INTERNATIONAL JOURNAL OF NOVEL RESEARCH AND DEVELOPMENT (IJNRD)
© 2026 IJNRD | Volume 11, Issue 11, November 2026 | ISSN: 2456-4184 | IJNRD.ORG

CLEAN ARCHITECTURE VS VERTICAL SLICE ARCHITECTURE

A Comparative Analysis for Enterprise .NET Systems

Manish Kumar
Sr. Consultant, CGI
Wilmington, Delaware, USA

Abstract : Enterprise .NET systems are constantly faced with a design dilemma: should they be structured as concentric layers (Clean Architecture, CA) or by feature (Vertical Slice Architecture, VSA)? Most of the advice on this topic is opinion, conference talks, or starter templates, with little quantitative evidence on the trade-offs. We survey the relevant literature, from the early days of modularity to modern DDD, team topologies, and architecture erosion, and then present a small, reproducible measurement. We generated an order management service in both styles, with 3, 10, and 30 features, and ran 5 typical change requests against each. We then simulated how often two independent changes would intersect in the same file. At 10 features, the CA version had 47 source files, versus 13 for VSA. The average change in CA touched 3.6 files, versus 1.6 in VSA. Across the five changes, CA touched 9 shared ("hotspot") files, versus 3 in VSA. In CA, parallel changes intersected ~32% of the time, regardless of size. In VSA, this rate decreased from 21% to 13% as the size increased. VSA had far more boilerplate duplication across slices, while CA provided a safe home for business rules. We conclude there is no clear winner. A hybrid approach, with slices for the application flow and a protected domain core for common rules, scored well in our weighted decision matrix for two enterprise profiles. We close with a checklist and the matrix itself**.**

*Keywords**: Clean Architecture; Vertical Slice Architecture; .NET; software architecture; modularity; coupling; cohesion; domain-driven design; maintainability; change impact.***

INTRODUCTION

Most enterprise .NET services start with a template. For a while now, the most popular templates have used the Clean Architecture approach: separate projects for Domain, Application, Infrastructure, and web API, with dependencies always inward to the business rules (Martin, 2017; Taylor, n.d.; Smith, n.d.). It's easy to see the appeal. It allows the business logic to be decoupled from frameworks and databases. You can test use cases without having to stand up infrastructure.

But a lot of teams have started pushing back. Their issue isn't that Clean Architecture is bad. It's just expensive in practice. If you want to add a new endpoint, you might have to edit a controller, a request, a command, a handler, a validator, a mapping profile, a repository interface, and its implementation. These files often live in four different projects. Bogard (2018) suggested a different approach called Vertical Slice Architecture. It builds around requests, puts everything a request needs together, and lets the slice decide how much abstraction it actually needs.

The debate is hot, but there's not much evidence. This paper tries to add some. Section 2 describes the method, Section 3 reviews the literature, Section 4 presents the results and a decision aid, and Section 5 concludes.

MATERIALS AND METHODOLOGY.

This study is a combination of structured literature review and controlled measurement. For the literature review, we gathered peer-reviewed literature on modularity, coupling, software evolution, architecture erosion, and the primary sources for each architectural approach. Since much of the literature on CA and VSA is found in books, blogs, and templates, rather than journals, we treat these sources as practitioner evidence and note this where appropriate.

For the measurement, we built a Python generator that generates the same order management service in both styles. We used modern .NET idioms, like MediatR request handlers, FluentValidation validators, EF Core, and minimal APIs. The CA version uses the typical 4 project approach, with a repository interface, mapping profile, and controller. The VSA version puts each feature into a single file with its command, validator, handler, and endpoint, on a thin shared kernel (Order entity, DbContext, Program.cs). We generated each version with 3, 10, and 30 features.

