Universidade Federal de Pernambuco
Centro de Informática

Graduação em Ciência da Computação

Estudo comparativo de desempenho entre aplicações Android Nativas vs. Flutter

André Ferreira Santos Sousa

Trabalho de Graduação

Recife - PE
17 de outubro de 2024

Universidade Federal de Pernambuco
Centro de Informática

André Ferreira Santos Sousa

Estudo comparativo de desempenho entre aplicações Android Nativas vs. Flutter

Trabalho apresentado ao Programa de Graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco como requisito parcial para obtenção do grau de Bacharel em Ciência da Computação.

Orientador: *Leopoldo Teixeira*

Recife - PE
17 de outubro de 2024

Ficha de identificação da obra elaborada pelo autor,
através do programa de geração automática do SIB/UFPE

Sousa, André Ferreira Santos.
Estudo comparativo de desempenho entre aplicações Android nativas vs. Flutter / André Ferreira Santos Sousa. - Recife, 2024.
29 p. : il., tab.

Orientador(a): Leopoldo Motta Teixeira
Trabalho de Conclusão de Curso (Graduação) - Universidade Federal de Pernambuco, Centro de Informática, Ciências da Computação - Bacharelado, 2024.
Inclui referências.

1. Android. 2. Desempenho. 3. Flutter. 4. Jetpack Compose. 5. Kotlin. I. Teixeira, Leopoldo Motta. (Orientação). II. Título.

000 CDD (22.ed.)

ANDRÉ FERREIRA SANTOS SOUSA

Estudo comparativo de desempenho entre aplicações Android Nativas vs. Flutter

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em 2024 da Universidade Federal de Pernambuco, como requisito parcial para obtenção do título de bacharel em Ciência da Computação.

Aprovado em: 17/10/2024

BANCA EXAMINADORA

_____

Prof. Dr. Leopoldo Motta Teixeira(Orientador)

Universidade Federal de Pernambuco

_____

Prof. Dr. Kiev Santos da Gama (Examinador Interno)

Universidade Federal de Pernambuco

Dedico este trabalho principalmente a minha mãe, que sempre batalhou muito para que eu estivesse aqui.

Agradecimentos

Agradeço a minha mãe, que diante de todas as dificuldades sempre deu seu máximo para garantir minha educação.

Agradeço ao meu pai por todos os conselhos e por ter me incentivado até o final.

Agradeço a minha madrinha, Lilian, por todo o suporte durante todos esses anos.

Agradeço a Erika, minha companheira, por todo o amor e incentivo.

Agradeço aos amigos feitos no curso, que sempre me ajudaram bastante e me acompanharam até o final dessa jornada, que não foi fácil.

Agradeço aos professores por todo o conhecimento que me repassaram.

Agradeço a Leopoldo pela orientação neste trabalho.

E por fim, agradeço ao Andre do passado por não ter desistido e se esforçado durante todo o curso de graduação diante de todos os problemas.

Resumo

Com a crescente demanda de aplicações móveis, surge uma necessidade de criação de aplicações de modo mais efetivo, rápido e menos custoso que o desenvolvimento invididual para cada sistema operacional. Diante dessa necessidade, surgiram os *frameworks* multiplataforma que visam atender esta necessidade do mercado.

Porém, ainda não se pode deixar de lado o desenvolvimento nativo para cada sistema, a depender da necessidade de desempenho da aplicação e não temos métricas claras quanto a diferença para justificar esta necessidade.

Este trabalho visa realizar testes de comparação de desempenho entre aplicações desenvolvidas de modo nativo utilizando Kotlin com Jetpack Compose e as desenvolvidas de modo multiplataforma utilizando o *framework* Flutter, escrito em Dart.

Palavras-chave: Android, Kotlin, Jetpack Compose, Flutter, Performance, Desempenho

Abstract

With the growing demand for mobile applications, there has been a need to create applications more effectively, quickly, and at a lower cost than developing individually for each operating system. To meet this need, cross-platform frameworks have emerged to address this market demand.

However, native development for each system cannot be disregarded, depending on the application's performance needs, and we lack clear metrics to justify this requirement.

This work aims to perform performance comparison tests between applications developed natively using Kotlin with Jetpack Compose and those developed cross-platform using the Flutter framework, written in Dart.

Keywords: Android, Kotlin, Jetpack Compose, Flutter, Performance

Lista de Figuras

Lista de Tabelas

Sumário

1. Introdução

1.1 Motivação

Nos últimos anos, as aplicações móveis vêm sendo cada vez mais utilizadas, devido ao acesso facilitado a dispositivos móveis. Os que utilizam o sistema operacional Android, possuem hoje a maior participação de mercado no mundo com 72,15% e uma porcentagem ainda maior no Brasil, com 82,38% [4], demonstrando que a escolha de uma ferramenta de desenvolvimento para sistemas Android que melhor se adeque à necessidade de sua aplicação é de grande importância para seus usuários.

Essas aplicações por muito tempo foram desenvolvidas de forma individual para cada sistema operacional, até o surgimento dos *frameworks* multiplataformas, que são ferramentas que facilitam o desenvolvimento para mais de um sistema operacional simultaneamente, diminuindo o esforço e custo necessários para as empresas e times de desenvolvedores. Os mais utilizados são o Flutter e o React Native [5].

Com isso, algumas empresas deixaram de desenvolver aplicativos de modo nativo, sem critérios claros de escolha quanto à necessidade de desempenho da aplicação. Um destes exemplos é o do Airbnb, que retornou ao desenvolvimento nativo, abandonando o React Native [6].

Da mesma forma que pode haver uma necessidade de voltar ao desenvolvimento nativo, também há casos em que a empresa migra sua aplicação para o desenvolvimento multiplataforma, visando economia com o número de desenvolvedores, tempo de desenvolvimento e diminuição de fluxos de trabalho, como é o caso da Headspace [7].

Escolher uma tecnologia para sua aplicação não é algo simples, considerando que quanto mais complexa for, mais atrelado àquela tecnologia seu produto estará [8]. A falta de informação sobre o tópico pode levar a migrações evitáveis de tecnologia, trazendo economia de tempo e dinheiro para a empresa.

1.2 Trabalhos Relacionados

Na literatura atual, há algumas discussões acerca da escolha e vantagens de cada tecnologia, em Meirelles et al. [9] foram realizadas pesquisas acerca do tema com estudantes de graduação e mestrado, além de desenvolvedores *mobile* experientes, coletando dados sobre a diferença de cada abordagem e quando escolher cada uma, mostrando fortes vantagens de desempenho e uso de hardware na opinião dos desenvolvedores quanto ao

nativo.

Em Olsson [10] temos um estudo comparativo de desempenho levando em conta aspectos como tamanho e complexidade do código, utilização de CPU e o sentimento do usuário ao utilizar a aplicação. No estudo é demonstrado um pico maior de utilização de CPU para o nativo mas com uma menor média de uso.

No estudo de Andersson [11], temos a comparação do tempo de execução de determinadas funções, utilização de CPU e uso de memória RAM. Aqui, os resultados demonstram uma melhor otimização do Android no quesito de uso de CPU mas em relação a memória, há uma vantagem para o Flutter.

1.3 Objetivo

O objetivo deste estudo é fazer uma comparação de desempenho através de testes manuais e de integração entre uma aplicação desenvolvida com Flutter e uma aplicação desenvolvida de modo nativo utilizando Kotlin e Jetpack Compose, obtendo e inferindo alguns dados estatísticos para fornecer métricas quantitativas que agreguem na escolha da ferramenta de desenvolvimento utilizada para sua aplicação.

O restante deste trabalho está organizado da seguinte maneira: O Capítulo 2 apresenta conceitos para o desenvolvimento móvel, apresentando as linguagens de programação e *frameworks* utilizados. No Capítulo 3, apresentamos o processo de desenvolvimento das aplicações utilizadas e no quarto capítulo discutimos como será a metodologia. Nos capítulos 5 e 6, apresentamos os resultados e discutimos acerca dos mesmos.

2. Background

Neste capítulo, apresentaremos conceitos básicos do desenvolvimento móvel, mostrando a principal linguagem atualmente utilizada no desenvolvimento nativo, que é o Kotlin, além das formas de construir as interfaces, com os layouts XML e o Jetpack Compose. Apresentaremos também o desenvolvimento multiplataforma utilizando Flutter e Dart, sua linguagem de programação.

