Aplicação da Arquitetura Clean no Desenvolvimento de Aplicações Móveis com Flutter

João Antônio da Silva Pereira Júnior; Elaine Barbosa de Figueiredo

Aplicação da Arquitetura Clean no Desenvolvimento de Aplicações Móveis com Flutter

Resumo (ou Sumário Executivo)

A crescente complexidade dos sistemas de software e a necessidade de maior organização estrutural motivaram a busca por arquiteturas que promovessem manutenibilidade, escalabilidade e independência de tecnologias. Nesse contexto, a Arquitetura Clean destacou-se por propor uma separação clara entre as camadas do sistema, facilitando o desenvolvimento sustentável e a testabilidade do código. Este trabalho teve como objetivo aplicar os princípios da Arquitetura Clean, em conjunto com o padrão Model-View-ViewModel (MVVM), no desenvolvimento de uma aplicação móvel do tipo *to-do list* (lista de tarefas), avaliando seus benefícios e desafios na prática. A metodologia adotada incluiu pesquisa bibliográfica e o desenvolvimento de um protótipo funcional seguindo esses paradigmas arquiteturais. A coleta de dados deu-se por meio de experimentos de desempenho no aplicativo construído, simulando diferentes cenários de carga e estratégias de construção de interface. Os resultados indicaram que a utilização da Arquitetura Clean e do padrão MVVM contribuiu significativamente para a modularização do código, reutilização de componentes e facilidade de testes, ao mesmo tempo em que foram identificadas técnicas para mitigar problemas de desempenho (como o uso de *isolates* e listas construídas sob demanda). Concluiu-se que, apesar do maior esforço inicial de planejamento e aprendizagem, a adoção conjunta da Arquitetura Clean e do padrão MVVM representa uma estratégia eficaz para projetos móveis que almejam alta qualidade, facilidade de manutenção e boa performance ao longo do ciclo de vida do software.

Palavras-chave: Clean Architecture; organização de código; engenharia de software; desenvolvimento móvel; boas práticas.

Introdução

A evolução acelerada da tecnologia e o crescimento exponencial do uso de dispositivos móveis têm impulsionado a necessidade de desenvolver aplicações que sejam, ao mesmo tempo, eficientes, escaláveis e de fácil manutenção. Nesse cenário, a organização do código e a separação das responsabilidades se tornam fundamentais para assegurar a qualidade e a sustentabilidade dos sistemas. A Clean Architecture, proposta por Robert C. Martin, apresenta uma abordagem que promove a separação de camadas, permitindo que a lógica de negócio permaneça independente das dependências externas, como interfaces de usuário e bancos de dados. Essa filosofia arquitetural tem se mostrado

especialmente relevante no contexto do desenvolvimento mobile, onde a complexidade do código e as constantes atualizações de plataformas exigem soluções robustas e adaptáveis.

O presente trabalho de conclusão de curso tem como objetivo analisar e demonstrar a aplicação dos princípios da Clean Architecture no desenvolvimento de um aplicativo todo app simples, que implementa operações CRUD (criação, leitura, atualização e exclusão) tanto utilizando um banco de dados local no dispositivo quanto integrando serviços em nuvem. A escolha desse tipo de aplicação se justifica pela sua abrangência e relevância prática, uma vez que aplicativos desse gênero são comuns no mercado e demandam uma estrutura de código que facilita futuras manutenções e expansões. Além disso, a integração entre armazenamento local e cloud evidencia os desafios e as vantagens de se adotar uma arquitetura modular, capaz de lidar com diferentes fontes de dados e oferecer uma experiência de uso consistente, independentemente do ambiente de execução.

A relevância deste estudo reside na necessidade de comprovar, por meio de uma aplicação prática, que a adoção de uma arquitetura limpa não apenas melhora a organização do código, mas também contribui para a redução de custos de manutenção e aumento da testabilidade dos sistemas. A proposta deste trabalho é, portanto, construir um protótipo funcional que sirva de estudo de caso para a validação dos benefícios teóricos defendidos pela Clean Architecture. Para tanto, o projeto contempla uma revisão bibliográfica aprofundada sobre os fundamentos da Clean Architecture e os princípios SOLID, bem como a implementação prática do aplicativo utilizando tecnologias de desenvolvimento mobile, com ênfase em frameworks que suportem a separação de responsabilidades.

Ao final do trabalho, espera-se demonstrar que a aplicação dos conceitos de Clean Architecture pode ser um diferencial competitivo no desenvolvimento de software, garantindo maior modularidade, facilidade de integração e manutenção, e, consequentemente, uma melhor experiência para o usuário final. Dessa forma, o estudo contribuirá para a disseminação de boas práticas na engenharia de software, servindo como referência para profissionais e pesquisadores interessados na criação de sistemas robustos e escaláveis.

Justificativa

A escolha do desenvolvimento de um aplicativo de lista de tarefas como estudo de caso justificou-se por sua abrangência e relevância prática. Aplicativos desse gênero são ubiquamente utilizados e demandam uma estrutura de código que suporte futuras manutenções e expansões, servindo, portanto, como um bom modelo para avaliar arquiteturas de software. Além disso, a integração entre armazenamento local e serviços em

nuvem no mesmo aplicativo evidencia a necessidade de uma arquitetura modular capaz de lidar com múltiplas fontes de dados, oferecendo uma experiência consistente independentemente do ambiente de execução.

A relevância deste estudo reside na oportunidade de demonstrar, por meio de uma aplicação concreta, os benefícios e eventuais desafios da adoção de uma arquitetura limpa em um cenário realista. Embora a literatura ressalte as vantagens teóricas da Arquitetura Clean – como independência de tecnologia, testabilidade e facilidade de mudança –, faz-se importante observar esses aspectos na prática, quantificando impactos em desempenho e verificando se o esforço adicional de planejamento arquitetural é recompensado pela qualidade do produto final. Em suma, ao comprovar a eficácia (ou limitações) da combinação Clean Architecture + MVVM em um aplicativo móvel simples porém completo, este trabalho pode oferecer orientações práticas a desenvolvedores e enriquecer a discussão sobre boas práticas de engenharia de software móvel.

