALCOMM TECHNOLOGIES, INC., s.d.a) e outras. Tem uma arquitetura de orquestrador e *plugins* que permite a implementação de uma forma nova de coleta. Oferece funcionalidades como *replay* de interação com interface, na qual o usuário utiliza o aplicativo uma vez e a ferramenta guarda as ações realizadas para repetir durante a automação.

A comparação de diferentes *frameworks* multiplataforma também é alvo de pesquisas, como em (NAWROCKI et al., 2021), que comparou React Native, Flutter e Xamarin com a solução nativa. Para isso, foram desenvolvidas 2 aplicações em cada *framework*, que foram geradas para Android e iOS. Uma aplicação simples, somente com uma tela em branco com um texto no centro, e outra aplicação mais avançada, que contém uma tela com vários botões, um tela de formulário e uma tela de listagem. Cada tela tinha um programa executando em plano de fundo, com a intenção de executar atividades de CPU, leitura e

[5] https://source.android.com/docs/core/power/values?hl=pt-br

escrita de arquivo, e outro realizando requisições. Métricas como tempo de inicialização da aplicação, uso de memória e CPU foram utilizadas para comparação. Flutter, mesmo que com consumo elevado de memória, se mostrou o mais performático entre os *frameworks* multiplataforma do estudo.

Matteo e Ombretta (CIMAN; GAGGI, 2017b) também apresentam outro trabalho que compara as diferentes plataformas com diferentes *frameworks*. O trabalho analisa o consumo de energia através de quatro dispositivos, dois com Android e dois com iOS. Phonegap, Titanium e MoSync foram os *frameworks* utilizados, em conjunto com uma aplicação web. Diferentes aplicações foram desenvolvidas para testar diferentes sensores nos dispositivos - acelerômetro, GPS, compasso, sensor de proximidade, sensor de luz, câmera, orientação e gravação de áudio. Os resultados foram comparados com uma aplicação nativa e mostraram um acréscimo no consumo ao utilizar *frameworks* multiplataforma. Os resultados mostraram, também, que o *framework* de melhor desempenho em uma plataforma não necessariamente é também em outra. Como exemplo, enquanto no iOS o Titanium é mais otimizado, no Android o Phonegap se mostra melhor em alguns cenários.

Biørn-Hansen e outros (BIØRN-HANSEN et al., 2020) compararam cinco *frameworks* multiplataforma, utilizando o desenvolvimento Android nativo como base da comparação. MAML/MD_2, NativeScript, React Native, Flutter e Ionic foram comparados para entender o custo adicional em relação ao nativo. Para comparação, o estudo construiu uma aplicação para cada *framework*, capaz de testar diferentes cenários. Dentre eles, pode-se mencionar o uso do acelerômetro, uma tela de listagem de contatos, leitura de arquivo do sistema, e geolocalização. Para coleta das informações, foram utilizadas as ferramentas presentes no *Android Studio*. Para cada cenário de teste, tempo de execução, uso de CPU e memória foram coletados. Os resultados mostraram um melhor desempenho da solução nativa, com *frameworks* multiplataforma apresentando resultados melhores em alguns casos. O MAML/MD_2 apresentou, em média, o melhor resultados entre os multiplataformas.

Huber (HUBER; DEMETZ; FELDERER, 2020) apresenta uma comparação entre os *frameworks* Ionic, React Native e o desenvolvimento nativo, em que o trabalho desenvolveu uma aplicação de listagem e gerenciamento de contatos - que foi reutilizada na presente pesquisa. Três dispositivos Android foram utilizados e as métricas de comparação foram uso de CPU, memória, contagem de quadros e memória de vídeo. Os resultados mostraram uma boa fluidez do React Native em comparação ao desenvolvimento nativo, porém o React apresentou maior uso de recursos nas outras métricas. Já o Ionic desempenhou pior na contagem de quadros gerados e utilizou os demais recursos de forma aproximada ao React Native.

Dos trabalhos encontrados durante a elaboração desta pesquisa, a grande parte foca em um determinado contexto para comparação. *Frameworks* multiplataforma evoluem rapidamente, de forma que trabalhos antigos podem perder relevância durante as atua-

lizações feitas nessas ferramentas. Muitos trabalhos focam em comparar o consumo de poucos recursos - normalmente energia - ou focam em implementações muito simples de aplicações que não necessariamente representam aplicações reais. Esta pesquisa visa contribuir com os trabalhos de comparação de *frameworks* utilizando os de maior popularidade da atualidade em diferentes cenários e com um conjunto elevado de métricas, para assim facilitar o processo de escolha de qual *framework* resolve determinados problemas de forma mais eficiente.

3 EBSERVER

A execução de testes em dispositivos móveis e a coleta de métricas geradas são etapas comumente repetitivas, e tendem a gerar uma grande quantidade de informações resultantes. Esse padrão pode ser visto tanto em trabalhos científicos, quanto em análises feitas na indústria, devido à vasta quantidade de configurações existentes dos dispositivos Android. Características como a quantidade de memória, capacidade de bateria e modelo do processador impactam nos mais diversos cenários, sendo interessante a realização de testes nas maiores quantidades possíveis de configuração para entender o real impacto do aplicativo em análise. A maneira mais simples e direta de coletar dados é de forma manual.

A execução manual é útil para observação de pequenas amostras dos cenários de testes. No início de análises, é importante para o entendimento dos dados coletados, para se ter noção dos arquivos gerados (quantidade, tamanho, etc.), do momento em que cada arquivo precisa ser gerado, e uma primeira avaliação dos resultados, a fim de confirmar - ou pivotar - a escolha das variáveis analisadas. Com a evolução do teste, a quantidade de variáveis analisadas tende a crescer, o que torna a coleta manual inviável. A quantidade de execuções cresce, assim como o tempo de execução. Há ainda casos em que o cenário de teste é muito rápido e a velocidade da coleta manual afeta o resultado do teste, além da suscetibilidade à falha humana.

Por isso, as pesquisas tornam-se propensas a envolver a criação de alguma ferramenta de automação para execução de testes e coleta das informações dos dispositivos. Com esse exato contexto, foi criado o Ebserver, uma ferramenta de automação de testes e coleta de dados, operados em dispositivos Android. Ele simplifica a organização e execução da ferramenta de coleta dos dados, sendo alheio a essa. O Ebserver é capaz de executar testes em múltiplos dispositivos simultaneamente e ainda elimina a necessidade de que um período de tempo seja determinado para para finalização do teste.

Graças à sua interface simples, e à possibilidade de configuração e customização para diferentes contextos, o Ebserver já foi usado em vários trabalhos científicos e maturado para ser uma ferramenta mais robusta. É uma ferramenta de simples entendimento e configuração, com o objetivo de automatizar o processo de testes e de fácil integração com diferentes ferramentas de coleta de dados. Por padrão, o Ebserver já é integrado com uma ferramenta de medição do consumo de recursos a nível de processos do sistema, e oferece duas formas de execução para atender a diferentes necessidades.

Este capitulo apresenta o Ebserver detalhando o funcionamento da ferramenta e seus componentes. A Seção 3.1 apresenta os cenários que se beneficiam do uso do Ebserver, a Seção 3.2 explica as formas de execução da ferramenta, e a Seção 3.3 apresenta a estrutura dos arquivos do Ebserver, explicando como configurar, a função de cada arquivo e como

os resultados são armazenados em arquivos.

3.1 CENÁRIOS DE USO

A implementação atual do Ebserver, focada no sistema operacional Android, é capaz de atender a uma grande gama de testes. Desde simples execuções de uma aplicação que não requer parâmetros de entrada, até testes mais complexos, que exigem simulação de interação com interface gráfica, ou programas cujo tempo até a finalização não seja conhecido em primeiro momento. Ambos testes de caixa branca e preta (NIDHRA; DONDETI, 2012) também são compatíveis com o Ebserver.

Para os testes de caixa branca, é preciso instrumentar a aplicação testada. Essa instrumentação deve adicionar uma comunicação entre o dispositivo e o Ebserver para controle do fluxo, na qual o dispositivo deve avisar ao Ebserver quando invocar os programas de coletas e quando interromper a coleta dos dados. Já os testes que envolvem aplicações com interação com elementos gráficos, não terão instrumentação direta na aplicação em teste, mas necessitam de ferramentas adicionais para interação com a interface. Até então, as ferramentas de interação com a interface gráfica que o Ebserver dá suporte são Expresso (GOOGLE LLC, s.d.e) e UiAutomator (GOOGLE LLC, s.d.k).

O Ebserver é de utilidade para os seguintes cenários:

- Quando procura-se uma ferramenta de automação simples e de rápida configuração, dessa forma menos tempo é desprendido para entendimento e uso da ferramenta, e mais tempo é utilizado para a análise dos testes.
- Quando há necessidade de uma ferramenta madura, já testada em outros trabalhos científicos.
- Quando é necessário um controle a nível de função, de forma a iniciar e finalizar o experimento em momentos específicos sem a determinação de um período de tempo fixo.
- Quando parâmetros são necessários para controle dos experimentos, e diferentes dispositivos necessitam de diferentes parâmetros para atender às limitações de hardware de cada um.
- Quando a velocidade é um requisito importante, já que o Ebserver consegue executar os testes em vários dispositivos simultaneamente sem configuração adicional.
- Quando é preciso adicionar mais métricas para coleta, visto que o Ebserver é flexível a isso de forma rápida, com poucas alterações no projeto.

3.2 MODOS DE EXECUÇÃO

Ebserver é construído utilizando o ambiente de execução JavaScript Node.JS e conta com dois modos de operação, **Execução Individual** e **Execução de UI**. O primeiro, é onde se concentra a maior contribuição deste trabalho. Nele, são oferecidas funcionalidades como a possibilidade de um controle mais granular do teste (em que funções específicas do código são testadas) e a capacidade de executar testes cujo tempo de duração da execução é desconhecido. Por outro lado, há a necessidade de acesso ao código fonte da aplicação testada, e não há possibilidade de execução de ações no dispositivo. Já o segundo modo, é uma extensão deste trabalho para habilitar a execução de testes que envolvem ações de interação no dispositivo, através de ferramentas modernas de automação - Expresso(GOOGLE LLC, s.d.e) e Ui Automator(GOOGLE LLC, s.d.k).

3.2.1 Execução Individual

Para deixar o Ebserver mais adaptável, sua comunicação com os dispositivos é feita via REST API. Dessa forma, diferentes aplicações, utilizando diferentes tecnologias, podem estabelecer conexão com a ferramenta de forma fácil. No modo de execução individual, é necessária a criação um arquivo de configuração com instruções dos parâmetros de execução de cada cenário de teste. Esse arquivo permite que a coleta das métricas aconteça de forma automática. A partir daí, uma instância do servidor Node pode ser criada para dar início ao teste.

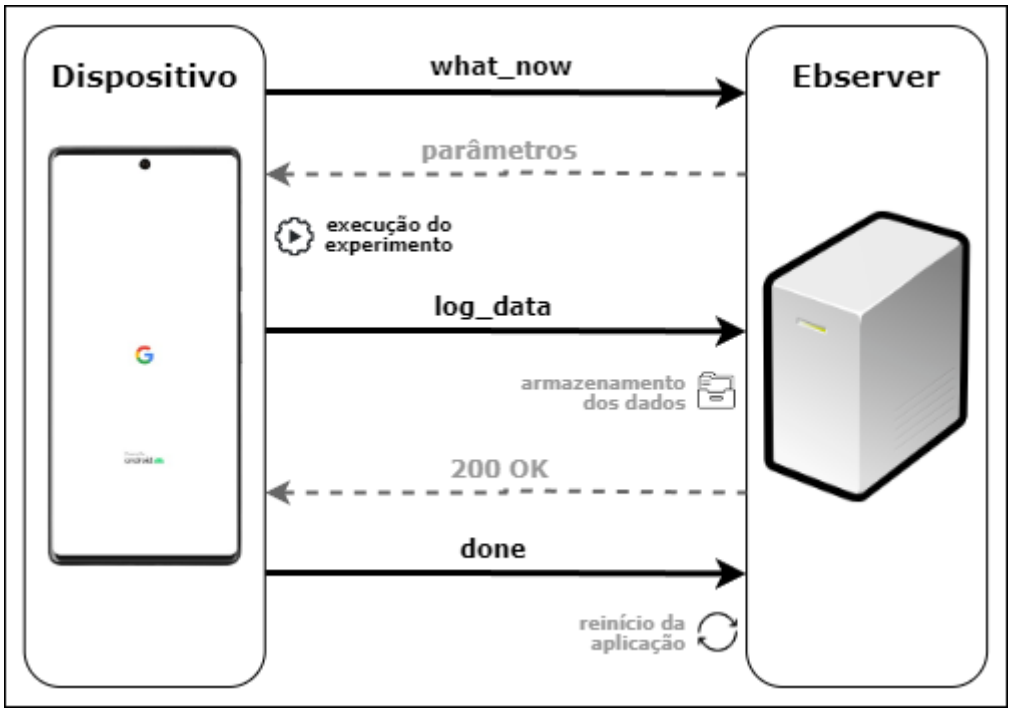
Como pré requisito da primeira execução, a aplicação testada deve ser inicializada manualmente no dispositivo. Assim, o aplicativo instrumentado faz a primeira chamada ao serviço do Ebserver, expondo o IP do dispositivo e enviando as informações necessárias para seguimento da coleta das métricas de forma automática. O Ebserver armazena as informações recebidas e as utiliza para reinicialização da aplicação entre os testes, execução de comandos e organização de pastas.

No lado do dispositivo, o código instrumentado da aplicação sob testes também deve ser responsável por comunicar ao Ebserver quando a coleta deve iniciar e finalizar. Com essa informação, alguns impactos nos resultados do teste podem ser mitigados, como *garbage collector*, compilação de código na inicialização da aplicação, entre outros fatores. Essa comunicação acontece em 3 diferentes momentos, nomeados ***what_now***, ***log_data*** e ***done***. Os dois primeiros estão relacionados à execução do experimento, enquanto o terceiro é dedicado à finalização do programa e inicialização para uma nova execução. Para o funcionamento correto, o dispositivo Android precisa ser instrumentado para fazer requisição HTTP em cada umas dessas etapas. O Ebserver utiliza-se de ferramentas de coleta através de comandos no terminal e direciona o resultados desses comandos para gerar arquivos. Os comandos ao ADB são cadastrados no arquivo **adbCommands.js** que são expostos para o arquivo **ebserver.js** a fim de serem executados executados em um

dos 3 momentos.

