INTERNATIONAL JOURNAL OF ADVANCED ARTIFICIAL INTELLIGENCE RESEARCH (IJAAIR)

eISSN: 3087-4297

Volume. 02, Issue. 12, pp. 01-12, December 2025"

Comparative Review of Clean Architecture and Vertical Slice Architecture Approaches for Enterprise .NET Applications

Serhii Yakhin
Senior .NET Software Engineer at Growe

Article received: 24/11/2025, Article Revised: 03/12/2025, Article Accepted: 14/12/2025, Article Published: 26/12/2025
DOI: https://doi.org/10.55640/ijaair-v02i12-01

ABSTRACT

The article presents a comparative analysis of the Clean Architecture and Vertical Slice Architecture (VSA) approaches in the context of designing enterprise .NET applications, conducted from the vantage point of their evolutionary interplay, organizational applicability, and impact on maintainability. The study's relevance is that the .NET ecosystem remains one of the most robust corporate development standards. At the same time, the rising cost of change and the acceleration of DevOps cycles demand architectures that simultaneously sustain the resilience of the domain model and enable high-velocity feature delivery. The objective is to identify patterns of complexity distribution across layers and functional slices, and to formulate criteria for selecting or combining the specified approaches at different stages of the software system's life cycle. The novelty lies in the synthetic consideration of Clean Architecture and Vertical Slice Architecture not as mutually exclusive but as mutually complementary architectural paradigms, joined within a single spectrum of product maturity. It is recorded that both approaches implement different phases of the corporate solution life cycle: vertical slices prevail in the early stages, providing rapid adaptation and feedback, whereas as the product and organizational complexity grow, the role of Clean Architecture increases, ensuring the consistency and durability of solutions. The key conclusion is that an optimal strategy for enterprise .NET systems is a hybrid configuration in which Clean Architecture defines the systemic core and standardized interaction boundaries. At the same time, Vertical Slice Architecture shapes the peripheral modules responsible for the independent evolution of functional areas. In addition, it is established that the effectiveness of the chosen architecture is determined not by the pattern itself but by the engineering maturity of the team, the presence of automated tests, review procedures, and quality metrics, without which the advantages of both models are neutralized. The article is of practical value to software architects, technical leads, and researchers designing enterprise solutions on the .NET platform and seeking optimal strategies to balance stability and the speed of change.

KEYWORDS

Clean Architecture, Vertical Slice Architecture, .NET, enterprise applications, architectural patterns, microservices architecture

1. Introduction

Corporate IT divisions continue to rely on the .NET ecosystem when a technology foundation resilient to change, a transparent maintenance model, and confidence in long-term vendor support are required. In 2025, the C# language retains nearly a third of the professional developer community (29.9% in the professionals segment, 27.8% in the overall audience), making it the fourth most popular programming language; accordingly, each new design methodology for .NET automatically concerns a broad swath of the industry [1]. Among web stacks, ASP.NET Core remains in the leading cohort: 21.3% of professional developers use it permanently, with 19.7% in the overall sample; this figure significantly outpaces classic ASP.NET and confirms that enterprises are shifting their focus to a more modern and modular platform. In the Stack Overflow

2024 survey, the Other Libraries and Frameworks category is again led by .NET, underscoring its reputation as a corporate standard despite competition from native and interpreted solutions [2].

In parallel, enterprise application landscapes have shifted mainly towards cloud-native, containerised, and increasingly event-driven solutions, characterised by high rates of change and continuous delivery practices. Within this environment, architectural styles must support not only change resilience but also horizontal scalability, operational fault tolerance, and graceful degradation under load. These qualities are no longer abstract non-functional requirements but directly measurable factors that influence delivery speed, defect rates, total cost of ownership (TCO), and the cognitive load borne by development teams. Against this backdrop, the .NET community in 2025 effectively converges on two leading architectural approaches for large-scale business systems: Clean Architecture, which represents the evolution of layered, domain-centric design, and Vertical Slice Architecture, which represents feature-oriented decomposition aligned with modern DevOps cadences.

As the unit cost of code changes increases, businesses have sought architectural techniques to limit the blast radius of modifications. The most mature among them is Clean Architecture. However, teams' ambitions to accelerate feature rollout and isolate risks have generated demand for a more fine-sliced practice, Vertical Slice Architecture.

The vendor itself supports this shift: in Microsoft's official .NET Microservices guide, it is emphasized that enterprises already reduce overheads through containerization and aim to decompose systems into independently deployable services. The materials note that Docker is becoming an industry standard, and that Azure Kubernetes Service and Service Fabric cloud services constitute the de facto infrastructure layer, thereby increasing the value of architectures that fit into a microservices context [3].

These trends are embedded coherently in the platform's evolution. The first wave of .NET (WinForms, WebForms, WCF) was based on a rigid three-layer architecture, with decoupling achieved through service-level abstractions. With the advent of .NET Core and open source in 2016, a second phase began: the community transferred the concepts of the Onion and Hexagonal approaches to the platform, which coalesced under the Clean Architecture brand into convenient templates. The third, current phase is driven by the need to develop independent functional slices in parallel without compromising overall domain integrity; here, vertical slices come to the fore, where each set of commands/queries has its own model, handlers, and infrastructural dependencies, all confined within a single repository folder [4]. The continuous .NET release cadence (years 6–7–8–9) further intensified pressure in favor of practices requiring minimal glue code and maximal feature isolation.

Thus, the subject of the present article, a comparison between Clean Architecture and Vertical Slice Architecture, is justified by both the ecosystem's scale and the industry's evolutionary demand: the former provides time-tested stratification and testability of the core. At the same time, the latter accelerates delivery and parallel teamwork.