Fundamentação Teórica

Arquitetura Clean: A Arquitetura Clean (ou Arquitetura Limpa) foi proposta por Robert C. Martin em 2012, reunindo princípios de design de software voltados a manter o sistema independente de frameworks, UI, banco de dados ou qualquer detalhe de infraestrutura (MARTIN, 2017). Em essência, essa arquitetura organiza o código em camadas concêntricas, definindo dependências unidirecionais: as camadas mais externas podem depender das mais internas, mas nunca o contrário. No núcleo ficam as **Entidades** (regras de negócio de nível mais alto, como modelos de dados fundamentais) e os **Casos de Uso** (regras de aplicação específicas, coordenando as entidades). Em volta, camadas de interface adaptadora e frameworks tratam dos detalhes de implementação, como interfaces gráficas, banco de dados ou APIs externas. Essa separação visa tornar a lógica de negócio do sistema agnóstica em relação a detalhes tecnológicos, promovendo independência e facilidade de mudança. Como consequência, obtém-se um sistema com componentes de baixo acoplamento e alta coesão, características que facilitam testes unitários e evoluções futuras no código sem que mudanças em uma parte afetem indevidamente outras partes. **Princípios SOLID:** Para alcançar essa estrutura limpa, são aplicados os princípios SOLID de design orientado a objetos, também difundidos por Martin (2002). Os cinco princípios SOLID são:

1. **Single Responsibility Principle (SRP):** cada classe deve ter uma única responsabilidade ou razão para mudar. No contexto deste projeto, por exemplo, a classe NoteDao ficou responsável apenas pelo acesso a dados (CRUD no banco local), enquanto o HomeViewModel dedica-se somente a orquestrar chamadas de casos de uso e expor o estado necessário para a UI, evitando misturar responsabilidades.
2. **Open/Closed Principle (OCP):** o software deve estar aberto para extensão, mas fechado para modificação. Isso significa que novos comportamentos devem ser acrescentados preferencialmente via novas classes ou métodos, e não alterando código existente. Na implementação realizada, foi possível adicionar um novo caso de uso (por exemplo, *GetNoteById*) sem modificar as classes já existentes de repositório, apenas estendendo o sistema com uma nova classe de caso de uso – evidenciando adesão ao OCP.
3. **Liskov Substitution Principle (LSP):** objetos de classes derivadas devem poder ser utilizados como objetos de suas classes base, sem quebrar a funcionalidade do sistema. Em nossa arquitetura, a interface NoteRepository define um contrato para operações de notas; a classe NoteRepositoryImpl (implementação concreta) pode ser substituída por outra implementação (por exemplo, uma versão fake ou de testes) sem que o restante do sistema seja afetado, desde que respeite a interface – garantindo substituibilidade conforme LSP.
4. **Interface Segregation Principle (ISP):** interfaces mais específicas são preferíveis a interfaces genéricas e muito abrangentes. Aplicando ISP, cada caso de uso no projeto (como AddNote, DeleteNote, etc.) foi implementado como uma classe ou interface separada com um único método (por exemplo, execute ou call), em vez de se definir uma interface única que contenha métodos para todas as operações de notas. Isso mantém os contratos de cada funcionalidade isolados e fáceis de implementar ou modificar individualmente.
5. **Dependency Inversion Principle (DIP):** módulos de alto nível não devem depender de módulos de baixo nível, mas ambos devem depender de abstrações. Em outras palavras, detalhes devem depender de regras de negócio, e não o contrário. No projeto, esse princípio manifestou-se ao

fazermos, por exemplo, o ViewModel depender de uma abstração de caso de uso (GetAllNotes), e não diretamente de um detalhe de banco de dados como o NoteDao. Da mesma forma, o repositório depende da interface NoteRepository (abstração) em vez de acoplar-se a uma classe concreta. Essa inversão de dependências garante que a lógica principal do sistema permaneça desacoplada de detalhes de implementação, possibilitando trocar componentes (por exemplo, a fonte de dados) sem impactar as regras de negócio.

Padrão MVVM: O padrão Model-View-ViewModel (MVVM) é uma variação dos padrões arquiteturais de separação de interface, derivada do Model-View-Presenter (MVP) e do Model-View-Controller (MVC). No MVVM, divide-se a aplicação em: **Model** (os dados ou lógica de negócio, que no caso deste projeto corresponde às entidades e casos de uso do domínio), **View** (a interface do usuário, isto é, as telas e widgets Flutter) e **ViewModel** (uma camada intermediária que contém a lógica de interface, gerenciando estado da UI e servindo de fonte de dados para a View). A ViewModel expõe propriedades e comandos para a View, frequentemente utilizando mecanismos reativos de data binding ou observables – no Flutter, por exemplo, utilizaram-se objetos ValueNotifier para notificar a interface sobre mudanças de estado. Com MVVM, a View não precisa saber nada sobre o Model diretamente; ela interage com a ViewModel, solicitando ações ou exibindo dados. Essa abordagem facilita testes (pode-se testar a ViewModel isoladamente, sem interface gráfica) e melhora a organização, pois evita “misturar” código de interface com lógica de processamento. Em aplicações Flutter, embora o framework não implemente nativamente *data binding* declarativo como em outras plataformas, é possível seguir MVVM combinando gerenciadores de estado (no caso, Provider/ValueNotifier) para conectar ViewModels às Widgets de UI.