2.1 Desenvolvimento Android Nativo

No desenvolvimento nativo, atualmente a linguagem padrão é o Kotlin, que é uma linguagem de programação criada pela Jetbrains [12] com diferentes propósitos de uso, sendo um deles o desenvolvimento de aplicações Android. Seu uso já é majoritário nesse meio, sendo utilizado por mais de 60% dos desenvolvedores profissionais [13]. A própria Google, dona do sistema operacional, firmou um compromisso para utilizar cada vez mais o Kotlin nesse ambiente, afirmando que é mais expressivo, conciso, com código mais seguro e interoperável que o Java [14].

Figura 2.1: Diferença de um mesmo código em Java e Kotlin. Fonte: Technology Rivers, 2023 [1].

Atualmente há duas formas de se criar interfaces com o Kotlin, a primeira é utilizando o layout XML no Android sendo a forma tradicional de criar a aparência de um aplicativo, utilizando o paradigma imperativo. Nele, a parte visual, como botões e textos, é definida em um arquivo XML separado e referenciado dentro do código. Para conectar essa aparência ao funcionamento da aplicação, é preciso utilizar métodos que buscam esses elementos e dizem o que eles devem fazer [15].

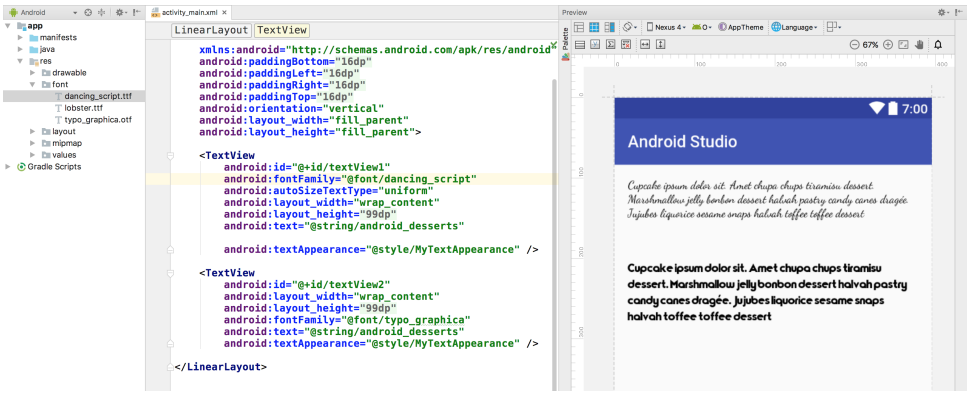

Figura 2.2: Exemplo de um layout em XML. Fonte: Android Developers, 2024 [2].

A mais nova é utilizando o Jetpack Compose, sendo mais moderno e simplificado de criar a aparência de um aplicativo Android, que utiliza do paradigma declarativo, lançado em 2021. Ao invés de utilizar arquivos separados, tudo é feito diretamente no código através da declaração de *Composables*. Isso permite que o visual da aplicação se atualize automaticamente conforme as informações mudam, tornando o processo mais rápido e fácil de entender [16].

2.1.1 Diferença entre Compose e XML

A principal diferença entre o layout XML e o Jetpack Compose está na forma como a interface do aplicativo Android é criada e gerenciada. O layout XML segue o método tradicional, onde a parte visual é separada do código e exige uma conexão manual entre o que é visto na tela e o que a aplicação faz, o que pode ser mais trabalhoso e menos eficiente. Já o Compose simplifica esse processo ao integrar tudo no código, facilitando o desenvolvimento ao permitir que a interface se atualize automaticamente quando algo muda e aumentando a flexibilidade.

Embora o Jetpack Compose seja o modo mais recente e recomendado para criação de interfaces, ainda enfrenta alguns problemas de desempenho em comparação com o layout em XML como mostrado no estudo de Wahlandt [17], que demonstra vantagens ao XML principalmente no que cerne ao tempo de criação de quadros.

2.2 Desenvolvimento multiplataforma

O Flutter é um framework também desenvolvido pela Google, usado para criar interfaces de usuário nativas para diversas plataformas, como Android, iOS, web e desktop, a partir de um único código. Ele utiliza Dart como linguagem base e adota uma abordagem declarativa, assim como o Compose, em que cada elemento é chamado de *widget*, e é usado para construção de interfaces. O Flutter permite o desenvolvimento rápido e a alta performance, com recursos como "hot reload"e *widgets* personalizáveis que facilitam a criação de aplicativos visualmente atraentes e responsivos. Além disso, permite também a criação e utilização de código nativo para tarefas que possam demandar interação com APIs específicas de um sistema.

Dart é uma linguagem de programação desenvolvida pela Google, projetada para ser eficiente, fácil de aprender e produtiva. Ela é orientada a objetos e possui tipagem estática, que foi originalmente criada para o desenvolvimento web mas evoluiu para ser a base do Flutter, com foco em oferecer uma sintaxe simples e intuitiva para o desenvolvimento de interfaces de usuário.

2.3 Renderização de quadros

No Android, um quadro é uma imagem estática que representa o estado atual da interface. Esses quadros são continuamente renderizados para fornecer uma experiência visual dinâmica. Durante cada ciclo de exibição, o sistema operacional coleta as instruções para desenhar elementos visuais e as envia para o sistema de renderização. Um conjunto contínuo de quadros gera a percepção de animação ou transição suave.

No nativo, esse processo de renderização envolve uma colaboração entre várias threads para garantir uma interface fluida e responsiva. O processo começa na UI Thread, onde eventos de interação do usuário e atualizações de layout são processados. A árvore de *layout* é construída para calcular o tamanho e a posição dos elementos na tela, seguida pela fase de desenho, onde as *views* são preparadas para serem exibidas. Em seguida, a Render Thread entra em ação, compondo os elementos da interface e enviando as instruções gráficas para a GPU.

A GPU é responsável pela rasterização, convertendo as instruções visuais da Render Thread em pixels que serão exibidos na tela. O processo completo da criação e renderização de um quadro precisa ser concluído em até 16.6ms para garantir uma taxa de 60 quadros por segundo (FPS), evitando atrasos e travamentos visuais, conhecidos como *jank* [18].

No Flutter, o processo de renderização é similar, com a UI Thread e a Raster Thread trabalhando em paralelo para garantir uma performance suave. A UI Thread é responsável por processar o quadro atual (N), realizando a construção e organização da árvore de *widgets*, e calculando as atualizações visuais com base nas mudanças de estado do aplicativo. Enquanto isso, a Raster Thread está processando o quadro anterior (N-1), convertendo as instruções da árvore de renderização em pixels que serão exibidos pela GPU. [19].

3. Desenvolvimento da aplicação

3.1 Criação

Para garantir testes de comparação justos entre as duas tecnologias, foi crucial escolher uma abordagem que permitisse desenvolver aplicações o mais semelhantes possível. Inicialmente foram consideradas duas opções: utilizar uma aplicação *open-source* existente e adaptá-la para a outra tecnologia ou desenvolver duas do início, buscando maximizar a uniformidade em cada funcionalidade desenvolvida.

Optamos por desenvolver duas novas aplicações. Essa decisão foi motivada pela falta de familiaridade prévia do pesquisador com ambas as linguagens e tecnologias envolvidas, além da necessidade de maior controle sobre o código e as escolhas de implementação. Essa abordagem também evitou a complexidade de compreender e modificar códigos já existentes, além de prevenir a utilização de aplicações excessivamente simples ou que possuiriam a necessidade de remover ou refatorar funcionalidades desnecessárias para o estudo.

A aplicação desenvolvida faz consultas a uma API de filmes, envolvendo ações como visualizar uma lista de filmes populares, buscar por filmes, visualizar seus detalhes e uma tela de filmes favoritos. Além dessas funcionalidades, há um gráfico de mercado atualizando continuamente o preço de criptomoedas utilizando uma API que fornece estes dados através de websockets, e uma lista aleatória que atualize cada componente em intervalos regulares de 100ms para forçar a atualização contínua dos componentes.

Para conseguirmos criar estas funcionalidades, necessitamos de bibliotecas externas em ambas as tecnologias e dando escolha as mais populares e que se adequassem melhor ao escopo da aplicação. No geral, as bibliotecas externas foram utilizadas para o gerenciamento de estados, injeção de dependências entre os componentes e serviços, requisições HTTP para chamadas as APIs, utilização de um banco de dados local para persistir informações, criação de gráfico, além das utilizadas para a criação dos testes.