We then ran 5 change requests against each version and diffed the file trees: (S1) Add a new feature; (S2) Add a field to a feature; (S3) Change a validation rule; (S4) Replace a query with a faster, hand-written read; (S5) Add audit logging to all commands. For each change, we measured the number of files changed, the number of projects touched, and the number of shared files (files that many features depend on) touched. Finally, we simulated 20,000 independent random change pairs for each configuration and measured how often a pair would intersect in at least 1 common file. This is a proxy for merge conflicts in parallel teams (Gall et al., 1998; Tornhill, 2015). We provide the generator and results with the paper so readers can reproduce the figures.

LITERATURE REVIEW.

3.1 Foundations: modularity, coupling and cohesion

Both styles have their roots in the same 1970s vocabulary. Parnas (1972) suggested we should break a system down into modules based on the design decisions most likely to change. This way, any change is confined to a single module. Stevens, Myers and Constantine (1974) introduced the concepts of coupling and cohesion as the two forces a good design should balance. Later, Martin (2003) re-expressed these in the form of package principles. The Common Closure Principle, "classes that change together belong together", is the strongest argument for VSA. The Dependency Inversion Principle, "depend on abstractions, not concretions", is the core of CA.

3.2 The layered and concentric architecture lineage

Clean Architecture is the latest step in a long line of designs that put business rules at the center (Figure 1). Classic n-tier applications stacked presentation, business and data layers. Cockburn (2005) reframed this as Hexagonal Architecture, or Ports and Adapters, in which the application talks to the outside world only through ports. Palermo (2008) described Onion Architecture with the domain model at the core. Martin (2012, 2017) combined these into Clean Architecture and its single Dependency Rule: source code dependencies may only point inward.

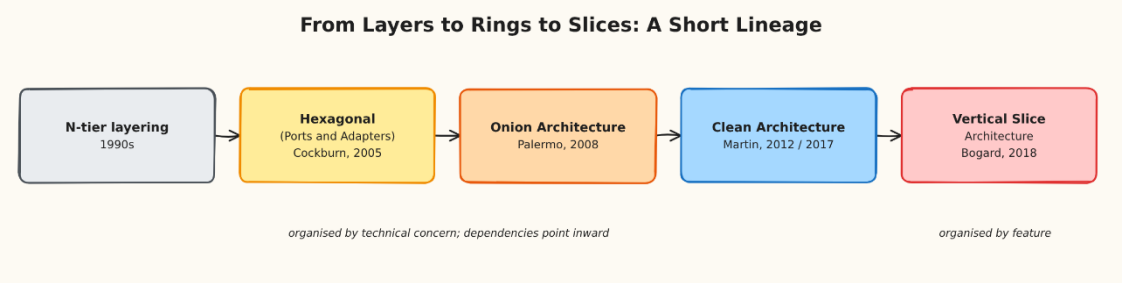

Figure 1. Layered designs to Clean Architecture to Vertical Slice Architecture.

.

3.3 Clean Architecture: claims and criticisms

Its claims include independence from the framework, UI and database, and easy unit testing of business rules (Martin, 2017). Microsoft's guidance for ASP.NET Core apps recommends this style for complex apps (Smith, 2023). Practitioners often complain about the ceremony: lots of small types per feature, mapping code between layers, and interfaces with exactly one implementation. Fowler (2015) also pointed out that layering by technical concern at the top level can make it harder to see the domain-oriented modules. If you have to touch every layer to make a change to a single feature, that's the "shotgun surgery" smell, which cohesion is supposed to prevent.

3.4 Vertical Slice Architecture: origins and principles

Bogard (2018) described VSA as focusing on distinct requests. Each slice should encapsulate everything from front end to back end, minimizing coupling between slices and maximizing coupling within a slice. VSA has roots in "package by feature" and command-query responsibility segregation (Young, 2010), which uses separate models for writing and reading. In .NET, VSA is typically implemented with a mediator library. Each request has its own file or folder. Each slice can use a rich domain model, a simple EF Core query, or raw SQL, depending on the needs of the request. The main criticisms of VSA are duplication between slices and the danger of business rules leaking into handlers, rather than being shared.