Metodologia ou Material e Métodos

Materiais e Ambiente de Desenvolvimento: O desenvolvimento foi realizado utilizando os seguintes recursos de hardware e software:

- **Hardware:** MacBook M1, SSD de 250 GB e 8gb de RAM.

- **SO e Simulador:** macOS Sequoia 15.1.1, simulador iPhone 16 Pro Max com iOS 1.8.1
- **Flutter:** 3.29.1 gerenciado via FVM.
- **Editor:** Visual Studio Code.
- **Persistência:** Drift + SQLite, configurado em app_db.dart
- **Injeção e estado:** provider + ValueNotifier / ValueListenableBuilder.
- **Estrutura de Projeto:** Clean Architecture + MVVM.

Procedimentos de Desenvolvimento: O estudo de caso consistiu na implementação de um aplicativo móvel de lista de tarefas seguindo a arquitetura proposta. Inicialmente, foram definidas as funcionalidades básicas de CRUD (criação, leitura, atualização e exclusão de notas/tarefas) utilizando persistência local. Em paralelo, estruturou-se o código em camadas conforme a Arquitetura Clean: camada de **Dados** (englobando a definição da tabela de notas, o DAO para operações SQLite e os *mappers* entre modelos de dados e entidades de domínio), camada de **Domínio** (incluindo a entidade de nota e os casos de uso *AddNote*, *GetAllNotes*, *DeleteNote*, *UpdateNote*, além de interfaces de repositório para abstrair o acesso aos dados) e camada de **Apresentação** (composta pelos *ViewModels* correspondentes às telas e pelas *Views* em Flutter). Cada componente foi implementado visando respeitar os princípios SOLID e o padrão MVVM. Por exemplo, na camada de domínio definiu-se a interface NoteRepository e implementou-se NoteRepositoryImpl delegando ao DAO, aplicando DIP; também foram criadas classes separadas para cada caso de uso, seguindo SRP e ISP. Na camada de apresentação, desenvolveu-se a HomeViewModel para gerenciar a tela principal (lista de notas) e a NoteEditAddViewModel para gerenciar o formulário de criação/edição de notas, expondo estados através de propriedades observáveis (notes, isLoading, error, etc.) e métodos que acionam os casos de uso do domínio. As telas Flutter (widgets HomePage e NoteEditAddPage) foram implementadas de forma declarativa, sem lógica de negócio embutida, utilizando ValueListenableBuilder e Provider para observar a ViewModel e reagir às mudanças de estado. Concomitantemente, integrou-se um serviço de nuvem para sincronização dos dados (permitindo que as notas fossem armazenadas remotamente além do armazenamento local). Essa integração buscou simular um cenário real de aplicativo conectado, aumentando a complexidade do caso de uso e testando a adaptabilidade

da arquitetura – a camada de dados, por exemplo, pôde ser estendida para enviar e obter dados de uma API remota sem modificar a camada de domínio, apenas acrescentando implementações adicionais de repositório. **Configuração de Injeção de Dependências:** No arquivo principal (main.dart), utilizou-se um MultiProvider para injetar as dependências necessárias a cada camada de forma hierárquica. Foram registrados, na inicialização do aplicativo, instâncias únicas do banco de dados (AppDb) e dos repositórios (NoteRepositoryImpl ligado ao DAO), bem como instâncias dos casos de uso (via Provider ou ProxyProvider conectando repositório aos casos de uso). Em seguida, os ViewModels foram fornecidos já configurados com seus respectivos casos de uso. Essa configuração garante que, ao longo do ciclo de vida do app, os componentes de camada inferior sejam reutilizados e as camadas superiores recebam as dependências adequadas, mantendo o desacoplamento entre elas. **Procedimentos de Teste e Avaliação:** Para avaliar o desempenho e validar as vantagens da arquitetura, foram conduzidos experimentos focados em dois aspectos: (1) responsividade da interface frente a operações de processamento pesado, e (2) eficiência na renderização de listas extensas na UI. Criaram-se cenários de teste nos quais uma quantidade significativa de dados era processada e exibida, medindo-se empiricamente o comportamento do aplicativo. Em um dos cenários, simulou-se a geração e exibição de um grande conjunto de itens (por exemplo, 1.000 tarefas) utilizando diferentes abordagens de construção de lista no Flutter – incluindo a construção padrão (carregando todos os itens de uma vez) e a construção sob demanda com ListView.builder() – para comparar o tempo de construção de interface e o uso de recursos. Também foram testadas variantes utilizando ListView.separated() (para listas com separadores entre itens) e a delegação do trabalho de construção para *isolates* (threads separadas) do Dart. As medições de tempo de construção e o número de frames perdidos durante o scroll foram observados, exibindo-se os resultados na própria interface para análise comparativa. Além disso, avaliou-se a influência do tipo de widget na performance de atualização da UI: componentes Stateless (imutáveis) com uso de keyword const onde possível, versus componentes *Stateful* (com estado interno), em situações de atualização frequente da tela. Todos os testes foram realizados no simulador citado, em modo *debug* e *release* para identificar possíveis diferenças. Os resultados obtidos em cada experimento foram coletados e são apresentados a seguir,

seguidos da discussão associada, relacionando-os às boas práticas recomendadas e aos objetivos propostos.