3.2 Desenvolvimento e refinamento

A primeira aplicação desenvolvida foi a Android nativa, utilizando Kotlin com Jetpack Compose e após finalizá-la, foi a vez de criá-la em Flutter. Pelo fato do Flutter e do Jetpack Compose possuírem um padrão declarativo de criação de componentes, tornou-se mais fácil a escrita e reutilização do conhecimento adquirido ao desenvolver a primeira

aplicação. Além disso, com a utilização do ChatGPT [20] a transcrição de partes mais semelhantes, como por exemplo a parte visual dos componentes e lógicas mais simples foram facilitadas. Ainda assim os códigos transformados nem sempre eram funcionais e necessitavam de diversos ajustes lógicos e visuais.

Para facilitar o entendimento das funcionalidades da aplicação, aqui estão listadas em tabela:

Tabela 3.1: Funcionalidades da aplicação

Tela	Objetivo	Funcionalidades utilizadas
Filmes populares	Listar os filmes por ordem de popularidade	Listagem de filmes realizada por demanda, chamada a API do TMDB, utilização de gerenciamento de estado.
Procurar filmes	Procurar filmes por nome	Chamada a API do TMDB, listagem dos filmes retornados por demanda, utilização de gerenciamento de estado.
Detalhes do filme	Detalhamento do filme, contendo informações como nome, descrição, data de criação, gêneros, avaliação e filmes similares.	Chamada a API do TMDB e renderização dos componentes após resultado, utilização de gerenciamento de estados, listagem de filmes retornados por demanda, chamada ao banco de dados local SQLite.
Filmes favoritos	Tela contendo uma lista com filmes favoritos	Listagem de filmes favoritos, utilização de gerenciamento de estados, chamadas ao banco de dados local SQLite.
Gráfico de Mercado	Contém gráfico de ações de mercado relacionadas a criptomoedas	Chamada a API da Binance, utilização de websockets para fluxo contínuo de dados (1s), criação e atualização de gráfico a medida em que novas informações chegam.
Lista aleatória	Lista aleatória	Lista aleatória com objetos contendo 6 atributos atualizados em um curto intervalo de tempo (100ms), forçando a renderização contínua de toda a lista.

```
@Composable
fun MovieFavoriteItem(
    modifier: Modifier = Modifier,
    movie: Movie,
    onClick: (movieId: Int) -> Unit
) {
    Card(modifier = modifier
        .fillMaxWidth()
        .clickable {
            onClick(movie.id)
        }) {
        Column(
            modifier = Modifier
                .fillMaxWidth()
                .background(black),
            verticalArrangement = Arrangement.spacedBy(8.dp)
        ) { this: ColumnScope
            Box(
                modifier = Modifier
                    .fillMaxWidth()
                    .height(200.dp)
            ) { this: BoxScope
                AsyncImageUrl(
                    imageUrl = movie.imageUrl,
                    contentScale = ContentScale.FillWidth,
                    modifier = Modifier
                        .fillMaxWidth()
                        .padding(8.dp)
                )
            }
            Text(
                text = movie.title,
                maxLines = 1,
                fontSize = 20.sp,
                fontWeight = FontWeight.SemiBold,
                color = white,
                overflow = TextOverflow.Ellipsis
            )
        }
```

Figura 3.1: Exemplo de Composable

```dart
class MovieFavoriteItem extends StatelessWidget {
  final Movie movie;

  const MovieFavoriteItem({
    Key? key,
    required this.movie,
  }) : super(key: key);

  @override
  Widget build(BuildContext context) {
    return Card(
      child: InkWell(
        onTap: () => {(Navigator.pushNamed(
        context, Routes.detailMovies,
        arguments: movie.id))},
        child: Column(
          crossAxisAlignment: CrossAxisAlignment.start,
          children: [
            Container(
              width: double.infinity,
              height: 200,
              decoration: BoxDecoration(
                image: DecorationImage(
                  image: NetworkImage(movie.imageUrl),
                  fit: BoxFit.cover,
                ),  // DecorationImage
              ),  // BoxDecoration
              padding: const EdgeInsets.all(8.0),
            ),  // Container
            Container(
              padding: const EdgeInsets.all(8.0),
              width: double.infinity,
              decoration: BoxDecoration(
                shape: BoxShape.rectangle,
                color: Colors.black,
                border: Border(
                  bottom: BorderSide(color: Colors.white)
                ),  // Border
              ),  // BoxDecoration
              child: Text(
```

Figura 3.2: Exemplo de Widget

3.3 Fluxos de teste

Definidas as funcionalidades da aplicação e os tipos de testes a serem realizados, foi necessário definir também quais os fluxos de tela a serem executados para os testes de integração e os manuais, que serão explicados posteriormente, com cada fluxo testando partes da aplicação que podem ser diferentes ou não afim de testar partes específicas que resultem numa variação maior das métricas a serem coletadas.

3.3.1 Fluxo 1

O primeiro fluxo será focado na rolagem de lista, na navegação da aplicação e no gerenciamento de estados, que são fluxos básicos de uma aplicação.

1. Inicia na tela inicial da aplicação;
2. Faz uma longa rolagem da lista, fazendo a busca por um filme através de sua avaliação;
3. Clica no filme desejado e abre sua tela de detalhes, aguardando seu carregamento;
4. Aperta no botão de favoritar;
5. Volta para a tela anterior;
6. Navega para a tela de filmes favoritados.

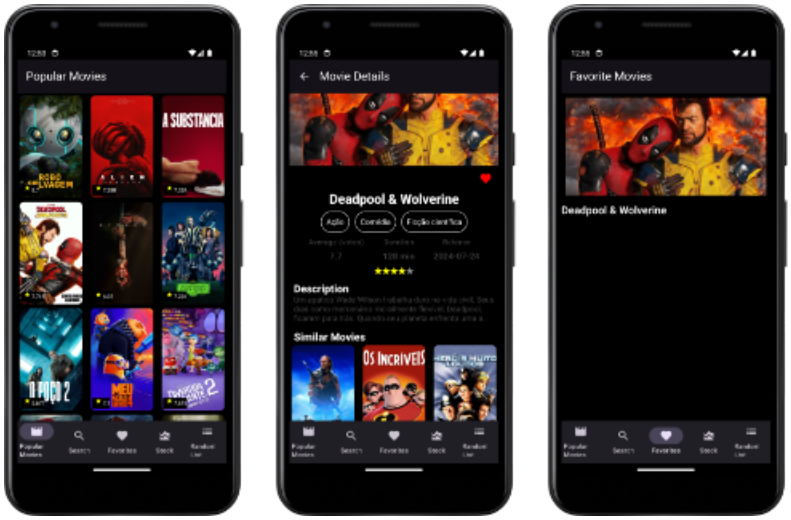

Figura 3.3: Telas presentes no Fluxo 1

3.3.2 Fluxo 2

1. Inicia na tela inicial da aplicação;
2. Navega para a tela de procurar filmes;
3. Pesquisa pelo filme "Garfield"e aguarda os resultados;
4. Clica no filme primeiro filme da lista de resultados e navega para tela de detalhes;
5. Volta para a tela de pesquisa;
6. Procura pelo filme "Star Wars";
7. Clica no filme primeiro filme da lista de resultados e navega para tela de detalhes;
8. Volta para a tela de pesquisa.

