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 in Modern Software Development: Ally or Villain?

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

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.

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.

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 Introduction

Modern software development has been marked by increasing system complexity, the need for frequent updates, and the pursuit of applications capable of keeping pace with technological and organizational changes. In this context, software architecture plays a fundamental role because it defines how a system's components are structured, organized, and related. An appropriate architecture can facilitate maintenance, testing, technology replacement, and software evolution, whereas unsuitable architectural decisions can create excessive dependencies, comprehension difficulties, and increased development costs.

Among the approaches used to organize software systems, Clean Architecture stands out. It was proposed to separate business rules from external application elements such as databases, graphical interfaces, frameworks, and third-party services. Its structure seeks to ensure that the most important parts of the system do not depend directly on specific technologies, allowing external changes to be made with less impact on the software's core logic. This proposal is related to principles such as separation of concerns, dependency inversion, low coupling, and high cohesion.

The application of Clean Architecture has been associated with benefits such as greater maintainability, testability, flexibility, scalability, and ease of system evolution. By organizing code into independent layers, this architecture can help teams better understand the responsibilities of each component and make changes without compromising other parts of the application. In addition, independence from frameworks and databases can facilitate technology replacement and the reuse of business rules in different contexts.

Despite these benefits, adopting Clean Architecture may also present limitations. In small systems, short-term projects, or applications with relatively simple business rules, creating multiple layers, interfaces, abstractions, and additional structures can increase the amount of code and make development more difficult. Rigid application of its principles can also cause unnecessary complexity, increase implementation time, and require greater technical expertise from the team. Therefore, its use should not automatically be considered suitable for every project; the context, requirements, and characteristics of the system must be analyzed.

Given this duality, the following research question is established: to what extent does the application of Clean Architecture contribute to the quality and evolution of modern software, and in which situations can its adoption generate unnecessary complexity, costs, and difficulties in the development process? Based on this question, the study seeks to understand whether Clean Architecture should be considered an ally of modern development or whether, when used inappropriately, it can become a harmful factor for projects.

This study is justified by the need to critically understand the consequences of adopting architectural models in software development. Although Clean Architecture is frequently presented as a solution to problems related to system maintenance and organization, its implementation requires planning, experience, and an analysis of the project's actual needs. Thus, discussing its advantages and limitations can help students, developers, software architects, and organizations make more appropriate decisions, avoiding both the construction of excessively coupled systems and the use of complex structures without a concrete need.

The general objective of this study is 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. To achieve this objective, the following specific objectives were defined: to present the main concepts, principles, and layers that comprise Clean Architecture, highlighting its relationship with the separation of concerns and independence among software components; to identify the benefits provided by the use of Clean Architecture, especially regarding maintainability, testability, scalability, flexibility, and system longevity; and to investigate the limitations and challenges associated with its implementation, evaluating the contexts in which its adoption may generate excessive complexity, increased costs, and difficulties during the development process.

Regarding methodological procedures, the study was conducted through bibliographic research with a qualitative, descriptive, and exploratory approach. The theoretical survey considered scientific articles published in Portuguese over the last ten years and made freely available in academic databases and open-access journals, including materials found on Google

Scholar. Studies related to Clean Architecture, SOLID principles, dependency inversion, modularity, maintainability, testability, technical debt, and monolithic and microservices architectures were selected. After selection, the materials were analyzed and compared to identify the main arguments for and against adopting Clean Architecture, as well as the contexts in which its use can contribute to or hinder software development.

2 Development

2.1 Software Architecture and the Development of Modern Systems

Software architecture corresponds to the structure that organizes a system's components, defines their responsibilities, and establishes the forms of communication among them. It involves decisions about technologies, patterns, modules, and integrations, directly influencing application operation, maintenance, and capacity for evolution throughout its life cycle (Ciriano; Pinto, 2025).

In modern systems, architectural decisions have become more important because of increasing complexity. Applications must serve different users, integrate external services, process information, and operate on various platforms. A poorly organized structure can make it difficult to add new features and increase the risk of failures during changes.