Resultados e Discussão

Implementação da Arquitetura: Após a implementação, verificou-se que o aplicativo foi devidamente estruturado nas camadas planejadas, e cada componente cumpriu seu papel conforme a Arquitetura Clean. Na **Camada de Dados**, foi definida a tabela NoteTable (contendo campos como ID, título, conteúdo, data de criação, etc.) e implementado o NoteDao responsável por todas as operações CRUD no banco local através do pacote Drift. Para manter a independência entre camadas, mapeamentos foram criados (no arquivo *note_mapper.da*rt) para converter os objetos do banco (tabela/DAO) em entidades de domínio (Note) e vice-versa. Na **Camada de Domínio**, a entidade Note encapsula os dados de uma nota, provendo, por exemplo, um método fábrica Note.create() que gera identificadores únicos (UUID) e timestamp automaticamente, assegurando que cada nota nova tenha ID distinto e carimbo de criação. Foram estabelecidos os casos de uso AddNote, GetAllNotes, GetNoteById, UpdateNote e DeleteNote, cada qual implementado em uma classe separada com um método principal (por convenção chamado call() ou execute()) isolando a lógica específica da operação. Esses casos de uso consomem um repositório de notas através da interface NoteRepository – cuja implementação concreta (NoteRepositoryImpl) delega as operações à camada de dados (DAO) apropriada. Esse desenho evidencia a aplicação do DIP: a lógica de negócio lida apenas com a abstração do repositório, sem conhecimento de como os dados são obtidos ou armazenados (poderiam vir de SQLite, de uma API REST ou outro, sem alterar o caso de uso). Na **Camada de Apresentação**, adotou-se o padrão MVVM. Foram criadas duas ViewModels principais: a HomeViewModel para gerenciar os dados da tela inicial (listagem de notas) e a NoteEditAddViewModel para gerenciar o estado do formulário de criação/edição de notas. A HomeViewModel expõe, por meio de propriedades observáveis (ValueNotifier), a lista de notas (notes), um indicador de carregamento (isLoading) e possíveis mensagens de erro (error). Ela provê métodos como loadNotes() – que aciona o caso de uso GetAllNotes e atualiza os estados isLoading, notes e error conforme o resultado – e deleteNote(id) – que invoca o caso de uso DeleteNote e atualiza a lista local em seguida. De forma

semelhante, a NoteEditAddViewModel mantém controladores para os campos de título e conteúdo (usando TextEditingController do Flutter), indicadores de estado de salvamento (isSaving) e erro, além do método save() – o qual determina se deve chamar AddNote ou UpdateNote conforme o contexto (nova nota ou edição de nota existente). Essa ViewModel cuida de toda a lógica de validação e submissão do formulário, mantendo a interface de edição o mais simples possível. As **Views** (interfaces do usuário em Flutter) – representadas principalmente por HomePage e NoteEditAddPage – foram implementadas de maneira declarativa e reativa. Ambas utilizam o ValueListenableBuilder (fornecido pelo Flutter) para observar as propriedades das ViewModels (via Provider) e reconstruir partes da interface automaticamente quando há mudanças, conforme o padrão MVVM. Por exemplo, a HomePage exibe uma lista de notas por meio de um widget de lista que escuta o notes da HomeViewModel; se uma nota é adicionada ou removida, a lista é atualizada reativamente sem necessidade de lógica imperativa na interface. Toda a lógica de negócios (carregar notas, remover nota, tratar erros) foi delegada ao ViewModel, de modo que o código da UI se restringiu a construir os widgets e reagir às mudanças de estado. Implementou-se também um gerenciador de layout (ViewLayoutManager<TModel>) para suportar responsividade: esse componente avalia o tipo de dispositivo/tamanho de tela e decide qual implementação de página utilizar (por exemplo, poderia selecionar entre uma página otimizada para mobile ou outra para tablet/desktops), permitindo manter uma única árvore de widgets adaptável. Embora este aplicativo seja simples e focado em mobile, essa preparação para diferentes *layouts* reforça a escalabilidade da solução. Por fim, a **Injeção de Dependências** na inicialização garantiu o encadeamento correto entre as camadas: ao iniciar o app, instanciou-se o banco de dados local AppDb e o providenciou a NoteDao; em seguida, criou-se NoteRepositoryImpl ligando-o ao DAO (via ProxyProvider, uma funcionalidade do Provider para construir dependências com base em outras já fornecidas). Após isso, instâncias dos casos de uso (AddNote, GetAllNotes, etc.) foram disponibilizadas, recebendo o repositório como dependência. Por último, as ViewModels foram instanciadas; por exemplo, a HomeViewModel foi fornecida já configurada com o caso de uso GetAllNotes (obtido do Provider) e, no momento de sua criação, já invoca loadNotes() para preencher a lista inicial. Esse encadeamento, implementado declarativamente no main.dart,

assegura que cada componente do sistema receba suas dependências sem acoplamentos explícitos no código, e que apenas um local central (a configuração do Provider) saiba como compor os objetos – alinhando-se às boas práticas de inversão de controle. Em resumo, a fase de implementação resultou em um aplicativo totalmente funcional, estruturado segundo Clean Architecture e MVVM. A separação de camadas tornou possível, por exemplo, executar testes unitários isolados dos casos de uso (substituindo o repositório real por um simulado) e das ViewModels (simulando diferentes comportamentos dos casos de uso) para verificar a correta reação da interface a estados de sucesso e erro – algo inviável em arquiteturas monolíticas. Essa modularidade atingida já sugere um atendimento aos objetivos de qualidade e manutenibilidade. A seguir, são apresentados os resultados dos experimentos de desempenho realizados para avaliar a segunda parte dos objetivos: a eficiência e responsividade da aplicação sob diferentes abordagens técnicas.

Desempenho e Responsividade: Foram conduzidos testes específicos para analisar o comportamento do aplicativo em cenários de carga e para verificar o impacto de decisões de implementação na performance da UI. Os experimentos focaram em duas dimensões principais: o uso de *isolates* para processamento em segundo plano e as estratégias de construção de listas extensas, além de analisar o efeito de widgets *Stateless* vs *Stateful* na atualização de interface. *Performance sem isolates* – Nos testes iniciais, sem o uso de *isolates*, observou-se uma degradação perceptível da responsividade da interface quando o aplicativo executava operações computacionalmente intensivas no *thread* principal (UI). Por exemplo, ao processar um grande conjunto de dados diretamente na tela principal (simulando a renderização de 1.000 itens ou realizando um cálculo pesado antes de atualizar a UI), o tempo de resposta do aplicativo aumentou significativamente e foram registrados *frames* perdidos durante a tarefa.