2. Materials and Methodology

The research was conducted as a comparative review of the Clean Architecture (CA) and Vertical Slice Architecture (VSA) paradigms in the context of enterprise .NET applications. The methodological foundation combines a systematic analysis of academic and industry sources, practical cases from Microsoft's official guidance, and secondary data on engineering patterns and production metrics. A total of 9 documents were analyzed, including peer-reviewed journal articles, open-access preprints, and .NET Core architecture guides.

The theoretical framework was built around models of sustainable design for the .NET ecosystem, where the layered concept of Clean Architecture is viewed as an evolution of the traditional N-Tier approach. The work by Nagib [5] was used to capture empirical differences between stratified systems and clean architectures in terms of code dependencies and distribution of responsibility. The conceptual arrangement of inner and outer architectural circles relied on Valente [6], which systematizes the practice of implementing a clean structure in Softengbook engineering templates. For analysis of the vertical-slice paradigm, studies by Dieste et al. [7] and Lytvynov & Hruzin [8] were employed, examining the roles of slicing and CQRS models in test-driven and event-driven design, respectively. Data on the impact of code segmentation and bounded contexts on solution maintainability were taken from Ibrahimi & Moudilos [9], which discusses low-code platforms but contains relevant conclusions on structural cohesion.

3. Results and Discussion

3.1. Overview of CA and VSA

The Clean Architecture approach (hereinafter CA) emerged in response to the growing enterprise need for robust separation of responsibilities within software systems. Its foundational principle is the so-called dependency rule, according to which the direction of relationships in source files always points inward: inner components containing business rules do not depend on outer ones, whereas user interfaces, databases, and frameworks must interact with the core only through abstractions. Contemporary analyses of CA's concentric layers emphasize that precisely such inversion governs the evolutionary flexibility of a solution, enabling changes to environmental details without rewriting critical logic [5].

The CA model is typically described by four circles, as shown in Figure 1.

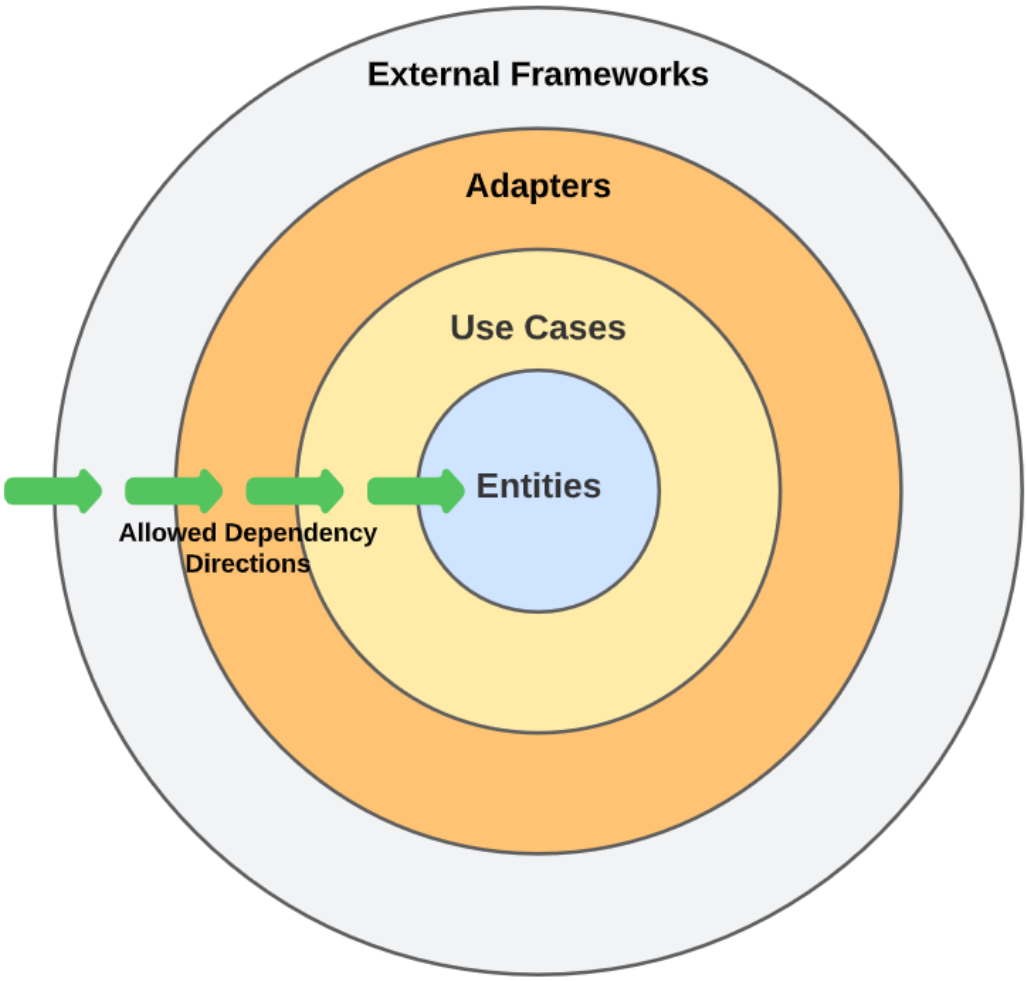

Figure 1: Clean Architecture [5]

At the very center reside domain entities; next follows the use-case layer, where user precedents are codified; then an infrastructural contour responsible for storage, integration, and network calls; finally, the outer circle forms the presentation layer, providing application code access to human or interservice interaction environments [6]. In .NET guidance, these levels are illustrated in practice through solution templates, allowing developers to immediately separate the domain model from the infrastructure and interfaces without overcomplicating project initiation. In enterprise .NET solutions, this conceptual layout is most commonly implemented as a four-project structure comprising Domain, Application, Infrastructure, and API layers, which mirrors the concentric circles while remaining aligned with standard tooling and deployment pipelines.

