RCMOS – Revista Científica Multidisciplinar O
Saber. ISSN: 2675-9128. São Paulo - SP.
**Ano VII, v.1 2026 | submissão: 11/07/2026 | aceito: 08/07/2026 | publicação: 14/07/2026**

# Clean Architecture no Desenvolvimento de Software Moderno: Aliada ou Vilã?

*Clean Architecture in Modern Software Development: Ally or Villain?*
*Clean Architecture en el Desarrollo de Software Moderno: ¿Aliada o Villana?*

Wanderley Junior Alves Trindade
ORCID: 0009-0005-7962-272X
Lattes: http://lattes.cnpq.br/1193694031486153

**Resumo:** A crescente complexidade dos sistemas modernos tornou as decisões arquiteturais fundamentais para garantir a organização, a qualidade e a evolução do software. Nesse contexto, a Clean Architecture propõe a separação entre as regras de negócio e os detalhes tecnológicos, como frameworks, bancos de dados, interfaces e serviços externos. O objetivo geral deste trabalho foi analisar criticamente a aplicação da Clean Architecture no desenvolvimento de software moderno, considerando seus princípios, benefícios, limitações e impactos na qualidade, na manutenção e na evolução dos sistemas. Metodologicamente, realizou-se uma pesquisa bibliográfica, de abordagem qualitativa e caráter descritivo e exploratório, baseada em artigos científicos em língua portuguesa, publicados nos últimos dez anos e disponíveis gratuitamente em periódicos e bases acadêmicas. Os resultados demonstraram que a Clean Architecture pode favorecer a manutenibilidade, a testabilidade, a flexibilidade e a longevidade dos sistemas, principalmente em projetos complexos, duradouros e sujeitos a mudanças frequentes. Entretanto, sua implementação também pode aumentar o número de camadas, classes, interfaces e abstrações, elevando os custos e a complexidade inicial, especialmente em aplicações pequenas ou com regras simples. Concluiu-se que a Clean Architecture não pode ser considerada exclusivamente uma aliada ou uma vilã, pois sua efetividade depende das características do projeto, da experiência da equipe e da aplicação proporcional de seus princípios.
**Palavras-chave:** Arquitetura de software. Clean Architecture. Desenvolvimento de software.

**Abstract:** The increasing complexity of modern systems has made architectural decisions fundamental to ensuring software organization, quality, and evolution. In this context, Clean Architecture proposes a separation between business rules and technological details, such as frameworks, databases, interfaces, and external services. The overall objective of this study was to critically analyze the application of Clean Architecture in modern software development, considering its principles, benefits, limitations, and impacts on system quality, maintenance, and evolution. Methodologically, a literature review was conducted using a qualitative, descriptive, and exploratory approach, based on Portuguese-language scientific articles published within the last ten years and freely available in journals and academic databases. The results demonstrated that Clean Architecture can enhance system maintainability, testability, flexibility, and longevity, particularly in complex, long-lasting projects subject to frequent changes. However, its implementation can also increase the number of layers, classes, interfaces, and abstractions, thereby raising initial costs and complexity, especially in small applications or those with simple rules. It was concluded that Clean Architecture cannot be viewed solely as an ally or a villain, as its effectiveness depends on project characteristics, team experience, and the proportionate application of its principles.
**Keywords:** Software architecture. Clean Architecture. Software development.

**Resumen:** La creciente complejidad de los sistemas modernos ha hecho que las decisiones arquitectónicas sean fundamentales para garantizar la organización, la calidad y la evolución del software. En este contexto, la Clean Architecture propone separar las reglas de negocio de los detalles tecnológicos, como frameworks, bases de datos, interfaces y servicios externos. El

objetivo general de este trabajo fue analizar críticamente la aplicación de la Clean Architecture en el desarrollo de software moderno, considerando sus principios, beneficios, limitaciones e impactos en la calidad, el mantenimiento y la evolución de los sistemas. Metodológicamente, se realizó una investigación bibliográfica de enfoque cualitativo y carácter descriptivo y exploratorio, basada en artículos científicos en lengua portuguesa publicados en los últimos diez años y disponibles gratuitamente en revistas y bases académicas. Los resultados demostraron que la Clean Architecture puede favorecer la mantenibilidad, la capacidad de prueba, la flexibilidad y la longevidad de los sistemas, principalmente en proyectos complejos, duraderos y sujetos a cambios frecuentes. Sin embargo, su implementación también puede aumentar el número de capas, clases, interfaces y abstracciones, elevando los costes y la complejidad inicial, especialmente en aplicaciones pequeñas o con reglas simples. Se concluyó que la Clean Architecture no puede considerarse exclusivamente una aliada o una villana, pues su efectividad depende de las características del proyecto, de la experiencia del equipo y de la aplicación proporcional de sus principios.
**Palabras clave:** Arquitectura de software. Clean Architecture. Desarrollo de software.

## 1 Introdução

O desenvolvimento de software moderno tem sido marcado pelo aumento da complexidade dos sistemas, pela necessidade de atualizações frequentes e pela busca por aplicações capazes de acompanhar mudanças tecnológicas e organizacionais. Nesse contexto, a arquitetura de software exerce papel fundamental, pois define a forma como os componentes de um sistema são estruturados, organizados e relacionados. Uma arquitetura adequada pode facilitar a manutenção, a realização de testes, a substituição de tecnologias e a evolução do software, enquanto decisões arquiteturais inadequadas podem gerar dependências excessivas, dificuldades de compreensão e aumento dos custos de desenvolvimento.

Entre as abordagens utilizadas para organizar sistemas de software, destaca-se a Clean Architecture, ou Arquitetura Limpa, proposta com a finalidade de separar as regras de negócio dos elementos externos da aplicação, como bancos de dados, interfaces gráficas, frameworks e serviços de terceiros. Sua estrutura busca garantir que as partes mais importantes do sistema não dependam diretamente de tecnologias específicas, permitindo que alterações externas sejam realizadas com menor impacto sobre a lógica central do software. Essa proposta está relacionada a princípios como separação de responsabilidades, inversão de dependência, baixo acoplamento e alta coesão.

A aplicação da Clean Architecture tem sido associada a benefícios como maior manutenibilidade, testabilidade, flexibilidade, escalabilidade e facilidade de evolução dos sistemas. Ao organizar o código em camadas independentes, essa arquitetura pode contribuir para que as equipes compreendam melhor as responsabilidades de cada componente e realizem alterações sem comprometer outras partes da aplicação. Além disso, a independência em

relação a frameworks e bancos de dados pode favorecer a substituição de tecnologias e a reutilização das regras de negócio em diferentes contextos.

Apesar desses benefícios, a adoção da Clean Architecture também pode apresentar limitações. Em sistemas pequenos, projetos de curta duração ou aplicações com regras de negócio pouco complexas, a criação de diversas camadas, interfaces, abstrações e estruturas adicionais pode aumentar o volume de código e dificultar o desenvolvimento. A aplicação rígida de seus princípios também pode provocar complexidade desnecessária, elevar o tempo de implementação e exigir maior conhecimento técnico da equipe. Dessa forma, sua utilização não deve ser considerada automaticamente adequada para todos os projetos, sendo necessário analisar o contexto, os requisitos e as características do sistema.