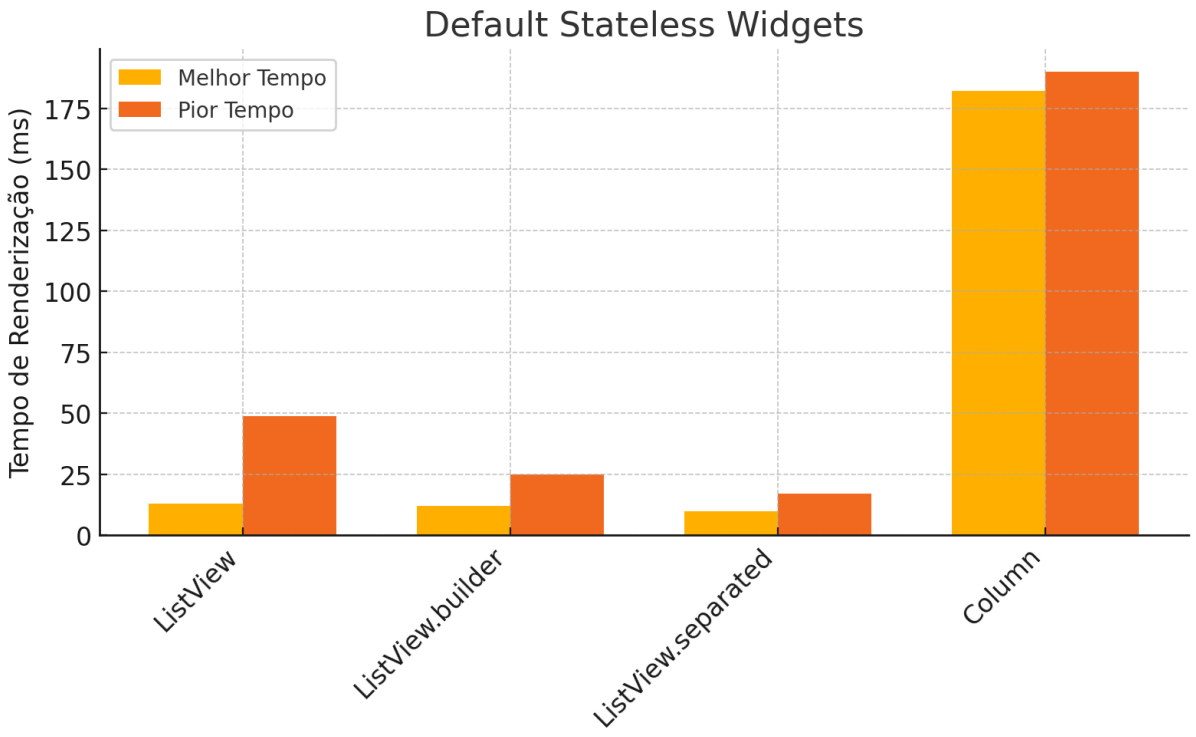

Figura 1. Comparação dos tempos mínimos (“Melhor Tempo”) e máximo (“Pior Tempo”) de renderização (em milissegundos) para diferentes widgets Stateless padrões.

Fonte: Dados originais da pesquisa (2025).

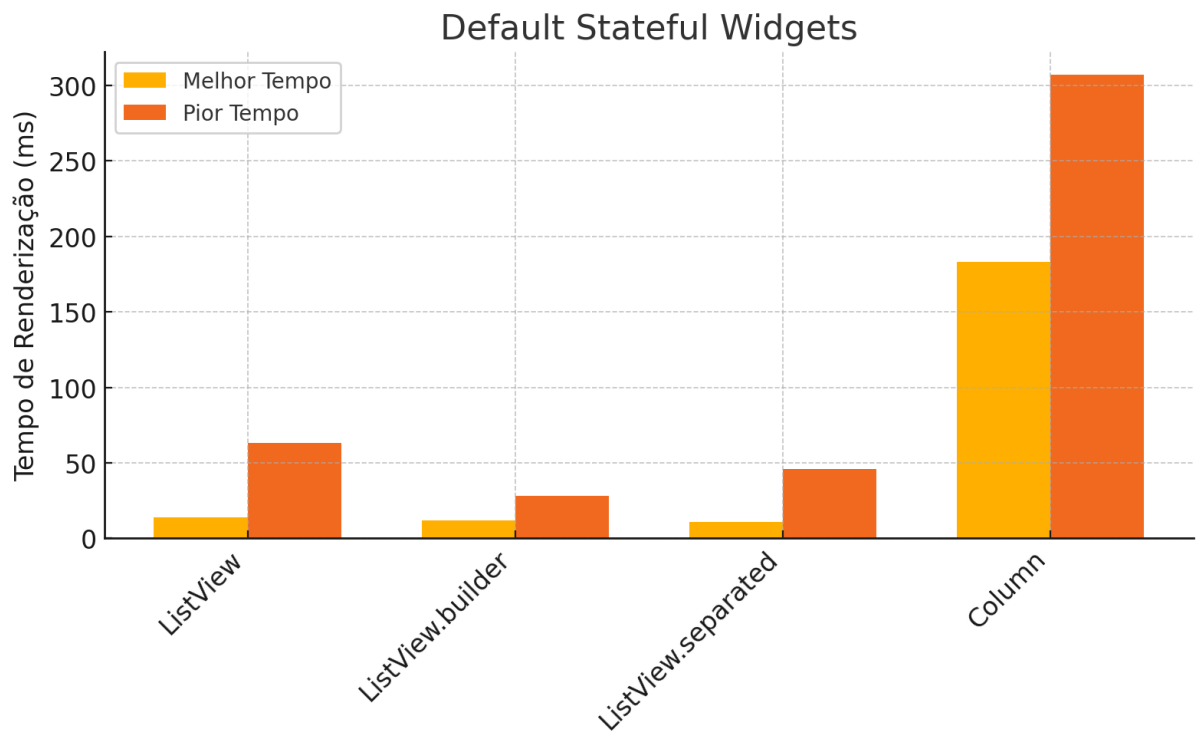

Figura 1. Comparação dos tempos mínimos (“Melhor Tempo”) e máximo (“Pior Tempo”) de renderização (em milissegundos) para diferentes widgets Stateful padrões.