Os 3 momentos servem para possibilitar a inicialização correta das aplicações, limpar os dados gerados em uma etapa de teste e reiniciar a aplicação para uma nova rodada de teste. A Figura 7 apresenta, de forma reduzida, a comunicação entre o dispositivo e o Ebserver. De forma mais detalha, as etapas de comunicação são descritas a seguir:

Figura 7 – Fluxo de execução de experimentos de forma simplificada



- **what_now** deve ser executado no início da medição. Nesse momento, o Ebserver executa os comandos para iniciar a leitura de forma que os comandos seguintes só analisem dados do momento da execução, descartando dados anteriores. No caso de comandos ADB, os parâmetros comuns para essa finalidade são --reset"ou --clear". Na primeira vez que essa comunicação acontece, o Ebserver cria uma instância do dispositivo em sessão para automatizar as demais chamadas. No lado do dispositivo, é esperado que qualquer preparação anterior à execução do teste já tenha sido executada. Por fim, o Ebserver retorna qual parâmetro o dispositivo deve executar para o teste requisitado.

- **log_data** deve ser chamado imediatamento após o fim da execução. O Ebserver executa os comandos responsáveis pela coleta dos dados e cria os arquivos com as respostas dos comandos. O retorno dessa comunicação para o dispositivo denota o momento de limpeza das informações permanentes geradas durante essa execução do teste. Tabelas em banco de dados, arquivos e informações em cache da aplicação são exemplos de dados a serem removidos.

- **done** deve ser chamado após a limpeza citada no **log_data**. Nesse momento, o Ebserver executa os comandos para fechar e reiniciar a aplicação para execução de um novo teste.

A requisição do dispositivo deve ser do verbo GET do padrão de comunicação HTTP. Em todas a chamadas é necessário enviar informações do teste em execução ao Ebserver. Essas informações são enviados no cabeçalho da chamada e são necessárias para execução correta dos comandos cadastrados e nomeação dos arquivos. São válidos os seguintes cabeçalhos:

- ***device*** informação necessária para identificação do dispositivo, utilizada na sessão e no nome da pasta para armazenar os testes realizados no dispositivo. O valor enviado é utilizado para encontrar o arquivo de configuração específico do dispositivo. Nos artigos que se utilizam do Ebserver, o **Build Model** do dispositivo tem sido utilizado para preencher esse campo.
- ***application_id*** necessário para execução de comandos pelo Ebserver. É com essa informação que o Ebserver sabe identificar qual aplicação deve fechar, iniciar e limitar os dados gerados pelos comandos somente ao teste em execução.
- ***test_type*** utilizado para separação dos diferentes testes que podem ser configurados no Ebserver, de forma que o Ebserver seja capaz de guardar uma variedade de configurações de testes a serem executados no dispositivo. Esse parâmetro precisa ser uma cópia do valor que está no arquivo de configuração do dispositivo.
- ***activity*** necessário para inicialização automática da aplicação, o campo deve corresponder ao nome da *Activity* principal da aplicação.
- ***framework*** utilizado para organizar os arquivos gerados por tipo de *framework* em teste.

3.2.2 Execução de UI

De forma complementar à **Execução individual**, a Execução de UI serve para testes cujo objetivo é simular a interação com a interface gráfica - ações como toque curto e longo, toque múltiplo, movimento de pinçar ou arrastar, entre outros. Esse tipo de interação não é possível de executar no modo individual pela natureza da comunicação existente, em que as entradas do experimento - nome e parâmetros - são passadas no início da execução e têm valor estático. Não obstante, o controle de início e parada é feito na própria aplicação a ser testada, sendo um teste necessariamente de caixa branca. Já na Execução de UI, não é necessário ter acesso ao código da aplicação testada, sendo um teste de caixa preta.

Ebsever suporta a integração com as ferramentas Espresso e UI Automator. Para executá-las, é necessário escrever um arquivo com o conjunto de instruções a serem feitas

na tela do dispositivo, utilizando as regras da própria ferramenta escolhida. Esse arquivo deve ser passado como parâmetro para o Ebserver, juntamente com a descrição de qual *framework* está em uso (como Nativo, Flutter, etc.) e a e quantidade de execuções da bateria de testes. Essas informações são escritas no Ebserver no no arquivo **testModule.js**. Os parâmetro de execução do Ebserver são:

1. Método a ser chamado
2. Localização da classe de teste
3. O identificador do pacote da aplicação a ser testada
4. Sufixo do teste informando qual ferramenta executar

Um exemplo dessa inicialização utilizando Espresso seria:

```
1 $ node ebserver.js sampleMethod sampleTestClass \
2 > com.example.sampleApplication \
3 > android.support.test.runner.AndroidJUnitRunner
```

Listing 3.1 – Exemplo de uso de Execução de UI

3.3 ESTRUTURA DE ARQUIVOS

Como o objetivo na construção do Ebserver foi deixá-lo simples, de forma a ter somente o necessário para automatizar a execução de experimento, o projeto tem somente cinco arquivos base para funcionamento e uma pasta para armazenar as configurações de cada dispositivo. Esses arquivos têm responsabilidade isolada de forma a facilitar a manutenção e extensão do serviço. Eles foram criados usando o *Single Responsibility Principle* (SRP) (MARTIN, s.d.). Os arquivos são **ebserver.js**, **adbComands.js**, **configurations.js**, **session.js** e **testModule.js** e têm o objetivo de:

- **ebserver.js**: Nesse arquivo tem-se o funcionamento básico da ferramenta e o controle dos dois modos possíveis de execução. Nele, o servidor Node é levantado para o fluxo de **Execução Individual** ser possível e, caso os parâmetros adicionais sejam passados na inicialização, o fluxo de **Execução de UI** é invocado. Nesse arquivo está a informação de tempo limite durante uma execução e nele são feitas chamadas aos demais arquivos do projeto.
- **collectDataComands.js**: Arquivo que centraliza a integração do Ebserver com as chamadas de comandos a serem aplicados nos dispositivos. Os comandos nele configurados integram com as ferramentas de coleta dos dados durante a execução e auxiliam na automação do processo, como fechar e abrir a aplicação novamente para a próxima iteração.

- **session.js**: Centraliza o gerenciamento da sessão, com funções como criação, atualização e obtenção da sessão corrente. A sessão, por sua vez, armazena qual configuração cada dispositivo deve executar, e informações para controlar o fluxo do experimento por dispositivo. O arquivo de sessão controla como acontece a sequência de cada teste a ser executado.

- **configurations.js**: Responsável pela leitura dos arquivos Json de configuração, presentes na pasta **configurationsFolder**, e disponibilização ao objeto de sessão como cada teste deve ser realizado.

- **testModule.js**: É chamado quando o Ebserver é inicializado com parâmetros para executar testes de interação com a interface gráfica do dispositivo. Nele, tem-se a configuração de quantidade de repetições necessárias e qual o *framework* foi utilizado para construção do programa a ser testado. Essa informação é utilizada para organização dos resultados.

3.3.1 Arquivos de configuração

Os arquivos de configuração orientam o Ebserver como deve funcionar a automação para cada dispositivo e ficam armazenados na pasta **configurationsFolder**. Esses arquivos são do formato Json e contém as informações dos tipos de programas, dos experimentos que cada tipo deve executar, e dos parâmetros de cada experimento. Esse arquivo também guarda os valores de início e fim das repetições que cada experimento deve executar. Para reconhecimento de qual dispositivo utilizado, cada arquivo deve ser nomeado com o identificador do dispositivo, a mesma informação que é enviada no cabeçalho **device** das chamadas de comunicação com o Ebserver.

Figura 8 – Estrutura do arquivo de configuração

```
{
    "programs": ["foo", "bar"],
    "foo": {
        "experiments": ["foo_1", "foo_2", "foo_3"],
        "parameters": {
            "foo_1": ["param1", "param2", "param3"],
            "foo_2": ["param1"],
            "foo_3": ["param1", "param2"]
        }

    },
    "bar": {
        "experiments": ["bar_1"],
        "parameters": {
            "bar_1": ["param1", "param2"]
        }
    },
    "start_execution": 1,
    "end_execution": 45
}
```

A Figura 8 mostra um exemplo de como um arquivo de configuração deve ser preenchido. Nesse exemplo, dois programas foram configurados. O Ebserver espera o atributo **programs** com a listagem de todos os programas, que ajuda a diferenciar as várias formas de executar experimentos em um dispositivo. Deve existir, também, um atributo no arquivo, do tipo objeto, que descreve como cada programa deve ser executado. Para isso, o atributo **experiments** lista todos os experimentos que serão executados durante uma bateria de testes. O atributo **parameters** guarda uma lista de parâmetros para cada experimento, e dessa forma é possível executar um mesmo experimento com diferentes parâmetros.

O arquivo também contém a informação de início e de fim da execução nos atributos **start_execution** e **end_execution**. Durante a execução dos testes é possível que haja cenários em que problemas externos invalidem a execução atual e seja necessário recomeçar o teste. A existência dos parâmetros mencionados permite que a execução de um teste seja iniciada, por exemplo, no momento exato em que houve o problema, ao invés de reiniciar o teste do zero. No uso do Ebserver, é possível, também, ter um arquivo padrão de configuração que será aplicado a todos os dispositivos que não têm arquivos de configuração dedicado, basta nomeá-lo como **default.json**.

3.3.2 Arquivos gerados

Os arquivos gerados são os resultados obtidos pela execução dos programas de coleta configurados. O Ebserver organiza esses arquivos em uma hierarquia de pastas para facilitar acesso e leitura. A organização de pastas é feita da seguinte forma:

experiment-results > framework > “dispositivo” > “programa-parâmetro”

A raiz dos experimentos é a pasta "experiment-results", dentro dela uma pasta para cada *framework* executado é criada. Uma pasta para cada dispositivo é criada dentro da pasta do *framework* e, em seguida, cada programa com seu parâmetro utilizado na execução. Os arquivos de coleta são, por fim, armazenados para análises futuras. O padrão utilizado para cada programa de coleta é guardar o dado gerado com o nome do comando, seguido de hífen e o número da execução em que esse dado foi obtido.

4 COMPARAÇÃO DOS FRAMEWORKS

Para atender ao objetivo 1 da pesquisa, de entendimento do consumo de recursos dos *frameworks* multiplataforma sobre o nativo, foi feito um estudo comparativo por meio de duas aplicações e 10 *benchmarks*. Procurou-se ter certa diversidade nas aplicações para que fosse possível realizar a análise em diferentes situações. Dessa forma, desenvolvedores e empresas de *software* ganham um arcabouço de informações para auxílio de uma tomada de decisão mais consciente na escolha de *frameworks* para desenvolvimento de aplicativos móveis Android.

Devido à grande quantidade de *frameworks* disponíveis no mercado, para que seja possível fazer um estudo mais aprofundado sobre o tema, fez-se necessário escolher um subconjunto dos *frameworks* disponíveis. Para a escolha, teve-se como foco as ferramentas com maior popularidade entre os desenvolvedores. Como base, foi utilizada a pesquisa do Stackoverflow (STACK OVERFLOW, 2023) de popularidade de *frameworks* multiplataforma para desenvolvimento móvel, e a seguir são listados os *frameworks* selecionados para este trabalho com suas respectivas porcentagens de popularidade:

- Flutter - 9,21%
- React Native - 9,14%
- Ionic - 3,33%
- .Net Maui - 2,46%

Adicionalmente, o *framework* de desenvolvimento nativo do Android também foi utilizado neste trabalho. Vale mencionar, ainda, que o *framework* Xamarin, mesmo tendo 3,61% de popularidade, não foi utilizado no trabalho visto que o ele está sendo substituído pelo .Net Maui (MICROSOFT, s.d.a).

4.1 PROGRAMAS

Para comparação mais aprofundada de *frameworks*, é necessário que o desenvolvimento de programas iguais seja replicado em cada um deles e que esses programas sejam testados com as mesmas condições. Na indústria existem diversos aplicativos disponíveis, para uma vasta gama de necessidades e cenários, e muitos aplicativos são utilizados por milhões de usuários. Porém, normalmente esses aplicativos não possuem seu código fonte disponível publicamente, e muitas das aplicações são complexas demais para replicação em pesquisas. Com isso, uma boa prática é a criação de aplicações mais simples e cujo objetivo único é a execução de testes.

Durante a presente pesquisa, foram desenvolvidas três aplicações para testes, com distintos focos. O tempo disponível para tal foi um fator limitante na quantidade de aplicações, visto que cada aplicação precisa, ainda, ser replicada nos cinco *frameworks* em análise. Não obstante, a execução de testes e verificação de resultados também precisa ser realizada em cada versão de cada aplicação. Apesar disso, os aplicativos criados visam atender a maior quantidade possível de perfil de consumo:

- Uso intenso de CPU e Armazenamento;
- Carregamento de muitas imagens e animação sobre elas;
- Listagens longas, interação do usuário e banco de dados (SQLite);

A seguir, são explicadas e detalhadas cada uma das aplicações desenvolvidas.

4.1.1 Computer Language Benchmark Game

Computer Language Benchmark Game (CLBG) (GOUY, 2023) é um projeto com o objetivo de comparação de linguagens de programação. Nele, implementações similares são adotadas em programas simples, para checar como desempenham em relação ao uso de Memória, tempo de CPU, tempo execução, e tamanho do código gerado. Uma das características interessantes do projeto é que ele não somente usa algoritmos iguais para comparar linguagens diferentes, mas os algoritmos precisam ser implementados seguindo as mesmas regras. Baseado nesse projeto, foi criada uma aplicação que utiliza uma única tela em branco para executar os algoritmos de *benchmark* do CLBG. Seu objetivo é focar nos *benchmarks*, sem muito impacto da construção e gerenciamento de componentes visuais.