The principal value of CA lies in the technological independence of the inner core. The absence of direct references to specific data storage or visualization packages facilitates migration across DBMSs and delivery media, increases the volume of code amenable to unit testing, and thereby reduces the risk of regressions under frequent releases. Authors of practical guides also highlight improved readability, since each domain rule resides in an isolated file and the boundaries between

layers are evident even to new team members [4].

However, clean separation exacts a price in the form of a more complex configuration. Data transfer between layers requires adapter models, repository interfaces, and mappers, which increases boilerplate and the time needed for initial setup. For applications with a narrow domain scope, such fragmentation can turn the architecture into a source of unnecessary indirection and slow feature delivery, especially when the team is small and lacks rigorous code review procedures.

The vertical slice views a software system not as an aggregation of horizontal technical layers but as a set of autonomous flows of requirement realization traversing the entire application. In an empirical study devoted to the foundation of the test-driven approach, each TDD cycle is interpreted as a complete vertical slice of a user story; the authors showed that such local closure improves observability of side effects and thus accelerates feedback between code and tests [7].

In practice, the idea dovetails with the separation of commands and queries: the Command/Query Responsibility Segregation architecture allows write and read operations to be placed into separate models, thereby naturally fixing slice boundaries. Both CA and VSA in enterprise .NET are typically applied within a domain-driven design (DDD) perspective, in which a rich domain model and its ubiquitous language define bounded contexts; CQRS commands and queries then realise individual use cases that cut across technical layers while remaining confined to a specific context. An analytical investigation of CQRS variants with event-sourcing records shows that the applicability of the classical implementation ranges from 47–61% depending on project priorities, whereas a simplified modification yields 39–53%, a difference explained by the trade-off between throughput and the complexity of migrating across variants [8].

A codebase under such a design is organized into slices: each folder contains the contract for a command or query, the handler, data-transfer models, and infrastructural adapters. Clear segregation enables distributing responsibility among parallel development teams without tight coordination. In typical .NET implementations, these slices are expressed as Request/Handler/Endpoint triplets, often wired through a mediator component, and in many cases they are hosted within a single solution or even a single project where handlers are allowed to access the persistence layer directly, provided transactional boundaries and testing requirements are satisfied.

Academic metrics indicate that a vertical slice improves testability and maintainability: compressed dependencies across slices reduce the number of callbacks, and maintenance-cycle indices decrease by more than twenty percent when moving from a layered scheme to a sliced one; at the same time, code cohesion increases, as measured by method weight and the proportion of reuse [9]. Risks also intensify: the same CQRS work notes that introducing asynchronous exchange and eventual consistency raise the entry threshold and require deliberate monitoring to detect desynchronization between read and write models, particularly in systems with high update frequency [8].

Accordingly, the vertical slice, whose algorithm is shown in Figure 2, forms highly cohesive yet isolated fragments, accelerating value delivery and simplifying team scaling, while shifting responsibility for transaction integrity and duplication elimination to architectural automation and engineering discipline. These properties render it a natural counterpoint to clean stratification and establish the basis for subsequent comparative analysis of the two approaches in enterprise .NET systems.