Software quality also depends on factors such as maintainability, reliability, flexibility, and testability. Even when it meets initial requirements, an application can become difficult to modify when it has excessive dependencies and poorly distributed responsibilities. Architecture helps reduce these problems by establishing boundaries between components (Escareli; Malara, 2022).

Maintenance involves corrections, improvements, updates, and adaptations to new needs. When responsibilities are not properly separated, small changes can affect different modules. Architectural planning should therefore consider not only the initial delivery but also the system's future evolution.

Frequent changes in frameworks, libraries, and databases also influence architecture. When core rules are directly tied to these technologies, replacing them may require extensive modifications. Reusable components and well-defined integration mechanisms reduce these dependencies and promote more flexible systems (Gomes; Gibertoni, 2022).

The separation between business rules and technological details is fundamental. Rules express the application's core processes, while interfaces, servers, and persistence mechanisms

function as supporting resources. Mixing these dimensions causes external changes to directly affect the software's primary operation.

Coupling among components must also be controlled. When modules have strong dependencies, isolated changes may require modifications in several parts of the system. Dependency inversion reduces this problem by directing core components to depend on abstractions rather than specific implementations (Gonçalves; Mendes, 2022).

In addition to low coupling, high cohesion improves code organization. A cohesive component brings together related functions and has a well-defined responsibility. This makes it easier to locate rules, understand the system, and make changes more safely.

Architecture also influences testability. Independent components can be verified in isolation, reducing the need to run the entire application. Unit, integration, and interface tests help identify failures early and increase confidence during modifications (Lima; Leal; Mesquita, 2023).

However, tests do not correct problems caused by an unsuitable architecture. When modules remain strongly dependent, validating them may require many configurations and external resources. Thus, test quality also depends on how responsibilities and dependencies have been organized.

A lack of planning can contribute to technical debt. Shortcuts, duplication, unsuitable dependencies, and a lack of documentation accelerate deliveries in the short term but make future changes more expensive and complex. In legacy systems, the accumulation of these decisions can limit application evolution (Serafim; Duarte; Duarte, 2022).

Conversely, an excessively elaborate architecture can also cause difficulties. Simple systems may not require many layers, interfaces, or abstractions. In such cases, the structure can increase the amount of code, extend development time, and make the system harder for the team to understand.

Monolithic and microservices architectures have advantages and limitations. A monolith brings functionality together in a single application, whereas microservices distribute responsibilities among independent services. The choice should consider business complexity, team size, scalability, and the capacity to manage the infrastructure (Pegrucci; Tiosso; Reis, 2020).

No approach eliminates every problem. A distributed architecture may promote scalability but create communication and integration difficulties. Likewise, multiple layers may improve separation of concerns while increasing the initial effort. Decisions must therefore consider the benefits and costs of each alternative.

In this context, Clean Architecture seeks to organize dependencies and protect the system's core rules. By keeping frameworks, databases, and interfaces in the outer layers, it aims to facilitate testing, maintenance, and technology replacement. However, creating boundaries and abstractions can also increase initial complexity (Donato; Zuchi, 2021).

Architecture should therefore be used as a means of meeting project needs. The choice must consider the context, available resources, team experience, and expected evolution. An appropriate structure supports quality and maintenance, whereas a disproportionate solution may generate unnecessary costs and difficulties.

2.2 Origin and Foundations of Clean Architecture

Clean Architecture emerged as a proposal to reduce dependencies between business rules and an application's technological elements. Its structure establishes boundaries among system responsibilities, protecting the core from changes in frameworks, databases, interfaces, and external services. In this way, it seeks to facilitate software maintenance and adaptation over time (Ferreira; Ferreira; Grande, 2022).

Its origin is related to models such as layered architecture, hexagonal architecture, and Onion Architecture. Despite differences among these approaches, all of them maintain that core rules should not depend directly on details responsible for presentation, persistence, or communication with external systems.

In layered architecture, responsibilities are usually distributed among presentation, application, domain, and infrastructure. This division makes it easier to identify the functions of each component, provided that dependencies are planned correctly (Ciriano; Pinto, 2025).

However, merely dividing software into layers does not guarantee core independence. When business rules remain dependent on specific infrastructure resources, technological changes can directly affect application logic.