3.5 DDD: common ground

Both camps mention DDD. Evans (2003) and Vernon (2013) discuss bounded contexts, aggregates that enforce invariants, and a shared language between developers and domain experts. CA has a dedicated, protected ring for the domain model. VSA sees the domain model as something used by slices as needed. If a slice enforces invariants, it will call into aggregates. If a reporting slice doesn't need to, it can just skip them. So the difference is less about whether the domain is modeled and more about where the domain model lives and who can bypass it.

3.6 Change, maintainability and coupling measurement

Lehman (1980) argued that systems in use must change and that their structure deteriorates unless effort is put into maintaining it. Change impact is therefore a reasonable yardstick for comparing architectures. Gall, Hajek and Jazayeri (1998) introduced logical coupling, files that change together in the version history, even without a direct code dependency. Tornhill (2015) popularized the idea of hotspot analysis, which looks at the frequency of change and the size of the files to find where change is concentrated. To our knowledge, neither of these approaches has been used in a published, head-to-head comparison between CA and VSA.

3.7 Testability and test strategy

CA is designed for the classic test pyramid (Cohn, 2009; Fowler, 2012). It favors many fast unit tests around use cases, with infrastructure swapped out for test doubles, and fewer integration tests. VSA practitioners often prefer tests per slice, running against a real database, which is more in line with the "testing trophy" idea that integration tests give the best value (Dodds, 2021). The trade-off is clear. CA's unit tests are fast and precise, but broken wiring can still pass. VSA's slice tests will catch wiring and query issues, but they're slower and require more infrastructure.

3.8 Architecture erosion and conformance

De Silva and Balasubramaniam (2012) surveyed how intended architectures erode as developers make expedient changes, and how tools can detect and prevent it. CA is better off here. It has explicit, checkable rules. For example, architecture tests can

fail the build when Domain references Infrastructure. VSA has fewer rules to break, but the erosion is different. It's more of a drift between slices, where they solve the same problem in slightly different ways.

3.9 Team and organizational fit

Conway (1968) noted that systems reflect the communication structure of the organizations that build them. MacCormack, Rusnak and Baldwin (2006) showed empirical support for this "mirroring" in real codebases. Skelton and Pais (2019) suggest teams should be aligned to streams of business change. Feature-aligned teams are a natural fit for vertical slices. Layered architectures tend to work well with specialists per layer, or a platform team that owns shared rules.

3.10 Hybrid approaches

Many enterprise teams mix the two in practice. That is, they have vertical slices for the application flow, but a protected domain project for the aggregates and invariants. Sometimes they even use a modular monolith, but split by bounded contexts (Grzybek, 2019). While this mix is quite common in practitioner literature, it is almost never mentioned in the research literature. Instead, the research literature seems to view the two styles as competitors. Our review found no studies that compare a hybrid with the pure styles.

RESULTS AND DISCUSSION

4.1 Overview

Figure 2 shows the same application arranged in both ways. CA divides the system horizontally based on technical roles. VSA divides the system vertically based on requests, with only a thin kernel that is shared