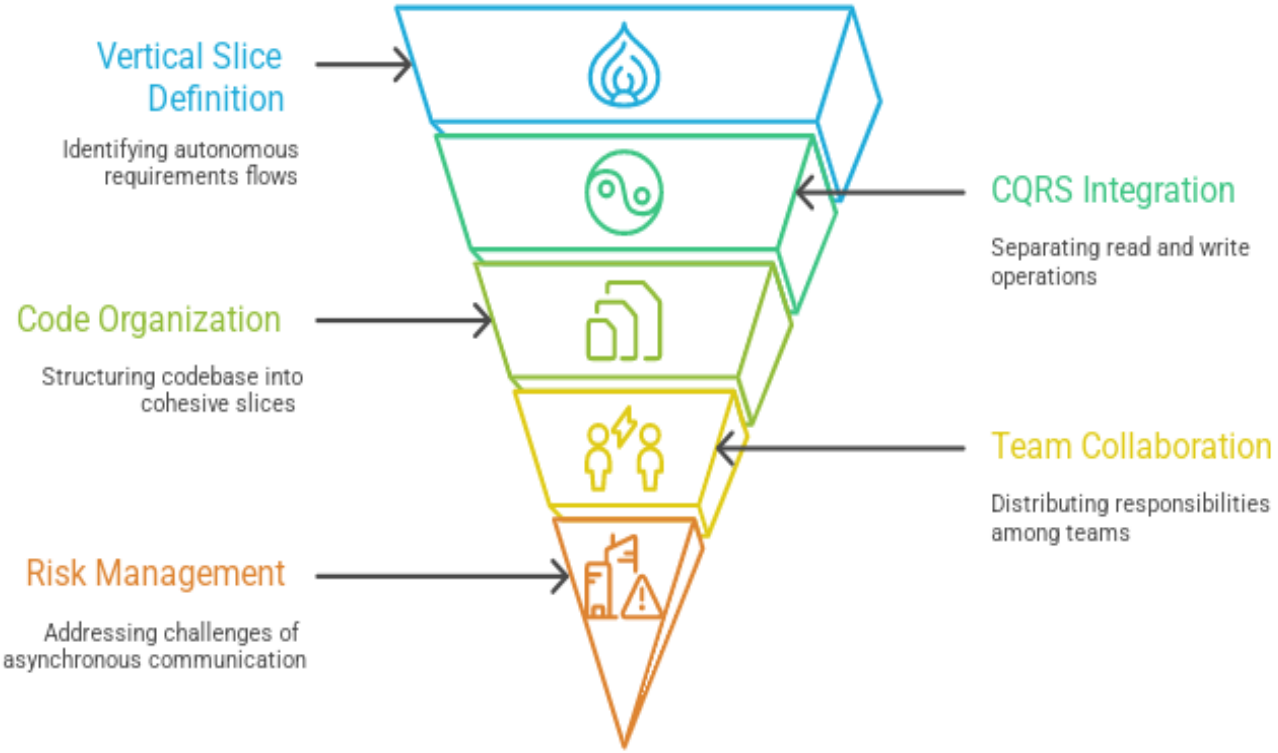

Figure 2: Implementing Vertical Slicing in Software Design

3.2. Comparison of CA and VSA

A juxtaposition of the two approaches reveals a stable divergence in how they allocate complexity between people and code. Clean Architecture places the main burden on initial domain modeling. By correctly separating the core from the infrastructure, a team obtains a system in which each change passes through narrowly delineated interfaces, triggering minimal side effects. The price of this advantage is the need for continuous discipline when creating new layers and adapters; any shortcuts at the start typically come back as hidden dependencies that are difficult to eradicate later.

Vertical Slice Architecture, by contrast, shifts labor intensity toward functional isolation. Each new fragment of logic evolves almost autonomously, that is, faster and without coordination with the rest of the solution. The result is a variegated yet readily extensible picture in which individual teams move in parallel without awaiting a centralized architectural authority. The downside is increased duplication and the complexity of cross-cutting refactoring: slice intersections become apparent only when accumulated discrepancies begin to hinder system-wide consistency.

These opposite emphases yield a regularity particularly pronounced in long-lived products. The more actively business requirements change, the more evident the benefit of vertical slices in the initial development period; however, as the system's organizational weight grows, the need for a shared model characteristic of Clean Architecture increases. Thus, both approaches describe different life-cycle phases: the vertical slice accumulates speed, while the stratified structure imparts shape to the accumulated experience and reduces further disorder.

Another regularity is observed: the success of each method depends less on the chosen directory scheme than on the maturity of engineering practices. With strict automated checks, continuous integration, and a rich test suite, even with a vertical structure, integrity remains acceptable; without these practices, the advantages of clean layered boundaries are lost. In other words, architectural decisions amplify or attenuate team properties; they do not replace quality procedures but merely render them more or less obligatory.

Accordingly, the opposition of the two archetypes, shown in Figure 3, becomes a spectrum in which each point describes a balance between speed and order. Practice indicates that projects gradually move along this spectrum: at early stages, vertical increments predominate; as maturity increases, a core formed by domain-isolation rules grows within them. The resulting conclusion reduces to the idea of compatibility: a judicious combination of local autonomy with a global framework can fuse the strengths of both approaches while minimizing their weaknesses.