Diante dessa dualidade, estabelece-se a seguinte questão-problema: em que medida a aplicação da Clean Architecture contribui para a qualidade e a evolução dos softwares modernos e em quais situações sua adoção pode gerar complexidade, custos e dificuldades desnecessárias ao processo de desenvolvimento? A partir dessa questão, o estudo busca compreender se a Clean Architecture deve ser considerada uma aliada do desenvolvimento moderno ou se, quando empregada de maneira inadequada, pode se transformar em um fator prejudicial para os projetos.

A realização deste trabalho justifica-se pela necessidade de compreender criticamente as consequências da adoção de modelos arquiteturais no desenvolvimento de software. Embora a Clean Architecture seja frequentemente apresentada como uma solução para problemas relacionados à manutenção e à organização dos sistemas, sua implementação exige planejamento, experiência e análise das necessidades reais do projeto. Assim, discutir suas vantagens e limitações pode auxiliar estudantes, desenvolvedores, arquitetos de software e organizações na tomada de decisões mais adequadas, evitando tanto a construção de sistemas excessivamente acoplados quanto o uso de estruturas complexas sem uma necessidade concreta.

O objetivo geral deste trabalho consiste em analisar criticamente a aplicação da Clean Architecture no desenvolvimento de software moderno, considerando seus princípios, benefícios, limitações e impactos na qualidade, na manutenção e na evolução dos sistemas. Para alcançar esse objetivo, foram definidos como objetivos específicos: apresentar os principais conceitos, princípios e camadas que compõem a Clean Architecture, destacando sua relação com a separação de responsabilidades e a independência entre os componentes do software; identificar os benefícios proporcionados pela utilização da Clean Architecture, especialmente em relação à manutenibilidade, testabilidade, escalabilidade, flexibilidade e longevidade dos

sistemas; e investigar as limitações e os desafios associados à sua implementação, avaliando em quais contextos sua adoção pode gerar complexidade excessiva, aumento de custos e dificuldades durante o processo de desenvolvimento.

Quanto aos procedimentos metodológicos, o estudo foi desenvolvido por meio de uma pesquisa bibliográfica, de abordagem qualitativa e caráter descritivo e exploratório. O levantamento teórico considerou artigos científicos publicados em língua portuguesa nos últimos dez anos, disponibilizados gratuitamente em bases acadêmicas e periódicos de acesso aberto, incluindo materiais localizados no Google Acadêmico. Foram selecionadas produções relacionadas à Clean Architecture, aos princípios SOLID, à inversão de dependência, à modularidade, à manutenibilidade, à testabilidade, à dívida técnica e às arquiteturas monolítica e de microsserviços. Após a seleção, os materiais foram analisados e comparados, buscando-se identificar os principais argumentos favoráveis e contrários à adoção da Clean Architecture, bem como os contextos em que sua utilização pode contribuir ou dificultar o desenvolvimento de software.

## 2 Desenvolvimento

### 2.1 Arquitetura de Software e o Desenvolvimento de Sistemas Modernos

A arquitetura de software corresponde à estrutura que organiza os componentes de um sistema, define suas responsabilidades e estabelece as formas de comunicação entre eles. Ela envolve decisões sobre tecnologias, padrões, módulos e integrações, influenciando diretamente o funcionamento, a manutenção e a capacidade de evolução da aplicação ao longo de seu ciclo de vida (Ciriano; Pinto, 2025).

Nos sistemas modernos, as decisões arquiteturais tornaram-se mais importantes devido ao aumento da complexidade. As aplicações precisam atender diferentes usuários, integrar serviços externos, processar informações e funcionar em variadas plataformas. Uma estrutura pouco organizada pode dificultar a inclusão de novas funcionalidades e aumentar os riscos de falhas durante as alterações.

A qualidade do software também depende de fatores como manutenibilidade, confiabilidade, flexibilidade e testabilidade. Mesmo atendendo aos requisitos iniciais, uma aplicação pode tornar-se difícil de modificar quando apresenta dependências excessivas e responsabilidades mal distribuídas. A arquitetura contribui para reduzir esses problemas ao estabelecer limites entre os componentes (Escareli; Malara, 2022).

A manutenção envolve correções, melhorias, atualizações e adaptações às novas necessidades. Quando não existe uma separação adequada de responsabilidades, pequenas alterações podem afetar diferentes módulos. Por isso, o planejamento arquitetural deve considerar não apenas a entrega inicial, mas também a evolução futura do sistema.

As mudanças frequentes em frameworks, bibliotecas e bancos de dados também influenciam a arquitetura. Quando as regras principais estão diretamente ligadas a essas tecnologias, sua substituição pode exigir modificações extensas. Componentes reutilizáveis e mecanismos de integração bem definidos reduzem essas dependências e favorecem sistemas mais flexíveis (Gomes; Gibertoni, 2022).

A separação entre regras de negócio e detalhes tecnológicos é fundamental. As regras expressam os processos centrais da aplicação, enquanto interfaces, servidores e mecanismos de persistência funcionam como recursos de apoio. Misturar essas dimensões faz com que mudanças externas afetem diretamente o funcionamento principal do software.

O acoplamento entre os componentes também deve ser controlado. Quando os módulos possuem dependências intensas, alterações isoladas podem exigir mudanças em várias partes do sistema. A inversão de dependência reduz esse problema ao orientar os componentes centrais a dependerem de abstrações, e não de implementações específicas (Gonçalves; Mendes, 2022).

Além do baixo acoplamento, a alta coesão favorece a organização do código. Um componente coeso reúne funções relacionadas e possui responsabilidade bem definida. Isso facilita a localização das regras, a compreensão do sistema e a realização de alterações com maior segurança.

A arquitetura também influencia a testabilidade. Componentes independentes podem ser verificados isoladamente, diminuindo a necessidade de executar toda a aplicação. Testes unitários, de integração e de interface ajudam a identificar falhas antecipadamente e aumentam a confiança durante as modificações (Lima; Leal; Mesquita, 2023).

Entretanto, os testes não corrigem problemas provocados por uma arquitetura inadequada. Quando os módulos permanecem fortemente dependentes, sua validação pode exigir muitas configurações e recursos externos. Assim, a qualidade dos testes depende também da forma como as responsabilidades e dependências foram organizadas.

A falta de planejamento pode contribuir para o surgimento de dívida técnica. Atalhos, duplicações, dependências inadequadas e ausência de documentação aceleram entregas no curto prazo, mas tornam as mudanças futuras mais caras e complexas. Em sistemas legados, o acúmulo dessas decisões pode limitar a evolução da aplicação (Serafim; Duarte; Duarte, 2022).

Por outro lado, uma arquitetura excessivamente elaborada também pode causar dificuldades. Sistemas simples podem não exigir muitas camadas, interfaces ou abstrações. Nesses casos, a estrutura pode aumentar o volume de código, prolongar o desenvolvimento e dificultar a compreensão pela equipe.