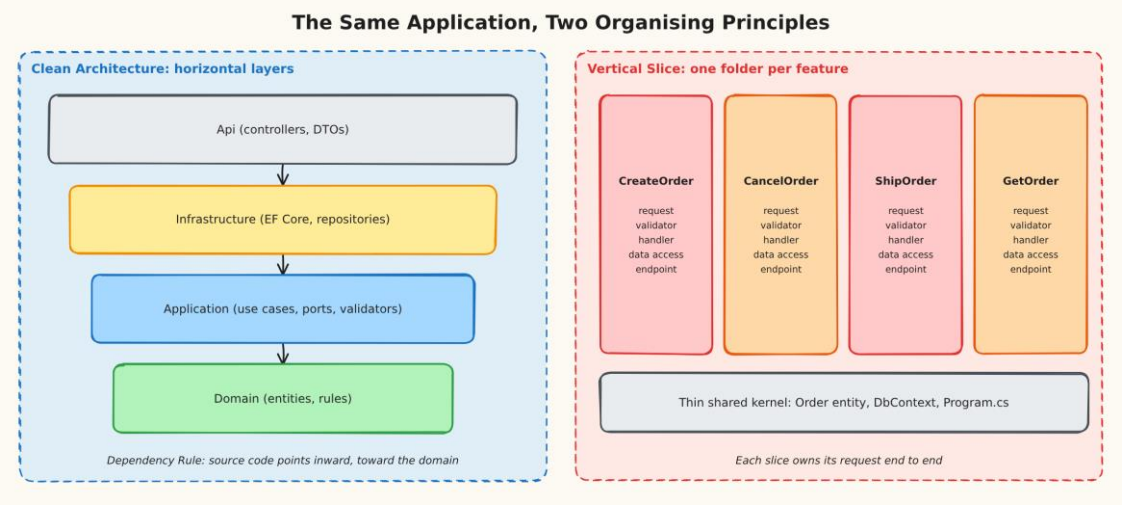

Figure 2. Same order-management service in Clean Architecture (left) and Vertical Slice Architecture (right).

The structural cost of each style becomes obvious as you add features (Figure 3). Each new feature added about 4.0 files in the CA version, but only 1.0 file in the VSA version. So with 30 features, the CA codebase has 127 files and the VSA codebase has 33. There was less difference in lines of code (579 for CA versus 437 for VSA). This is because VSA can pack the same pieces into fewer, but longer files

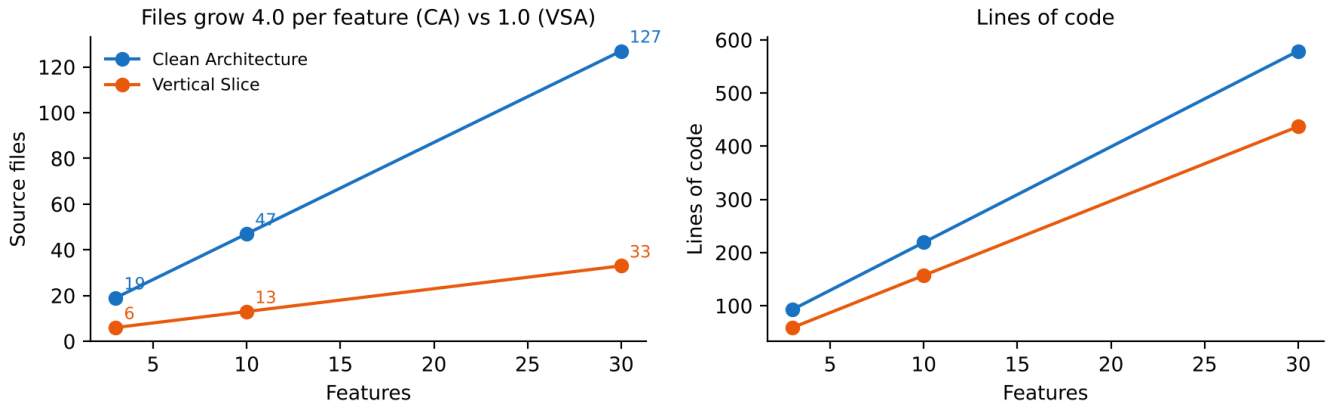

Figure 3. Measured: source files and lines of code as the number of features increases.

4.2 Comparison

Figure 4 shows the files that were touched for each change request in the application with 10 features. The biggest difference between the styles was when adding a feature (S1). CA touched 8 files in three different projects. Four of these files were shared files. VSA only touched one new file. For optimizing one query (S4), CA touched four files. This is because the repository interface, implementation and dependency registration all needed to be changed. VSA only touched two files. Changing a validation rule (S3) and adding audit logging to all commands (S5) cost the same for both styles. The way cross-cutting behavior was handled was the same. One pipeline behavior was registered once.