Hexagonal architecture, or Ports and Adapters, proposes isolating the core from input and output mechanisms. Ports define communication contracts, while adapters connect the system to interfaces, databases, and external services (Silva; Costa, 2024).

Onion Architecture also organizes the system into concentric circles. Entities and domain rules remain at the center, while services, interfaces, and infrastructure occupy the outer layers. Dependencies must always point inward.

Clean Architecture brings together elements of these proposals and determines that inner layers should not know external details. Thus, entities and use cases remain independent of the database, framework, and interface used by the user (Donato; Zuchi, 2021).

This protection exists because business rules represent the system's most important and stable part. Tools and technologies can be replaced, while the domain expresses the processes and behaviors that justify the application's existence.

Dependency inversion reinforces this foundation. Instead of having the core depend directly on concrete infrastructure classes, it uses abstractions and contracts. The outer layers then implement the needs defined by the inner parts, reducing coupling (Gonçalves; Mendes, 2022).

Dependency inversion is not limited to the use of injection tools. It involves reorganizing the relationship among modules so that the most important components do not depend on those subject to frequent changes.

Testability is also a foundation of Clean Architecture. When business rules are separated from databases, interfaces, and external services, they can be verified in isolation. Real dependencies can be replaced by simulated implementations, making tests faster and more targeted (Lima; Leal; Mesquita, 2023).

This ease increases safety during changes because tests help confirm that previous behaviors have been preserved. However, a testable structure must be planned, since later separating strongly coupled components may require considerable effort.

Independence from frameworks is also relevant. These tools should be treated as resources used by the application, not as elements responsible for controlling its core logic. Keeping specific configurations in the outer layers reduces the impacts of future technological replacements (Ferreira; Ferreira; Grande, 2022).

This does not mean rejecting frameworks and libraries. These resources accelerate development and offer useful solutions. The objective is to prevent their characteristics from completely determining the structure of business rules.

The same principle applies to databases. Information should be accessed through contracts, preventing entities from being modeled exclusively according to tables, documents, or resources of a specific persistence technology (Gomes; Gibertoni, 2022).

This separation can reduce the impacts of replacing the storage mechanism or integrating new data sources. However, it does not guarantee automatic replacements because each technology has its own characteristics.

Interface independence allows the same use cases to be accessed by web applications, mobile systems, command-line interfaces, or external integrations. In this way, core logic can be reused across different platforms without duplication (Nascimento; Sotto, 2020).

Nevertheless, Clean Architecture should not be understood as a rigid model of folders or names. Its application may vary according to the language, system size, and team experience. The essential point is to preserve the direction of dependencies and separate business rules from technological details.

An excessively formal adoption may generate interfaces, converters, and classes with no concrete usefulness. These elements should be created only when they help organize responsibilities, reduce dependencies, or facilitate testing, thereby avoiding unnecessary complexity (Donato; Zuchi, 2021).

Clean Architecture therefore seeks to protect the core, invert dependencies, and keep technologies in the outer layers. These foundations can support maintenance, testability, and adaptation, provided that they are applied in proportion to project needs.

2.3 Principles and Organization of Clean Architecture Layers

Clean Architecture organizes a system into layers with distinct responsibilities, keeping business rules protected from technological details. In its traditional representation, the core contains the most important elements, while interfaces, databases, frameworks, and external services remain in the peripheral layers. This structure seeks to reduce the impact of technological changes on the application's core logic (Ferreira; Ferreira; Grande, 2022).

Implementation does not require a fixed number of layers. Its organization may vary according to the language, project complexity, and team needs. The main point is to separate responsibilities correctly and ensure that internal components do not depend directly on external technologies.

Entities occupy the innermost layer and represent the domain's general and stable rules. They may bring together data, behaviors, and constraints related to system operation while remaining independent of databases, graphical interfaces, and specific libraries (Donato; Zuchi, 2021).

These entities should not function merely as storage structures. They must express relevant concepts and concentrate domain behaviors. When rules are scattered across controllers and external services, the system loses cohesion and becomes more difficult to understand.