Utilizar esses *benchmarks* é de bastante utilidade pela maior garantia na similaridade das implementações. Dado que o melhor código de uma linguagem pode ser comparado com o de outra linguagem, essa comparação é interessante, pois diferentes *frameworks* multiplataforma utilizam diferentes conjuntos de tecnologia e linguagem. Em específico, esses algoritmos são interessantes para analisar o comportamento de diferentes *frameworks* em relação a cenários de alto consumo de memória, CPU, armazenamento (leitura e escrita dos dados gerados em arquivos) e energia. Vale mencionar que nem todos os *benchmarks* disponíveis exercitam os quatro recursos, mas essa característica também pode ser interessante para verificação de especificidades, como a abertura de arquivo de entrada com tamanho elevado, e alto consumo de memória necessária para execução do programa.

Os *benchmarks* disponíveis no projeto são:

- fannkuch-redux
- n-body

- spectral-norm
- mandelbrot
- pidigits
- regex-redux
- fasta
- k-nucleotide
- reverse-complement
- binary-trees

Alguns *benchmarks* necessitam de um parâmetro de entrada relativamente grande ou geram uma grande quantidade de dados. Para executar esses *benchmarks* no dispositivo móvel, eles foram alterados para que alguns dos parâmetros de entrada fossem lidos de arquivos de texto armazenados no dispositivo. Da mesma forma, a saída de informações dos *benchmarks* também foi redirecionada para ser armazenada em um arquivo de texto.

4.1.2 RotatingApp

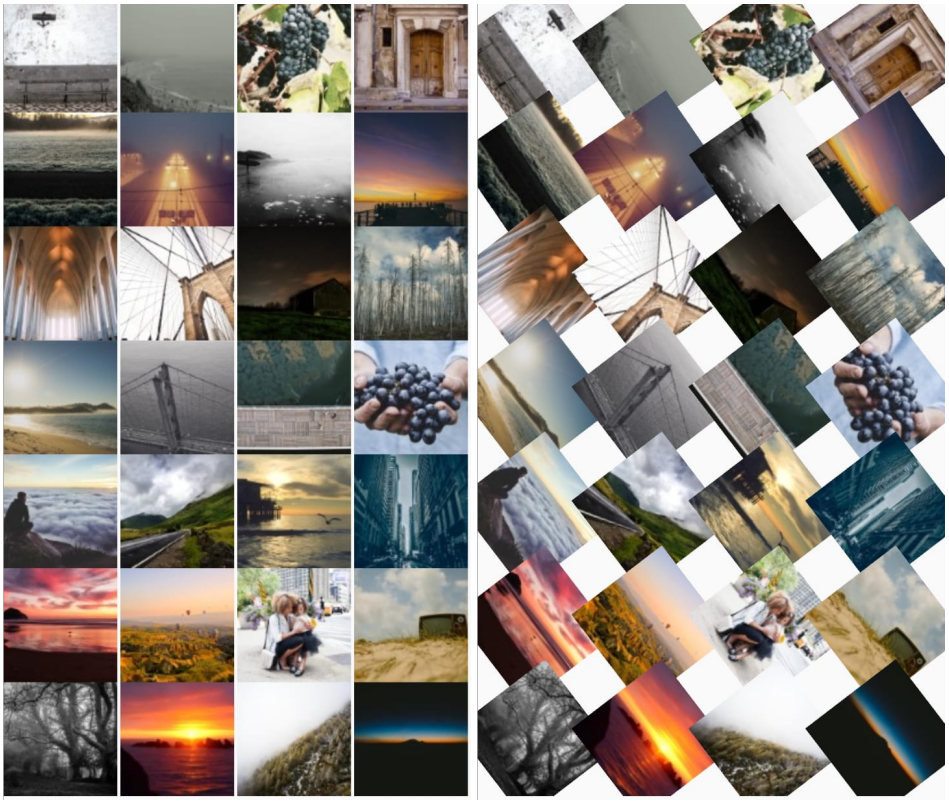
Diferentemente dos *benchmarks* do CLBG, o RotatingApp é uma aplicação que foi desenvolvida durante este trabalho com a função de testar componente visuais, animações gráficas e capacidade dos *frameworks* em lidar com grande quantidade de imagens na tela. Trata-se de um programa simples, que resume-se na exibição e rotação de n imagens na tela, dado o banco de 28 imagens pré-definidas e armazenadas na aplicação. Elas têm resolução de 100 x 100 pixeis e tamanho entre 1 e 7 megabytes, foram selecionadas de forma aleatória no Lorem Picsum[1]. Essa aplicação foi pensada para entender como cada framework lida com quantidade muito elevada de animações e alta taxa de atualização da interface gráfica.

As imagens são carregadas na aplicação de forma a preencher uma estrutura de listagem padrão do *framework* implementado, sempre na mesma ordem. A quantidade de imagens a serem carregadas e a quantidade de colunas da listagem consistem nos parâmetros de entrada do programa. A quantidade de colunas permite o controle para que todas as imagens fiquem dentro do limite visível de tela do dispositivo, de acordo com sua resolução e proporção. Em caso de o parâmetro de quantidade de imagens ser superior às 28 disponíveis, a sequência é repetida até o alcance do valor desejado.

A Figura 9 ilustra o funcionamento do programa, e o seguinte fluxo é seguido na sua execução:

[1] Lorem Picsum <https://picsum.photos>

Figura 9 – Execução do aplicativo RotatingApp com 28 imagens e 4 colunas. À esquerda pode-se ver a posição inicial das fotos carregadas; à direita são exibidas as imagens rotacionando



1. Carregamento das imagens na listagem;
2. Rotação de todas as imagens 9 vezes, durante um período de 18 segundos

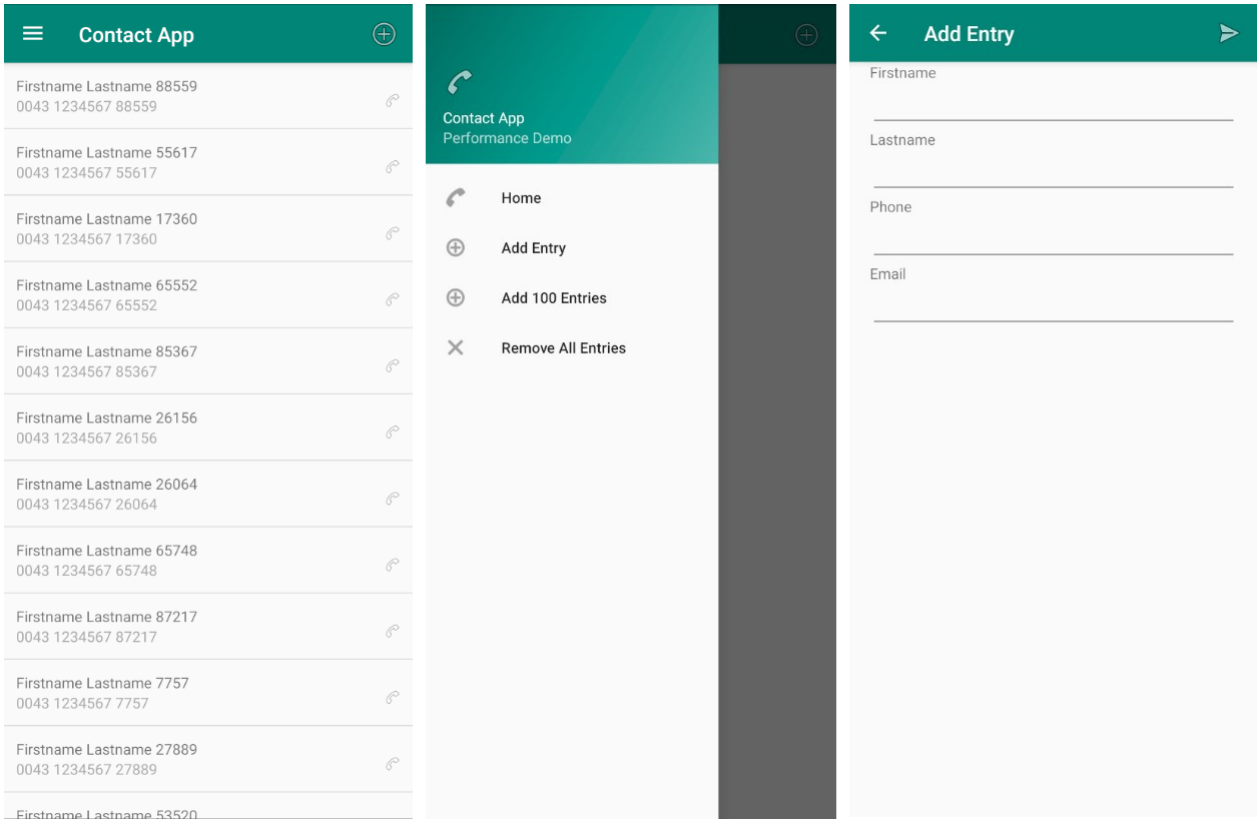
Para garantir que todas as aplicações se comportem de forma similar, a implementação padrão apresentada na documentação de cada *framework* foi utilizada, sem utilização de bibliotecas externas.

4.1.3 ContactApp

No trabalho (HUBER; DEMETZ; FELDERER, 2022) foi criada uma aplicação - ContactApp - para simular a interação do usuário no gerenciamento de contatos. Basicamente o usuário tem acesso a uma lista de contatos, que é armazenada no banco de dados do dispositivo, e essa lista pode ser alterada com adição e remoção de contatos. A aplicação conta com 2 telas simples e um *drawer* com algumas ações rápidas de gerenciamento dos contatos. Originalmente, essa aplicação foi desenvolvida para comparar Flutter, *framework* Nativo, React Native e Ionic. Neste trabalho, foi desenvolvida a versão em .Net Maui desta aplicação.

O funcionamento da aplicação começa na tela de listagem, na qual são carregados os contatos existentes na base de dados do dispositivo. A partir dessa tela, o usuário pode

Figura 10 – Execução do aplicativo ContactApp - da esquerda para a direita tem-se a listagem de contatos; *drawer* com ações; e tela de criação e edição de contatos



abrir o *drawer* de opções, através de um clique no menu sanduíche localizado na parte superior esquerda da tela, e ter acesso às seguintes funcionalidades:

- ***Home***: Fecha o *drawer*, retornando à tela principal
- ***Add Entry***: Fecha o *drawer* e navega para a tela de formulário para cadastro de novo contato
- ***Add 100 Entries***: Adiciona 100 novos usuários na base de dados, com informações aleatórias, e fecha o *drawer*
- ***Remove All Entries***: Remove todos os contatos da base de dados e fecha o *drawer*

A tela de formulário contém 4 campos, para preenchimento sobre as informações do novo contato, e 2 ícones na barra superior. A seta para esquerda volta para a tela de listagem, enquanto o ícone à direita completa o fluxo de adição do contato na base de dados e depois retorna para a tela de listagem. Essa aplicação é pensada para simular um aplicativo de listagem e cadastro de informações simples, para poder comparar como os diferentes *frameworks* se comportam com navegação entre telas, listagem e banco de dados.

Os testes executados consistem em:

1. ***OpenCloseDrawer***: Abertura e fechamento do *drawer*, 5 vezes
2. ***ScrollDownList***: 5 Deslizes para baixo na listagem
3. ***ScrollUpList***: 5 Deslizes para cima na listagem
4. ***SwitchScreens***: Navegação para tela de cadastro
5. ***EnterFormData***: Adição de contato via tela de formulário

4.2 CONFIGURAÇÃO

Para desenvolvimento das aplicações testadas, foram utilizadas as linguagens de programação JavaScript[2] para os *frameworks* Ionic e React Native; Dart[3] para Flutter, e C#[4] para .Net Maui. No caso do *framework* nativo, ambas as linguagens Java[5] e Kotlin[6] são possíveis de ser utilizadas e consideradas oficiais para tal. Neste trabalho, Java foi a linguagem escolhida para construção das aplicações nativas, dada a ainda relevância da linguagem no desenvolvimento Android (PETERS; SCOCCIA; MALAVOLTA, 2021b).

Mesmo que vários dos *frameworks* testados consigam compilar para diferentes plataformas - como Web, macOs, iOS, Windows -, todos os testes deste trabalho foram executados no sistema operacional Android, versão 13. Foi utilizado o aparelho Galaxy S20 FE 5G[7], da marca Samsung. Todas as aplicações utilizaram as bibliotecas padrão indiciadas na documentação de cada *framework*, nas seguintes versões dos *frameworks*:

- **Flutter**: v3.0.3
- **React Native**: v0.68.2
- **Ionic**: v6.2.7
- **.Net MAUI**: v7.0.202

Para os testes dos programas, cada um foi executado 45 vezes, e as 15 primeiras execuções de cada foram consideradas aquecimento e removidas das análises, para tentar garantir certa estabilidade do dispositivo. O Ebserver - apresentado no Capítulo 3 - foi utilizado para acelerar o processo de coleta dos recursos durante os testes. Os comandos pré-configurados na ferramenta simplificaram a configuração das execuções.

[2] JavaScript <https://developer.mozilla.org/en-US/docs/Web/JavaScript>
[3] Dart <https://dart.dev/>
[4] C# <https://dotnet.microsoft.com/pt-br/languages/csharp>
[5] Java <https://www.java.com/pt-BR/>
[6] Kotlin <https://kotlinlang.org/>
[7] Samsung - Galaxy S20 FE 5G <https://www.samsung.com/br/smartphones/galaxy-s/galaxy-s20-fe-5g-cloud-mint-128gb-sm-g781bzgjzto/>

4.2.1 CLBG

Todos os *benchmarks* disponíveis no CLBG foram utilizados. Desses, as versões de código síncronas foram priorizadas, pois as versões assíncronas necessitam da execução de diferentes *Threads*, e essa prática tem certas limitações no React Native. Novamente, como o objetivo é comparar todos os *frameworks*, a implementação dos programas precisa ser feita nas condições mais similares possível, e se torna necessário limitar certas configurações.