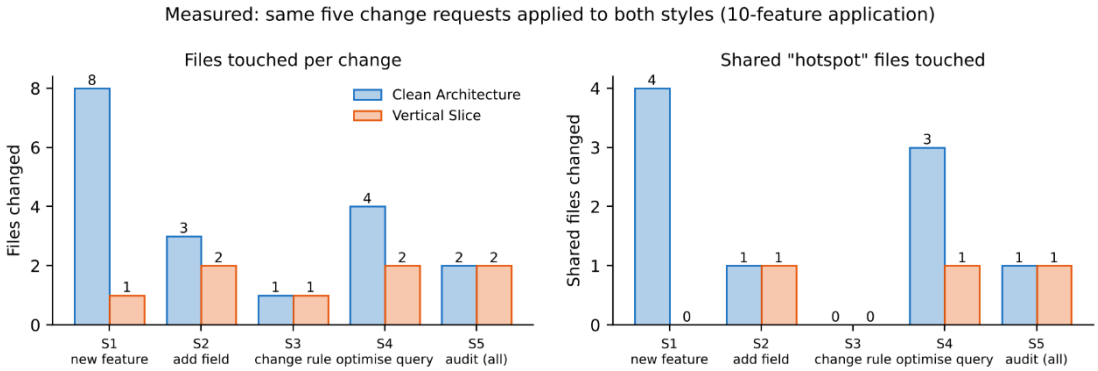

Figure 4. Measured: files and shared files touched by five change requests in the 10-feature application.

Shared files matter because they are where parallel work can collide. Figure 5 shows the result of the conflict simulation. In CA, about one out of three independent changes touched a common file. This rate did not improve as the application grew. This is because every new feature still had to edit the same controller, repository and mapping files. In VSA, the rate decreased from 21% with 5 features to 13% with 80 features. This is because most of the changes occur within the slice. The shared kernel is small.

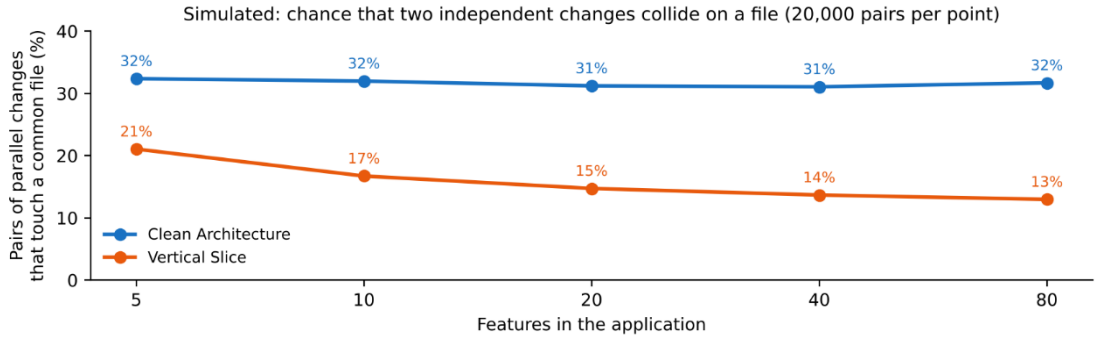

Figure 5. Simulated: probability that two independent changes touch a common file.

Measure (10-feature application)	Clean Architecture	Vertical Slice	Favors
Source files	47	13	VSA
Files added per new feature	4.0	1.0	VSA
Shared files in the codebase	7	3	VSA
Average files touched per change (S1-S5)	3.6	1.6	VSA
Average projects touched per change	2.4	1.6	VSA
Shared files touched across S1-S5	9	3	VSA
Parallel-change collision rate (10 / 80 features)	32% / 32%	17% / 13%	VSA
Repeated boilerplate lines	7.6%	65.9%	CA
Single home for shared business rules	Yes (Domain ring)	Optional	CA
Checkable dependency rules	Strong	Weak	CA

Table 1. Measured and structural comparison. The first eight rows come from the generated codebases; the last two summarize the literature.