Arquiteturas monolíticas e de microsserviços apresentam vantagens e limitações. O monólito reúne as funcionalidades em uma única aplicação, enquanto os microsserviços distribuem responsabilidades entre serviços independentes. A escolha deve considerar a complexidade do negócio, o tamanho da equipe, a escalabilidade e a capacidade de administrar a infraestrutura (Pegrucci; Tiosso; Reis, 2020).

Nenhuma abordagem elimina todos os problemas. Uma arquitetura distribuída pode favorecer a escalabilidade, mas criar dificuldades de comunicação e integração. Da mesma forma, muitas camadas podem melhorar a separação de responsabilidades, porém aumentar o esforço inicial. Por isso, as decisões devem considerar os benefícios e os custos de cada alternativa.

Nesse contexto, a Clean Architecture busca organizar as dependências e proteger as regras centrais do sistema. Ao manter frameworks, bancos de dados e interfaces nas partes externas, procura facilitar testes, manutenção e substituições tecnológicas. Contudo, a criação de limites e abstrações também pode aumentar a complexidade inicial (Donato; Zuchi, 2021).

Portanto, a arquitetura deve ser utilizada como um meio para atender às necessidades do projeto. A escolha precisa considerar o contexto, os recursos disponíveis, a experiência da equipe e a expectativa de evolução. Uma estrutura adequada favorece a qualidade e a manutenção, enquanto uma solução desproporcional pode gerar custos e dificuldades desnecessárias.

## 2.2 Origem e Fundamentos da Clean Architecture

A Clean Architecture surgiu como uma proposta para reduzir as dependências entre as regras de negócio e os elementos tecnológicos da aplicação. Sua estrutura estabelece limites entre as responsabilidades do sistema, protegendo o núcleo contra mudanças em frameworks, bancos de dados, interfaces e serviços externos. Com isso, busca facilitar a manutenção e a adaptação do software ao longo do tempo (Ferreira; Ferreira; Grande, 2022).

Sua origem está relacionada a modelos como arquitetura em camadas, arquitetura hexagonal e Onion Architecture. Apesar das diferenças entre essas abordagens, todas defendem

que as regras centrais não devem depender diretamente dos detalhes responsáveis pela apresentação, persistência ou comunicação com sistemas externos.

Na arquitetura em camadas, as responsabilidades costumam ser distribuídas entre apresentação, aplicação, domínio e infraestrutura. Essa divisão facilita a identificação das funções de cada componente, desde que as dependências sejam planejadas corretamente (Ciriano; Pinto, 2025).

Entretanto, apenas dividir o software em camadas não garante a independência do núcleo. Quando as regras de negócio continuam dependentes de recursos específicos da infraestrutura, mudanças tecnológicas podem afetar diretamente a lógica da aplicação.

A arquitetura hexagonal, ou Ports and Adapters, propõe isolar o núcleo dos mecanismos de entrada e saída. As portas definem contratos de comunicação, enquanto os adaptadores conectam o sistema a interfaces, bancos de dados e serviços externos (Silva; Costa, 2024).

A Onion Architecture também organiza o sistema em círculos concêntricos. Entidades e regras de domínio permanecem no centro, enquanto serviços, interfaces e infraestrutura ficam nas camadas externas. As dependências devem sempre ser direcionadas para o interior.

A Clean Architecture reúne elementos dessas propostas e determina que as camadas internas não conheçam detalhes externos. Assim, entidades e casos de uso permanecem independentes do banco de dados, do framework e da interface utilizada pelo usuário (Donato; Zuchi, 2021).

Essa proteção ocorre porque as regras de negócio representam a parte mais importante e estável do sistema. Ferramentas e tecnologias podem ser substituídas, enquanto o domínio expressa os processos e comportamentos que justificam a existência da aplicação.

A inversão de dependência reforça esse fundamento. Em vez de o núcleo depender diretamente de classes concretas da infraestrutura, ele utiliza abstrações e contratos. As camadas externas passam a implementar as necessidades definidas pelas partes internas, reduzindo o acoplamento (Gonçalves; Mendes, 2022).

A inversão de dependência não se resume ao uso de ferramentas de injeção. Trata-se de reorganizar a relação entre os módulos, fazendo com que os componentes mais importantes não dependam daqueles sujeitos a mudanças frequentes.

A testabilidade também constitui um fundamento da Clean Architecture. Quando as regras de negócio estão afastadas de bancos de dados, interfaces e serviços externos, podem ser verificadas isoladamente. Dependências reais podem ser substituídas por implementações simuladas, tornando os testes mais rápidos e direcionados (Lima; Leal; Mesquita, 2023).

Essa facilidade aumenta a segurança durante as alterações, pois os testes ajudam a confirmar se os comportamentos anteriores foram preservados. Contudo, uma estrutura testável precisa ser planejada, já que separar posteriormente componentes fortemente acoplados pode exigir grande esforço.

A independência de frameworks também é relevante. Essas ferramentas devem ser tratadas como recursos utilizados pela aplicação, e não como elementos responsáveis por controlar sua lógica central. Manter configurações específicas nas camadas externas reduz os impactos de futuras substituições tecnológicas (Ferreira; Ferreira; Grande, 2022).

Isso não significa rejeitar frameworks e bibliotecas. Esses recursos aceleram o desenvolvimento e oferecem soluções úteis. O objetivo é impedir que suas características determinem completamente a estrutura das regras de negócio.

O mesmo princípio aplica-se aos bancos de dados. O acesso às informações deve ocorrer por contratos, evitando que as entidades sejam modeladas exclusivamente conforme tabelas, documentos ou recursos de uma tecnologia específica de persistência (Gomes; Gibertoni, 2022).

Essa separação pode reduzir os impactos da troca do mecanismo de armazenamento ou da integração com novas fontes de dados. Entretanto, não garante substituições automáticas, pois cada tecnologia possui características próprias.

A independência da interface permite que os mesmos casos de uso sejam acessados por aplicações web, sistemas móveis, interfaces de linha de comando ou integrações externas. Dessa forma, a lógica central pode ser reutilizada sem duplicação entre diferentes plataformas (Nascimento; Sotto, 2020).

Apesar disso, a Clean Architecture não deve ser entendida como um modelo rígido de pastas ou nomes. Sua aplicação pode variar conforme a linguagem, o tamanho do sistema e a experiência da equipe. O essencial é preservar a direção das dependências e separar as regras de negócio dos detalhes tecnológicos.

A adoção excessivamente formal pode gerar interfaces, conversores e classes sem utilidade concreta. Esses elementos devem ser criados apenas quando ajudam a organizar responsabilidades, reduzir dependências ou facilitar testes, evitando complexidade desnecessária (Donato; Zuchi, 2021).

Portanto, a Clean Architecture procura proteger o núcleo, inverter as dependências e manter as tecnologias nas camadas externas. Esses fundamentos podem favorecer manutenção, testabilidade e adaptação, desde que sejam aplicados de forma proporcional às necessidades do projeto.