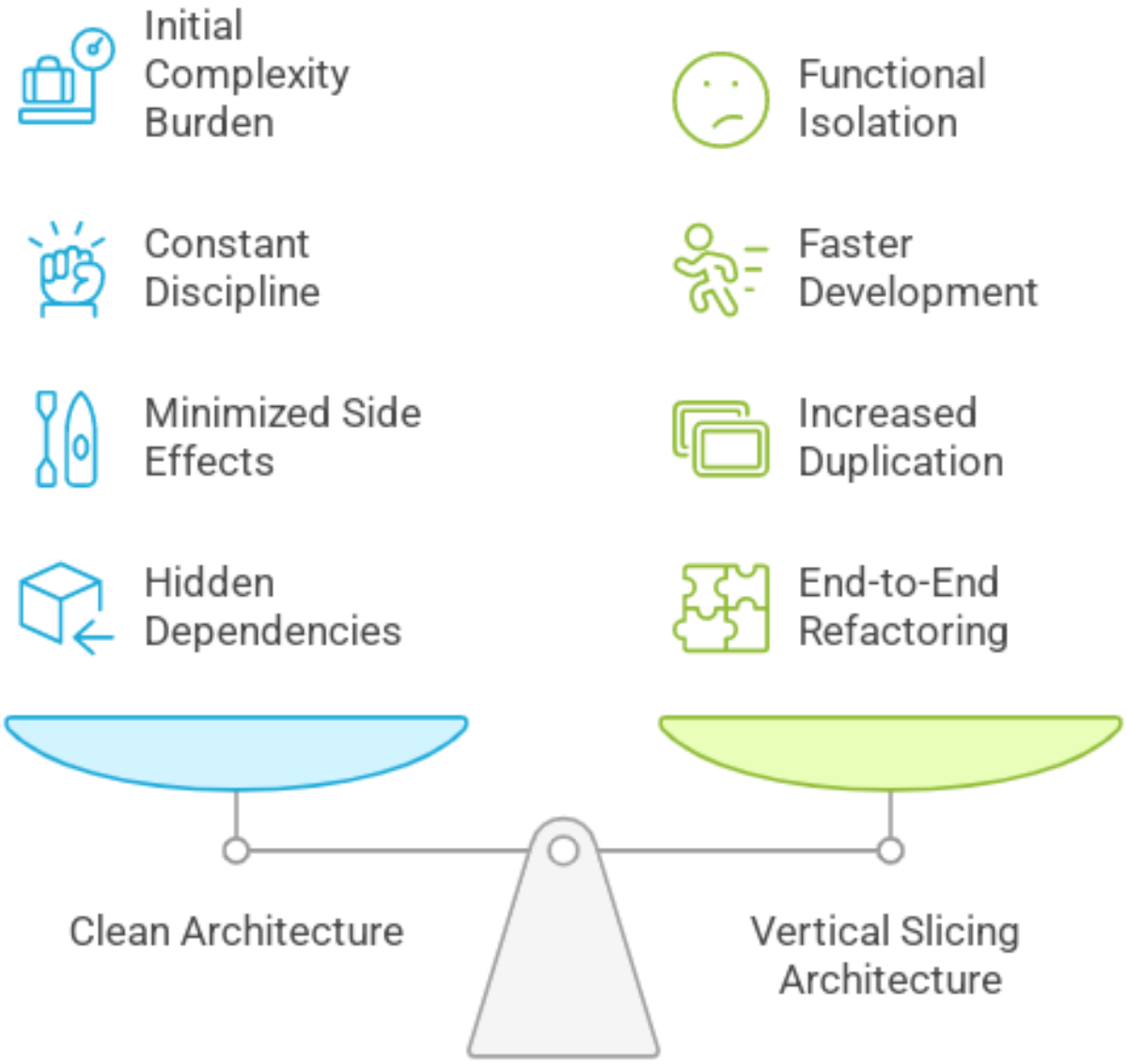

Figure 3: Balancing Speed and Order in Software Architecture

3.3. Symbiotic (Hybrid) Architecture

An internal architectural strategy always reflects the product's form and the tempo of its evolution; therefore, the choice between a layered framework and vertical increments is best considered through the prism of specific contexts. In a large monolith, where a single runtime hosts a substantial mass of domain rules, management of the common core plays a central role. Here, Clean Architecture reveals its potential: strict segregation of domain, use cases, and infrastructure curbs the spontaneous growth of couplings, and modular partitioning within the monolith enables the maintenance of different subsystems without compromising integrity. Even in such an environment, vertical slices find application at the command/function level: small logic packets designed according to the command-query principle reduce knowledge fragmentation and create local autonomy zones, which is especially useful when several groups develop different parts of one product in parallel.

If the environment is distributed and many processes are deployed in independent containers, another trade-off arises for the architect: delivery speed versus interservice consistency. The vertical-slice style naturally fits a microservices topology: each service forms its own set of commands, queries, and read models, and slice boundaries coincide with deployment boundaries. Such localization simplifies horizontal team scaling; whether adding functionality or an experimental branch, changes affect only the selected service. However, as the system grows, the need arises for common agreements on data formats and error-handling patterns; here, the stratified approach serves as an overlay layer, defining a standardized core of reusable conventions.

In the world of young products, priority shifts toward

minimizing the time between idea and market feedback. A startup team launching a minimally viable solution is rarely ready to invest in detailed engineering modeling. A vertical slice allows the team to describe the set of input and output messages in detail along a user scenario's path and deliver value immediately. It also facilitates abrupt course changes by decoupling new functionality from already released components. At a later stage, when the product trajectory stabilizes and speed yields to resilience, it is helpful to gradually introduce elements of Clean Architecture, shaping the layers around the core to create predictable boundaries for subsequent growth.

A different situation arises with legacy systems that have evolved for years without architectural guidance and now suffer from tight couplings, hidden dependencies, and brittle code. For such cases, a hybrid of both approaches is particularly effective. A selected functional area is extracted into a vertical slice, within which a thorough reworking of the domain model is performed in accordance with the canons of Clean Architecture. The slice is then reintegrated, but now as an isolated module with clear inputs and outputs. Iterative repetition of the procedure enables restructuring of the legacy without halting business operations: each iteration yields measurable improvements, while the risk of regressions remains localized.

Consequently, practical scenarios show that Clean Architecture and vertical slices are not mutually exclusive poles but tools that are combined depending on domain weight, organizational structure, and the rhythm of change. Conscious alternation of methods allows the initial delivery of functionality with maximal agility and the subsequent creation of conditions for the sustainable evolution of the entire application ecosystem.

A combination of a layered core with feature-oriented verticals has gradually become a natural response to the observed duality of design needs: on the one hand, organizations require a durable anchor in the form of a resilient domain model; on the other, they need the freedom to release isolated fragments of functionality quickly without jeopardizing existing code. The hybrid contour is built around a minimal domain center that includes entities and rules independent of infrastructure; use cases are deployed over it, while storage and interaction details are left to vertical slices located along the perimeter. Each slice contains its own command or query handler, local read/write models, and adapters to external services, allowing the team responsible for a specific task to fully control the life cycle of its piece of the system.

A simple hybrid layout is depicted in Figure 4 as a multi-project .NET solution, where a set of Clean Architecture projects (Domain, Application, Infrastructure, and API) form the central core. In contrast, feature-oriented vertical-slice projects or folders plug into the Application and API layers via well-defined contracts.