Fonte: Dados originais da pesquisa (2025).

A interface apresentava pequenos travamentos ou lentidão, indicando que o loop principal de renderização estava sendo bloqueado por trabalho excessivo. Esse comportamento era esperado, pois, conforme descrito na documentação oficial do Flutter, operações muito extensas executadas no *isolate* principal podem causar engasgos na interface do usuário (FLUTTER, 2023a). Em outras palavras, a UI ficou momentaneamente congelada enquanto toda a carga era processada de forma síncrona. *Performance com isolates** – Introduzindo *isolates* (isolados de execução concorrente) para lidar com tarefas pesadas, foi possível mitigar esse impacto negativo. Reestruturou-se o código de forma que determinadas operações (como a conversão de um grande lote de objetos JSON em instâncias de notas, ou a preparação de uma lista extensa) fossem executadas em um isolate separado, usando a função utilitária Isolate.run() do Dart. Com essa mudança, a interface manteve-se fluida mesmo sob carga de processamento elevada, já que a lógica

pesada passou a rodar em paralelo, liberando o *thread* principal para continuar respondendo às interações do usuário.

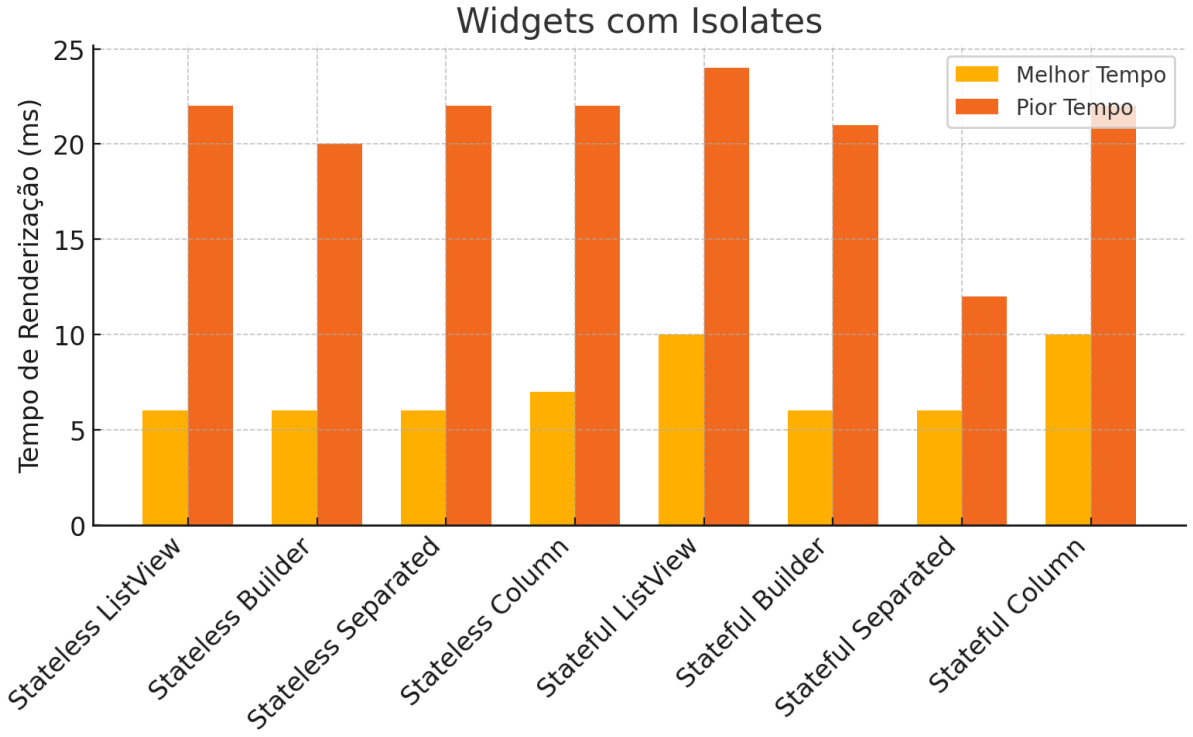

Figura 1. Comparação dos tempos mínimos (“Melhor Tempo”) e máximo (“Pior Tempo”) de renderização (em milissegundos) para diferentes widgets Flutter utilizando isolates.

Fonte: Dados originais da pesquisa (2025).