VSA is favored on change locality, but that measure does not capture the purpose of CA. VSA has much more repeated boilerplate: about 66% of non-trivial lines were repeated, compared with 8% in CA. This is because each slice has its own command, validator and handler boilerplate. The number is inflated because our generated code is intentionally uniform. However, the trend is consistent with the criticisms in the literature. More importantly, our scenarios contained no shared business rules, such as a discount policy that 10 slices must all respect. In CA, there is one obvious home for such a rule. In VSA, it is easy to copy the rule to multiple handlers

4.3 Hybrid architecture

The results suggest that the two styles address different problems. VSA addresses the problem of "how do we structure the application flow so that changes remain local?". CA addresses the problem of "how do we protect the rules that must never be broken?". A hybrid (Figure 6) would have slices for every request and a small protected core domain for aggregates, value objects and invariants. Slices that change state interact with this core domain. Read-only slices, such as reports and lookups, can query a read model directly. This is in the spirit of CQRS (Young, 2010). Infrastructure remains behind ports. The shared kernel in this

approach only grows when there is a truly shared rule. Architecture tests can still ensure that the core does not depend on the infrastructure.

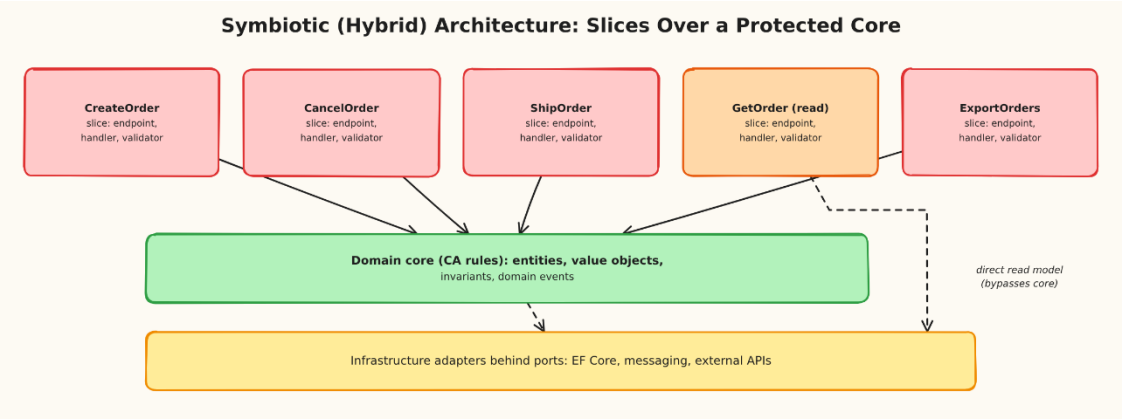

Figure 6. A hybrid: feature slices over a protected domain core, with direct read models for queries.

.

4.4 Decision

We translate our findings into a short checklist, weighted matrix and decision flow. The checklist is intended for architecture review. If most items in one column describe your situation, lean toward that style.

Lean toward Clean Architecture if...	Lean toward Vertical Slice if...
Many features share complex, long-lived business rules	Most features are independent requests with little shared logic
The domain is regulated and invariants must be provably enforced	Requirements change often and speed of change matters most
A central team owns the domain model	Several feature teams work in parallel on one codebase
You want build-time checks of dependency direction	You want each request readable in one place
Developers rotate often and need one consistent structure	Endpoints differ widely (CRUD, reports, integrations)

Table 2. Architecture review checklist.

Table 3 scores each style from 1 to 5 on eight criteria. The scores are our judgment, anchored in the measurements above and the literature in Section 3. We then weight the criteria for two enterprise profiles: (A) a regulated domain with complex rules and three teams, for example payments or lending; and (B) a fast-moving, mostly CRUD retail API with six feature teams.