RCMOS – Revista Científica Multidisciplinar O
Saber. ISSN: 2675-9128. São Paulo - SP.
**Ano VII, v.1 2026 | submissão: 11/07/2026 | aceito: 08/07/2026 | publicação: 14/07/2026**

### 2.3 Princípios e Organização das Camadas da Clean Architecture

A Clean Architecture organiza o sistema em camadas com responsabilidades distintas, mantendo as regras de negócio protegidas dos detalhes tecnológicos. Em sua representação tradicional, o núcleo concentra os elementos mais importantes, enquanto interfaces, bancos de dados, frameworks e serviços externos permanecem nas camadas periféricas. Essa estrutura busca reduzir o impacto de mudanças tecnológicas sobre a lógica central da aplicação (Ferreira; Ferreira; Grande, 2022).

A implementação não exige uma quantidade fixa de camadas. Sua organização pode variar conforme a linguagem, a complexidade do projeto e as necessidades da equipe. O aspecto principal consiste em separar corretamente as responsabilidades e garantir que os componentes internos não dependam diretamente das tecnologias externas.

As entidades ocupam a camada mais interna e representam as regras gerais e estáveis do domínio. Elas podem reunir dados, comportamentos e restrições relacionados ao funcionamento do sistema, permanecendo independentes de bancos de dados, interfaces gráficas e bibliotecas específicas (Donato; Zuchi, 2021).

Essas entidades não devem funcionar apenas como estruturas de armazenamento. Elas precisam expressar conceitos relevantes e concentrar comportamentos do domínio. Quando as regras são espalhadas por controladores e serviços externos, o sistema perde coesão e torna-se mais difícil de compreender.

A camada de casos de uso representa as ações oferecidas pela aplicação. Esses componentes coordenam o fluxo de informações, utilizam as entidades e executam regras específicas. Dessa forma, descrevem o que o sistema realiza sem depender diretamente da forma de apresentação ou armazenamento dos dados (Silva; Costa, 2024).

Cada caso de uso deve possuir uma responsabilidade bem definida. Ao receber uma solicitação, ele valida os dados, interage com as entidades e utiliza contratos para acessar recursos externos. Essa organização facilita os testes e permite que diferentes interfaces compartilhem as mesmas funcionalidades.

Os adaptadores de interface transformam os dados entre os formatos internos e externos. Nessa camada podem estar controladores, apresentadores, conversores, gateways e implementações de repositórios. Sua função é permitir a comunicação com o núcleo sem expor as regras de negócio às particularidades tecnológicas (Gomes; Gibertoni, 2022).

Os controladores encaminham as solicitações aos casos de uso, enquanto os apresentadores organizam as respostas. Esses componentes não devem concentrar regras de negócio, pois o acúmulo de responsabilidades enfraquece a separação proposta pela arquitetura.

Na camada externa encontram-se frameworks, bancos de dados, servidores, interfaces e serviços de terceiros. Esses elementos são considerados detalhes de implementação e devem fornecer recursos tecnológicos sem controlar as regras do domínio (Ciriano; Pinto, 2025).

As configurações de inicialização, conexão, rotas e gerenciamento de dependências também permanecem nessa camada. Como esses recursos mudam com maior frequência, mantê-los afastados do núcleo reduz a propagação dos impactos tecnológicos pelo sistema.

A regra de dependência determina que as relações do código apontem para as camadas internas. Entidades não conhecem casos de uso, e casos de uso não acessam diretamente a infraestrutura. As camadas externas implementam os contratos definidos pelo núcleo, preservando a independência das regras de negócio (Gonçalves; Mendes, 2022).

Quando um caso de uso necessita armazenar informações, por exemplo, ele utiliza uma interface em vez de acessar diretamente uma classe de banco de dados. A implementação concreta permanece na infraestrutura e é fornecida durante a execução, fazendo com que o núcleo conheça apenas o contrato.

Esse processo está relacionado à inversão de dependência. O núcleo define aquilo de que necessita, enquanto as camadas externas adaptam-se a essas exigências. A injeção de dependência pode ser utilizada para fornecer implementações reais ou simuladas, favorecendo a substituição de componentes durante os testes (Lima; Leal; Mesquita, 2023).

A inversão de dependência é um princípio arquitetural, enquanto a injeção corresponde a uma técnica. O simples uso de uma ferramenta de injeção não garante baixo acoplamento, principalmente quando os módulos internos continuam conhecendo classes concretas da infraestrutura.

Os princípios SOLID também contribuem para essa organização. Eles orientam a definição de responsabilidades, a criação de contratos específicos, a possibilidade de extensão e a dependência de abstrações. Quando aplicados com equilíbrio, favorecem componentes mais coesos e fáceis de testar (Ferreira; Ferreira; Grande, 2022).

Entretanto, esses princípios não devem provocar fragmentação excessiva. Criar interfaces e classes para operações simples pode aumentar o número de arquivos sem gerar independência relevante. Cada abstração deve possuir uma finalidade concreta, como facilitar testes, permitir substituições ou isolar tecnologias.

O fluxo de dados atravessa as camadas por meio de contratos definidos. Uma solicitação parte da interface, passa pelo controlador e chega ao caso de uso, que executa as regras e interage com entidades. A resposta percorre o caminho inverso, sendo adaptada ao formato esperado pelo usuário (Donato; Zuchi, 2021).

Objetos próprios podem ser utilizados para transportar dados entre as camadas e evitar que as entidades dependam de frameworks. Contudo, o excesso de modelos e conversores pode gerar código repetitivo. A decisão deve equilibrar independência e simplicidade.

O baixo acoplamento e a alta coesão são resultados esperados dessa organização. O primeiro limita o conhecimento entre componentes, enquanto a segunda concentra atividades relacionadas. Isso reduz os efeitos inesperados das mudanças e facilita a localização das regras no sistema (Escareli; Malara, 2022).

A divisão em camadas, porém, não garante automaticamente uma boa arquitetura. Um projeto pode possuir muitas pastas e interfaces e ainda apresentar regras duplicadas ou responsabilidades inadequadas. A qualidade depende da definição e da preservação dos limites durante o desenvolvimento.

A Clean Architecture pode ser empregada tanto em aplicações monolíticas quanto em microsserviços. No monólito, organiza internamente os módulos; nos microsserviços, pode estruturar cada serviço de forma independente. Portanto, essas abordagens atuam em níveis diferentes e não são necessariamente concorrentes (Kanizawa; Pinto, 2022).

Em sistemas distribuídos, entretanto, continuam existindo dificuldades como falhas de rede, consistência de dados e monitoramento. A Clean Architecture melhora a organização interna dos serviços, mas não elimina a complexidade operacional desse modelo.

A separação das camadas também favorece a testabilidade. Casos de uso e entidades podem ser verificados sem executar bancos de dados ou interfaces reais, utilizando implementações simuladas. Ainda assim, testes de integração permanecem necessários para validar a comunicação entre os componentes (Romanini; Sotto, 2019).

Portanto, a Clean Architecture busca proteger as regras de negócio e controlar as dependências tecnológicas por meio de entidades, casos de uso, adaptadores e componentes externos. Quando aplicada de forma proporcional, pode melhorar a clareza, a manutenção e os testes. Contudo, deve evitar divisões e abstrações que não apresentem utilidade concreta para o projeto.