The use-case layer represents the actions offered by the application. These components coordinate information flow, use entities, and execute specific rules. They therefore describe what the system does without depending directly on how data are presented or stored (Silva; Costa, 2024).

Each use case should have a well-defined responsibility. Upon receiving a request, it validates the data, interacts with entities, and uses contracts to access external resources. This organization facilitates testing and allows different interfaces to share the same functionality.

Interface adapters transform data between internal and external formats. This layer may include controllers, presenters, converters, gateways, and repository implementations. Its function is to enable communication with the core without exposing business rules to technological particularities (Gomes; Gibertoni, 2022).

Controllers forward requests to use cases, while presenters organize responses. These components should not concentrate business rules because accumulating responsibilities weakens the separation proposed by the architecture.

Frameworks, databases, servers, interfaces, and third-party services are found in the outer layer. These elements are considered implementation details and should provide technological resources without controlling domain rules (Ciriano; Pinto, 2025).

Startup, connection, routing, and dependency management configurations also remain in this layer. Because these resources change more frequently, keeping them away from the core reduces the spread of technological impacts throughout the system.

The dependency rule determines that code relationships point toward the inner layers. Entities do not know use cases, and use cases do not access infrastructure directly. The outer layers implement the contracts defined by the core, preserving the independence of business rules (Gonçalves; Mendes, 2022).

When a use case needs to store information, for example, it uses an interface instead of directly accessing a database class. The concrete implementation remains in the infrastructure and is supplied at runtime, so the core knows only the contract.

This process is related to dependency inversion. The core defines what it needs, while the outer layers adapt to these requirements. Dependency injection can be used to provide real or simulated implementations, facilitating component replacement during testing (Lima; Leal; Mesquita, 2023).

Dependency inversion is an architectural principle, whereas injection is a technique. Simply using an injection tool does not guarantee low coupling, especially when internal modules continue to know concrete infrastructure classes.

SOLID principles also contribute to this organization. They guide the definition of responsibilities, the creation of specific contracts, extensibility, and dependence on abstractions. When applied in a balanced manner, they promote more cohesive components that are easier to test (Ferreira; Ferreira; Grande, 2022).

However, these principles should not cause excessive fragmentation. Creating interfaces and classes for simple operations can increase the number of files without generating meaningful independence. Each abstraction should have a concrete purpose, such as facilitating testing, enabling replacements, or isolating technologies.

Data flow crosses the layers through defined contracts. A request begins at the interface, passes through the controller, and reaches the use case, which executes the rules and interacts with entities. The response follows the reverse path and is adapted to the format expected by the user (Donato; Zuchi, 2021).

Dedicated objects can be used to transport data between layers and prevent entities from depending on frameworks. However, too many models and converters can generate repetitive code. The decision must balance independence and simplicity.

Low coupling and high cohesion are expected results of this organization. The former limits knowledge among components, while the latter concentrates related activities. This reduces unexpected effects of changes and makes it easier to locate rules in the system (Escareli; Malara, 2022).

Dividing a system into layers, however, does not automatically guarantee good architecture. A project can have many folders and interfaces and still contain duplicated rules or unsuitable responsibilities. Quality depends on defining and preserving boundaries throughout development.

Clean Architecture can be used in both monolithic applications and microservices. In a monolith, it organizes modules internally; in microservices, it can structure each service independently. These approaches therefore operate at different levels and are not necessarily competing alternatives (Kanizawa; Pinto, 2022).

In distributed systems, however, difficulties such as network failures, data consistency, and monitoring remain. Clean Architecture improves the internal organization of services but does not eliminate the operational complexity of this model.

Layer separation also improves testability. Use cases and entities can be verified without running real databases or interfaces by using simulated implementations. Even so, integration tests remain necessary to validate communication among components (Romanini; Sotto, 2019).

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 therefore seeks to protect business rules and control technological dependencies through entities, use cases, adapters, and external components. When applied proportionately, it can improve clarity, maintenance, and testing. However, it should avoid divisions and abstractions that have no concrete usefulness for the project.