Os *benchmarks* disponíveis no CLBG utilizam parâmetros de entrada que precisam ser calibrados considerando as limitações do *hardware* que vai ser testado. Como os *frameworks* multiplataforma adicionam camadas de complexidade para sua execução, e os dispositivos de foco deste trabalho são dispositivos móveis, foi necessário executar algumas vezes todos os *benchmarks* para entender quais parâmetros seriam mais adequados para o tipo de dispositivo. Como exemplo, o aumento de alguns parâmetros dos *benchmarks* afeta notavelmente o tempo de execução do algoritmo. Uma execução muito rápida pode finalizar antes mesmo de os dados de consumo serem coletados, enquanto uma execução muito demorada pode gerar erros do próprio SO Android devido ao uso excessivo de algum recurso. A exemplo, memória foi um fator limitante para finalização correta do código em cenários de parâmetros máximos de algumas execuções.

Portanto, os testes foram configurados para ter tempo de execução mínimo de 0.5 segundos, e máximo de 120 segundos, para garantir uma coleta de dados completa e sem erros. Consequentemente, a escolha dos parâmetros do CLBG, e como eles afetam a execução, precisam respeitar o tempo de execução estabelecido. Considerando, também, o tempo disponível para construção deste trabalho, em todos os *benchmarks* do CLBG, os testes foram executados com 2 parâmetros diferentes - listados na Tabela 2 - , com o intuito de entender o comportamento da aplicação, se há diferença no consumo de recursos, e entender possíveis limitações.

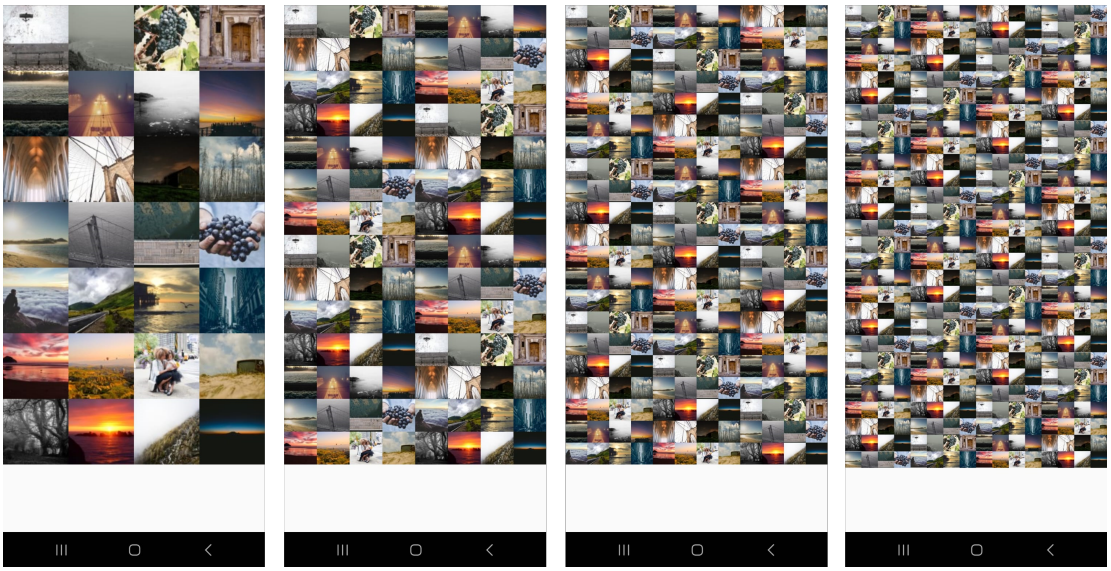
4.2.2 RotatingApp

Os parâmetros do RotatingApp definem a quantidade de imagens que aparecem na tela, além da quantidade de colunas. É necessário alinhar as imagens para o tamanho de tela do dispositivo testado, de forma que nenhuma fique fora da tela. Essa configuração é importante, pois otimizações são comumente aplicadas pelos *frameworks* em listagens para não colocar elementos não visíveis em memória, o que poderia invalidar os testes visto que nesse cenário nem todas as imagens estariam sendo rotacionadas a depender da otimização. Para o dispositivo utilizado neste estudo, a proporção de $7 : 4$ garantiu a visibilidade das imagens.

Para testar cenários de estresse no dispositivo, com diferentes quantidades de imagens, o valores utilizados durante os testes foram: 28, 112, 252 e 448. Esses números foram escolhidos pois respeitam a proporção escolhida, utilizando 4, 6, 8 e 12 colunas,

respectivamente. A Figura 11 exibe como fica o aplicativo com cada um dos parâmetros.

Figura 11 – RotatingApp com 28, 112, 252 e 448 imagens, respectivamente



4.2.3 ContactApp

Para a execução dos testes é feita uma inicialização manual de aplicação e a opção de adicionar 100 contatos no *drawer* é clicada para possibilitar os testes de *scroll* da listagem.

4.3 RESULTADOS

Esta seção apresenta os resultados obtidos na execução dos experimentos. Os programas e *benchmarks* foram divididos em seções para facilitar o entendimento de cada resultado por aplicação. As implicações dos resultados congregados são apresentadas logo em seguida.

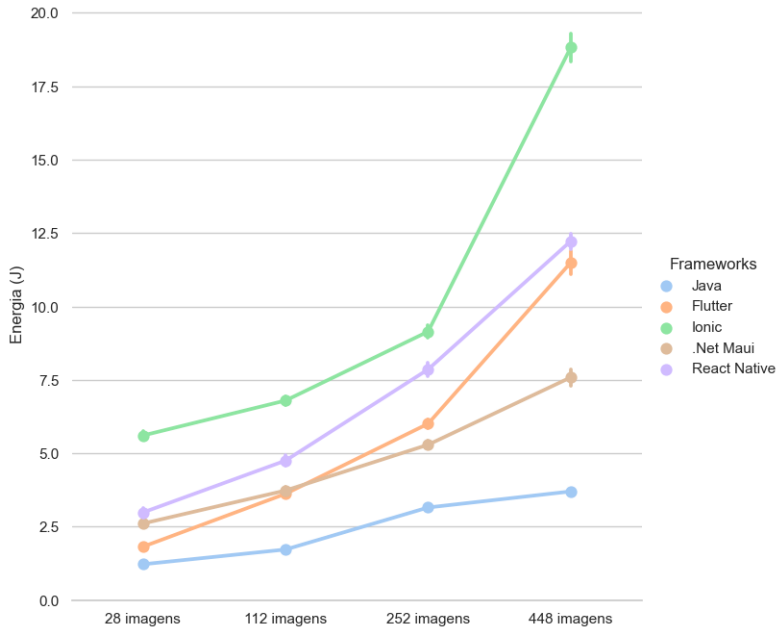
4.3.1 RotatingApp

O objetivo do RotatingApp é testar o comportamento de cada *framework* com animação e como eles escalam com o aumento na quantidade das imagens. As métricas utilizadas nesse teste são: uso de memória, consumo de energia durante execução, tempo de CPU, tamanho do APK gerado, e quantidade de *janky frames* - uma medida para entender a fluidez da animação. A métrica de *janky frames* calcula o percentual de *frames* que demoraram mais tempo para serem gerados do que o tempo de atualização da tela.

Os resultados mostraram um desempenho aquém, do Ionic, em comparação com os demais *frameworks*. Enquanto o Flutter apresentou escalabilidade ruim para o acréscimo de imagens, o Ionic apresentou um maior uso de CPU no caso de poucas imagens, maior consumo de energia em todos os testes e pior desempenho gráfico.

No contexto de consumo de energia, todos os *frameworks* multiplataforma desempenharam pior do que o *framework* nativo, com uma boa margem de diferença. Conforme

Figura 12 – Média e desvio padrão do Consumo de Energia durante 30 execuções do RotatingApp



mostrado na Figura 12, Flutter apresenta o menor consumo entre os *frameworks* multiplataforma. No cenário de animação de 28 imagens, ele demonstra um aumento de uso de energia, em relação ao nativo, de aproximadamente 48%; enquanto o Ionic apresenta o pior desempenho, com aumento de 358% contra o nativo.

Com o aumento da quantidade de imagens, o intervalo na diferença de consumo mantém certa similaridade entre os *frameworks*, sendo a execução do Ionic, no cenário de 448 imagens, a de consumo mais discrepante - aumento de 408% em relação ao nativo - e de maior valor absoluto no consumo energético. De todos os *frameworks*, o que tem custo energético mais afetado com a escalabilidade das imagens é o Flutter, com uma diferença de 530% entre a maior e a menor execução. React Native e .Net Maui apresentam resultados medianos, com aumento de 230% e 104%, respectivamente, em relação ao *framework* nativo no cenário de 448 imagens.

No consumo de memória, ilustrado na Figura 13, os *frameworks* se mantiveram relativamente constantes durante as execuções, com exceção do React Native. No cenário de 448 imagens, esse *framework* utilizou 148% mais memória do que solução nativa e 61% mais do que o segundo pior *framework* - .Net Maui.

Em relação ao uso de CPU, expresso na Figura 14, os diferentes *frameworks* apresentaram um padrão comum em todos os cenários, com exceção do Ionic, nos primeiros três, e Flutter, nos dois últimos. Enquanto o React Native apresenta pior desempenho com 28 imagens, com 45% mais uso do que o segundo pior *framework* e 118% mais consumo em relação ao nativo, essa situação muda no cenário de 448 imagens. Comparando a variação entre os cenários de 112 e 252 imagens, o Flutter apresenta um crescimento vertiginoso de 142% no uso de CPU, se tornando o *framework* de maior consumo nessa execução, e

Figura 13 – Média e desvio padrão do uso de memória durante 30 execuções do RotatingApp

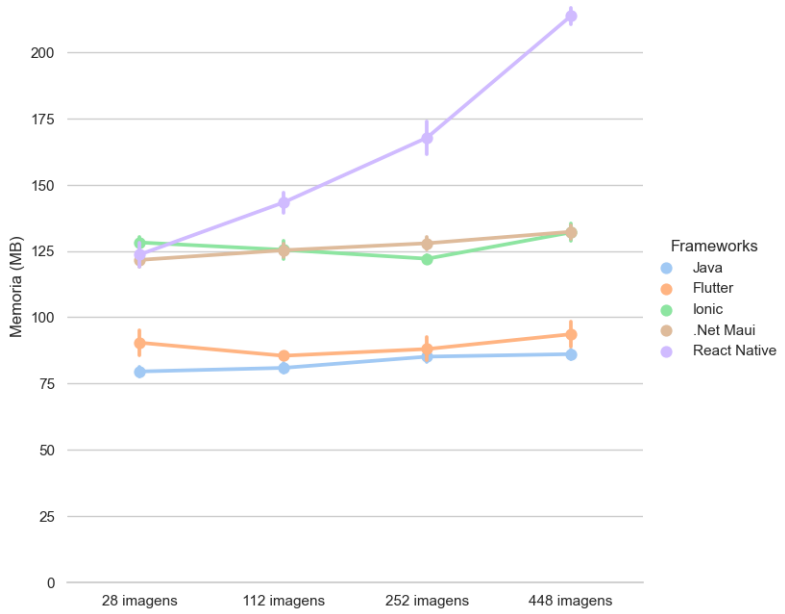
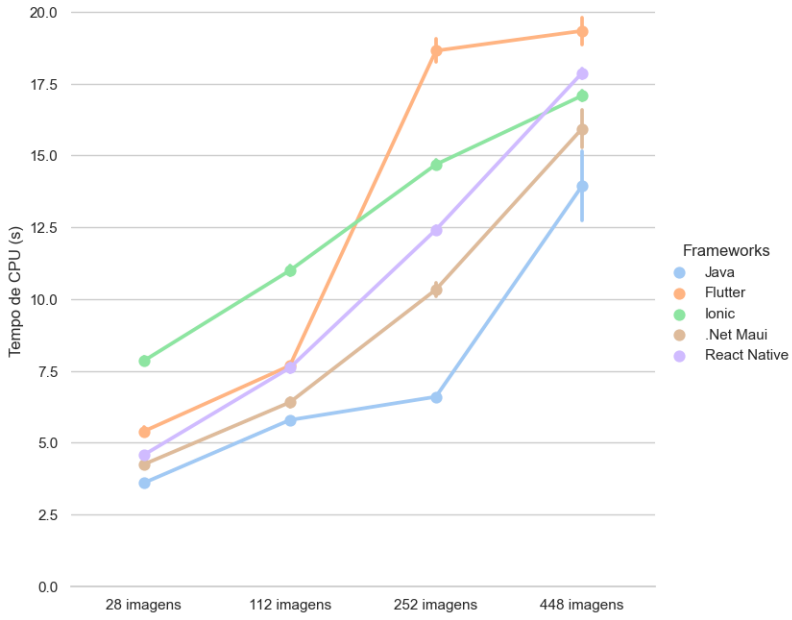


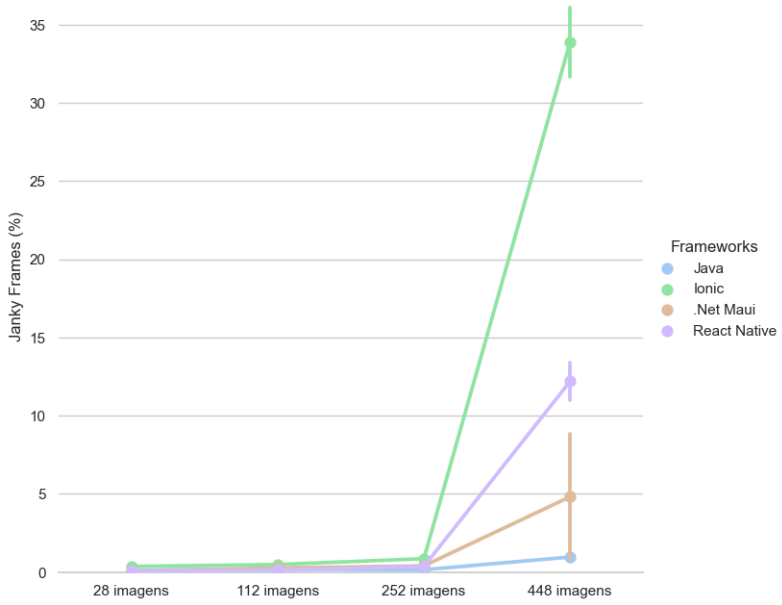
Figura 14 – Média e desvio padrão do uso de CPU durante 30 execuções do RotatingApp



ainda permanecendo nesse posto com o aumento de imagens para a quantidade máxima testada, 448.