## 2.4 Clean Architecture como Aliada do Desenvolvimento de Software

A Clean Architecture pode atuar como aliada do desenvolvimento de software ao organizar o código e reduzir as dependências entre os componentes. Ao separar as regras de negócio de frameworks, bancos de dados, interfaces e serviços externos, protege o núcleo da aplicação contra mudanças tecnológicas e facilita sua manutenção e evolução (Donato; Zuchi, 2021).

A definição clara das responsabilidades também favorece a compreensão do sistema. Entidades concentram regras do domínio, casos de uso coordenam funcionalidades, adaptadores convertem dados e componentes externos fornecem recursos tecnológicos. Essa divisão reduz a dispersão das regras e facilita a identificação dos pontos que precisam ser alterados.

A manutenibilidade é uma das principais vantagens da abordagem. Em sistemas pouco organizados, classes extensas, responsabilidades misturadas e dependências excessivas fazem com que pequenas mudanças afetem diferentes módulos. A separação em camadas limita esses impactos e mantém as alterações próximas aos componentes envolvidos (Escareli; Malara, 2022).

Essa organização também facilita a inclusão de funcionalidades, a adaptação a novas regras e a atualização de tecnologias. Ao considerar a evolução desde o início, a equipe reduz o esforço necessário para compreender o código e realizar modificações futuras.

A independência em relação aos frameworks contribui para preservar as regras de negócio quando uma tecnologia precisa ser atualizada ou substituída. Embora a troca ainda exija esforço, manter os detalhes tecnológicos nas camadas externas reduz a quantidade de componentes afetados (Ferreira; Ferreira; Grande, 2022).

A testabilidade constitui outro benefício relevante. Quando entidades e casos de uso não dependem diretamente de bancos de dados, redes ou interfaces, seus comportamentos podem ser verificados isoladamente com dependências simuladas. Isso favorece testes mais rápidos e facilita a identificação de falhas.

A combinação entre testes unitários, de integração e de interface amplia a qualidade do sistema. Entretanto, a existência de camadas não garante, por si só, uma aplicação testável. Os componentes precisam possuir responsabilidades bem definidas e respeitar a direção das dependências (Lima; Leal; Mesquita, 2023).

A flexibilidade tecnológica também pode ser favorecida pelo uso de abstrações. Bancos de dados, serviços externos e outros recursos podem ser substituídos com menor impacto sobre o núcleo, desde que o acesso ocorra por meio de contratos bem definidos.

Essa característica é útil em sistemas utilizados por diferentes plataformas. Aplicações web, móveis e serviços de integração podem compartilhar os mesmos casos de uso, reduzindo duplicações e garantindo maior consistência das regras de negócio (Gonçalves; Mendes, 2022).

A reutilização também ocorre quando funcionalidades comuns permanecem concentradas no núcleo. Em vez de repetir validações e processos em diferentes interfaces, os casos de uso fornecem um ponto único de execução. Contudo, os componentes não devem ser transformados em estruturas excessivamente genéricas apenas para aumentar o reaproveitamento.

Em sistemas maiores, os limites entre as camadas podem facilitar a divisão do trabalho entre equipes. Grupos distintos podem atuar no domínio, nas interfaces ou na infraestrutura com menor interferência entre suas atividades. Uma estrutura previsível também favorece revisões de código e a entrada de novos integrantes (Ciriano; Pinto, 2025).

A Clean Architecture pode ser aplicada tanto em monólitos quanto em microsserviços. Em sistemas monolíticos, organiza internamente os módulos; nos microsserviços, separa as regras de cada serviço dos mecanismos de comunicação e persistência. Entretanto, não resolve problemas próprios de sistemas distribuídos, como falhas de rede e consistência de dados.

A separação das responsabilidades também pode reduzir a dívida técnica. Quando regras, interfaces e detalhes tecnológicos estão misturados, cada alteração tende a aumentar a complexidade. Limites bem definidos facilitam refatorações e impedem que soluções temporárias se espalhem pelo código (Serafim; Duarte; Duarte, 2022).

Apesar disso, a arquitetura não elimina problemas causados por falta de testes, documentação ou planejamento. Seus benefícios dependem de práticas consistentes e do respeito aos limites definidos durante o desenvolvimento.

A longevidade do software também pode ser favorecida. Sistemas corporativos permanecem em uso por muitos anos e atravessam mudanças de tecnologias. Quando as regras centrais estão isoladas, interfaces, ferramentas e mecanismos de armazenamento podem ser adaptados sem exigir a reescrita completa da aplicação (Silva; Saito, 2021).

Os princípios SOLID reforçam essa organização ao favorecer responsabilidade única, segregação de interfaces e inversão de dependência. Entretanto, interfaces e abstrações devem ser utilizadas apenas quando solucionam problemas concretos, evitando o aumento desnecessário do número de classes e arquivos.

A previsibilidade da estrutura melhora ainda a comunicação entre os desenvolvedores. Quando a equipe compreende a função de cada camada, torna-se mais fácil definir onde incluir

novas regras, quais componentes podem acessar serviços externos e quais dependências são permitidas (Silva; Costa, 2024).

Assim, a Clean Architecture atua como aliada principalmente em projetos com regras complexas, vida útil prolongada, múltiplas integrações e necessidade frequente de evolução. Nesses contextos, o esforço inicial pode ser compensado pela maior facilidade de manutenção, teste e adaptação tecnológica.

Em síntese, a abordagem pode favorecer manutenibilidade, testabilidade, flexibilidade, reutilização e longevidade. Contudo, seus benefícios dependem de uma aplicação proporcional às características do projeto. Quando utilizada de forma rígida ou sem necessidade concreta, pode comprometer a simplicidade do desenvolvimento (Donato; Zuchi, 2021).

## 2.5 Limitações e Desafios da Implementação da Clean Architecture

Embora a Clean Architecture favoreça a organização, a manutenção e a testabilidade, sua implementação também apresenta limitações. A criação de camadas, contratos, interfaces e adaptadores pode ajudar a controlar dependências em sistemas complexos, mas, quando aplicada sem necessidade, aumenta o volume de código e torna o desenvolvimento mais trabalhoso (Donato; Zuchi, 2021).

Um dos principais desafios é a complexidade inicial. A equipe precisa definir antecipadamente quais regras pertencem ao domínio, quais operações serão tratadas como casos de uso e quais recursos permanecerão nas camadas externas. Esse planejamento exige mais tempo do que estruturas baseadas diretamente nos recursos oferecidos pelos frameworks.

A divisão de uma funcionalidade entre entidades, controladores, casos de uso, repositórios e modelos de dados pode dificultar a compreensão do fluxo. Embora cada componente tenha uma responsabilidade específica, a necessidade de consultar muitos arquivos pode tornar a leitura menos direta, especialmente para desenvolvedores pouco familiarizados com a abordagem (Ferreira; Ferreira; Grande, 2022).