2.4 Clean Architecture as an Ally of Software Development

Clean Architecture can act as an ally of software development by organizing code and reducing dependencies among components. By separating business rules from frameworks, databases, interfaces, and external services, it protects the application core from technological changes and facilitates its maintenance and evolution (Donato; Zuchi, 2021).

Clearly defined responsibilities also improve understanding of the system. Entities concentrate domain rules, use cases coordinate functionality, adapters convert data, and external components provide technological resources. This division reduces the scattering of rules and makes it easier to identify the points that need to be changed.

Maintainability is one of the approach's main advantages. In poorly organized systems, large classes, mixed responsibilities, and excessive dependencies cause small changes to affect different modules. Layer separation limits these impacts and keeps modifications close to the components involved (Escareli; Malara, 2022).

This organization also facilitates the addition of features, adaptation to new rules, and technology updates. By considering evolution from the outset, the team reduces the effort required to understand the code and make future modifications.

Independence from frameworks helps preserve business rules when a technology must be updated or replaced. Although replacement still requires effort, keeping technological details in the outer layers reduces the number of affected components (Ferreira; Ferreira; Grande, 2022).

Testability is another relevant benefit. When entities and use cases do not depend directly on databases, networks, or interfaces, their behaviors can be verified in isolation with simulated dependencies. This supports faster tests and makes failures easier to identify.

Combining unit, integration, and interface tests improves system quality. However, the existence of layers does not, by itself, guarantee a testable application. Components must have well-defined responsibilities and respect the direction of dependencies (Lima; Leal; Mesquita, 2023).

Technological flexibility can also be promoted through abstractions. Databases, external services, and other resources can be replaced with less impact on the core, provided that access occurs through well-defined contracts.

This characteristic is useful in systems used by different platforms. Web and mobile applications and integration services can share the same use cases, reducing duplication and ensuring greater consistency in business rules (Gonçalves; Mendes, 2022).

Reuse also occurs when common functionality remains concentrated in the core. Instead of repeating validations and processes in different interfaces, use cases provide a single execution point. However, components should not be turned into excessively generic structures merely to increase reuse.

In larger systems, boundaries between layers can facilitate the division of work among teams. Different groups can work on the domain, interfaces, or infrastructure with less interference among their activities. A predictable structure also supports code reviews and the onboarding of new members (Ciriano; Pinto, 2025).

Clean Architecture can be applied to both monoliths and microservices. In monolithic systems, it organizes modules internally; in microservices, it separates each service's rules from communication and persistence mechanisms. However, it does not solve problems inherent to distributed systems, such as network failures and data consistency.

Separating responsibilities can also reduce technical debt. When rules, interfaces, and technological details are mixed, each change tends to increase complexity. Well-defined boundaries facilitate refactoring and prevent temporary solutions from spreading throughout the code (Serafim; Duarte; Duarte, 2022).

Nevertheless, architecture does not eliminate problems caused by a lack of testing, documentation, or planning. Its benefits depend on consistent practices and respect for the boundaries defined during development.

Software longevity can also be improved. Corporate systems remain in use for many years and undergo technological changes. When core rules are isolated, interfaces, tools, and storage mechanisms can be adapted without requiring a complete rewrite of the application (Silva; Saito, 2021).

SOLID principles reinforce this organization by supporting single responsibility, interface segregation, and dependency inversion. However, interfaces and abstractions should be used only when they solve concrete problems, avoiding an unnecessary increase in the number of classes and files.

A predictable structure also improves communication among developers. When the team understands the function of each layer, it becomes easier to define where to add new rules, which components may access external services, and which dependencies are permitted (Silva; Costa, 2024).

Thus, Clean Architecture acts as an ally primarily in projects with complex rules, long life cycles, multiple integrations, and frequent need for evolution. In these contexts, the initial effort may be offset by easier maintenance, testing, and technological adaptation.

In summary, the approach can promote maintainability, testability, flexibility, reuse, and longevity. However, its benefits depend on an application proportionate to project characteristics. When used rigidly or without a concrete need, it can compromise development simplicity (Donato; Zuchi, 2021).