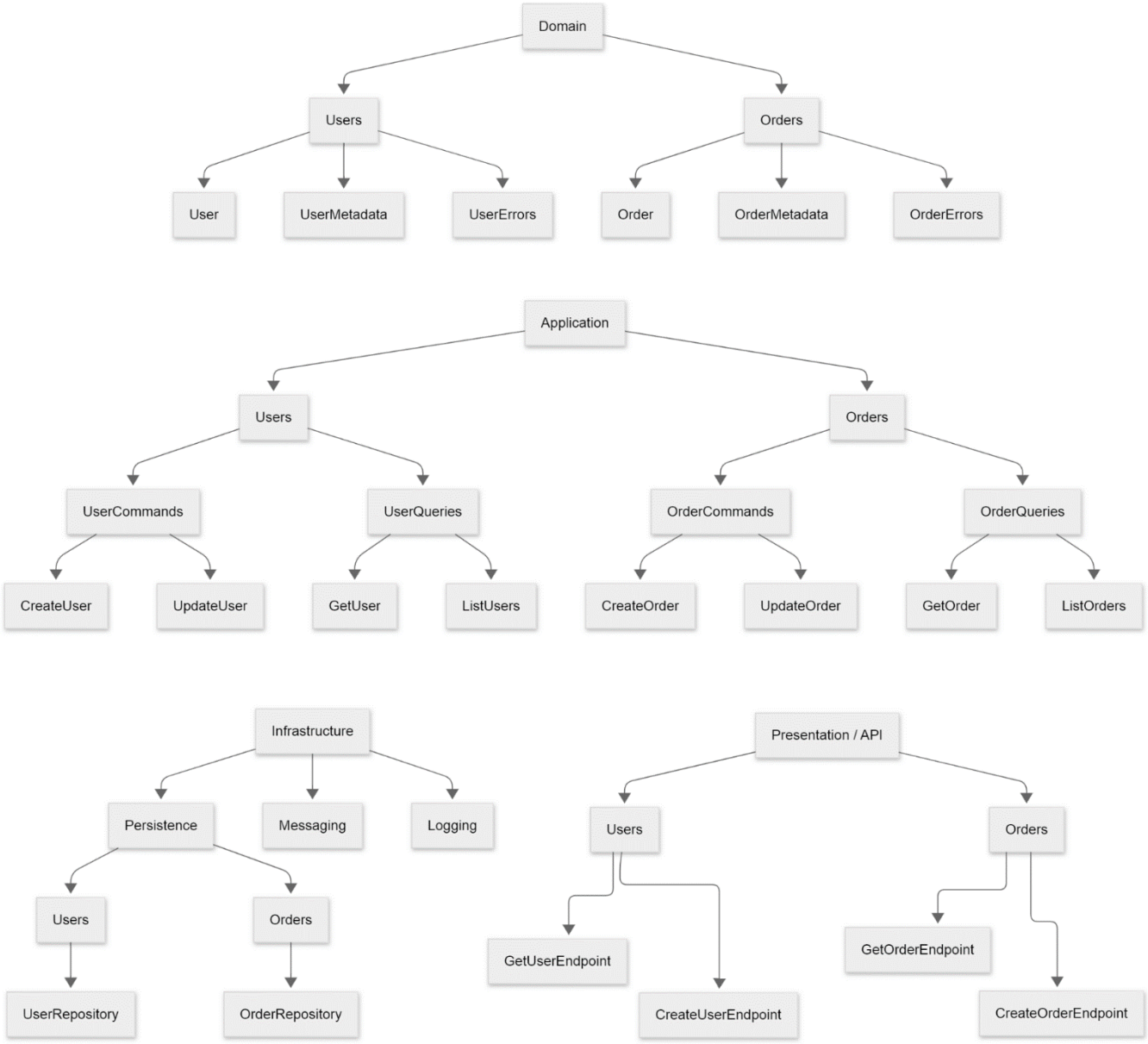

Figure 4: Hybrid architecture diagram

Practice indicates that it is most convenient to begin with a tiny core that describes only key domain invariants and to define a unified dependency-injection contract immediately. Each new vertical appears as a cross-cutting augmentation attached to the container configuration and accessible via a mediator bus. Within such a slice, it is helpful to adhere to a closed set of ports: input as a command or query, output as an event or projection, with all other logic wrapped in private classes invisible outside the folder. Duplication of intersecting data representations is efficiently resolved by using shared packages with read-only models, thereby avoiding dragging infrastructure into the core while simultaneously reducing contract drift.

A hybrid solution contains pitfalls. The most frequent is gradual stratification of terminology, when different verticals begin to name the same concepts differently, thereby multiplying entities that are subsequently difficult to reconcile into uniformity. A second risk is the sprawl of adapter layers: gateway interfaces and mappers may grow faster than the domain core and, over time, turn into a new edge-monolith. These threats are mitigated by a regular merging ritual, a periodic review of boundaries in which the architectural committee compares models and elevates common elements into the core, as well as by static analysis tools that detect cyclic dependencies. Additional protection is provided by contract tests that monitor compatibility of events and external APIs; they formalize exchange rules and prevent silent version drift.

3.4. Decision Framework (Checklists & Matrices)

To decide whether a system needs clean stratification, vertical slices, or a mixed option, it is helpful to traverse the branches of a mental decision tree. At the top, the question of change velocity is posed: if business rules change weekly, vertical design prevails; if edits are rare and the risk of error is high, a strict layered framework is preferable. Next, team scale is evaluated: a small co-located group more easily agrees on vertical autonomy, whereas a large distributed department requires a single core with strict boundary conventions. The next level addresses process maturity: the presence of automated checks and end-to-end quality metrics increases the long-term viability of slices, whereas weak discipline dictates centralized layer control. The final node asks about the criticality of data consistency; if temporary inconsistency is tolerable and compensating mechanisms exist, verticals are more justified; otherwise, a monolithic core takes priority, as shown in Figure 5.

