AYI NEDJIMI Consultants

Architecture Vertical Slice + Clean Lite : Guide 2026

Catégorie : Articles Techniques Lecture : 12 min Publié le : 23/03/2026 Auteur : Ayi NEDJIMI

Guide 2026 : VSA + Clean Architecture Lite pour équipes 3–12 devs. Structure dossiers, 5 règles d'or, matrices langages/stacks et prompt maître IA.

La combinaison **Vertical Slice Architecture (VSA)** et **Clean Architecture Lite** s'est imposée en 2026 comme le standard de référence pour les projets professionnels à moyen et long terme, avec des équipes de 3 à 12 développeurs et une durée de vie de 2 à 8 ans. Cette approche hybride fusionne deux philosophies complémentaires : VSA découpe l'application en tranches verticales autonomes par fonctionnalité, tandis que Clean Architecture Lite isole le cœur métier des détails techniques à l'intérieur de chaque tranche. Ce modèle offre le meilleur ratio entre vélocité de développement et maintenabilité, avec un avantage inattendu : les LLM modernes (Cursor, Claude, Windsurf) y sont significativement plus performants, travaillant sur un contexte regroupé par fonctionnalité plutôt que dispersé dans des couches techniques distantes. Ce guide s'adresse aux architectes logiciels et tech leads qui souhaitent adopter ce standard immédiatement.

Principes de l'Architecture Hybride VSA + Clean Lite

1. Comparatif Stratégique (Benchmark 2026)

Avant d'adopter VSA + Clean Lite, il est utile de la positionner face aux alternatives :

- **Clean Architecture pure** : excellente isolation du domaine, mais verbosité élevée et changements dispersés dans de multiples couches. Recommandée pour les systèmes à logique métier très complexe (finance, assurance).
- **Architecture Microservices** : scalabilité granulaire, mais complexité opérationnelle massive injustifiée pour moins de 15 développeurs. C'est une évolution d'une slice mature, pas un point de départ.
- **Monolithe MVC classique** : rapidité initiale, mais couplage croissant et difficulté de maintenance au-delà de 2 ans.
- **VSA + Clean Lite** : localisation maximale des changements, onboarding rapide, chaque tranche est un contexte IA-complet. **Style recommandé par défaut en 2026.**

L'avantage IA mérite une mention spéciale : les LLM modernes sont bien plus performants lorsqu'ils travaillent sur un contexte regroupé par fonctionnalité (Slice) plutôt que dispersé dans des couches techniques distantes. Chaque dossier de feature est un module autonome que l'IA peut comprendre et modifier sans effets de bord imprévus.

2. Principes Fondamentaux et Best Practices Dev

A. Organisation par Features (Slices)

- *Règle d'or* : Un développeur doit pouvoir réaliser **90% de sa tâche en restant dans un seul dossier** de fonctionnalité.
- *Cohésion* : Si deux fonctionnalités changent toujours ensemble, elles appartiennent probablement à la même Slice.

B. Le Core Minimaliste et Transversal

On évite le piège des dossiers `Shared` obèses. Le Core ne contient que :

- Cross-cutting concerns : Logger, corrélation d'IDs, télémétrie.
- Primitives de base : une classe `Result<T>` pour éviter les exceptions de flux, les interfaces `IDomainEvent`.
- Ports génériques : uniquement les abstractions strictement nécessaires (`IUnitOfWork` , `IBus`).

C. Best Practices au sein de la Slice

- *Domain "Sourd et Aveugle"* : logique métier pure. Utilisez des Value Objects pour la validation structurelle (`Price` , `Email`).
- *Result Pattern* : les Use Cases retournent un objet `Result<T>`. Les erreurs sont des données, pas des interruptions de flux.
- *Validation à deux niveaux* : Input (schéma/format) et Business (règles d'état).
- *Médiateur et Découplage* : bus internes pour découpler les déclencheurs (API, Workers) des exécuteurs (Handlers).
- *CQRS Lite* : séparation des modèles de lecture (optimisés performance) et d'écriture (optimisés intégrité).

D. La Règle de Dépendance Stricte

La dépendance doit toujours pointer vers le centre (Domain) :

- **Domain (Niveau 0)** : Cœur pur. Entities, ValueObjects, Enums, DomainServices.
- **Application (Niveau 1)** : Orchestration. UseCases, DTOs, Interfaces (Ports).
- **Infrastructure (Niveau 2)** : Détails. Persistence, Clients API, File System.
- **API / Presentation (Niveau 2)** : Entrée. Controllers, Middlewares, Mapping.