2.5 Limitations and Challenges of Implementing Clean Architecture

Although Clean Architecture supports organization, maintenance, and testability, its implementation also has limitations. Creating layers, contracts, interfaces, and adapters can help control dependencies in complex systems, but when applied unnecessarily, it increases the amount of code and makes development more laborious (Donato; Zuchi, 2021).

One of the main challenges is initial complexity. The team must define in advance which rules belong to the domain, which operations will be treated as use cases, and which resources will remain in the outer layers. This planning requires more time than structures based directly on resources offered by frameworks.

Dividing functionality among entities, controllers, use cases, repositories, and data models can make the flow harder to understand. Although each component has a specific responsibility, the need to consult many files can make reading less direct, especially for developers unfamiliar with the approach (Ferreira; Ferreira; Grande, 2022).

The learning curve also represents an obstacle. Concepts such as dependency inversion, domain, abstractions, and adapters must be understood correctly. Otherwise, the project may merely reproduce a folder structure while keeping business rules dependent on frameworks and infrastructure resources.

This superficial implementation increases the number of components without achieving the expected independence. Adopting Clean Architecture therefore requires training, documentation, and code reviews, as well as greater technical knowledge among developers (Gonçalves; Mendes, 2022).

Initial costs can also increase, especially in inexperienced teams. Training, supervision, and mistaken decisions about boundaries between layers can extend development and require later refactoring. The benefits usually become apparent mainly during system maintenance and evolution.

In projects with short deadlines, creating contracts and abstractions can delay the delivery of simple features. The team must balance development speed with the need to avoid a structure that will be difficult to maintain in the future (Ciriano; Pinto, 2025).

Another risk is creating unnecessary abstractions. Interfaces and adapters are useful when they protect the core from relevant dependencies, facilitate testing, or enable replacements. However, creating an interface for every class can increase complexity without offering proportional benefits.

Abstractions should solve concrete problems. When an implementation is simple, stable, and unlikely to be replaced, direct access may be more appropriate. Excessive use of intermediaries makes the code harder to read and increases maintenance effort (Escareli; Malara, 2022).

Duplicating models can also increase the amount of code. The same information may be represented by entities, persistence models, input objects, and response structures. Although this preserves layer independence, such separation requires constant conversion between formats.

In systems with many fields and simple operations, any change may require modifications to several models and converters. The team should assess whether the isolation obtained justifies the effort or whether certain structures can be shared without compromising business rules (Silva; Costa, 2024).

In small projects, prototypes, or short-lived applications, fully adopting Clean Architecture may be disproportionate. Creating multiple layers can increase development time without offering meaningful maintenance or flexibility gains.

In these contexts, a simplified structure can provide greater agility. If the product grows, new boundaries can be introduced gradually. However, this strategy must prevent temporary solutions from accumulating and forming technical debt that is difficult to control (Donato; Zuchi, 2021).

Excessive planning can also lead to the creation of structures intended for future changes that may never occur. Interfaces, adapters, and extension points based only on remote possibilities increase current costs and can make the system more difficult to understand.

Conversely, a lack of planning promotes high coupling and technical debt. The challenge is to prepare the system for likely changes without creating components aimed only at hypothetical scenarios (Serafim; Duarte; Duarte, 2022).

Integration with frameworks represents another difficulty. Many tools offer their own structures for controllers, entities, and data access. Complete isolation of the core may require additional adapters and prevent the use of some conveniences provided by the chosen technology.

Technological independence does not mean rejecting frameworks. These resources can accelerate development and standardize tasks. In certain projects, accepting a limited and known dependency may be more appropriate than creating an extensive structure solely for a possible future replacement (Ferreira; Ferreira; Grande, 2022).

Testing also requires balance. Although Clean Architecture facilitates isolated tests, excessive use of mock objects can produce results that differ from real behavior. A feature may pass unit tests and fail during integration with databases, networks, or external services.

Unit tests should therefore be combined with integration and interface tests. Moreover, changes to contracts may require updating many mock objects, increasing the maintenance work for the tests themselves (Lima; Leal; Mesquita, 2023).

In distributed systems, Clean Architecture does not eliminate the problems of microservices. Network communication, unavailability, security, data consistency, and monitoring remain challenges. Combining the two approaches can significantly increase the number of components managed by the team.