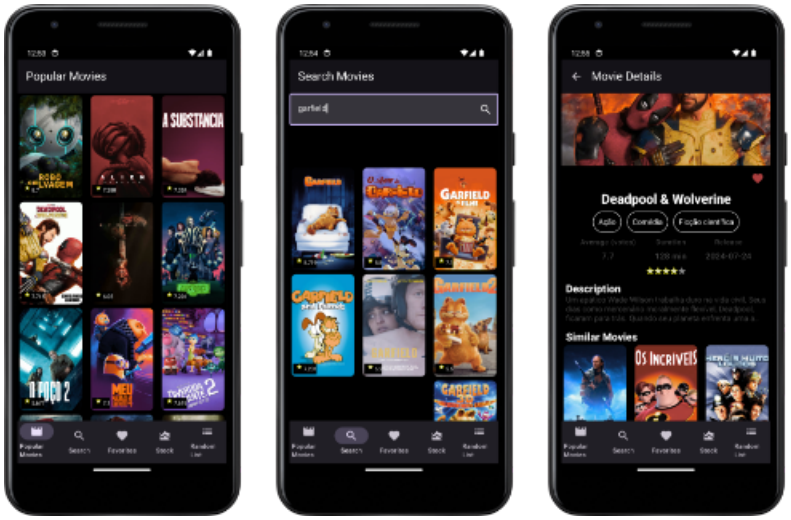

Figura 3.4: Telas presentes no Fluxo 2

3.3.3 Fluxo 3

1. Inicia na tela inicial da aplicação;
2. Navega para a tela de Mercado;
3. Faz uma rolagem até o final da lista;
4. Faz uma rolagem até o início da lista;
5. Realiza um zoom no gráfico um;
6. Realiza um zoom no gráfico dois;
7. Remove o zoom do gráfico um;
8. Remove o zoom do gráfico dois;

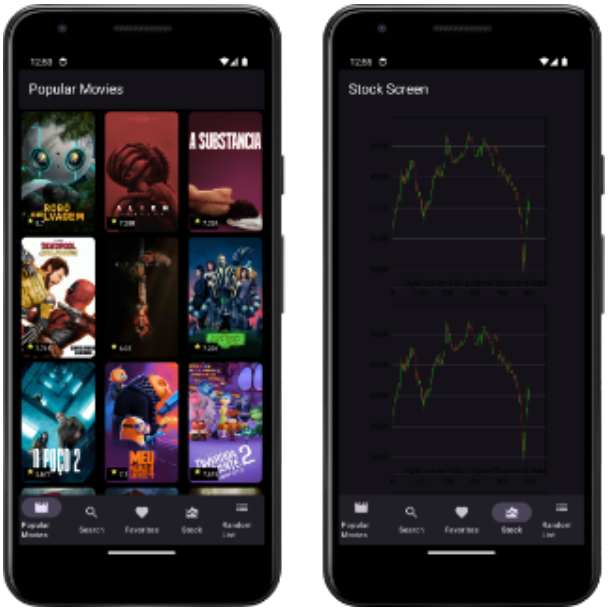

Figura 3.5: Telas presentes no Fluxo 3

3.3.4 Fluxo 4

1. Inicia na tela inicial da aplicação;
2. Navega para a tela de lista aleatória;
3. Faz uma rolagem até o item 50.

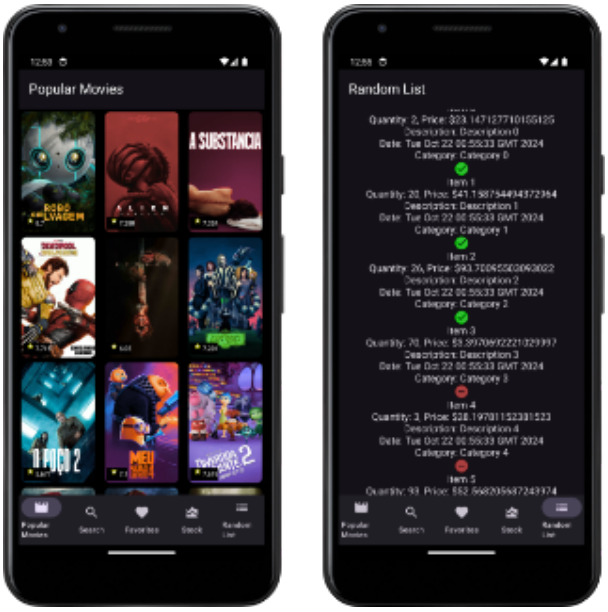

Figura 3.6: Telas presentes no Fluxo 4

4. Metodologia

Para a comparação de desempenho, foram utilizados testes manuais e de integração em diferentes fluxos de tela com o objetivo de capturar métricas específicas ao longo de suas execuções. A diversificação dos tipos de teste foi necessária devido à impossibilidade de coletar todas as métricas de uma única forma, principalmente em aplicações nativas que não possuem uma ferramenta de desenvolvimento que coleta métricas [21], diferentemente do Flutter, que dispõe de um analisador de desempenho próprio que fornece diversas métricas em tempo de desenvolvimento, chamado Flutter Developer [22].

Foi utilizado um dispositivo físico na execução dos testes para garantir resultados mais fiéis ao de um usuário final, com cada fluxo de teste de integração sendo executado dez vezes e manuais, cinco. Nos testes de integração, todos os quatro fluxos foram executados, já nos testes manuais foram utilizados apenas os fluxos um, três e quatro. Além disso, não haviam outras aplicações rodando em segundo plano para evitar interferência externa nas métricas obtidas. As configurações de hardware do dispositivo são as seguintes:

Tabela 4.1: Informações do dispositivo usado para os testes

Dispositivo	Galaxy S22 Ultra
Sistema Operacional	Android 14 com One UI 6.1
CPU	SM8450
GPU	Adreno 730
RAM	12GB

4.1 Métricas de comparação

Como métricas de comparação definimos algumas quantitativas, focados diretamente no desempenho da aplicação, e que acabam por influenciar diretamente na execução e fluidez ao ser executado pelo usuário final, são elas:

1. Tempo necessário para renderizar um quadro
2. Pico de utilização de memória RAM
3. Pico de utilização de CPU

4. Tamanho da aplicação após instalação (Armazenamento)
5. Tempo para início da aplicação

4.2 Testes de Integração

Os testes de integração compõem uma das fases do teste de software, em que a aplicação é testada como um todo, podendo ser chamado também de teste *end-to-end*, executando um fluxo de telas e/ou funcionalidades que busca validar a utilização dos componentes envolvidos de forma conjunta, verificando o comportamento da aplicação.

Com o Kotlin, estes testes podem ser feitos utilizando a biblioteca Macrobenchmark [23], que permite a execução dos testes de integração e da obtenção de algumas métricas capturadas durante o fluxo de teste, porém neste estudo vamos utilizá-los apenas para obter o tempo necessário para renderização de um quadro e para checar o tempo necessário para o início da aplicação, que são capturados através dos métodos *StartupTimingMetric* e *FrameTimingMetric*, como mostrado nas imagens abaixo.

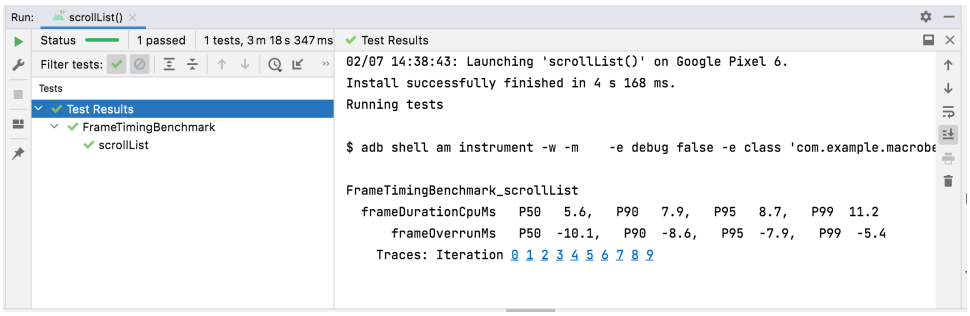

Figura 4.1: Exemplo de resultado do FrameTimingMetric. Fonte: Android Developers, 2024 [3].

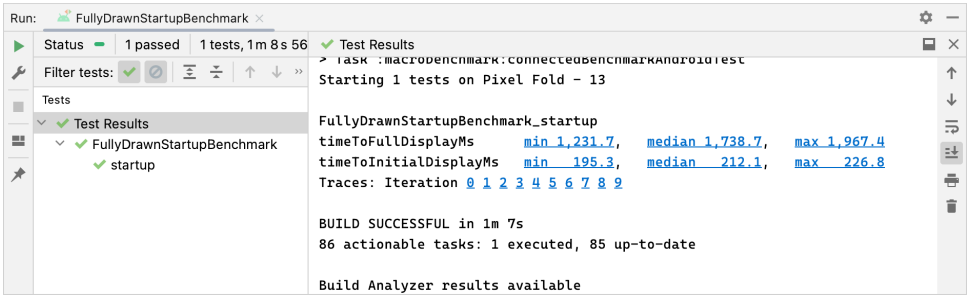

Figura 4.2: Exemplo de resultado do StartupTimingMetric. Fonte: Android Developers, 2024 [3].