Para comparar o desempenho de captura dos dados de *frames* gerados pela aplicação durante o teste, foi utilizado o comando `adb shell dumpsys gfxinfo`. Cada execução gerou, aproximadamente, 1065 *frames*. Outra informação obtida com esse comando é a quantidade de *janky frames*, que representa a contagem de quadros que demoraram a serem gerados. Na prática, essa quanto maior essa métrica, mais o usuário tem a sen-

Figura 15 – Média e desvio padrão de *janky frames* durante 30 execuções do RotatingApp



sação de que o aplicativo está travando, e menor é a sensação de fluidez na animação das imagens. A Figura 15 apresenta a média de *janky frames*, e apesar da quantidade crescente nas execuções, ela ainda é irrelevante, considerando 28, 112 e 252 imagens. O problema acontece quando a quantidade de imagens cresce para 448, momento no qual os *frameworks* começam a ter problema para lidar com a quantidade imagens sendo animadas simultaneamente. .Net Maui apresentou bastante variação de desempenho entre as execuções, mas teve sua média de *janky frames* menor em relação aos outros *frameworks* multiplataforma. O Ionic apresentou o pior desempenho entre todos, com resultado 172% maior do que o *React Native*, e 3380% do que o *framework* nativo.

Considerando a simplicidade da aplicação, os arquivos de instalação gerados por cada *framework* foram relativamente pequenos. O aplicativo nativo, por exemplo, tem somente 4,42MB. Como os *frameworks* multiplataforma adicionam camadas extras de abstração, é esperado um crescimento no tamanho deles, tal que o Ionic gerou um arquivo de 6,96MB, o Flutter de 16,6MB, o .Net Maui de 30,2MB e o React Native de 30,6MB.

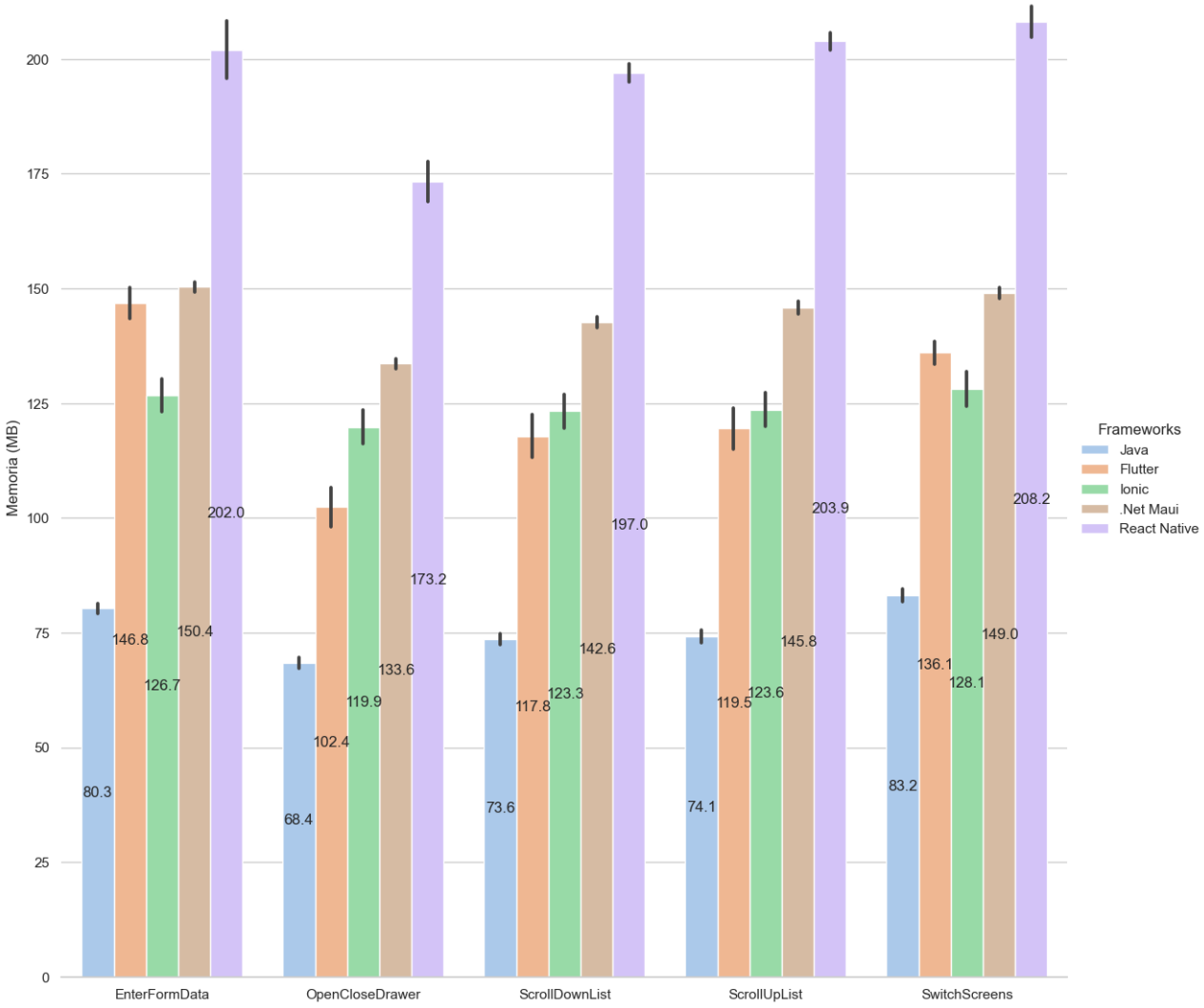
4.3.2 ContactApp

O ContactApp exercita funções visuais básicas do sistema, como clique, deslize, navegação entre telas e acesso a banco de dados. Com essas características em foco, as métricas utilizadas para comparação dos diferentes frameworks foram: uso de memória, consumo de energia durante execução, tempo de CPU e tamanho do APK gerado. Diferentemente do RotatingApp, que faz uso intenso de animações, a análise de *janky frames* não foi utilizada como métrica de medição para o ContactApp.

Os *frameworks* mantiveram um comportamento comparativo quase constante nos re-

sultados, pouca diferença foi notada de um teste para outro. React Native apresentou o pior resultado na maioria dos testes, enquanto o *framework* nativo foi constantemente melhor que os multiplataforma, com exceção do consumo energético em um dos testes.

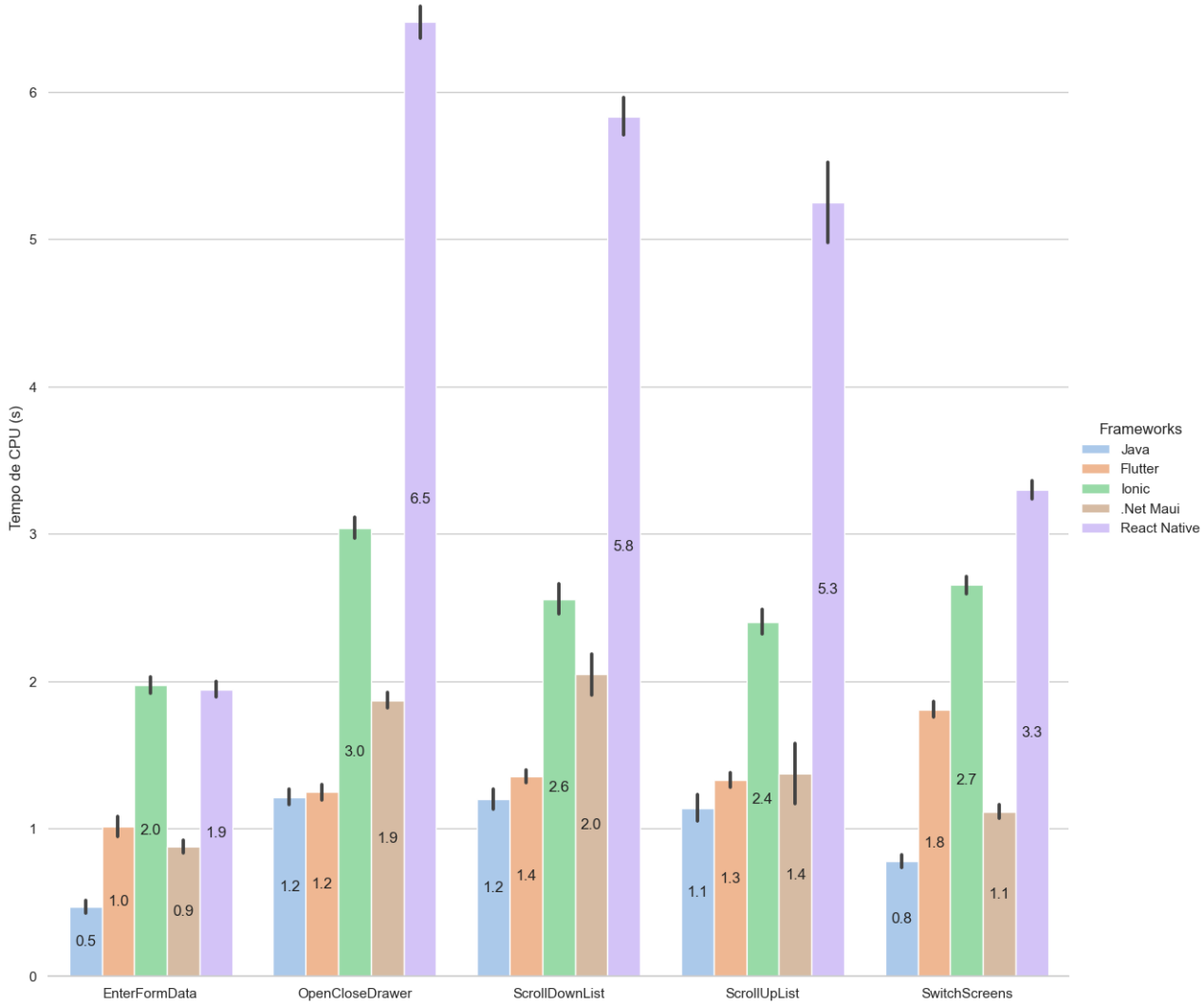
Figura 16 – Média e desvio padrão de uso de memória do ContactApp durante cada cenário por 30 execuções



No uso de memória, conforme visto na Figura 16, o React Native foi constantemente mais pesado que os outros *frameworks*, com uma média de uso em todos os cenários de 159% a mais do que a solução nativa. Os demais *frameworks* apresentam uso similar, apenas .Net Maui desempenhou pior que o Flutter e o Ionic, por pequena margem. Já entre Flutter e Ionic, o Flutter tem melhor desempenho em três cenários, e o Ionic em dois, com médias de consumo, em todos os cenários, aproximadamente iguais.

Quanto ao uso de CPU, visualizado na Figura 17, nota-se um maior consumo em todos os *frameworks* nos cenários de abertura do *drawer* (*OpenCloseDrawer*) e de deslizes na listagem na direção vertical (*ScrollDownList* e *ScrollUpList*), exceto pelo Ionic

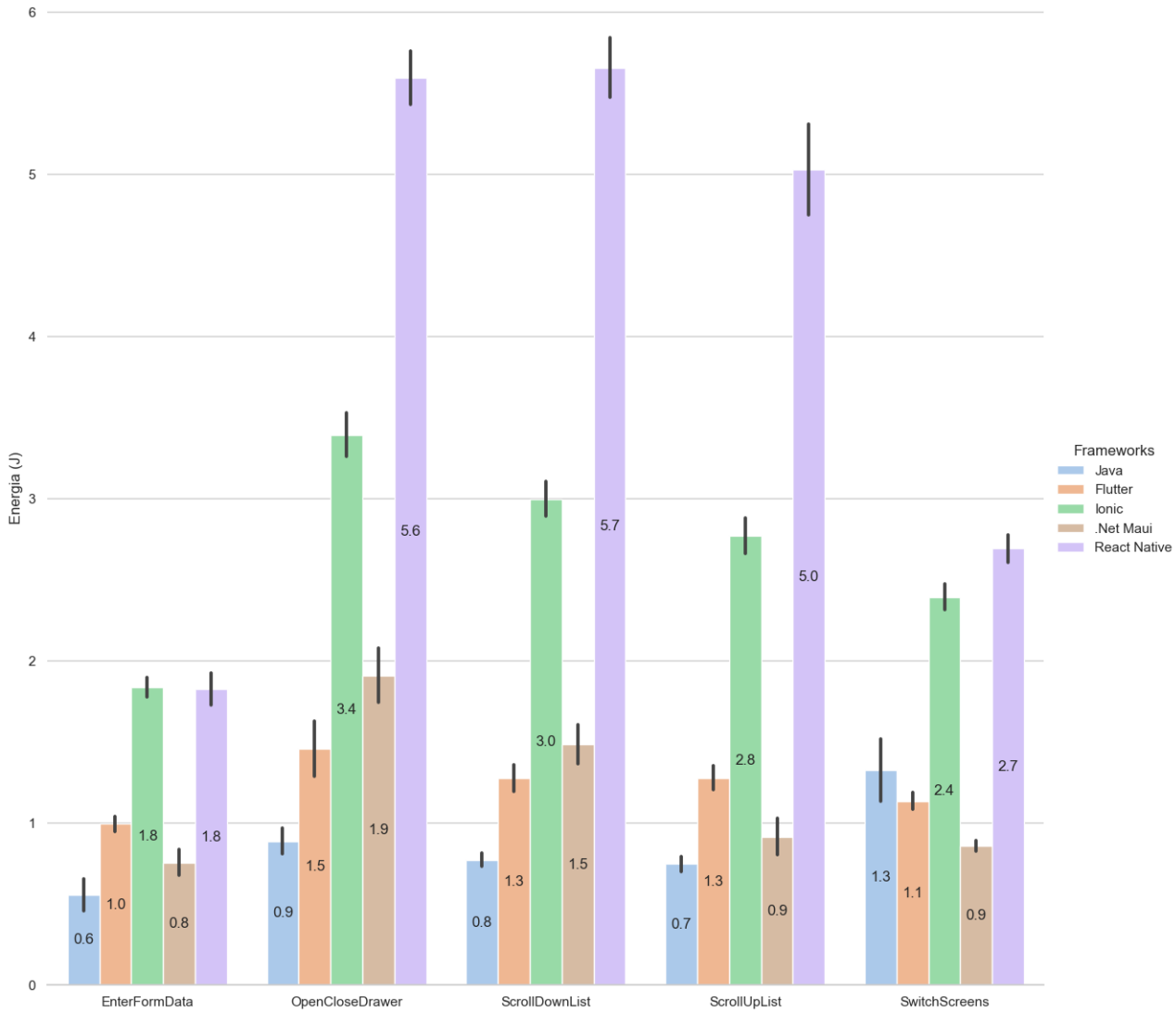
Figura 17 – Média e desvio padrão de utilização de CPU pelo ContactApp durante cada cenário por 30 execuções