In small teams, this complexity may exceed the expected benefits. When there are no concrete requirements for distribution or independent scalability, an organized monolith may represent a simpler and more efficient solution (Kanizawa; Pinto, 2022).

Architectural consistency must also be maintained over time. Short deadlines, team changes, and urgent deliveries may lead to business rules being placed in controllers or to use cases accessing infrastructure directly.

Avoiding this degradation requires code reviews, documentation, internal standards, and continuous monitoring. Architecture does not remain organized solely because of decisions made at the beginning of the project (Escareli; Malara, 2022).

Rigid application of the principles can also prevent simple and suitable solutions. Interfaces and intermediaries should not be created merely to reproduce a theoretical model. Architecture should guide decisions while allowing adaptations to the system's actual characteristics.

Pragmatism requires comparing the costs and benefits of each separation. A dependency may be acceptable when it is limited, known, and unlikely to change, whereas an abstraction is justified when it protects important rules or supports reliable testing (Gomes; Gibertoni, 2022).

The limitations of Clean Architecture therefore do not make the approach unsuitable, but they demonstrate that its adoption depends on context. Initial complexity, increased code, the learning curve, excessive abstractions, and maintenance costs must be considered.

In long-lasting systems with complex rules, the architectural investment may be offset over time. In simple applications, partial adoption may be more appropriate. Thus, Clean Architecture becomes a possible villain when it is applied indiscriminately and without relation to the project's concrete needs (Donato; Zuchi, 2021).

2.6 Suitability of Clean Architecture for Different Development Contexts

The adoption of Clean Architecture should consider the characteristics of each project, such as application size, rule complexity, team experience, and expected evolution. Although its principles support separation of concerns and dependency control, full application is not always necessary. The architecture should be proportionate to software needs, avoiding both disorganization and the creation of excessively complex structures (Donato; Zuchi, 2021).

In systems with complex business rules and frequent updates, separation among the domain, use cases, interfaces, and infrastructure can facilitate understanding, testing, and maintenance. By keeping core rules in independent components, changes tend to have less impact on other parts of the application.

Long-term corporate projects can benefit from this organization because they generally pass through different teams, changes in business rules, and technological replacements. Preserving the application core reduces the impacts caused by replacing frameworks, databases, and interfaces, contributing to system continuity (Ferreira; Ferreira; Grande, 2022).

However, not every corporate application requires many layers and abstractions. Administrative systems with simple flows and few changes can be organized through smaller structures. The choice should be based on functional complexity and maintenance costs, not merely on the size of the organization.

In small projects, fully applying Clean Architecture may be disproportionate. Creating interfaces, adapters, and models for each layer can increase effort without providing significant gains. In such cases, a simpler structure can preserve quality and facilitate development (Escareli; Malara, 2022).

This does not mean abandoning its principles. Separation between business rules and presentation, definition of responsibilities, and reduced coupling can be applied in a simplified manner. The team can adopt only the necessary elements, avoiding a structure larger than the problem requires.

Startups and products under validation also require balance. An extensive architecture can delay delivery of the first version, while a lack of organization can hinder future growth. It is therefore possible to begin with a simple structure and introduce new boundaries as requirements become clearer (Serafim; Duarte; Duarte, 2022).

Team experience directly influences results. Developers familiar with SOLID principles and dependency inversion tend to apply the approach more safely. In inexperienced teams, a lack of understanding can generate fragmented components and purposeless abstractions. Documentation, training, and code reviews are therefore fundamental.

Clean Architecture can also be applied to monolithic systems. An application deployed as a single unit can have well-defined modules, independent use cases, and separation between domain and infrastructure. In many projects, an organized monolith is more appropriate than immediately adopting microservices (Pegrucci; Tiosso; Reis, 2020).

In microservices, its principles can internally organize each service, protecting business rules from communication and persistence mechanisms. However, this combination increases the number of components and does not eliminate challenges such as network failures, data consistency, monitoring, and deployment complexity.

In legacy systems, Clean Architecture can guide gradual modernization. Progressively creating boundaries, use cases, and adapters makes it possible to isolate parts of old code and reduce dependencies without requiring a complete application rewrite, thereby reducing transformation risks (Silva; Saito, 2021).