No Flutter, estas mesmas métricas serão obtidas através da biblioteca integration_test, para o tempo de renderização de quadros, e através do comando *"flutter run –profile –startup-time"* [24]. Diferente do Macrobenchmark, aqui os resultados obtidos são um pouco mais descritivos e resultam em arquivos json.

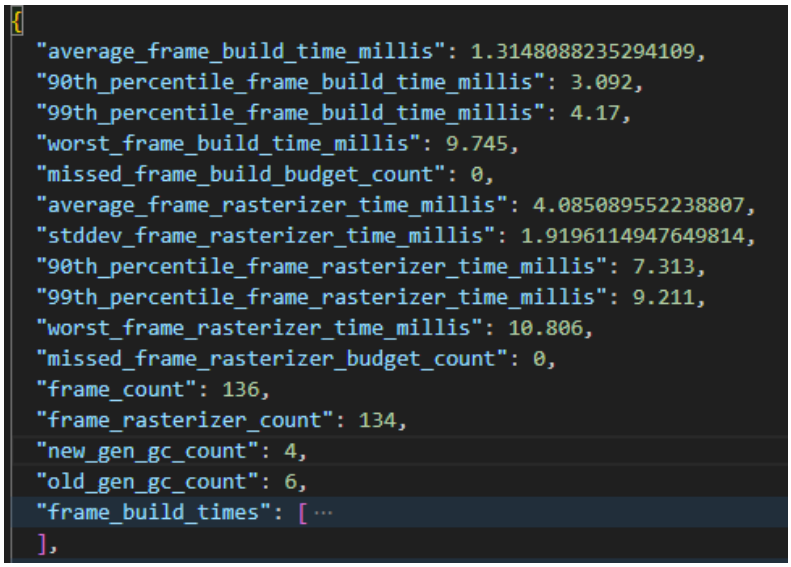

Figura 4.3: Exemplo de resultado do integration_test

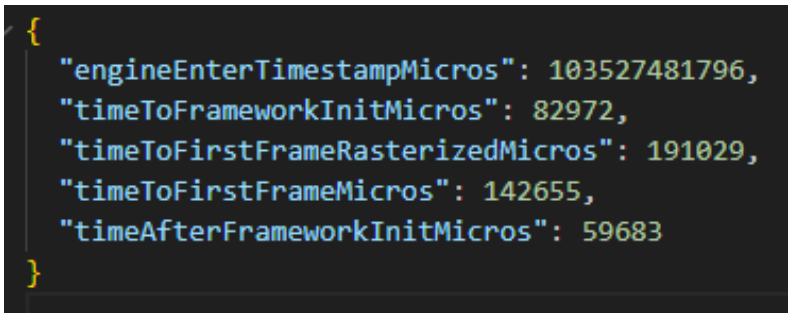

Figura 4.4: Exemplo de resultado do comando flutter run –profile –startup-time

4.3 Testes Manuais

Os testes manuais foram utilizados para executar os fluxos de tela como um usuário comum e capturando as métricas de memória RAM utilizada pela aplicação e o uso de CPU, que são disponibilizadas em um gráfico pelo analisador de desempenho do Android Studio [25], que é um ambiente de desenvolvimento integrado (IDE) próprio para o desenvolvimento Android. Além deles, o tamanho utilizado pela aplicação após a instalação foi checado de forma manual no dispositivo.

Por fim, na imagem abaixo temos um simples fluxograma de como cada métrica será capturada.

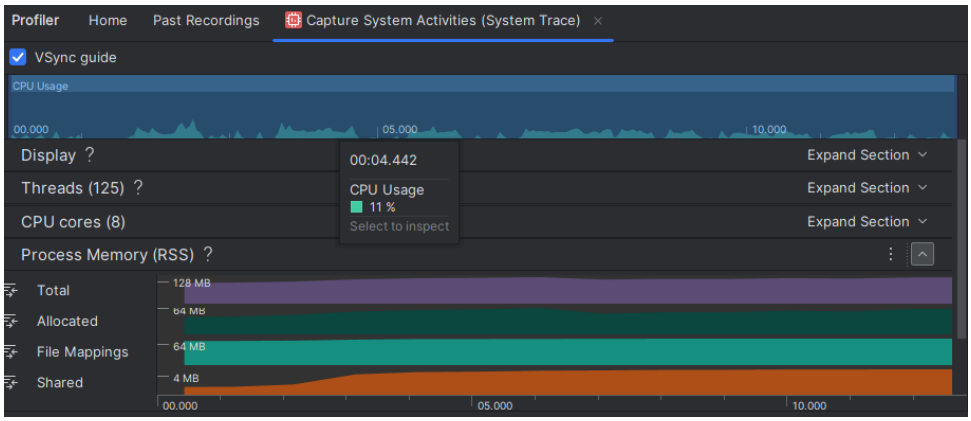

Figura 4.5: Exemplo de gráfico gerado com o Android Studio Profiler

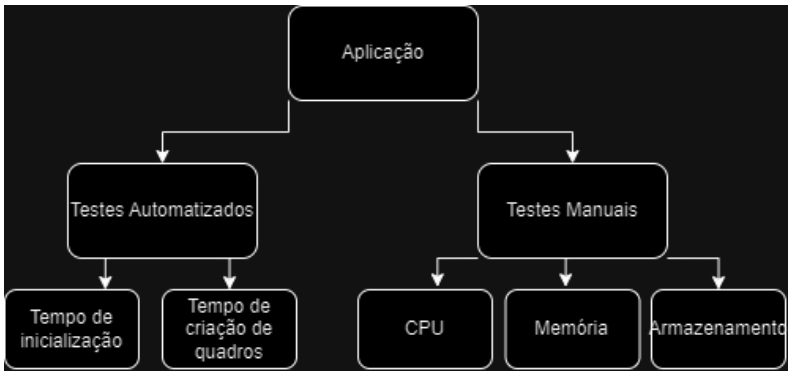

Figura 4.6: Metodologia de testes

5. Resultados

5.1 Armazenamento

Na primeira métrica, focado na comparação do armazenamento utilizado pela aplicação, foi obtida de forma direta, com cada aplicativo sendo instalado em sua versão final. Isso inclui etapas como ofuscação e otimização do código, representando o que o usuário final experimenta ao instalá-lo.

Tabela 5.1: Resultado do armazenamento utilizado pela aplicação

Armazenamento	**Tamanho**
Flutter	26,69MB
Android	5,81MB

Nesta métrica, a aplicação nativa apresentou uma grande vantagem, ocupando apenas 21,7% do tamanho da aplicação gerada com Flutter. Embora essa diferença possa ser irrelevante para dispositivos móveis modernos, como o utilizado neste estudo, com 256 GB de armazenamento disponível, ela se torna significativa em dispositivos mais simples, pois soma-se ao que pode ser consumido através da utilização de *cache* e outros dados da aplicação que serão salvos no dispositivo para melhorar a experiência do usuário.

5.2 Tempo de inicialização

Uma métrica importante ao usar uma aplicação é o tempo de inicialização, pois quanto maior o tempo de espera, maior a chance do usuário desistir de utilizá-la. A Google observou que, em páginas móveis, quando o tempo de carregamento aumenta de 1 para 5 segundos, a chance de abandono cresce em cerca de 90% [26], prejudicando bastante seu uso.

Para este teste, a cada iteração a aplicação era removida do segundo plano, o que poderia trazer interferências de um pré-carregamento. Os dados obtidos estão em milissegundos e representam o tempo desde o clique para abertura da aplicação até a renderização do primeiro quadro.

Ao analisar os resultados, observa-se uma vantagem significativa para o Flutter, que apresenta uma média de 155,7 ms, em comparação com 298,3 ms do Android nativo, representando um tempo de inicialização 47,81% mais rápido para o Flutter. Essa diferença

Tabela 5.2: Resultado do teste de inicialização

Startup	Itr 1	Itr 2	Itr 3	Itr 4	Itr 5	Itr 6	Itr 7	Itr 8	Itr 9	Itr 10
Flutter	210	168	142	146	144	153	146	154	152	142
Android	301	330	243	287	278	297	282	306	301	361

pode proporcionar uma experiência mais fluida e satisfatória para os usuários, reduzindo a probabilidade de abandono da aplicação logo no início de seu uso.