e pelo Flutter. Respectivamente, o cenário de troca de tela (*SwitchScreens*) teve resultado equiparável aos citados anteriormente, e, no caso do Flutter, a troca de tela foi seu pior desempenho de CPU. De forma geral, nota-se o padrão do React Native utilizar mais CPU do que os demais, seguido de Ionic, .Net Maui e Flutter; enquanto o nativo desempenha melhor do que todos os *frameworks* multiplataforma. De todos os testes realizados, o preenchimento do formulário e inserção dos dados inseridos no banco de dados (*EnterFormData*) apresentou menos uso de CPU para todos os *frameworks*. Enquanto o React Native apresentou um consumo muito mais elevado do que os outros *frameworks* nos outros cenários, neste, o Ionic apresentou mais uso do recurso, mesmo que somente 1% a mais. Em resumo, o Flutter utilizou 40% mais CPU do que o nativo, seguido do .Net Maui, com 51% de aumento, Ionic com 163% e React Native com 474%.

No consumo de energia, os resultados dos *frameworks* se comportam de maneira similar

Figura 18 – Média e desvio padrão de consumo de energia do ContactApp durante cada cenário por 30 execuções



aos resultados de CPU, com um ponto fora do padrão anterior, em que o nativo consome mais energia do que .Net Maui e Flutter no cenário de troca de tela. Respectivamente, o nativo apresenta 54% e 7% de aumento em relação aos multiplataforma.

O tamanho do APK de cada framework foi de 1,16MB para o nativo, 2,98MB para Ionic, 17,5MB para Flutter, 28,6MB para React Native e 33,6MB para .Net Maui.

4.3.3 CLBG

A aplicação para o CLBG tem o objetivo de comparar os diferentes *frameworks* em cenário de uso intenso de CPU. Em contraste com os outros dois programas, o tempo de execução também é um atributo importante e é usado como métrica, em conjunto com o consumo energético, uso de memória e tamanho do APK.

Tabela 2 – Benchmarks e seus parâmetros de entrada

Benchmark	Menor parâmetro ↓	Maior parâmetro ↑
binarytree	21	22
fannkuch	11	12
knucleotide	2.500.000	5.000.000
nbody	22.500.000	405.000.000
pidigits	13.000	19.500
regex	2.500.000	10.000.000
revcomp	5.000.000	40.000.000
spectral	6.500	29.250
fasta	35.000.000	180.000.000
mandelbrot	4.500	27.000

Como explicado anteriormente, foi necessário escolher os parâmetros de entrada dos *benchmarks* de forma a realizar uma comparação válida entre os *frameworks*. Essa escolha pode ter impacto devido a fatores como a linguagem usada e à forma como o *framework* é estruturado. Existem programas que, naturalmente, utilizam uma quantidade elevada de memória, e estudos sugerem não haver impacto no consumo de energia (PINTO et al., 2016; PEREIRA et al., 2020), mas seu consumo elevado pode impossibilitar execução de uma aplicação. A escolha de mais de um parâmetro é vantajosa para exercitar os *benchmarks* em mais cenários, de maior e menor necessidade dos recursos.

Usando essas informações e as regras de tempo mínimo e máximo para cada execução, os parâmetros foram escolhidos de forma a ter um maior número de *benchmarks* possível que executem o menor parâmetro, e ter pelo menos dois que executem o parâmetro de maior valor. Por fim, a Tabela 2 exibe os parâmetros que foram utilizados durante as execuções. Alguns dos *benchmarks* utilizam arquivos de entrada e, para esses, os parâmetros representam o nome do arquivo existente na aplicação. Para simplificação, os símbolos ↑ e ↓ são utilizados para representar os parâmetros de maior e menor valores, respectivamente.

Alguns *benchmarks* chegaram a consumir mais de 2GB de memória do sistema. No cenário do *benchmark* **fasta**, o React Native consumiu 2,683GB, enquanto a solução nativa usou somente 24MB. Já em relação ao tempo de execução, o Ionic levou 55 segundos para finalizar o *benchmark* **revcomp**, enquanto o Flutter o executou em aproximadamente 1 segundo. Nesse mesmo cenário, o React Native não terminou a execução a tempo. Essas discrepâncias de comportamento impossibilitaram mudanças no parâmetro de entrada para que pudesse incluir o React Native na comparação, pois reduzir o parâmetro resultaria na exclusão do Flutter, devido à sua execução abaixo do tempo mínimo.

A Tabela 3 mostra quais *frameworks* executaram os *benchmarks* com cada parâmetro selecionado. Em verde, são indicados os cenários de sucesso na execução, e em vermelho

Tabela 3 – Relação dos benchmarks que executaram com os parâmetros selecionados

		Framework				
Benchmark	Parâmetro	Flutter	Nativo	RNative	Ionic	Maui
binarytree	21					
	22					
fannkuch	11					
	12					
knucleotide	2.500.000					
	5.000.000					
nbody	22.500.000					
	405.000.000					
pidigits	13.000					
	19.500					
regex	2.500.000					
	10.000.000					
revcomp	5.000.000					
	40.000.000					
spectral	6.500					
	29.250					
fasta	35.000.000					
	180.000.000					
mandelbrot	4.500					
	27.000					

os que não terminaram a execução dentro dos limites de tempo definidos. O Flutter conseguiu executar todos os algoritmos, seguido do nativo que não conseguiu executar um, Ionic falhou em dois, .Net Maui em sete e, por fim, o React Native executou menos da metade dos testes e não executou nenhum dos *benchmarks* com o maior parâmetro.

A Figura 19 exibe a média dos resultados obtidos durante as execuções dos *benchmarks*. Vários cenários apresentam diferenças significativas de desempenho entre os *frameworks*. Há casos em que o uso de recurso de uma plataforma é várias vezes maior que as demais, como no *revcomp* ↓, em que o Ionic teve 861% mais consumo de energia que o segundo pior resultado. Esse tipo de resultado está relacionado com o fato de alguns *frameworks* não conseguirem executar alguns *benchmarks* com os parâmetros escolhidos.

É notável também o caso do React Native, que desempenhou de forma inferior em relação aos demais *frameworks*, com exceção de alguns casos. No consumo de energia, desempenhou melhor que o *binarytree* ↓, em que consumiu 59% menos que o .Net Maui. Enquanto no tempo de execução no *fannkuch* ↓, o React executou 3% mais rápido do que a solução nativa. Já no uso de memória, ele foi mais eficiente que o Ionic em dois cenários,

com 20% menos uso no *fannkuch* ↓ e 26% a menos no *mandelbrot* ↓.

4.3.3.1 Tempo de execução

Diferentemente dos programas com interface gráfica, a velocidade de execução dos *benchmarks* com a solução nativa - escrita com linguagem Java - não se mostrou a melhor ferramenta em comparação com alguns dos *frameworks*. Soluções multiplataformas se mostraram mais rápidas em 16 dos 20 cenários, sendo o nativo mais rápido em *fasta* ↓ ↑, *nbody* ↓ ↑ . Dos 16 restantes, o Flutter e o .Net Maui foram mais rápidos em 8 cenários cada. Com o Flutter sendo melhor em *binarytree* ↓↑, *mandelbrot* ↓↑, *pidigits* ↓↑ e *revcomp* ↓↑, enquanto o .Net Maui foi melhor em *fannkuch* ↓↑, *knucleotide* ↓↑, *regex* ↓↑ e *spectral* ↓ ↑.

React Native teve o pior desempenho em todos os cenários que executou com sucesso, enquanto o Ionic apresentou um desempenho mediano, não sendo o melhor em nenhum cenário e sendo o pior em *binarytree* ↑, *knucleotide* ↑, *spectral* ↓, com um caso - *revcomp* ↓ - cuja velocidade foi muitas vezes mais lenta do que os demais *frameworks*, sendo 19 vezes mais lento do que o segundo mais lento, o .Net Maui.

Dos cenários em que os *frameworks* multiplataforma foram mais rápidos que o nativo, o Flutter executou 32% mais rápido no *binarytree* ↓, 45% no *binarytree* ↑, 8% no *mandelbrot* ↓, 31% no *mandelbrot* ↑, 82% no *pidigits* ↓ e *pidigits* ↑, 2% no *revcomp* ↓ e 12% no *revcomp* ↑. Já o .Net Maui, executou 79% mais rápido no *fannkuch* ↓, 83% no *fannkuch* ↑, 46% no *knucleotide* ↓, 63% no *knucleotide* ↑, 72% no *regex* ↓, 4% no *spectral* ↓ e 27% no *spectral* ↑.

Esses resultados mostram que *frameworks* multiplataformas podem apresentar desempenho superior e, portanto, ser uma ferramenta mais adequada quando esse recurso tiver uma maior prioridade.

4.3.3.2 Consumo de energia

O Java consumiu menos energia em 8 cenários - *binarytree* ↓, *fasta* ↓↑, *knucleotide* ↓↑, *nbody* ↓ e *spectral* ↓↑ -; Flutter em 10 - *binarytree* ↑, *fannkuch* ↓↑, *mandelbrot* ↓↑, *nbody* ↑, *pidigits* ↓↑ e *revcomp* ↓↑ -; e o Ionic em 2 - *regex* ↓↑. Diferentemente das outras métricas, o .Net Maui não apresentou melhores resultados em nenhum cenário de energia, tendo valores elevados, comparativamente com os demais, no *binarytree* ↓ com 142% mais consumo do que o segundo pior resultado, e *spectral* ↑ com 64% mais consumo.

O React Native, nas demais métricas, ficou aquém dos outros *frameworks* em quase todos os resultados, sendo melhor somente do que o .Net Maui em seu pior caso. Em geral, o nativo desempenhou bem, considerando os resultados dos demais *frameworks*, com resultados bons em muitos e medianos nos outros, com exceção do *fannkuch* ↑, no qual apresentou um resultado 89% pior do que o segundo pior resultado do cenário, o .Net Maui.

O Flutter se destacou positivamente no *fannkuch* ↓↑ e *pidigits* ↓↑, consumindo uma quantidade muito menor de energia que os demais, com 77% menos consumo do que o Ionic no *pidigits* ↓, 83% menos do que nativo no *pidigits* ↑, 42% do que o .Net Maui no *fannkuch* ↓ e 46% do que o Ionic no *fannkuch* ↑. Resultados mostram, ainda mais, um bom desempenho dos *frameworks* multiplataforma para fazer cálculos complexos. Eles são, não somente, rápidos como energeticamente econômicos, e alguns cenários, como os que o Flutter se destacou, o ganho de desempenho é muito significativo.

4.3.3.3 Uso de Memória

Em vários *benchmarks*, é possível observar que o aumento dos parâmetros de entrada não resulta em um maior consumo de memória. Alguns casos como *fannkuch*, *pidigits*, *spectral*, *mandelbrot* e *nbody* apresentaram, até mesmo, redução no consumo. Enquanto nesses *benchmarks* todos os *frameworks* tiveram o mesmo comportamento, alguns outros destoaram do padrão, como *binarytree* no Ionic e *fasta* no Nativo. O motivo pode estar relacionado à forma de implementação utilizada pelo CLBG e gerenciamento de memória adotada por cada *framework*. Nota-se, também, uma não correlação entre o uso de memória e o consumo de energia.

A implementação nativa teve um menor impacto no uso de memória dentre os *frameworks*, sendo a que menos utiliza o recurso em 14 cenários e a que mais utiliza no *binarytree* ↑. Também não aconteceu do uso de memória aumentar muito radicalmente entre os parâmetros ↓ e ↑ dos *benchmarks*. o Ionic variou entre um menor uso de memória em 5 cenários e maior em 8, no *nbody* ↓ somente foi mais eficiente do que o React Native. O React Native, por sua vez, apresentou um consumo elevado de memória, sendo o que mais utiliza, ou o segundo que mais utiliza em todas as execuções que terminaram corretamente. Já o Flutter e o .Net Maui apresentaram resultados medianos, nos quais somente o Flutter foi mais eficaz que os demais *frameworks*, no *spectral* ↑. Nos 14 cenários em que os dois *frameworks* executaram corretamente os *benchmarks*, o Flutter obteve melhores resultados em 12.

O tamanho do APK de cada framework foi de 240MB para o nativo, 243MB para Ionic, 253MB para Flutter, 263MB para React Native e 267MB para .Net Maui. O tamanho elevado das aplicações se deve aos arquivos que são usados como parâmetro de entrada de alguns *benchmarks*.