Structure et Organisation du Code

3. Structure de Dossiers Détaillée (Standard 2026)

La structure canonique d'un projet VSA + Clean Lite garantit qu'un nouveau développeur identifie instantanément les capacités métier de l'application en lisant simplement `src/features/` :

```
src/
├── core/                       # Commun à toutes les tranches (LÉGER)
│   ├── domain/                 # Result.ts, Entity.ts, AggregateRoot.ts
│   ├── application/            # IPipelineBehavior.ts, CommonDTOs.ts
│   └── infrastructure/         # DatabaseContext.ts, HttpClientBase.ts
├── features/                   # Vertical Slices
│   ├── ordering/               # Exemple : Slice Commandes
│   │   ├── domain/             # ── LOGIQUE MÉTIER PURE ──
│   │   │   ├── entities/       # Order.ts (Aggregate Root)
│   │   │   ├── value-objects/  # Address.ts, OrderId.ts
│   │   │   ├── services/       # Logique complexe multi-entités
│   │   │   └── exceptions/     # Erreurs métier (Business rules)
│   │   ├── application/        # ── ORCHESTRATION ──
│   │   │   ├── commands/       # Write (CreateOrderCommand + Handler)
│   │   │   ├── queries/        # Read  (GetOrderQuery + Handler)
│   │   │   ├── common/         # IOrderRepository.ts (Port), Mappers
│   │   │   └── events/         # Intégration / Notification events
│   │   ├── infrastructure/     # ── DÉTAILS TECHNIQUES ──
│   │   │   ├── persistence/    # Implémentation Repository (SQL, Mongo)
│   │   │   └── gateway/        # Clients API externes
│   │   └── api/                # ── POINT D'ENTRÉE ──
│   │       ├── controllers/    # Controller dédié à la feature
│   │       └── requests/       # Schémas de validation (Zod, Fluent)
│   └── payments/               # (Même structure interne)
└── entrypoints/                # Setup global & Configuration
```

Chaque dossier de feature est un module isolé, testable et déployable indépendamment si nécessaire. Pour les projets containerisés avec Docker/Kubernetes, cette structure facilite également la migration éventuelle vers des microservices — chaque slice devenant un service distinct si elle grossit suffisamment.

4. Les 5 Règles d'Or Inviolables (Maintenance 2–8 ans)

1. **Pas d'import cross-feature de couches hautes** : une feature ne doit jamais importer le Handler ou le Repository d'une autre. Utilisez des Events pour communiquer entre features.
2. **Duplication > Mauvaise Abstraction** : ne partagez pas de DTOs entre tranches. La duplication permet l'indépendance — une abstraction prématurée crée un couplage destructeur sur le long terme.
3. **Le Domain ne lance pas d'exception de flux** : utilisez le pattern `Result<T>`. Les erreurs sont des cas métier, pas des exceptions de programmation.
4. **Zéro Logique Métier dans les Controllers** : le controller est un simple passe-plat qui valide l'entrée et délègue au Handler via le médiateur.
5. **Persistence Ignorance (Lite)** : l'application définit ses besoins via des interfaces (Ports), l'infrastructure s'y plie via les Adapters. Le Domain ne connaît pas la BDD.

Intelligence Artificielle et Architecture

5. Le Prompt Maître "Expert 2026" (Pour IA de Codage)

Utilisez ce prompt pour une génération de haute précision avec Cursor, Windsurf ou Claude :

```
Agis en tant qu'Expert Architecte Logiciel Senior.
Génère la feature : [NOM_FEATURE] en respectant strictement
l'architecture Vertical Slice + Clean Architecture Lite.

1. Structure des dossiers :
   Crée les sous-dossiers domain/, application/,
   infrastructure/ et api/ au sein de la feature.

2. Domain (Cœur Métier) :
   - Crée un Aggregate Root avec des méthodes expressives
     (ex: .ship() au lieu de .setStatus('shipped')).
   - Utilise des Value Objects pour les données sensibles
     (Emails, Montants, Identifiants).
   - La logique de validation métier doit résider ici.

3. Application (Use Cases) :
   - Implémente le pattern CQRS : sépare une Commande
     (écriture) et une Query (lecture).
   - Chaque Handler doit retourner un objet Result<T>.
   - Définit les interfaces (Ports) pour la persistance
     et les services tiers.

4. Infrastructure et API :
   - Implémente le Repository avec [DATABASE_TYPE].
   - Côté API, utilise [ZOD/FLUENT VALIDATION] pour
     valider le contrat d'entrée avant le Handler.
   - Assure le mapping Entités ↔ DTOs
     (pas de fuite du Domain vers l'extérieur).

5. Qualité et Tests :
   - Génère un test unitaire pour la logique du Domain.
   - Génère un test d'intégration simulant un appel API
     complet pour cette feature.
   - Nommage explicite + commentaires sur les règles
     métier complexes.
```

6. Pourquoi ce Choix pour un Projet de 7 Ans ?

- **Anti-Fragilité** : supprimer une feature n'impacte pas les autres. L'isolation des tranches signifie zéro propagation des régressions lors des suppressions ou refactorings majeurs.
- **Onboarding Rapide** : un nouveau développeur lit l'arborescence `features/` et comprend immédiatement les capacités métier. Pas besoin de décoder des couches techniques abstraites.
- **IA-Ready** : chaque dossier de feature est un module autonome compréhensible par un LLM sans contexte supplémentaire — idéal pour les workflows Cursor/Claude en 2026.