Nos testes, a diferença foi clara: operações que antes causavam congelamentos passaram a ocorrer em segundo plano, e a UI pôde exibir, por exemplo, um indicador de progresso enquanto aguardava o resultado, sem travar. Quando a computação terminava, o resultado era então retornado ao *isolate* principal e aplicado à interface (por exemplo, exibindo os dados processados). Esse resultado corrobora as recomendações da literatura e da própria documentação do Flutter, que incentivam delegar computações custosas a threads separadas para manter a responsividade (FLUTTER, 2023a). Portanto, constatou-se que o uso de *isolates* é uma solução eficaz para manter a experiência do usuário suave em cenários de

grande carga de processamento, atendendo às expectativas de desempenho sem exigir mudanças na arquitetura geral do aplicativo. *Listas extensas e construção sob demanda* – Outro conjunto de testes comparou diferentes abordagens de construção de listas com um grande número de elementos na interface Flutter. Inicialmente, utilizou-se o construtor padrão de listas (ListView estático), que cria todos os widgets de uma vez. Com uma quantidade muito grande de itens, esse método resultou em tempos de construção altos e consumo elevado de memória, já que todos os 1.000 elementos (no experimento) eram instanciados independentemente de estarem visíveis. O aplicativo apresentou lentidão no carregamento inicial da lista e uso intensivo de recursos. Em seguida, testou-se o uso de ListView.builder(), que cria os widgets sob demanda conforme a lista é rolada. Como esperado, o desempenho melhorou drasticamente: o tempo inicial de construção da interface reduziu-se, e a rolagem manteve-se suave, pois o *framework* gerava apenas os itens visíveis na tela a cada momento. O comportamento observ ado alinhou-se às boas práticas documentadas – o próprio Flutter recomenda o uso de listas construídas sob demanda (*builder*) para listas longas, em vez do construtor padrão, exatamente para evitar o custo de construir elementos não visíveis (FLUTTER, 2023b). Adicionalmente, avaliou-se o uso de ListView.separated(), que é similar ao builder mas insere separadores entre os itens; esse método não apresentou diferença significativa de performance em relação ao builder comum, mostrando-se igualmente eficiente para o propósito de listas extensas. Por fim, combinou-se a técnica de *lazy load* (construção sob demanda) com o uso de *isolates*: gerou-se a lista de 1.000 itens em um isolate em segundo plano, enquanto a UI principal exibia um *spinner* de carregamento. Uma vez gerada a coleção de dados, ela foi entregue à UI e exibida via ListView.builder(). Esse arranjo mostrou o menor tempo de bloqueio de interface dentre todos os cenários testados – a interface permaneceu responsiva durante toda a preparação dos dados, e a exibição dos itens ocorreu gradualmente, conforme necessário. Assim, comprovou-se que a combinação de *isolates* para preparar os dados e *builders* para renderizá-los é altamente eficaz para lidar com grandes volumes de informação em aplicativos Flutter, garantindo uma experiência fluida ao usuário mesmo em casos de uso estressantes. *Widgets Stateless vs Stateful* – Durante o desenvolvimento, notou-se a importância de escolher corretamente entre widgets *Stateless* e *Stateful* para cada componente da UI, pois isso pode influenciar

tanto a simplicidade do código quanto a performance. Nos testes de interface, comparou-se o comportamento de componentes implementados das duas formas em cenários de atualização frequente. Constatou-se que, para elementos majoritariamente estáticos ou que mudam em resposta a estados gerenciados externamente (por exemplo, valores fornecidos pelo ViewModel), os widgets Stateless combinados com o uso adequado de constantes (const) mostraram-se mais eficientes. Tais widgets evitam reconstruções desnecessárias, pois o Flutter consegue otimizar e reaproveitar instâncias constantes, resultando em menor carga de processamento para re-renderizar a interface. Por outro lado, em situações que exigem que um componente gerencie seu próprio estado interno e atualize partes da UI localmente (por exemplo, um contador animado dentro de um item de lista, ou um campo de texto que valida sua entrada em tempo real), o uso de *StatefulWidgets* foi benéfico e praticamente inevitável. Nesses casos, os widgets com estado permitiram atualizações pontuais de subárvores da interface sem notificar todo o aplicativo, focando a recomposição somente onde necessário. Em resumo, os experimentos indicaram que a escolha entre Stateless e Stateful deve considerar o comportamento dinâmico exigido: widgets sem estado, aliadas a técnicas de otimização (uso de const e listas sob demanda), foram suficientes e eficientes na maioria dos casos testados, enquanto widgets com estado próprio se justificaram apenas em cenários de interações locais mais complexas. Essa observação reforça uma das vantagens do padrão MVVM adotado: grande parte do estado da aplicação ficou centralizada nas ViewModels (fora da UI), permitindo que muitas das telas fossem construídas com widgets Stateless simples, que são mais leves. Somente componentes específicos (como formulários) utilizaram StatefulWidgets dentro de um escopo limitado. Assim, a arquitetura facilitou o uso de widgets imutáveis sempre que possível, contribuindo para a performance e para a previsibilidade das renderizações. **Discussão dos Resultados:** De modo geral, os resultados obtidos demonstram que a adoção dos princípios arquiteturais propostos trouxe benefícios concretos tanto para a qualidade do software quanto para seu desempenho, atendendo aos objetivos do trabalho. Em termos de **qualidade de código e manutenibilidade**, a Arquitetura Clean aliada ao MVVM resultou em um aplicativo altamente modular. Cada camada pôde ser desenvolvida e testada quase independentemente, e alterações em uma parte (por exemplo, trocar o banco de