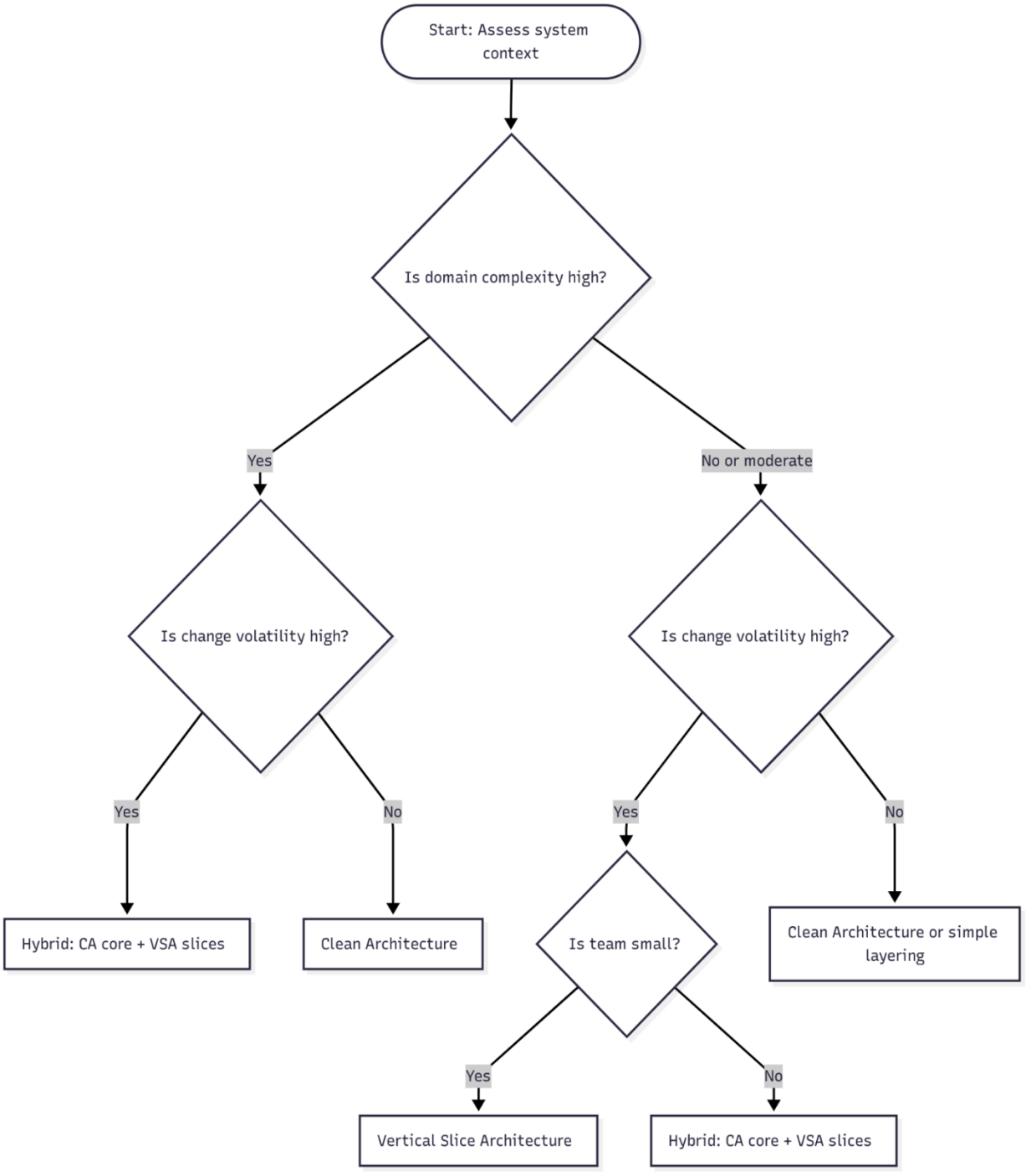

Figure 5: Architecture selection diagram

Before final selection, a short checklist assists an architectural committee. It identifies whether multiple independent delivery cadences are required, when a database replacement is expected, how many different teams will write code in parallel, the cost of an hour of downtime, whether there is a budget for regular large-scale refactorings, and how often stakeholders demand audits of the storage model. Answers to these items, taken together, delineate the spectrum point at which the project will develop most naturally. Complementing the checklist, an evaluation matrix can be used (Table 1) to score CA, VSA, and a hybrid configuration against key criteria such as time-to-first-feature, long-term maintainability, developer onboarding effort, observability and diagnostics, regulatory and compliance support, scalability characteristics, team autonomy, and overall total cost of ownership (TCO).