Integration with different interfaces can also justify its adoption. Web and mobile applications and external services can share the same use cases when core logic remains independent of presentation. However, in systems with a single simple and stable interface, less extensive separation may be sufficient.

Testability requirements are another important criterion. In systems where failures can cause significant losses, isolating rules facilitates unit testing and the replacement of external dependencies with controlled implementations. This characteristic helps validate core behaviors with greater confidence (Lima; Leal; Mesquita, 2023).

The frequency of technological changes should also be considered. Contracts and adapters are most useful when there is dependence on external services, different data sources,

or technologies likely to be replaced. When the resources used are stable, the cost of creating and maintaining abstractions may exceed the expected benefits.

Available time and resources likewise influence the decision. Clean Architecture usually requires greater initial investment in defining layers, contracts, and data flows. In long-term projects, this effort may be offset by easier maintenance, but in temporary systems the return may not occur (Ciriano; Pinto, 2025).

There is therefore no single answer to whether Clean Architecture is an ally or a villain. It can facilitate the development of complex systems and make the construction of simple applications unnecessarily difficult. Its adoption should consider domain complexity, the number of integrations, expected growth, project duration, team experience, and testing needs.

The approach can also be applied partially and progressively. A system can begin with separation between domain and infrastructure and incorporate new use cases, adapters, and abstractions as concrete needs arise. In this way, the architecture follows software evolution without being treated as a rigid model (Donato; Zuchi, 2021).

Thus, Clean Architecture acts as an ally when it protects relevant rules, facilitates testing, reduces dependencies, and supports evolution. It becomes a possible villain when applied indiscriminately, increasing the amount of code, costs, and difficulty of understanding. Its effectiveness depends on critical, pragmatic use consistent with the conditions of each project.

3 Conclusion

This study made it possible to critically analyze the application of Clean Architecture in modern software development, considering its foundations, benefits, limitations, and different contexts of use. Based on the literature review, it was possible to understand that this approach seeks to protect business rules from technological changes by organizing the system through separation of concerns and control of dependencies among components.

The main benefits identified include improvements in system maintainability, testability, flexibility, and capacity for evolution. By keeping the application core independent of frameworks, databases, interfaces, and external services, Clean Architecture can reduce the impacts caused by technological changes and facilitate isolated testing. In addition, defining clear boundaries among entities, use cases, adapters, and infrastructure contributes to a more understandable organization of the code.

However, important limitations were also identified. Implementing Clean Architecture can increase the number of classes, interfaces, contracts, converters, and files, making development initially slower and more complex. This problem is more evident in small projects, prototypes, temporary applications, or systems with simple business rules, in which the effort required to create an extensive structure may not be offset by future benefits.

Another challenge is related to team experience. Proper adoption of the approach requires knowledge of separation of concerns, SOLID principles, dependency inversion, and layer organization. When these concepts are not understood, the project may present only a formal division of folders without achieving the expected independence. In this scenario, Clean Architecture increases code complexity without effectively improving its quality.

The research question was therefore answered by finding that Clean Architecture contributes to software quality and evolution when applied to systems with complex business rules, multiple integrations, a frequent need for changes, and an expectation of long-term use. Conversely, its adoption can generate costs, difficulties, and unnecessary abstractions when it occurs rigidly, indiscriminately, or incompatibly with the project's actual needs.

It was also found that adopting Clean Architecture does not have to occur in full. Its principles can be applied partially and progressively according to system growth and complexity. Separation between domain and infrastructure, for example, can be implemented initially, while new layers and abstractions are added only when concrete maintenance, testing, or technology replacement needs arise.

It is therefore concluded that Clean Architecture should not be considered exclusively an ally or a villain of modern software development. It is an architectural tool whose effectiveness depends on context, planning, and how its principles are used. When applied critically, proportionately, and pragmatically, it can contribute to building systems that are more organized, testable, and prepared to evolve. However, when adopted merely as a trend or mandatory model, it can compromise simplicity, increase costs, and hinder the development process.

4 References

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.