dados SQLite por outra solução, ou modificar a interface visual) exigiriam mudanças mínimas nas outras camadas. Essa separação clara de concernes melhora a **facilidade de manutenção** e a **testabilidade** do sistema – de fato, pôde-se criar testes unitários para as classes de caso de uso e ViewModels sem necessidade de infraestrutura complexa, graças ao uso de interfaces e *mocks* para as dependências. Isso evidencia que a arquitetura atingiu o propósito de melhorar a organização interna do aplicativo, tornando-o mais robusto frente a evoluções. No que tange ao **desempenho**, que era uma preocupação secundária do estudo, os ensaios mostraram que é possível mitigar eventuais penalizações introduzidas pela camada extra de abstração. Inicialmente poderia supor-se que uma arquitetura mais elaborada (com mais classes, camadas e indireção) pudesse incorrer em overhead perceptível. Contudo, os resultados indicam que quaisquer custos adicionais foram desprezíveis diante dos ganhos estruturais – e onde houve risco de impacto (como nas situações de processamento pesado), estratégias conhecidas de otimização puderam ser empregadas sem a necessidade de alterar a arquitetura. O uso de *isolates* e de padrões eficientes de construção de UI garantiu que o aplicativo permanecesse **performático**, mesmo com a estrutura em camadas. Assim, demonstrou-se ser possível conciliar uma arquitetura de software limpa e bem estruturada com uma aplicação ágil em Flutter, não havendo conflito entre escrever um código organizado e atingir boa performance. Vale ressaltar que a adoção da Arquitetura Clean e do MVVM, como qualquer abordagem, trouxe também desafios: percebeu-se um **aumento no tempo de desenvolvimento inicial**, devido à necessidade de criar múltiplas classes e estabelecer as estruturas de comunicação entre camadas, em comparação a uma implementação direta e menos estratificada. Além disso, exigiu-se da equipe uma curva de aprendizado para compreender e aplicar corretamente os conceitos de injeção de dependências, separação de camadas e gerenciamento de estado reativo. Entretanto, esses esforços iniciais se pagaram na forma de um código final mais legível e confiável. Em aplicações de maior porte ou em equipes de desenvolvimento colaborativo, espera-se que tais benefícios se tornem ainda mais evidentes, reforçando a escalabilidade do projeto. Em suma, os experimentos e análises realizadas confirmaram que a utilização conjunta da Arquitetura Clean e do padrão MVVM no desenvolvimento de um aplicativo de lista de tarefas atendeu aos objetivos propostos. A abordagem

proporcionou um código organizado, de fácil manutenção e amplamente testável, ao mesmo tempo em que – com as devidas técnicas de otimização – entregou um aplicativo eficiente e responsivo. No próximo tópico, são apresentadas as considerações finais, sintetizando as conclusões do trabalho e propondo possíveis trabalhos futuros ou melhorias.

Considerações Finais

Concluiu-se que a aplicação dos princípios da Arquitetura Clean e do padrão MVVM no desenvolvimento do aplicativo de lista de tarefas cumpriu os objetivos propostos, resultando em um código modular, manutenível e alinhado às boas práticas de engenharia de software móvel. Os experimentos de performance permitiram avaliar de forma objetiva o impacto de diferentes abordagens na responsividade e eficiência da aplicação. Verificou-se que operações computacionais intensivas, quando executadas na thread principal, degradavam a experiência do usuário; contudo, esse impacto pôde ser mitigado eficazmente com o uso de isolates, mantendo a interface fluida mesmo sob carga de processamento. Também se observou que a escolha entre widgets Stateless e Stateful deve considerar o comportamento dinâmico da interface: widgets sem estado, combinados com técnicas como o uso de constantes e listas construídas sob demanda, mostraram-se eficientes na maioria dos casos, enquanto widgets com estado próprio foram vantajosos apenas em cenários de atualização frequente de componentes específicos. Em suma, o projeto demonstr**ou** ser possível conciliar uma arquitetura de software limpa e bem estruturada com uma aplicação performática em Flutter, atendendo aos requisitos de qualidade e desempenho estabelecidos no início do trabalho.

Naturalmente, este estudo concentrou-se em um caso de uso específico (um aplicativo de notas/to-do) e em um conjunto delimitado de tecnologias (Flutter, SQLite, Provider). Como trabalho futuro, sugere-se avaliar a aplicabilidade e os efeitos da Arquitetura Clean e MVVM em projetos de maior escala ou em outros domínios de aplicação, bem como investigar o uso de frameworks adicionais de gerenciamento de estado (como Bloc ou Redux) em conjunção com a arquitetura proposta. Adicionalmente, poderia ser aprofundada a análise quantitativa de performance, registrando métricas precisas de tempo de execução e uso de memória para cada cenário, a fim de complementar as observações qualitativas aqui

apresentadas. De todo modo, os resultados obtidos reforçam a ideia de que investir em uma boa arquitetura e seguir princípios sólidos de design é vantajoso mesmo em aplicações mobile, proporcionando um desenvolvimento mais confiável e preparando a base para evoluções e manutenção futura do software

Agradecimento

Agradeço à Elaine Barbosa de Figueiredo, minha orientadora, pelo suporte técnico e pelas valiosas orientações ao longo de todo o desenvolvimento deste trabalho. Sou grato também à minha família e amigos que me apoiaram nesta trajetória. Um agradecimento especial ao meu amigo Arlan, que sempre esteve ao meu lado como mentor e confidente.

Referências

FLUTTER. *Concurrency and isolates – Flutter Documentation*. 2023. Disponível em: https://docs.flutter.dev/perf/isolates. Acesso em: 27 abr. 2025.

FLUTTER. *Work with long lists – Flutter Documentation (Cookbook)*. 2023. Disponível em: https://docs.flutter.dev/cookbook/lists/long-lists. Acesso em: 27 abr. 2025.

MARTIN, Robert C. *Agile Software Development: Principles, Patterns, and Practices*. Upper Saddle River: Prentice Hall, 2002.

MARTIN, Robert C. *Clean Architecture: A Craftsman's Guide to Software Structure and Design*. Upper Saddle River: Prentice Hall, 2017.

SMITH, Josh. Patterns – WPF Apps With The Model-View-ViewModel Design Pattern. *MSDN Magazine*, fev. 2009. Disponível em: https://learn.microsoft.com/en-us/archive/msdn-magazine/2009/february/patterns-wpf-apps-with-the-model-view-viewmodel-design-pattern. Acesso em: 27 abr. 2025.

Apêndice A – Repositório do Projeto TODO

Dados originais de pesquisa.

Link do repositório: https://github.com/JotaJrJr/tcc_app