A curva de aprendizagem também representa um obstáculo. Conceitos como inversão de dependência, domínio, abstrações e adaptadores precisam ser compreendidos corretamente. Caso contrário, o projeto pode apenas reproduzir uma divisão de pastas, mantendo as regras de negócio dependentes de frameworks e recursos de infraestrutura.

Essa implementação superficial aumenta a quantidade de componentes sem alcançar a independência esperada. Por isso, a adoção da Clean Architecture exige capacitação,

documentação e revisões de código, além de maior conhecimento técnico por parte dos desenvolvedores (Gonçalves; Mendes, 2022).

Os custos iniciais também podem aumentar, sobretudo em equipes inexperientes. Treinamentos, acompanhamento e decisões equivocadas sobre os limites entre as camadas podem prolongar o desenvolvimento e exigir refatorações posteriores. Seus benefícios costumam aparecer principalmente durante a manutenção e a evolução do sistema.

Em projetos com prazos reduzidos, a criação de contratos e abstrações pode atrasar a entrega de funcionalidades simples. A equipe precisa equilibrar a velocidade de desenvolvimento com a necessidade de evitar uma estrutura difícil de manter no futuro (Ciriano; Pinto, 2025).

Outro risco é a criação de abstrações desnecessárias. Interfaces e adaptadores são úteis quando protegem o núcleo de dependências relevantes, facilitam testes ou permitem substituições. Contudo, criar uma interface para cada classe pode aumentar a complexidade sem oferecer benefícios proporcionais.

As abstrações devem resolver problemas concretos. Quando uma implementação é simples, estável e possui pouca possibilidade de substituição, o acesso direto pode ser mais adequado. O uso excessivo de intermediários dificulta a leitura do código e aumenta o esforço de manutenção (Escareli; Malara, 2022).

A duplicação de modelos também pode elevar o volume de código. Uma mesma informação pode ser representada por entidades, modelos de persistência, objetos de entrada e estruturas de resposta. Apesar de preservar a independência das camadas, essa separação exige conversões constantes entre formatos.

Em sistemas com muitos campos e operações simples, qualquer alteração pode exigir mudanças em vários modelos e conversores. A equipe deve avaliar se o isolamento obtido compensa o esforço ou se determinadas estruturas podem ser compartilhadas sem comprometer as regras de negócio (Silva; Costa, 2024).

Em projetos pequenos, protótipos ou aplicações de curta duração, a adoção integral da Clean Architecture pode ser desproporcional. A criação de diversas camadas pode aumentar o tempo de desenvolvimento sem oferecer ganhos relevantes de manutenção ou flexibilidade.

Nesses contextos, uma estrutura simplificada pode proporcionar maior agilidade. Caso o produto cresça, novos limites podem ser introduzidos gradualmente. Entretanto, essa estratégia deve evitar que soluções temporárias se acumulem e formem uma dívida técnica difícil de controlar (Donato; Zuchi, 2021).

RCMOS – Revista Científica Multidisciplinar O
Saber. ISSN: 2675-9128. São Paulo - SP.
**Ano VII, v.1 2026 | submissão: 11/07/2026 | aceito: 08/07/2026 | publicação: 14/07/2026**

O excesso de planejamento também pode levar à criação de estruturas destinadas a mudanças futuras que talvez nunca ocorram. Interfaces, adaptadores e pontos de extensão baseados apenas em possibilidades remotas aumentam os custos atuais e podem tornar o sistema mais difícil de compreender.

Por outro lado, a falta de planejamento favorece o alto acoplamento e a dívida técnica. O desafio está em preparar o sistema para mudanças prováveis sem criar componentes voltados apenas para cenários hipotéticos (Serafim; Duarte; Duarte, 2022).

A integração com frameworks representa outra dificuldade. Muitas ferramentas oferecem estruturas próprias para controladores, entidades e acesso a dados. O isolamento completo do núcleo pode exigir adaptadores adicionais e impedir o aproveitamento de algumas facilidades fornecidas pela tecnologia utilizada.

A independência tecnológica não significa rejeitar frameworks. Esses recursos podem acelerar o desenvolvimento e padronizar tarefas. Em determinados projetos, aceitar uma dependência limitada e conhecida pode ser mais adequado do que criar uma estrutura extensa apenas para uma possível substituição futura (Ferreira; Ferreira; Grande, 2022).

Os testes também exigem equilíbrio. Embora a Clean Architecture facilite testes isolados, o uso excessivo de objetos simulados pode produzir resultados distantes do comportamento real. Uma funcionalidade pode ser aprovada nos testes unitários e falhar durante a integração com bancos de dados, redes ou serviços externos.

Por isso, os testes unitários devem ser combinados com testes de integração e de interface. Além disso, alterações nos contratos podem exigir a atualização de muitos objetos simulados, aumentando o trabalho de manutenção dos próprios testes (Lima; Leal; Mesquita, 2023).

Em sistemas distribuídos, a Clean Architecture não elimina os problemas dos microsserviços. Comunicação pela rede, indisponibilidade, segurança, consistência de dados e monitoramento continuam sendo desafios. A combinação das duas abordagens pode elevar significativamente a quantidade de componentes administrados pela equipe.

Em equipes pequenas, essa complexidade pode superar os benefícios esperados. Quando não existem requisitos concretos de distribuição ou escalabilidade independente, um monólito organizado pode representar uma solução mais simples e eficiente (Kanizawa; Pinto, 2022).

A consistência arquitetural também precisa ser mantida ao longo do tempo. Prazos curtos, mudanças na equipe e entregas urgentes podem levar à inclusão de regras de negócio em controladores ou ao acesso direto da infraestrutura pelos casos de uso.

Para evitar essa degradação, são necessárias revisões de código, documentação, padrões internos e acompanhamento contínuo. A arquitetura não permanece organizada apenas pelas decisões tomadas no início do projeto (Escareli; Malara, 2022).

A aplicação rígida dos princípios pode ainda impedir soluções simples e adequadas. Interfaces e intermediários não devem ser criados apenas para reproduzir um modelo teórico. A arquitetura deve orientar decisões, mas permitir adaptações conforme as características reais do sistema.

O pragmatismo exige comparar os custos e os benefícios de cada separação. Uma dependência pode ser aceitável quando é restrita, conhecida e pouco sujeita a mudanças, enquanto uma abstração se justifica quando protege regras importantes ou favorece testes confiáveis (Gomes; Gibertoni, 2022).

Portanto, as limitações da Clean Architecture não tornam a abordagem inadequada, mas demonstram que sua adoção depende do contexto. Complexidade inicial, aumento do código, curva de aprendizagem, excesso de abstrações e custos de manutenção precisam ser considerados.

Em sistemas duradouros e com regras complexas, o investimento arquitetural pode ser compensado ao longo do tempo. Em aplicações simples, uma adoção parcial pode ser mais adequada. Assim, a Clean Architecture torna-se uma possível vilã quando é aplicada indiscriminadamente e sem relação com as necessidades concretas do projeto (Donato; Zuchi, 2021).