Além disso, o Flutter demonstrou uma menor variação nos tempos medidos, apresentando um desvio padrão de 21,5 ms, enquanto o Android registrou um desvio padrão de 34,8 ms, demonstrando uma maior previsibilidade na inicialização da aplicação.

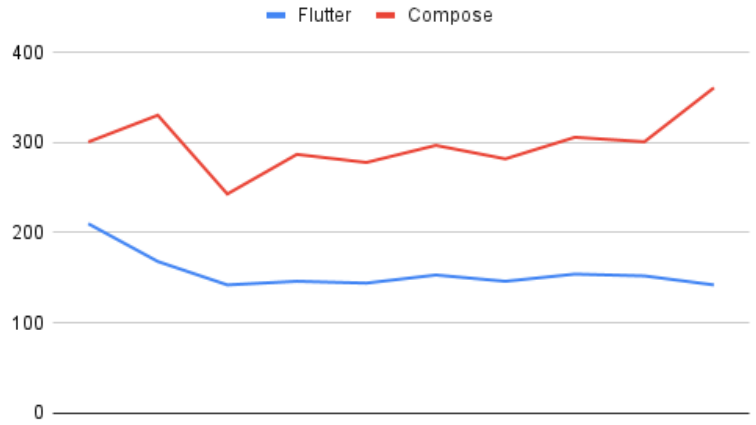

Figura 5.1: Tempo de inicialização

5.3 Tempo de Renderização de Quadros

Nesta seção será analisado o tempo levado para a renderização de um quadro, importante métrica para a fluidez da aplicação em que quanto menor o tempo, melhor. O processo completo da criação e renderização de um quadro precisa ser concluído em até 16.6ms para garantir uma taxa de 60 quadros por segundo (FPS). Caso um quadro ultrapasse este limiar de tempo, ele já pode ser considerado um *jank*. [18].

Os dados a seguir mostram os resultados em milissegundos dos percentis 50, 90 e 95 para o tempo de renderização de quadros em diferentes fluxos. As análises comparativas demonstram o desempenho das aplicações Flutter e nativa, focando na diferença percentual entre as duas ferramentas.

Para o fluxo um, o Flutter apresentou um desempenho melhor em comparação ao Compose em todos os percentis analisados. No percentil 50, o tempo de renderização médio para o Flutter foi de 5,9ms, enquanto o Compose registrou 8,3ms, resultando em uma melhoria de aproximadamente 28,9%. Nos percentis 90 e 95, Flutter também superou o Compose, com tempos de renderização menores em 25,9% e 10%, respectivamente.

Tabela 5.3: Percentis do tempo de renderização de quadros para o fluxo um

Fluxo 1	**50 Percentil**	**90 Percentil**	**95 Percentil**
Flutter	5,9	11,7	18
Compose	8,3	15,8	20

Tabela 5.4: Percentis do tempo de renderização de quadros para o fluxo dois

Fluxo 2	**50 Percentil**	**90 Percentil**	**95 Percentil**
Flutter	4,4	6,4	10,4
Compose	6,4	12,3	17

No fluxo dois, o Flutter continuou demonstrando um desempenho superior em todos os percentis, com uma diferença de respectivamente 31,3%, 48% e 38,8%. Além disto, o Compose registrou a incidência de *janks* próximos ao 95 percentil, que não acontece no Flutter.

Tabela 5.5: Percentis do tempo de renderização de quadros para o fluxo três

Fluxo 3	**50 Percentil**	**90 Percentil**	**95 Percentil**
Flutter	7	34,3	49,2
Compose	13,2	29,7	36,5

Para o fluxo três, no percentil 50 continuamos a ver vantagens para o Flutter, sendo 46,9% mais rápido na renderização de metade dos quadros. Ainda assim não há a existência de *janks* neste percentil, mas o resultado do Compose indica uma certa tendência a ter estes travamentos mais cedo que o Flutter. Nos percentis 90 e 95, o Compose apresentou menores tempos de renderização, com uma melhora de 13,4% e 25,8%, respectivamente.

Tabela 5.6: Percentis do tempo de renderização de quadros para o fluxo quatro

Fluxo 4	**50 Percentil**	**90 Percentil**	**95 Percentil**
Flutter	7,6	20,6	28,3
Compose	10,8	26,7	31,9

No último fluxo, o Flutter apresentou novamente tempos de renderização melhores em todos os percentis analisados. No percentil 50 houve uma diferença de 29,6%, e nos percentis 90 e 95, diferenças de 22,9% e 11,3% respectivamente, mas ainda com a presença de *janks* em ambos.

5.3.1 Análise Geral

De forma geral, o Flutter apresentou um desempenho superior ao Compose, sendo melhor no percentil 50 em todos os fluxos, com uma melhora média de 35,7%. Nos

percentis 90 e 95, o único fluxo que não apresentou superioridade foi no fluxo 3, com o Compose sendo 13,4% e 25,8% melhor nestes dois percentis. No restante, o Flutter obteve uma superioridade média de 29,38% e 17,7%. Isto demonstra uma boa superioridade para o Flutter no quesito de renderização de quadros, de modo que sua aplicação, nestes casos, é mais fluída. Abaixo, temos um gráfico com o tempo de renderização para cada percentil destas tecnologias.

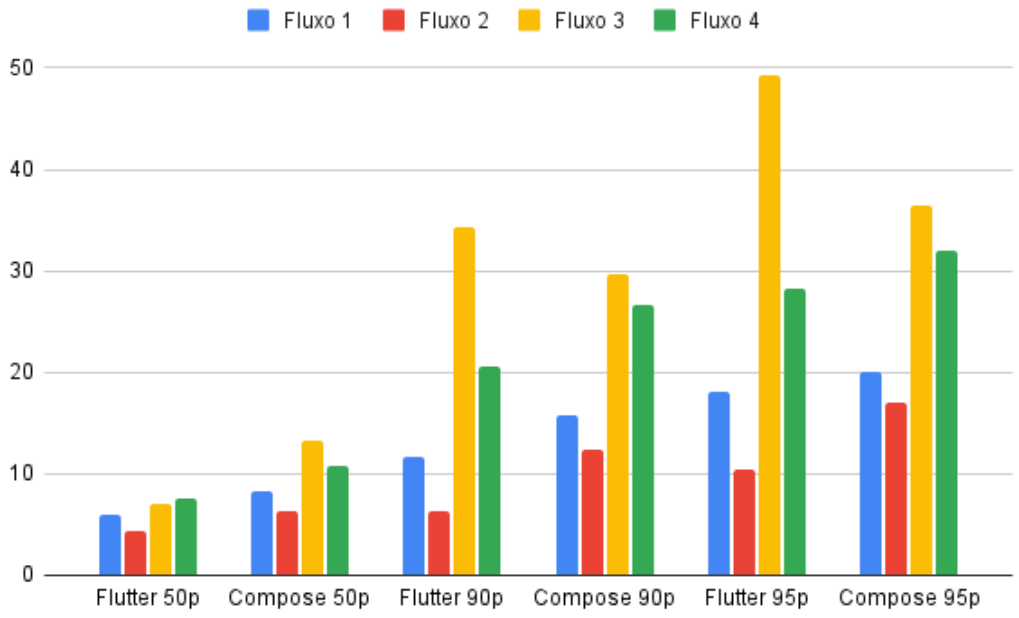

Figura 5.2: Tempo para renderização de quadros por percentil e *framework*

5.4 RAM

Para este teste, a métrica foi colhida com o pico de memória RAM em cada iteração por três diferentes fluxos da aplicação através do Android Profiler, via Android Studio. Os resultados estão em megabytes.

Tabela 5.7: Resultado do teste de memória RAM para o fluxo um

Fluxo 1	**Itr 1**	**Itr 2**	**Itr 3**	**Itr 4**	**Itr 5**
Flutter	373	365	370	388	368
Compose	180	157	161	169	160

Para o fluxo um, o consumo médio de memória RAM foi de 165,4 MB para o aplicativo nativo e de 372,8 MB para o Flutter, representando um aumento de aproximadamente 125,4% no uso de memória por parte do Flutter. Apesar disso, os resultados do Flutter mostraram maior consistência, com um desvio padrão de 8,3 MB, em comparação ao

Compose, que teve um desvio padrão de 9,5 MB, indicando uma variabilidade ligeiramente maior no último.