7. Matrice Langages × Styles Architecturaux

Cette matrice évalue l'adéquation de chaque langage/framework avec les principaux styles architecturaux, en tenant compte de l'écosystème, des patterns natifs et de la maturité des outils. **★★★★★ = Excellent support natif — ★☆☆☆☆ = Inadapté**

Langage / Framework	VSA + Clean Lite	Clean/ Onion Pur	Microservices	CQRS/ES	Event-Driven	Serverless
C# / .NET 8+	★★★★★	★★★★★	★★★★★	★★★★★	★★★★☆	★★★★★
TypeScript / NestJS	★★★★☆	★★★★☆	★★★★★	★★★☆☆	★★★★★	★★★★★
Python / FastAPI	★★★★★	★★★★☆	★★★☆☆	★★★☆☆	★★★★☆	★★★★☆
Python / Django	★★★☆☆	★★★☆☆	★★☆☆☆	★★☆☆☆	★★☆☆☆	★★★☆☆
Rust / Actix-Web	★★★★☆	★★★☆☆	★★★★☆	★★★☆☆	★★★★☆	★★★☆☆
Go / Gin	★★★★☆	★★★☆☆	★★★★★	★★★☆☆	★★★★★	★★★★☆
Java / Spring Boot	★★★★★	★★★★★	★★★★★	★★★★★	★★★★☆	★★★★☆
Kotlin / Ktor	★★★★☆	★★★★☆	★★★★★	★★★★★	★★★★★	★★★★☆
Elixir / Phoenix	★★★★☆	★★★☆☆	★★★★☆	★★★★☆	★★★★★	★★★☆☆
PHP / Laravel	★★★☆☆	★★★☆☆	★★★☆☆	★★☆☆☆	★★☆☆☆	★★★☆☆

8. Matrice Stack × Architecture Recommandée

Cette matrice associe chaque stack à l'architecture la plus naturelle. La colonne "Architecture Conseillée" représente le style le plus naturel — toute stack peut adopter n'importe quelle architecture, avec un coût d'adaptation variable. Voir aussi les guides d'architecture sécurité avancée.

Stack	Architecture Conseillée	Cas d'usage	Points forts	Limites
C# / .NET 8+	VSA + Clean Lite	Enterprise, B2B SaaS	MediatR, FluentValidation, tooling Visual Studio/Rider, Azure natif	Verbosité C#
TypeScript / NestJS	VSA + Clean Lite	APIs full-stack, BFF	Full-stack unifié, serverless mature, structure modulaire NestJS	Écosystème npm instable
Python / FastAPI	VSA + Clean Lite	APIs IA/ML, data pipelines	Écosystème ML inégalé, Pydantic, Depends() pour IoC	Pas de compilateur
Python / Django	Monolithe Modulaire	CMS, e-commerce, CRUD	Productivité maximale, admin auto-générée	Peu adapté aux architectures distribuées
Rust / Actix-Web	VSA + Clean Lite	Systèmes critiques, edge, HFT	Performances brutes, sécurité mémoire compilée	Courbe d'apprentissage élevée
Go / Gin	Microservices	Cloud-native, DevOps, infra	Binaires légers, goroutines, interfaces implicites	DDD verbeux sans génériques avancés
Java / Spring Boot	Clean Architecture / Onion	Enterprise, finance, assurance	Spring Modulith, Axon CQRS/ES, écosystème le plus complet	JVM overhead, démarrage lent
Kotlin / Ktor	CQRS / Event Sourcing	Backend moderne, Android API	Coroutines, data classes immutables, interop Java	Communauté plus petite
Elixir / Phoenix	Event-Driven	Temps réel, IoT, chat	BEAM tolérance aux fautes, millions de processus, LiveView	Courbe d'apprentissage, recrutement difficile
PHP / Laravel	Monolithe Modulaire	Startups, CMS, MVP	Rapidité, hébergement économique, communauté massive	Réputation architecturale mitigée

9. Justification des Choix — Notes Révisées

Rust / Actix-Web — VSA + Clean Lite : ★★★☆☆ → ★★★★☆
Le système de modules et crates de Rust impose des frontières entre slices à la compilation. Un module Rust rend ses types internes privés par défaut, et le compilateur *enforce* la règle de dépendance : si `domain/` ne déclare pas de dépendance vers `infrastructure/`, l'import est physiquement impossible. Les traits jouent le rôle de Ports/Interfaces de manière naturelle. La note initiale de 3/5 était pénalisée par l'absence de framework VSA-ready (pas d'équivalent MediatR), mais la robustesse structurelle compilée compense largement.