Criterion	CA	VSA	Hybrid	Weight A	Weight B
Protects shared domain rules	5	2	4	25	5
Change locality	2	5	4	10	25
Parallel team throughput	2	5	4	10	25
Onboarding consistency	4	3	4	15	10
Unit testability of rules	5	3	4	20	10
Low ceremony	2	5	3	5	15
Enforceable structure	4	3	4	10	5
Duplication control	5	2	4	5	5
Weighted score, profile A (regulated)	4.00	3.20	3.95	100	
Weighted score, profile B (fast-moving)	2.90	4.20	3.85		100

Table 3. Weighted decision matrix. Scores (1-5). Weights sum to 100 per profile.

CA won profile A (4.00 versus 3.20 for VSA). VSA won profile B (4.20 versus 2.90). Hybrid is within 0.05 of the winner in profile A. Hybrid is within 0.35 of the winner in profile B. If the profile is uncertain or likely to change, the hybrid is the choice with the smallest worst-case shortfall (0.35, against 0.80 for VSA and 1.30 for CA). Figure 7 summarizes this in a decision flow.

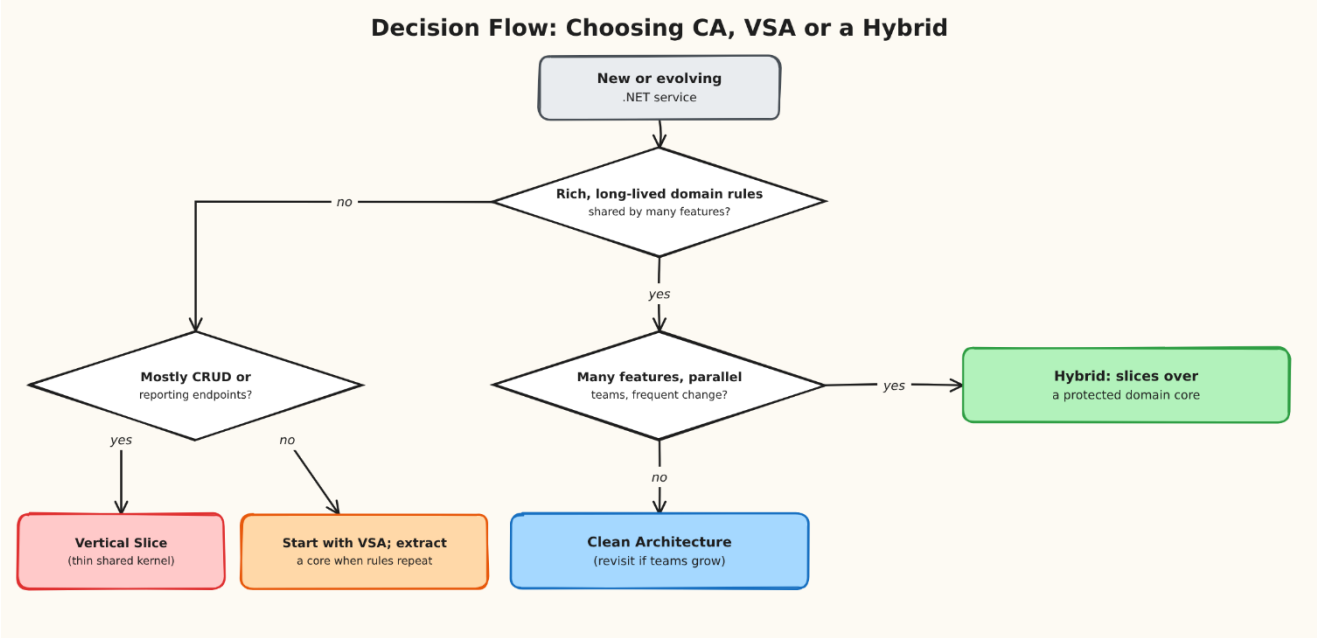

Figure 7. Decision flow for choosing Clean Architecture, Vertical Slice Architecture or a hybrid.

.