Tabela 5.8: Resultado do teste de memória RAM para o fluxo três

Fluxo 3	**Itr 1**	**Itr 2**	**Itr 3**	**Itr 4**	**Itr 5**
Flutter	232	221	227	224	229
Compose	174	153	161	156	175

No fluxo três, o consumo médio de memória foi de 226,6 MB para o Flutter e 163,8 MB para o Compose. Isso representa um aumento percentual de aproximadamente 38,3% no uso de memória para o Flutter em comparação ao Compose. O desvio padrão foi de 4,2 MB para o Flutter e 9,3 MB para o Compose, sugerindo que o Flutter manteve um comportamento mais consistente em termos de uso de memória, enquanto o Compose apresentou uma maior variabilidade.

Tabela 5.9: Resultado do teste de memória RAM para o fluxo quatro

Fluxo 4	**Itr 1**	**Itr 2**	**Itr 3**	**Itr 4**	**Itr 5**
Flutter	224	212	206	208	218
Compose	155	141	139	138	141

Para o fluxo quatro, o Flutter teve um consumo médio de memória de 213,6 MB, enquanto o Android teve 142,8 MB, o que representa um aumento percentual de 49,6% para o Flutter. O desvio padrão foi de 7,1 MB para o Flutter e 6,8 MB para o Compose, indicando que ambos os *frameworks* apresentaram variabilidade semelhante neste cenário, embora o Flutter ainda tenha tido um consumo maior.

Análise Geral

Considerando todos os fluxos analisados, o Compose consistentemente utilizou menos memória RAM em comparação ao Flutter, com diferenças percentuais variando de 38,3% a 125,4%. Em dois dos três cenários (um e três), o Flutter mostrou maior consistência (menor desvio padrão) no consumo de memória. No entanto, essa maior consistência vem acompanhada de um consumo significativamente maior de recursos, o que pode impactar dispositivos com menor capacidade de memória RAM disponível.

Na imagem abaixo, conseguimos perceber melhor estas diferenças. No Compose conseguimos perceber também que o uso de memória variou muito pouco durante cada fluxo, enquanto o Flutter teve uma redução considerável do fluxo 1 para o 3.

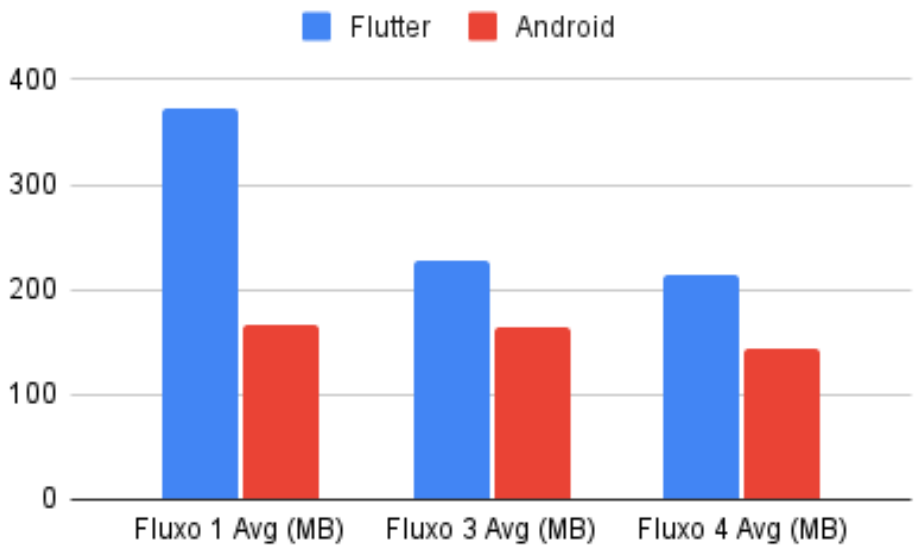

Figura 5.3: Pico de memória médio por fluxo

5.5 CPU

Tabela 5.10: Resultado do teste de CPU para o fluxo um

Fluxo 1	**Itr 1**	**Itr 2**	**Itr 3**	**Itr 4**	**Itr 5**
Flutter	36,5%	36%	38,8%	38,4%	45,6%
Compose	34%	48,1%	45,2%	40%	36,6%

No fluxo um, o pico médio de CPU para Flutter foi de 39,06% e para Compose foi de 40,78%, indicando que o Flutter teve um pico de cerca de 4,22% menor. O desvio padrão do Flutter foi de 3,55%, enquanto para o Compose foi de 5,44%, mostrando maior variabilidade no uso de CPU pelo Compose.

Tabela 5.11: Resultado do teste de CPU para o fluxo três

Fluxo 3	**Itr 1**	**Itr 2**	**Itr 3**	**Itr 4**	**Itr 5**
Flutter	20,9%	22,1%	24,8%	18,5%	18,3%
Compose	33,4%	27,2%	26,9%	29,8%	28,4%

No cenário três, o pico médio de CPU para Flutter foi de 20,92%, enquanto para o Compose foi de 29,14%, resultando em um pico de CPU cerca de 28,20% menor para o Flutter. Os desvios padrões foram 2,57% para Flutter e 2,61% para Compose, sugerindo que ambos tiveram variabilidade semelhante.

Para o cenário quatro, o pico médio de CPU foi de 13,84% para o Flutter e 20,22% para o Compose, indicando um uso de CPU cerca de 31,53% menor para o Flutter. O

Tabela 5.12: Resultado do teste de CPU para o fluxo quatro

Fluxo 4	**Itr 1**	**Itr 2**	**Itr 3**	**Itr 4**	**Itr 5**
Flutter	13%	13,8%	14,6%	11,9%	13,9%
Compose	16,1%	21%	19,3%	21,7%	22%

desvio padrão foi de 1,03% para o Flutter e 2,50% para o Compose, mostrando maior consistência no uso de CPU por parte do Flutter.

Análise Geral

Considerando todos os cenários, o Flutter consistentemente utilizou menos CPU em comparação ao Compose. As diferenças percentuais variaram de 4,22% a 31,53%, sendo o uso de CPU significativamente menor para o Flutter, especialmente nos cenários um e quatro. Além disso, o Flutter apresentou menor variabilidade, sugerindo um comportamento mais consistente, enquanto o Compose demonstrou maior oscilação nos valores obtidos.

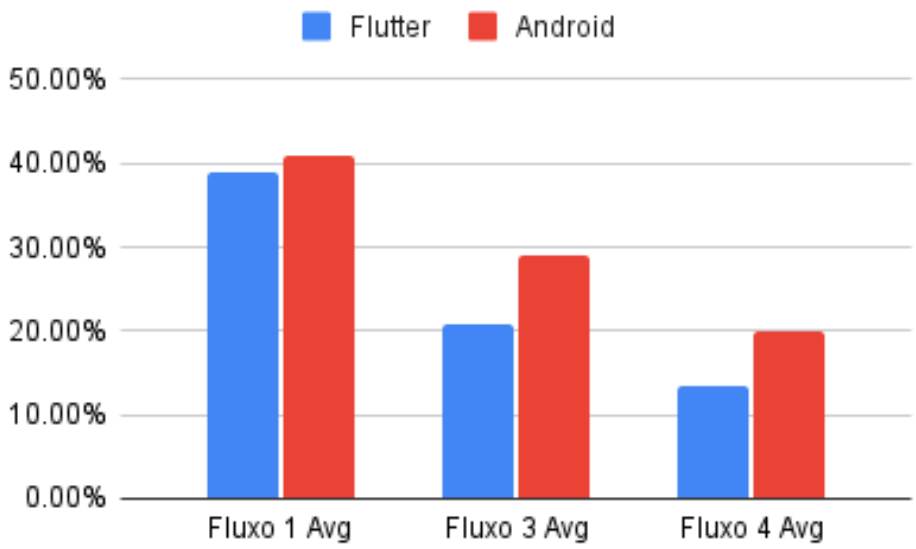

Figura 5.4: Pico médio de CPU por fluxo

5.6 Discussão

Este estudo revelou algumas diferenças entre o desempenho das aplicações desenvolvidas de forma nativa, com Kotlin e Jetpack Compose, e as criadas com Flutter. A principal conclusão é que o Flutter se mantém competitivo com o nativo, obtendo melhores resultados nas métricas do pico de CPU, renderização de quadros e no tempo de inicialização. Essa competitividade no desempenho, aliada a outras vantagens como o fato de ser multiplataforma, pode tornar o Flutter uma ótima escolha.