## 2.6 Adequação da Clean Architecture aos Diferentes Contextos de Desenvolvimento

A adoção da Clean Architecture deve considerar as características de cada projeto, como tamanho da aplicação, complexidade das regras, experiência da equipe e expectativa de evolução. Embora seus princípios favoreçam a separação de responsabilidades e o controle das dependências, sua aplicação integral nem sempre é necessária. A arquitetura deve ser proporcional às necessidades do software, evitando tanto a desorganização quanto a criação de estruturas excessivamente complexas (Donato; Zuchi, 2021).

Em sistemas com regras de negócio complexas e atualizações frequentes, a separação entre domínio, casos de uso, interfaces e infraestrutura pode facilitar a compreensão, os testes e a manutenção. Ao manter as regras centrais em componentes independentes, as alterações tendem a produzir menor impacto sobre as demais partes da aplicação.

Projetos corporativos de longa duração podem beneficiar-se dessa organização, pois geralmente passam por diferentes equipes, mudanças nas regras de negócio e substituições tecnológicas. A preservação do núcleo da aplicação reduz os impactos provocados pela troca de frameworks, bancos de dados e interfaces, contribuindo para a continuidade do sistema (Ferreira; Ferreira; Grande, 2022).

Entretanto, nem toda aplicação corporativa exige muitas camadas e abstrações. Sistemas administrativos com fluxos simples e poucas mudanças podem ser organizados por meio de estruturas menores. A escolha deve basear-se na complexidade das funcionalidades e nos custos de manutenção, e não apenas no porte da organização.

Em projetos pequenos, a aplicação integral da Clean Architecture pode ser desproporcional. A criação de interfaces, adaptadores e modelos para cada camada pode aumentar o esforço sem proporcionar ganhos significativos. Nesses casos, uma estrutura mais simples pode preservar a qualidade e facilitar o desenvolvimento (Escareli; Malara, 2022).

Isso não significa abandonar seus princípios. A separação entre regras de negócio e apresentação, a definição de responsabilidades e a redução do acoplamento podem ser aplicadas de maneira simplificada. A equipe pode adotar somente os elementos necessários, evitando uma estrutura maior do que o problema exige.

Startups e produtos em validação também demandam equilíbrio. Uma arquitetura extensa pode retardar a entrega da primeira versão, enquanto a ausência de organização pode dificultar o crescimento futuro. Assim, é possível iniciar com uma estrutura simples e introduzir novos limites conforme os requisitos se tornam mais claros (Serafim; Duarte; Duarte, 2022).

A experiência da equipe influencia diretamente os resultados. Desenvolvedores familiarizados com princípios SOLID e inversão de dependência tendem a aplicar a abordagem com maior segurança. Em equipes iniciantes, a falta de compreensão pode gerar componentes fragmentados e abstrações sem finalidade. Por isso, documentação, capacitação e revisões de código são fundamentais.

A Clean Architecture também pode ser aplicada em sistemas monolíticos. Uma aplicação implantada como uma única unidade pode possuir módulos bem definidos, casos de uso independentes e separação entre domínio e infraestrutura. Em muitos projetos, um monólito organizado é mais adequado do que a adoção imediata de microsserviços (Pegrucci; Tiosso; Reis, 2020).

Nos microsserviços, seus princípios podem organizar internamente cada serviço, protegendo as regras de negócio dos mecanismos de comunicação e persistência. Contudo, essa

combinação aumenta a quantidade de componentes e não elimina desafios como falhas de rede, consistência de dados, monitoramento e complexidade de implantação.

Em sistemas legados, a Clean Architecture pode orientar uma modernização gradual. A criação progressiva de limites, casos de uso e adaptadores permite isolar partes do código antigo e reduzir dependências sem exigir a reescrita completa da aplicação, diminuindo os riscos da transformação (Silva; Saito, 2021).

A integração com diferentes interfaces também pode justificar sua adoção. Aplicações web, móveis e serviços externos podem compartilhar os mesmos casos de uso quando a lógica central permanece independente da apresentação. Entretanto, em sistemas com uma única interface simples e estável, uma separação menos extensa pode ser suficiente.

Os requisitos de testabilidade são outro critério importante. Em sistemas nos quais falhas podem gerar prejuízos relevantes, o isolamento das regras facilita a realização de testes unitários e a substituição de dependências externas por implementações controladas. Essa característica contribui para validar os comportamentos centrais com maior segurança (Lima; Leal; Mesquita, 2023).

A frequência das mudanças tecnológicas também deve ser observada. Contratos e adaptadores são mais úteis quando há dependência de serviços externos, diferentes fontes de dados ou tecnologias sujeitas à substituição. Quando os recursos utilizados são estáveis, o custo de criar e manter abstrações pode superar os benefícios esperados.

O tempo e os recursos disponíveis igualmente influenciam a decisão. A Clean Architecture costuma exigir maior investimento inicial na definição das camadas, contratos e fluxos de dados. Em projetos duradouros, esse esforço pode ser compensado pela facilidade de manutenção, mas em sistemas temporários o retorno pode não ocorrer (Ciriano; Pinto, 2025).

Portanto, não existe uma resposta única para definir a Clean Architecture como aliada ou vilã. Ela pode facilitar o desenvolvimento de sistemas complexos e tornar desnecessariamente difícil a construção de aplicações simples. Sua adoção deve considerar a complexidade do domínio, o número de integrações, o crescimento esperado, a duração do projeto, a experiência da equipe e a necessidade de testes.

A abordagem também pode ser aplicada de forma parcial e progressiva. O sistema pode iniciar com a separação entre domínio e infraestrutura e incorporar novos casos de uso, adaptadores e abstrações conforme surgem necessidades concretas. Dessa forma, a arquitetura acompanha a evolução do software sem ser tratada como um modelo rígido (Donato; Zuchi, 2021).

Assim, a Clean Architecture atua como aliada quando protege regras relevantes, facilita testes, reduz dependências e favorece a evolução. Torna-se uma possível vilã quando é aplicada indiscriminadamente, aumentando o volume de código, os custos e a dificuldade de compreensão. Sua efetividade depende de uma utilização crítica, pragmática e coerente com as condições de cada projeto.

## 3 Conclusão

A realização deste trabalho permitiu analisar criticamente a aplicação da Clean Architecture no desenvolvimento de software moderno, considerando seus fundamentos, benefícios, limitações e diferentes contextos de utilização. A partir da revisão bibliográfica, foi possível compreender que essa abordagem busca proteger as regras de negócio das mudanças tecnológicas, organizando o sistema por meio da separação de responsabilidades e do controle das dependências entre os componentes.

Entre os principais benefícios identificados estão a melhoria da manutenibilidade, da testabilidade, da flexibilidade e da capacidade de evolução dos sistemas. Ao manter o núcleo da aplicação independente de frameworks, bancos de dados, interfaces e serviços externos, a Clean Architecture pode reduzir os impactos provocados por alterações tecnológicas e facilitar a realização de testes isolados. Além disso, a definição de limites claros entre entidades, casos de uso, adaptadores e infraestrutura contribui para uma organização mais compreensível do código.