CONCLUSION

Clean Architecture and Vertical Slice Architecture are not different answers to the same question. CA protects business rules and makes dependency direction checkable. VSA keeps change local and allows parallel teams to work without stepping on each other's toes. We put numbers to the trade-off by measuring equivalent .NET codebases. VSA had about a quarter of the files. It touched less than half as many files per change. It roughly halved the chance of parallel changes colliding. CA repeated much less code. In principle it also offers a single, enforceable home for shared rules, although our scenarios contained none.

For enterprise teams, the practical advice is to structure the application flow by feature and protect only the domain rules that are truly shared. If the domain is rich and regulated, and there is one team that owns it, then use pure CA. If the features are mostly independent and change fast, then use pure VSA. This study has clear limitations: the code is generated rather than real, only five change types were examined, our scenarios contained no shared business rules, the decision-matrix scores are judgments rather than measurements, and the conflict model does not account for line-level merging. Future work should measure co-change coupling, lead time and defect rates in real repositories that have used each style for several years;

REFERENCES

[1] Bogard, J. (2018, April 19). Vertical slice architecture. jimmybogard.com.
[2] Cockburn, A. (2005). Hexagonal architecture. alistair.cockburn.us.
[3] Cohn, M. (2009). Succeeding with agile: Software development using Scrum. Addison-Wesley.
[4] Conway, M. E. (1968). How do committees invent? Datamation, 14(4), 28-31.
de Silva, L., & Balasubramaniam, D. (2012). Controlling software architecture erosion: A survey. Journal of Systems and Software, 85(1), 132-151.
[5] Dodds, K. C. (2021). The testing trophy and testing classifications. kentcdodds.com.
[6] Evans, E. (2003). Domain-driven design: Tackling complexity in the heart of software. Addison-Wesley.
[7] Fowler, M. (2012). TestPyramid. martinfowler.com.
[8] Fowler, M. (2015). PresentationDomainDataLayering. martinfowler.com.
[9] Gall, H., Hajek, K., & Jazayeri, M. (1998). Detection of logical coupling based on product release history. Proceedings of the International Conference on Software Maintenance (ICSM), 190-198.
[10] Grzybek, K. (2019). Modular monolith: A primer. kamilgrzybek.com.
[11] Lehman, M. M. (1980). Programs, life cycles, and laws of software evolution. Proceedings of the IEEE, 68(9), 1060-1076.
[12] MacCormack, A., Rusnak, J., & Baldwin, C. Y. (2006). Exploring the structure of complex software designs: An empirical study of open source and proprietary code. Management Science, 52(7), 1015-1030.
[13] Martin, R. C. (2003). Agile software development: Principles, patterns, and practices. Prentice Hall.
[14] Martin, R. C. (2012, August 13). The clean architecture. blog.cleancoder.com.
[15] Martin, R. C. (2017). Clean architecture: A craftsman's guide to software structure and design. Prentice Hall.
[16] Palermo, J. (2008). The onion architecture: Part 1. jeffreypalermo.com.
[17] Parnas, D. L. (1972). On the criteria to be used in decomposing systems into modules. Communications of the ACM, 15(12), 1053-1058.
[18] Skelton, M., & Pais, M. (2019). Team topologies: Organizing business and technology teams for fast flow. IT Revolution.
[19] Smith, S. (2023). Architecting modern web applications with ASP.NET Core and Azure. Microsoft. learn.microsoft.com.
[20] Smith, S. (n.d.). CleanArchitecture [Solution template]. github.com/ardalis/CleanArchitecture.
Stevens, W. P., Myers, G. J., & Constantine, L. L. (1974). Structured design. IBM Systems Journal, 13(2), 115-139.

Taylor, J. (n.d.). CleanArchitecture [Solution template]. github.com/jasontaylordev/CleanArchitecture.
Tornhill, A. (2015). Your code as a crime scene. Pragmatic Bookshelf.
Vernon, V. (2013). Implementing domain-driven design. Addison-Wesley.
Young, G. (2010). CQRS documents. cqrs.files.wordpress.com..

Copyright & License:

© Authors retain the copyright of this article. This work is published under the Creative Commons Attribution 4.0 International License (CC BY 4.0), permitting unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.