Por outro lado, o desenvolvimento nativo com Jetpack Compose se destacou no uso de recursos do dispositivo, como armazenamento e memória RAM. A aplicação nativa utilizou significativamente menos espaço de armazenamento, o que é um ponto de vantagem em dispositivos com recursos mais limitados. O consumo de memória RAM também foi consideravelmente menor, o que indica que, para aplicações que exigem otimização de recursos, especialmente em dispositivos de entrada, o Jetpack Compose ainda pode ser a melhor opção.

5.7 Limitações

Uma limitação importante deste estudo é que foi conduzido com uma aplicação de complexidade moderada, que por si só pode não explorar todas as potencialidades ou limitações de ambas as tecnologias. Aplicações mais complexas podem trazer à tona outros desafios e diferenças de desempenho que não foram totalmente capturados aqui.

A utilização do ChatGPT na geração do código pode também não ter empregado as melhores práticas para cada tecnologia, podendo ter gerado algum gargalo não identificado. Outro ponto a ser considerado é o fato de que o Jetpack Compose, sendo uma tecnologia mais recente, lançada em 2021, ainda está em constante evolução e ainda é mais instável e com um desempenho inferior as XML views [17].

Além disto, ele foi realizado em um único dispositivo com configurações de topo de linha, o que pode influenciar nos resultados obtidos evitando gargalos de desempenho. É possível que, em outros cenários ou dispositivos, essas diferenças se comportem de maneira distinta.

6. Conclusão

Este trabalho teve como objetivo comparar o desempenho entre aplicações desenvolvidas de forma nativa, utilizando Kotlin com Jetpack Compose, e de forma multiplataforma, utilizando Flutter. Através da criação de uma mesma aplicação destas duas formas, visamos fornecer métricas de desempenho que possam facilitar a escolha da ferramenta a ser utilizada.

Os resultados indicaram uma vantagem significativa do Flutter em termos de tempo de inicialização, uso de CPU e renderização de quadros. Essa eficiência no tempo de resposta e no gerenciamento de recursos do dispositivo faz do Flutter uma boa opção para projetos que priorizam rapidez no mercado e simplicidade de manutenção, sem uma grande perca de desempenho.

Por outro lado, o desenvolvimento nativo com Jetpack Compose destacou-se no consumo de armazenamento e memória RAM, tornando-se mais adequado para dispositivos com menos recursos. Esses resultados ajudam a esclarecer a escolha entre essas tecnologias e mostram que, atualmente, tanto Flutter quanto Jetpack Compose podem oferecer soluções viáveis e eficientes.

Em comparação com estudos anteriores, aqui vemos um resultado semelhante quanto ao pico de uso de CPU, se mostrando maior nas aplicações nativas, porém quanto a memória RAM, o maior uso foi na aplicação Flutter. Diferentemente dos estudos anteriores que se utilizaram de aplicações simples, a aplicação utilizada aqui possui um nível médio de complexidade, chegando mais próximo a uma aplicação do mundo real, o que pode ter influenciado na diferença dos resultados anteriores. Além disto adicionamos algumas métricas diferentes como o tempo de renderização de quadros, tempo de inicialização e o armazenamento utilizado.

Para pesquisas futuras, seria interessante expandir os testes para uma variedade maior de dispositivos e incluir aplicações mais complexas que possam exigir mais das tecnologias analisadas, utilizando também tanto os layouts XML quanto o Compose. Além disso, com o avanço contínuo do Jetpack Compose, novos estudos poderão revelar uma evolução em seu desempenho, tornando a comparação ainda mais dinâmica.

Referências Bibliográficas

[1] Technology Rivers. Kotlin vs java for native android app development. https://technologyrivers.com/blog/kotlin-vs-java-native-android-app-development/, 2023. Accessed: 2024-10-14.

[2] Android Developers. Fonts in xml - android developers. https://developer.android.com/develop/ui/views/text-and-emoji/fonts-in-xml#fonts-in-code, 2024. Accessed: 2024-10-14.

[3] Android Developers. Capture macrobenchmark metrics. https://developer.android.com/topic/performance/benchmarking/macrobenchmark-metrics, 2024. Accessed: 2024-10-19.

[4] StatCounter. Market share stats. https://gs.statcounter.com/os-market-share/mobile/brazil, 2024. Accessed: 2024-08-30.

[5] Statista. Worldwide software developer working hours. https://www.statista.com/statistics/869224/worldwide-software-developer-working-hours, 2024. Accessed: 2024-8-30.

[6] Airbnb Engineering. Sunsetting react native. https://medium.com/airbnb-engineering/sunsetting-react-native-1868ba28e30a, 2018. Accessed: 2024-08-31.

[7] NearCoast. Headspace's leap to flutter: A game changer in mobile app development. https://www.nearcoast.com/headspaces-leap-to-flutter-a-game-changer-in-mobile-app-development/, 2023. Accessed: 2024-08-31.

[8] Rahul C. P. Raj and Seshubabu Tolety. A study on approaches to build cross-platform mobile applications and criteria to select appropriate approach. *2012 Annual IEEE India Conference (INDICON)*, pages 625–629, 2012.

[9] Paulo Meirelles, Carla Rocha, Felipe Assis, Rodrigo Siqueira, and Alfredo Goldman. A students' perspective of native and cross-platform approaches for mobile application development, 06 2019.

[10] Markus Olsson. A comparison of performance and looks between flutter and native applications: When to prefer flutter over native in mobile application development. Bachelor's thesis, Blekinge Institute of Technology, https://urn.kb.se/resolve?urn=urn:nbn:se:bth-19712, 2020. Accessed: 2024-10-07.

[11] Henrik Andersson. A comparison of the performance of an android application developed in native and cross-platform: Using the native android sdk and flutter. Bachelor's thesis, Blekinge Institute of Technology, https://urn.kb.se/resolve?urn=urn:nbn:se:bth-23080, 2022. Accessed: 2024-10-07.

[12] Kotlin Team. Kotlin programming language. https://kotlinlang.org/, 2024. Accessed: 2024-09-02.

[13] Android Developers. Build better apps with kotlin. https://developer.android.com/kotlin/build-better-apps, 2024. Accessed: 2024-09-02.

[14] Android Developers. Your first kotlin android app. https://developer.android.com/kotlin/first?hl=pt-br, 2024. Accessed: 2024-09-02.

[15] Android Developers. Declaring layout in android. https://developer.android.com/develop/ui/views/layout/declaring-layout, 2024. Accessed: 2024-09-07.

[16] Android Developers. Layouts basics in jetpack compose. https://developer.android.com/develop/ui/compose/layouts/basics, 2024. Accessed: 2024-09-07.

[17] Wahlandt, L., Brännholm, A. A comparative analysis of jetpack compose and xml views. Bachelor's thesis, University of Skövde, 2024.

[18] Android Developers. Rendering performance vitals. https://developer.android.com/topic/performance/vitals/render, 2024. Accessed: 2024-09-12.

[19] Flutter Developers. Flutter architectural overview. https://docs.flutter.dev/resources/architectural-overview, 2024. Accessed: 2024-09-12.

[20] OpenAI. Openai chatgpt. https://openai.com/chatgpt/, 2024. Accessed: 2024-09-30.

[21] Android Developers. Profiling in android studio. https://developer.android.com/studio/profile, 2024. Accessed: 2024-09-15.

[22] Flutter Developers. Flutter ui performance. https://docs.flutter.dev/perf/ui-performance, 2024. Accessed: 2024-09-15.

[23] Android Developers. Macrobenchmarking overview. https://developer.android.com/topic/performance/benchmarking/macrobenchmark-overview, 2024. Accessed: 2024-09-20.

[24] Flutter Developers. Profiling flutter with integration tests. https://docs.flutter.dev/cookbook/testing/integration/profiling, 2024. Accessed: 2024-09-20.

[25] Android Developers. Android studio. https://developer.android.com/studio, 2024. Accessed: 2024-09-20.

[26] Think with Google. Mobile page speed: New industry benchmarks. https://www.thinkwithgoogle.com/marketing-strategies/app-and-mobile/mobile-page-speed-new-industry-benchmarks/, 2024. Accessed: 2024-09-24.

[27] Flutter Developers. Flutter devtools. https://docs.flutter.dev/tools/devtools, 2024. Accessed: 2024-10-01.