4.3.4 Considerações

Foram testadas duas aplicações com interface gráfica e 10 *benchmarks* com uso intenso de CPU. Os resultados mostram um custo atrelado aos *frameworks* multiplataforma em aplicações com interface gráfica como foco. Comparando os *frameworks*, os que utilizam componentes nativos mostram bom desempenho com animações e alta taxa de atualização da tela, já no fluxo do ContactApp, o .Net Maui e o Flutter se destacaram entre os

Tabela 4 – Média dos resultados dos programas de interface gráfica em comparação ao *framework* nativo

Cenário	Framework	Energia (%)	CPU (%)	Memória (%)
Animação (rotação de imagens)	Flutter	134	71	8
	Ionic	311	69	53
	.Net Maui	96	23	53
	React Native	184	42	96
ScrollUpList	Flutter	71	16	61
	Ionic	272	111	67
	.Net Maui	23	20	97
	React Native	575	360	175
EnterFormData	Flutter	79	117	83
	Ionic	231	322	58
	.Net Maui	36	88	87
	React Native	229	316	151
ScrollDownList	Flutter	66	13	60
	Ionic	289	113	68
	.Net Maui	93	70	94
	React Native	634	385	168
OpenCloseDrawer	Flutter	64	3	50
	Ionic	283	151	75
	.Net Maui	115	54	95
	React Native	531	433	153
SwitchScreens	Flutter	-14	132	64
	Ionic	81	241	54
	.Net Maui	-35	43	79
	React Native	103	324	150

multiplataforma, com menor custo em relação à implementação nativa, enquanto o React Native apresentou alto uso adicional de recursos.

De forma resumida, a Tabela 4 mostra uma compilação da média dos resultados dos aplicativos testados nesse trabalho. Os dados representam a porcentagem de uso de cada recurso para cada cenário testado em comparação com os resultados obtidos nas aplicações nativas.

De forma geral, a solução nativa apresenta mais desempenho para esses casos, enquanto Flutter e .Net Maui mostram resultados mais interessantes entre os *frameworks* multiplataformas. O Ionic apresentou desempenho aquém no RotatingApp, com percentagem alta de *janky frames* e uso elevado de recursos. No ContactApp, por sua vez, desempenhou melhor que o React Native, porém inferior aos demais. Já o React Native, apresentou o pior desempenho no geral, com dados interessantes somente no RotatingApp, mas apresentou

alto consumo de recursos em todos os cenários do ContactApp.

Já nos *benchmarks*, a realidade muda. O desenvolvimento nativo não se impõe como a melhor solução indisputada. Regularmente, as soluções multiplataforma se mostram mais eficazes, com destaque ao Flutter e .Net Maui. O Ionic e React Native tendem a consumir mais recursos, em especial o React Native, que teve, em média, o pior desempenho de todos.

4.3.4.1 Implicações

A comparação dos recursos mostrou que *frameworks* baseados em web, como React Native e Ionic, desempenharam pior que os demais *frameworks* multiplataforma e o nativo. O React Native apresentou desempenho muito aquém, tornando ele uma péssima escolha para execução de programas com uso intenso de CPU. Ele demonstrou alto uso dos recursos para as aplicações com interface gráfica, com alto consumo de energia, CPU e memória, porém, no *smartphone* testado, apresentou bom desempenho gráfico. De forma geral, os resultados mostram que o React Native pode ser uma ferramenta interessante no desenvolvimento de aplicações simples, pois mesmo com a integração com a camada nativa, seu desempenho não é interessante para aplicações complexas que necessitem de uma quantidade maior de recursos.

Já o Ionic, apresentou resultados intermediários, em média melhores do que o React Native. Nos programas testados, apresentou desempenho superior ao React Native, porém inferior a todos os outros *frameworks*. Com o aumento na quantidade de imagens do RotatingApp, apresentou uma quantidade elevada de *Janky frames*, que implica numa perda de desempenho com a grande quantidade de animações aplicadas. Esse cenário, por outro lado, representa uma quantidade muito diminuta de aplicações - que precisam de tantas imagens em tela, e ainda de forma animada. Já nos *benchmarks*, conseguiu executar a grande maioria dos cenários (18 de 20), somente um a menos que o nativo, e obteve desempenho mediano, sendo poucas vezes o de pior resultado. Dessa forma, o Ionic parece ser uma solução interessante para quem busca desenvolver aplicativos com tecnologias web, com cuidado somente na quantidade de animações necessárias.

Enquanto o *framework* nativo apresentou desempenho médio muito superior aos demais, o Flutter e .Net Maui chegaram a consumir menos energia do que a solução nativa na troca de telas do ContactApp. Os outros resultados das aplicações com interface gráfica mostraram um melhor desempenho nesses dois *frameworks* do que no Ionic e no React Native, porém o consumo de recursos foi bastante elevado em comparação com o nativo. Flutter e .Net Maui mostram bons resultados no cenário de alto consumo de CPU, com desempenho frequentemente superior ao do *framework* nativo, o que torna os dois *frameworks* as escolhas mais seguras - dentre os multiplataforma desta pesquisa - para desenvolvimento de aplicações com bom desempenho, havendo cenários em que o desempenho ultrapassa o do nativo.

De forma geral, esses resultados podem auxiliar desenvolvedores na tomada de decisão sobre utilização de *frameworks* de desenvolvimento móvel Android. Para as empresas mantenedoras dos *frameworks*, também há ganho em dar atenção a esse tipo de trabalho, de forma a identificar possíveis melhorias e otimizações nos seus produtos.

Figura 19 – Média e desvio padrão de utilização dos recursos durante a execução dos *benchmarks* em 30 repetições

5 CONCLUSÕES

Este trabalho apresenta a comparação dos maiores *frameworks* do mercado, com o objetivo de entender o custo que é adicionado na utilização deles para o desenvolvimento de aplicações Android. O processo de comparação gerou uma quantidade muito elevada de arquivos, dada a quantidade de variáveis como quantidade de *frameworks* analisados, quantidade de programas e *benchmarks*, e a quantidade de vezes que foi necessário executar cada cenário de teste. Não somente, ainda há a necessidade de organização das ferramentas de coletas e dos arquivos resultados de cada coleta. Este trabalho utilizou 2 aplicativos diferentes e 10 *benchmarks* em 5 *frameworks*. O RotatingApp contou com 4 cenários de teste, o ContactApp com 5 cenários, e cada *benchmark*, com 2, gerando um total de 245 cenários. Desses, 45 execuções foram realizadas e até 4 arquivos foram gerados com as informações de coleta de cada execução, resultando em quase 20.000 arquivos para serem analisados *a posteriori*.

Todo esse contexto exigiu a automatização do processo para garantir velocidade e reduzir a propensão de erros humanos. Portanto, foi criado o Ebserver, uma ferramenta de automação para auxiliar no processo de análise de diferentes formas de implementação de aplicativos Android. O Ebserver já oferece integração, sem a necessidade de alterações, com ferramentas de coletas de métricas de informações de energia, memória, CPU e performance gráfica. Foi construído para ser uma ferramenta de simples utilização e extensível, para que outros cenários de testes sejam adicionados sem complexidade.

Com dois possíveis modos de execução, permite a utilização de ferramentas já conhecidas no mercado para realizar testes de interface - no modo de UI -, assim como a realização de testes mais granulares, sendo capaz de coletar dados a nível de métodos, dada a instrumentação necessária - no modo individual. A ferramenta já foi utilizada por outros trabalhos e se mostrou bastante eficaz em agilizar os testes realizados. As ferramentas de coletas já integradas no Ebserver são disponibilizadas pela mantenedora da plataforma Android, e essas métricas vêm sendo usadas veementemente pela academia e pela indústria.

Com o Ebserver, foi mais simples coletar dados das duas aplicações e 10 *benchmarks* para analisar o desempenho de *frameworks* multiplataforma em diferentes cenários. Se os mesmos testes tivessem sido feitos manualmente, facilmente o gerenciamento das execuções sairia do controle e poderia levar a algum erro e/ou perda de informações. Um *script* escrito na linguagem Python foi usado para leitura dos dados de coleta e foi disponibilizado junto ao projeto do Ebserver.

Os resultados da pesquisa mostraram que há um custo adicional na escolha de uma ferramenta multiplataforma em relação ao desenvolvimento nativo. Cada *framework* tem sua força e fraqueza que vai além da performance por si só. Apesar disso, o uso de recursos

é um fator importante na experiência do usuário final, seja na velocidade para terminar a execução de um algoritmo mais custoso, na fluidez de animações ou no simples fato da aplicação usar pouco armazenamento do dispositivo.

Frameworks baseados em web apresentaram performance média inferior aos demais testados. Enquanto o desenvolvimento nativo se mostrou mais performático nos programas com interface gráfica, nos *benchmarks* que impõem um intenso consumo de CPU, houve uma disputa maior entre os *frameworks* e o nativo, com vários cenários em que o nativo apresentou performance inferior.

Este estudo foi desenvolvido como evolução de dois outros trabalhos publicados em conferências internacionais - (OLIVEIRA et al., 2023b) e (OLIVEIRA et al., 2023d) -, os quais são de coautoria do autor desta dissertação.

5.1 TRABALHOS FUTUROS

Dada a grande quantidade de variáveis possíveis no desenvolvimento de uma aplicação móvel, é de grande dificuldade fazer uma comparação entre os principais *frameworks* que atenda a todas as variações de produtos existentes. Este trabalho adiciona, a esse escopo de pesquisa, uma ferramenta de automação, e aplicações desenvolvidas em 4 *frameworks* multiplataformas, além do Android nativo com Java. Bagagem construída para realização de um processo de comparação. Novos *frameworks* para desenvolvimento móvel surgem e ganham atenção dos desenvolvedores, sempre havendo espaço para novas comparações. O *Compose Multiplatform* (JETBRAINS S.R.O., s.d.) é um exemplo, e não consta no trabalho devido à data de lançamento, que foi posterior ao desenvolvimento da pesquisa.

As aplicações em foco neste trabalho exercitam casos específicos que não atendem a todos os cenário de uso, de forma que a adição de novas aplicações, ou incrementação das existentes, é interessante para aumentar o leque de análise. Cenários como uso de sensores, aplicação de mapa, processamento de texto, áudio e vídeo, desenvolvimento de jogos, entre outras possibilidades, são complementos bastante interessantes considerando a indústria de *software*.

Um outro fator atrativo para o enriquecimento da pesquisa é realização de testes em mais dispositivos, com processadores, quantidade de memória, tamanho e velocidade de armazenamento diferentes. Essas características podem resultar em dados diferentes dos que encontrados nessa pesquisa, mas a heterogeneidade agrega conteúdo aos resultados. Nesta pesquisa, somente um dispositivo foi usado, o que já oferece uma noção de como cada *framework* se comporta com os programas e *benchmarks* propostos, mas a adição de mais dispositivos traz maior confiabilidade nos dados e na comparação.

REFERÊNCIAS

AIRBNB, INC. *Building a Cross-Platform Mobile Team*. s.d. <https://medium.com/airbnb-engineering/building-a-cross-platform-mobile-team-3e1837b40a88>. Acesso em: 2023-11-27.

AIRBNB, INC. *React Native at Airbnb*. s.d. <https://medium.com/airbnb-engineering/react-native-at-airbnb-f95aa460be1c>. Acesso em: 2023-11-27.

AIRBNB, INC. *React Native at Airbnb: The Technology*. s.d. <https://medium.com/airbnb-engineering/react-native-at-airbnb-the-technology-dafd0b43838>. Acesso em: 2023-11-27.

AIRBNB, INC. *Sunsetting React Native*. s.d. <https://medium.com/airbnb-engineering/sunsetting-react-native-1868ba28e30a>. Acesso em: 2023-11-27.

AIRBNB, INC. *What's Next for Mobile at Airbnb*. s.d. <https://medium.com/airbnb-engineering/whats-next-for-mobile-at-airbnb-5e71618576ab>. Acesso em: 2023-11-27.

APPLE INC. *JavaScriptCore Documentation*. s.d. <https://docs.webkit.org/Deep%20Dive/JSC/JavaScriptCore.html>. Acesso em: 2023-11-27.

BIØRN-HANSEN, A.; RIEGER, C.; GRØNLI, T.-M.; MAJCHRZAK, T. A.; GHINEA, G. An empirical investigation of performance overhead in cross-platform mobile development frameworks. *Empirical Software Engineering*, Springer, v. 25, p. 2997–3040, 2020.

CIMAN, M.; GAGGI, O. An empirical analysis of energy consumption of cross-platform frameworks for mobile development. *Pervasive and Mobile Computing*, Elsevier, v. 39, p. 214–230, 2017.

CIMAN, M.; GAGGI, O. An empirical analysis of energy consumption of cross-platform frameworks for mobile development. *Pervasive and Mobile Computing*, Elsevier, v. 39, p. 214–230, 2017.

CIMAN, M.; GAGGI, O. et al. Evaluating impact of cross-platform frameworks in energy consumption of mobile applications. In: *WEBIST (1)*. [S.l.: s.n.], 2014. p. 423–431.

COUNTERPOINT TECHNOLOGY MARKET RESEARCH. *Global Smartphone Shipments Market Data (Q1 2022 – Q4 2023)*. 2024. Disponível em: <https://www.counterpointresearch.com/insights/global-smartphone-share/>. Acesso em: 10 mar 2024.

DISCORD INC. *An Exciting Update to Discord for Android*. s.d. <https://discord.com/blog/android-react-native-framework-update>. Acesso em: 2023-11-27.

DISCORD INC. *Why Discord is Sticking with React Native*. s.d. <https://discord.com/blog/why-discord-is-sticking-with-react-native>. Acesso em: 2023-11-27.

EL-KASSAS, W. S.; ABDULLAH, B. A.; YOUSEF, A. H.; WAHBA, A. M. Taxonomy of cross-platform mobile applications development approaches. *Ain Shams Engineering Journal*, Elsevier, v. 8, n. 2, p. 163–190, 2017.

FACEBOOK, INC. *Hermes Engine Documentation*. s.d. <https://hermesengine.dev/>. Acesso em: 2023-11-27.

FERREIRA, J.; SANTOS, B.; OLIVEIRA, W.; ANTUNES, N.; CABRAL, B.; FERNANDES, J. P. On security and energy efficiency in android smartphones. In: IEEE. *2023 IEEE/ACM 10th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*. [S.l.], 2023. p. 87–95.

GOOGLE LLC. *Android Debug Bridge Documentation*. s.d. <https://developer.android.com/tools/adb>. Acesso em: 2023-11-27.