Table 1: Evaluation matrix

Criterion	Clean Architecture (CA)	Vertical Slice Architecture (VSA)	Hybrid (CA core + VSA slices)
Time-to-first-feature	Slower start due to upfront layering and abstractions	Fastest initial delivery; minimal upfront structure	Moderate: faster than pure CA, slightly slower than pure VSA
Long-term maintainability	High, if discipline is preserved; clear dependency rule	Medium; local cohesion is high, but duplication accumulates	High; shared core stabilises invariants, slices localise volatility
Onboarding	Medium; concepts are clear but structure is heavier	Easy for small scopes; harder as slices proliferate	Medium; core rules plus slice patterns require some guidance
Observability	Good centralisation of cross-cutting concerns	Requires consistent patterns across slices	Best potential: shared observability standards with slice-level detail
Compliance / Auditability	Strong, due to explicit boundaries and shared model	Harder; scattered rules and models complicate evidence collection	Strong; policy lives in core, enforcement hooks exist in vertical slices
Scalability	Scales well at the code level; runtime scaling via APIs	Scales by replicating and isolating slices/services	Flexible: core scales logically, slices scale operationally
Team autonomy	Moderate; changes pass through centralised layers	High; teams own slices end-to-end	High; autonomy at slice level within shared contractual boundaries
Total Cost of Ownership	Lower in the long run, higher initial investment	Low initial cost, risk of rising costs over time due to divergence	Balanced; controlled long-term cost with acceptable initial investment

For fixing the final decision, a flexible notation is effective: at a high level, a context diagram with separated boundaries shows which feature sectors communicate and through which protocols; at the middle level, a component diagram colors core layers in one hue and slices in another to emphasize dependency directions visually; at the lowest level, decision cards record the reasons for the chosen balance and the conditions for its revision. This minimal document set allows retention of a coherent overall picture without overburdening

developers with excessive bureaucracy, while preserving the possibility of architectural evolution alongside the product itself.

4. Conclusion

A comparative analysis of Clean Architecture and Vertical Slice Architecture for enterprise .NET applications demonstrates that both models constitute not so much mutually exclusive as mutually complementary strategies for structuring code and organizing the engineering process. Clean Architecture ensures the resilience of the systemic foundation and the predictability of evolution: strict inversion of dependencies and independence of the domain core from infrastructural details form a platform capable of withstanding long-term change without loss of integrity. In this context, it functions as an instrument of strategic control, an architectural skeleton that prevents a product from sliding into chaos and ensures manageability even amid high staff turnover and growing project complexity.

Vertical slices, by contrast, operate in the tactical dimension. Their strength lies in the ability to rapidly realize new user scenarios while minimizing the impact radius of each change. Thanks to the cross-cutting organization of commands and queries, each functional unit becomes a self-sufficient micro-context, ideally aligned with the cadence of the modern DevOps cycle and the practice of microservice delivery. At the same time, such flexibility is inevitably coupled with the risk of logic duplication and divergence of terminological models across slices; these effects require compensating mechanisms, either a centralized core or a disciplined system of contract tests and architectural reviews.

The most productive scenario is a hybrid one, where Clean Architecture serves as the central support, and vertical slices act as dynamic peripheral modules. In such a configuration, the inner layer defines the domain's standard rules, while outer slices provide implementation variability and independent delivery cadences. This symbiosis yields a balanced state between rate of change and architectural rigor, an attribute in high demand in the corporate environment, where domain stability must coexist with rapid responsiveness to changing business requirements.

The choice between the two approaches should be treated not as a binary decision but as a position on a spectrum of product maturity. At early life-cycle stages, vertical slices accelerate feedback and adapt the product to the market, whereas as domain invariants accumulate and organizational mass increases, the value of Clean Architecture grows, ensuring manageability and reuse. In the long run, system resilience is determined less by the architectural template itself than by the culture of its application: given automated tests, observability, and engineering discipline, both approaches exhibit a synergistic effect, reinforcing each other's strengths.

Thus, the comparative study confirms that, for the .NET ecosystem, currently at a mature stage of integrating microservice and domain-oriented practices, the best outcome is achieved not by opposing Clean Architecture and Vertical Slice Architecture, but by judiciously combining them. In this combination, the platform's evolutionary tendency is manifested: a movement from monolithic stratification toward modular autonomy while preserving conceptual integrity, which defines the strategic direction of architectural development of enterprise .NET systems in the coming years.

References

1. "Technology," *Stackoverflow*, 2025. https://survey.stackoverflow.co/2025/technology/ (accessed Oct. 01, 2025).
2. "Technology," *Stackoverflow*, 2024. https://survey.stackoverflow.co/2024/technology (accessed Oct. 02, 2025).
3. Microsoft Learn, ".NET Microservices. Architecture for Containerized .NET Applications," *Microsoft Learn*, 2023. https://learn.microsoft.com/en-us/dotnet/architecture/microservices/ (accessed Oct. 02, 2025).
4. L. F. Al-Qora'n and A. A.-S. Ahmad, "Modular Monolith Architecture in Cloud Environments: A Systematic Literature Review," *Future Internet*, vol. 17, no. 11, pp. 496–496, Oct. 2025, doi: https://doi.org/10.3390/fi17110496.
5. S. F. Nagib, "Separation of Responsibilities in Practice: Comparison between N-Tier Architecture and Clean Architecture," *Zenodo*, vol. 2, no. 14, pp. 1–8, 2025, doi: https://doi.org/10.5281/zenodo.15164674.
6. M. T. Valente, "Building Software with a Clean Architecture," *Softengbook*, 2025. https://softengbook.org/articles/clean-architecture (accessed Oct. 05, 2025).
7. O. Dieste *et al.*, "The role of slicing in test-driven

development," *Arxiv*, Jul. 2024, doi: https://doi.org/10.48550/arxiv.2407.13258.

8. O. Lytvynov and D. Hruzin, "Decision-making on Command Query Responsibility Segregation with Event Sourcing architectural variations," *Technology audit and production reserves*, vol. 4, no. 2(84), pp. 37–59, Aug. 2025, doi: https://doi.org/10.15587/2706-5448.2025.337168.
9. I. Ibrahimi and D. Moudilos, "Model Slicing on Low-code Platforms," *Zenodo*, Dec. 2022, doi: https://doi.org/10.5281/zenodo.8160946.