Entretanto, também foram identificadas limitações importantes. A implementação da Clean Architecture pode aumentar a quantidade de classes, interfaces, contratos, conversores e arquivos, tornando o desenvolvimento inicialmente mais lento e complexo. Esse problema é mais evidente em projetos pequenos, protótipos, aplicações temporárias ou sistemas com regras de negócio simples, nos quais o esforço de criar uma estrutura extensa pode não ser compensado por benefícios futuros.

Outro desafio está relacionado à experiência da equipe. A adoção adequada da abordagem exige conhecimento sobre separação de responsabilidades, princípios SOLID, inversão de dependência e organização das camadas. Quando esses conceitos não são compreendidos, o projeto pode apresentar apenas uma divisão formal de pastas, sem alcançar a independência esperada. Nesse cenário, a Clean Architecture aumenta a complexidade do código sem melhorar efetivamente sua qualidade.

Dessa forma, a questão-problema foi respondida ao se constatar que a Clean Architecture contribui para a qualidade e a evolução do software quando é aplicada em sistemas que possuem regras de negócio complexas, múltiplas integrações, necessidade frequente de mudanças e expectativa de longa duração. Por outro lado, sua adoção pode gerar custos, dificuldades e abstrações desnecessárias quando ocorre de forma rígida, indiscriminada ou incompatível com as necessidades reais do projeto.

Também se verificou que a adoção da Clean Architecture não precisa ocorrer de maneira completa. Seus princípios podem ser aplicados de forma parcial e progressiva, de acordo com o crescimento e a complexidade do sistema. A separação entre domínio e infraestrutura, por exemplo, pode ser implementada inicialmente, enquanto novas camadas e abstrações são acrescentadas somente quando surgirem necessidades concretas de manutenção, testes ou substituição tecnológica.

Conclui-se, portanto, que a Clean Architecture não deve ser considerada exclusivamente uma aliada ou uma vilã do desenvolvimento de software moderno. Ela representa uma ferramenta arquitetural cuja efetividade depende do contexto, do planejamento e da forma como seus princípios são utilizados. Quando aplicada de maneira crítica, proporcional e pragmática, pode contribuir para a construção de sistemas mais organizados, testáveis e preparados para evoluir. Contudo, quando adotada apenas como tendência ou modelo obrigatório, pode comprometer a simplicidade, elevar os custos e dificultar o processo de desenvolvimento.

## 4 Referências

BATISTA, M. V.; PINTO, G. S. Arquitetura de microfrontend. Revista Interface Tecnológica, v. 20, n. 2, p. 208–218, 2023. Disponível em: https://revista.fatectq.edu.br/interfacetecnologica/article/view/1803.

CIRIANO, G.; PINTO, G. S. A evolução da arquitetura de software: do monólito aos microsserviços. Revista Interface Tecnológica, v. 22, n. 2, p. 122–133, 2025. Disponível em: https://revista.fatectq.edu.br/interfacetecnologica/article/view/2302.

DONATO, D. P.; ZUCHI, J. D. Arquitetura limpa: como minimizar o custo de um software. Revista Interface Tecnológica, v. 18, n. 2, p. 229–239, 2021. Disponível em: https://revista.fatectq.edu.br/interfacetecnologica/article/view/1288.

ESCARELI, G. P.; MALARA, R. D. Manutenibilidade de software: ferramentas que auxiliam e garantem a qualidade. RECIMA21 – Revista Científica Multidisciplinar, v. 3, n. 12, p. e3122450, 2022. Disponível em: https://recima21.com.br/index.php/recima21/article/view/2450.

FERREIRA, V. B. S.; FERREIRA, C. A.; GRANDE, E. T. G. Estado da arte da pesquisa em: Clean Architecture e princípios de SOLID. Research, Society and Development, v. 11, n. 16,

p. e335111637198, 2022. Disponível em: https://rsdjournal.org/rsd/article/download/37198/31702/418592.

GOMES, Y. L.; GIBERTONI, D. Web services: componentes reutilizáveis no processo de desenvolvimento orientado ao reuso. Revista Interface Tecnológica, v. 19, n. 2, p. 282–293, 2022. Disponível em: https://revista.fatectq.edu.br/interfacetecnologica/article/view/1531.

GONÇALVES, B. P.; MENDES, O. L. Princípio da inversão de dependência na qualidade de software: aplicação da injeção de dependência no desenvolvimento de software. Revista Interface Tecnológica, v. 19, n. 1, p. 34–46, 2022. Disponível em: https://revista.fatectq.edu.br/interfacetecnologica/article/view/1362.

KANIZAWA, D. T.; PINTO, G. S. Arquitetura de microsserviços. Revista Interface Tecnológica, v. 19, n. 2, p. 308–318, 2022. Disponível em: https://revista.fatectq.edu.br/interfacetecnologica/article/view/1538.

LIMA, D. P. C.; LEAL, E. D.; MESQUITA, L. J. S. Benefícios da utilização da pirâmide de teste em uma aplicação mobile. Revista Interface Tecnológica, v. 20, n. 1, p. 63–75, 2023. Disponível em: https://revista.fatectq.edu.br/interfacetecnologica/article/view/1607.

NASCIMENTO, C. H. P.; SOTTO, E. C. S. Microfrontend: um estudo sobre o conceito e aplicação no frontend. Revista Interface Tecnológica, v. 17, n. 1, p. 153–165, 2020. Disponível em: https://revista.fatectq.edu.br/interfacetecnologica/article/view/798.

PEGRUCCI, M.; TIOSSO, F.; REIS, H. M. Análise de um estudo de caso para comparar as arquiteturas monolítica e de microsserviços. Revista Interface Tecnológica, v. 17, n. 2, p. 325 –337, 2020. Disponível em: https://revista.fatectq.edu.br/interfacetecnologica/article/view/1029.

ROMANINI, I. Z.; SOTTO, E. C. S. Selenium Web Driver na evolução dos testes manuais. Revista Interface Tecnológica, v. 16, n. 2, p. 112–123, 2019. Disponível em: https://revista.fatectq.edu.br/interfacetecnologica/article/view/627.

SERAFIM, G. P.; DUARTE, A. M. D.; DUARTE, D. Análise quantitativa da aquisição de dívida técnica em sistemas legados. Revista de Sistemas e Computação, v. 12, n. 1, p. 10–19, 2022. Disponível em: https://revistas.unifacs.br/index.php/rsc/article/viewFile/7569/4507.

SILVA, R. A.; COSTA, J. M. Arquitetura modular e escalável para desenvolvimento de aplicativos móveis no Escritório de Soluções Criativas. JNT – Facit Business and Technology Journal, v. 1, n. 55, p. 780–797, 2024. Disponível em: https://revistas.faculdadefacit.edu.br/index.php/JNT/article/download/3051/2082.

SILVA, R. C.; SAITO, R. Aplicando arquitetura de microsserviços no desenvolvimento de software. Revista de Ubiquidade, v. 4, n. 2, p. 58–84, 2021. Disponível em: https://revistas.anchieta.br/index.php/RevistaUbiquidade/article/view/1859.