GOOGLE LLC. *Android documentation*. s.d. <https://developer.android.com/docs>. Acesso em: 2023-11-27.

GOOGLE LLC. *Android NDK documentation*. s.d. <https://developer.android.com/ndk/guides>. Acesso em: 2023-11-27.

GOOGLE LLC. *Android studio launch*. s.d. <https://android-developers.googleblog.com/2013/05/android-studio-ide-built-for-android.html>. Acesso em: 2024-02-11.

GOOGLE LLC. *Espresso*. s.d. <https://developer.android.com/training/testing/espresso>. Acesso em: 2023-11-27.

GOOGLE LLC. *LogCat*. s.d. <https://developer.android.com/studio/command-line/logcat>. Acesso em: 2023-11-27.

GOOGLE LLC. *Macrobenchmark*. s.d. <https://developer.android.com/topic/performance/benchmarking/microbenchmark-overview>. Acesso em: 2023-11-27.

GOOGLE LLC. *Microbenchmark*. s.d. <https://developer.android.com/topic/performance/benchmarking/microbenchmark-overview>. Acesso em: 2023-11-27.

GOOGLE LLC. *monkeyrunner*. s.d. <https://developer.android.com/studio/test/monkeyrunner/index.html>. Acesso em: 2023-11-27.

GOOGLE LLC. *Profile your app performance*. s.d. <https://developer.android.com/studio/profile>. Acesso em: 2023-11-27.

GOOGLE LLC. *UI Automator*. s.d. <https://developer.android.com/training/testing/other-components/ui-automator>. Acesso em: 2023-11-27.

GOOGLE LLC. *UI/Application Exerciser Monkey*. s.d. <https://developer.android.com/studio/test/other-testing-tools/monkey>. Acesso em: 2023-11-27.

GOUY, I. *The Computer Language Benchmarks Game. Web*. 2023. Accessed: 2023-11-27. Disponível em: <https://benchmarksgame-team.pages.debian.net/benchmarksgame/>.

GRADLE INC. *Gradle Build Tool*. s.d. <https://gradle.org/>. Acesso em: 2024-02-11.

HASAN, S.; KING, Z.; HAFIZ, M.; SAYAGH, M.; ADAMS, B.; HINDLE, A. Energy profiles of java collections classes. In: *Proceedings of the 38th International Conference on Software Engineering*. [S.l.: s.n.], 2016. p. 225–236.

HINDLE, A.; WILSON, A.; RASMUSSEN, K.; BARLOW, E. J.; CAMPBELL, J. C.; ROMANSKY, S. Greenminer: A hardware based mining software repositories software energy consumption framework. In: *Proceedings of the 11th working conference on mining software repositories*. [S.l.: s.n.], 2014. p. 12–21.

HODGES, A. *Alan Turing: the enigma*. New York: Simon and Schuster, 1983. ISBN 978-0-671-49207-6 978-0-671-52809-6.

HUBER, S.; DEMETZ, L.; FELDERER, M. Analysing the performance of mobile cross-platform development approaches using ui interaction scenarios. In: SPRINGER. *Software Technologies: 14th International Conference, ICSOFT 2019, Prague, Czech Republic, July 26–28, 2019, Revised Selected Papers 14*. [S.l.], 2020. p. 40–57.

HUBER, S.; DEMETZ, L.; FELDERER, M. A comparative study on the energy consumption of progressive web apps. *Information Systems*, Elsevier, v. 108, p. 102017, 2022.

HUBER, S.; DÖLLER, M.; FELDERER, M. On the energy-efficiency of hybrid ui components for mobile cross-platform development. In: SPRINGER. *International Conference on Web Engineering*. [S.l.], 2023. p. 247–261.

IONIC. *Capacitor Documentation*. s.d. <https://capacitorjs.com/>. Acesso em: 2023-11-27.

JETBRAINS S.R.O. *Compose Multiplatform*. s.d. <https://www.jetbrains.com/lp/compose-multiplatform/>. Acesso em: 2023-11-27.

KOTLIN FOUNDATION. *Kotlin Docs - Null safety*. [S.l.], s.d. Acesso em: 2023-11-27.

LI, Y.; YANG, Z.; GUO, Y.; CHEN, X. Droidbot: a lightweight ui-guided test input generator for android. In: IEEE. *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*. [S.l.], 2017. p. 23–26.

LINARES-VÁSQUEZ, M.; BAVOTA, G.; BERNAL-CÁRDENAS, C.; OLIVETO, R.; PENTA, M. D.; POSHYVANYK, D. Mining energy-greedy api usage patterns in android apps: an empirical study. In: *Proceedings of the 11th working conference on mining software repositories*. [S.l.: s.n.], 2014. p. 2–11.

MALAVOLTA, I.; GRUA, E. M.; LAM, C.-Y.; VRIES, R. D.; TAN, F.; ZIELINSKI, E.; PETERS, M.; KAANDORP, L. A framework for the automatic execution of measurement-based experiments on android devices. In: *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. [S.l.: s.n.], 2020. p. 61–66.

MARTIN, R. C. *The Principles of OOD*. s.d. <http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod>. Acesso em: 2023-11-27.

MATOS, F. D.; OLIVEIRA, W.; CASTOR, F.; REGO, P.; TRINTA, F. Multi-language offloading service: An android service aimed at mitigating the network consumption during computation offloading. In: *Proceedings of the Brazilian Symposium on Multimedia and the Web*. [S.l.: s.n.], 2022. p. 329–338.

MELHORPLANO.NET. *A evolução do celular*. s.d. <https://melhorplano.net/tecnologia/evolucao-do-celular>. Acesso em: 2024-02-11.

META OPEN SOURCE. *React Core Components*. s.d. <https://reactnative.dev/docs/components-and-apis>. Acesso em: 2023-11-27.

META OPEN SOURCE. *React Documentation*. s.d. <https://react.dev/>. Acesso em: 2023-11-27.

META PLATFORMS, INC. *Fabric Renderer*. s.d. <https://reactnative.dev/architecture/fabric-renderer>. Acesso em: 2023-11-27.

MICROSOFT. *What is .NET MAUI?* s.d. <https://learn.microsoft.com/en-us/dotnet/maui/what-is-maui?view=net-maui-8.0>. Acesso em: 2023-11-27.

MICROSOFT. *Xamarin Documentation*. s.d. <https://learn.microsoft.com/en-us/xamarin/>. Acesso em: 2023-11-27.

MONSOON SOLUTIONS, INC. *High Voltage Power Monitor*. s.d. <https://www.msoon.com/high-voltage-power-monitor>. Acesso em: 2023-11-27.

MOSKALA, M.; WOJDA, I. *Android Development with Kotlin*. [S.l.]: Packt Publishing Ltd, 2017.

MOTOROLA MOBILITY LLC. *Motorola Milestones*. s.d. Disponível em: <https://www.motorola.com/us/about/motorola-history-milestones>. Acesso em: 10 mar 2024.

NAWROCKI, P.; WRONA, K.; MARCZAK, M.; SNIEZYNSKI, B. A comparison of native and cross-platform frameworks for mobile applications. *Computer*, IEEE, v. 54, n. 3, p. 18–27, 2021.

NIDHRA, S.; DONDETI, J. Black box and white box testing techniques-a literature review. *International Journal of Embedded Systems and Applications (IJESA)*, v. 2, n. 2, p. 29–50, 2012.

NUCCI, D. D.; PALOMBA, F.; PROTA, A.; PANICHELLA, A.; ZAIDMAN, A.; LUCIA, A. D. Petra: a software-based tool for estimating the energy profile of android applications. In: IEEE. *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*. [S.l.], 2017. p. 3–6.

NUCCI, D. D.; PALOMBA, F.; PROTA, A.; PANICHELLA, A.; ZAIDMAN, A.; LUCIA, A. D. Software-based energy profiling of android apps: Simple, efficient and reliable? In: IEEE. *2017 IEEE 24th international conference on software analysis, evolution and reengineering (SANER)*. [S.l.], 2017. p. 103–114.

OLIVEIRA, W.; MORAES, B.; CASTOR, F.; FERNANDES, J. P. Analyzing the resource usage overhead of mobile app development frameworks. In: *Proceedings of the 27th International Conference on Evaluation and Assessment in Software Engineering*. [S.l.: s.n.], 2023. p. 152–161.

OLIVEIRA, W.; MORAES, B.; CASTOR, F.; FERNANDES, J. a. P. Analyzing the resource usage overhead of mobile app development frameworks. In: *Proceedings of the 27th International Conference on Evaluation and Assessment in Software Engineering*. New York, NY, USA: Association for Computing Machinery, 2023. (EASE '23), p. 152–161. ISBN 9798400700446. Disponível em: <https://doi.org/10.1145/3593434.3593487>.

OLIVEIRA, W.; MORAES, B.; CASTOR, F.; FERNANDES, J. P. Ebserver: Automating resource-usage data collection of android applications. In: *2023 IEEE/ACM 10th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*. [S.l.: s.n.], 2023. p. 55–59.

OLIVEIRA, W.; MORAES, B.; CASTOR, F.; FERNANDES, J. Ebserver: Automating resource-usage data collection of android applications. In: *2023 IEEE/ACM 10th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*. Los Alamitos, CA, USA: IEEE Computer Society, 2023. p. 55–59. Disponível em: <https://doi.ieeecomputersociety.org/10.1109/MOBILSoft59058.2023.00014>.

OLIVEIRA, W.; OLIVEIRA, R.; CASTOR, F. A study on the energy consumption of android app development approaches. In: IEEE. *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. [S.l.], 2017. p. 42–52.

OLIVEIRA, W.; OLIVEIRA, R.; CASTOR, F.; PINTO, G.; FERNANDES, J. P. Improving energy-efficiency by recommending java collections. *Empirical Software Engineering*, Springer, v. 26, n. 3, p. 55, 2021.

PALMIERI, M.; SINGH, I.; CICCHETTI, A. Comparison of cross-platform mobile development tools. In: IEEE. *2012 16th International Conference on Intelligence in Next Generation Networks*. [S.l.], 2012. p. 179–186.

PALOMBA, F.; NUCCI, D. D.; PANICHELLA, A.; ZAIDMAN, A.; LUCIA, A. D. On the impact of code smells on the energy consumption of mobile applications. *Information and Software Technology*, Elsevier, v. 105, p. 43–55, 2019.

PEREIRA, R.; CARÇÃO, T.; COUTO, M.; CUNHA, J.; FERNANDES, J. P.; SARAIVA, J. Spelling out energy leaks: Aiding developers locate energy inefficient code. *Journal of Systems and Software*, Elsevier, v. 161, p. 110463, 2020.

PETERS, M.; SCOCCIA, G. L.; MALAVOLTA, I. How does migrating to kotlin impact the run-time efficiency of android apps? In: *2021 IEEE 21st International Working Conference on Source Code Analysis and Manipulation (SCAM)*. [S.l.: s.n.], 2021. p. 36–46.

PETERS, M.; SCOCCIA, G. L.; MALAVOLTA, I. How does migrating to kotlin impact the run-time efficiency of android apps? In: IEEE. *2021 IEEE 21st International Working Conference on Source Code Analysis and Manipulation (SCAM)*. [S.l.], 2021. p. 36–46.

PINTO, C. M.; COUTINHO, C. From native to cross-platform hybrid development. In: *2018 International Conference on Intelligent Systems (IS)*. [S.l.: s.n.], 2018. p. 669–676.

PINTO, G.; LIU, K.; CASTOR, F.; LIU, Y. D. A comprehensive study on the energy efficiency of java's thread-safe collections. In: IEEE. *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. [S.l.], 2016. p. 20–31.

QUALCOMM TECHNOLOGIES, INC. *Introducing Trepn Profiler 6.0*. s.d. <https://www.qualcomm.com/news/onq/2015/04/introducing-trepn-profiler-60>. Acesso em: 2023-11-27.

QUALCOMM TECHNOLOGIES, INC. *Snapdragon Profiler*. s.d. Disponível em: <https://developer.qualcomm.com/software/snapdragon-profiler>. Acesso em: 19 mar 2024.

RUA, R.; SARAIVA, J. Pyanadroid: A fully-customizable execution pipeline for benchmarking android applications. In: IEEE. *2023 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. [S.l.], 2023. p. 586–591.

RUA, R.; SARAIVA, J. a. E-manafa: Energy monitoring and analysis tool for android. In: . New York, NY, USA: Association for Computing Machinery, 2023. (ASE22). ISBN 9781450394758. Disponível em: <https://doi.org/10.1145/3551349.3561342>.

STACK OVERFLOW. *Stack Overflow Developer Survey 2023*. 2023. <https://survey.stackoverflow.co/2023/>. Acesso em: 2023-11-27.

STATCOUNTER. *Mobile Operating System Market Share Worldwide (Feb 2023 - Feb 2024)*. 2024. Disponível em: <https://gs.statcounter.com/os-market-share/mobile/worldwide>. Acesso em: 16 mar 2024.

THE APACHE SOFTWARE FOUNDATION. *Cordova Documentation*. s.d. <https://cordova.apache.org/docs/en/latest/>. Acesso em: 2023-11-27.

TORVALDS, L. *Git*. s.d. <https://git-scm.com/>. Acesso em: 2024-02-11.

WEI, X.; GOMEZ, L.; NEAMTIU, I.; FALOUTSOS, M. Profiledroid: Multi-layer profiling of android applications. In: *Proceedings of the 18th annual international conference on Mobile computing and networking*. [S.l.: s.n.], 2012. p. 137–148.

XANTHOPOULOS, S.; XINOGALOS, S. A comparative analysis of cross-platform development approaches for mobile applications. In: *Proceedings of the 6th Balkan Conference in Informatics*. [S.l.: s.n.], 2013. p. 213–220.

ZIMMERLE, C.; OLIVEIRA, W.; GAMA, K.; CASTOR, F. Reactive-based complex event processing: An overview and energy consumption analysis of cep. js. In: *Proceedings of the XXXIII Brazilian Symposium on Software Engineering*. [S.l.: s.n.], 2019. p. 84–93.