Python / FastAPI — VSA + Clean Lite : ★★★★☆ → ★★★★★
FastAPI mérite la note maximale. Les routers sont des slices naturelles, Pydantic impose un typage strict aux frontières, et le système `Depends()` est un mécanisme d'injection de dépendances natif. Avec les type hints et les Protocols Python 3.12+, le pattern Ports/Adapters est propre et idiomatique. L'absence de compilateur rend la discipline architecturale encore plus précieuse — c'est la structure qui protège.

Go / Gin — VSA + Clean Lite : ★★★☆☆ → ★★★★☆
Les packages Go imposent des frontières naturelles (un package = un dossier = une responsabilité), et les interfaces implicites sont un avantage pour VSA : on définit l'interface côté consommateur (Domain), et l'Infrastructure l'implémente sans le savoir. C'est le pattern Ports/Adapters dans sa forme la plus pure.

Java / Spring Boot — VSA + Clean Lite : ★★★★☆ → ★★★★★
Spring Modulith est un module officiel de Spring explicitement conçu pour Vertical Slice. Il fournit la vérification des frontières entre modules, l'intégration avec les événements de domaine, et la documentation automatique de l'architecture. Avec Spring Modulith, Java est aussi bien outillé que C#/.NET pour VSA + Clean Lite.

Kotlin — CQRS/ES : ★★★★☆ → ★★★★★
Axon Framework supporte Kotlin nativement, et les coroutines se marient parfaitement avec le traitement d'événements asynchrones. Les data classes Kotlin sont idéales pour modéliser les événements (immuables par défaut).

Elixir / Phoenix — VSA + Clean Lite : ★★★☆☆ → ★★★★☆
Les Contexts Phoenix sont des slices par design. Quand on génère un projet Phoenix, le framework propose déjà un découpage par domaine métier (Accounts, Catalog, Orders…). Les Behaviours Elixir servent de Ports naturels.

10. Guide des Styles Architecturaux

VSA + Clean Architecture Lite — Découpage par fonctionnalité métier avec isolation du domaine à l'intérieur de chaque tranche. Couplage inter-features interdit (Events uniquement). Idéal pour 3–15 devs, 2–10 ans. Avantages : localisation maximale, IA-ready, testabilité élevée. Limites : duplication volontaire, discipline d'équipe requise.

Clean Architecture / Onion Architecture (Pur) — Structure en cercles concentriques. Domain au centre, entouré par Application, puis Infrastructure et UI. Dépendances toujours vers le centre. Idéal pour les systèmes à logique métier complexe (finance, assurance, santé). Avantages : isolation totale du domaine, mocks faciles, indépendance des frameworks. Limites : verbosité, surcoût pour les CRUD simples.

Architecture Microservices — Services indépendants, chacun déployable séparément, avec sa propre BDD. Communication par API REST, gRPC ou messages asynchrones (Kafka, RabbitMQ). Idéal pour 15+ devs avec infrastructure cloud mature. Avantages : scalabilité granulaire, liberté technologique, résilience. Limites : complexité opérationnelle massive, transactions distribuées, debugging difficile.

CQRS / Event Sourcing — Séparation des modèles de lecture et d'écriture. L'ES stocke l'historique complet des événements. Idéal pour les systèmes nécessitant un audit trail complet. Avantages : traçabilité totale, projections multiples, performances lecture optimisées. Limites : complexité significative, cohérence éventuelle.

Architecture Event-Driven — Composants communiquant via des événements asynchrones (Kafka, RabbitMQ, NATS). Idéal pour les systèmes temps réel et les intégrations inter-systèmes. Avantages : découplage maximal, scalabilité horizontale, extensibilité. Limites : debugging complexe, cohérence éventuelle.

Architecture Serverless — Fonctions éphémères (AWS Lambda, Azure Functions) déclenchées par des événements. Idéal pour les workloads sporadiques, webhooks, MVPs rapides. Avantages : zéro gestion d'infrastructure, scaling automatique, coût quasi nul au repos. Limites : cold starts, vendor lock-in.

11. Guide des Stacks Technologiques

C# / .NET 8+ — Fortement typé, orienté objet et fonctionnel (Microsoft). Frameworks clés : ASP.NET Core, Entity Framework Core, MediatR, FluentValidation, MassTransit, Dapper. Points forts : écosystème le plus complet pour VSA + Clean Architecture, tooling Visual Studio/Rider exceptionnel, support Azure natif, performances AOT. Lire la référence Clean Architecture d'Uncle Bob pour approfondir les principes fondamentaux.

TypeScript / Node.js — Typage statique sur JavaScript, modèle événementiel non-bloquant. Frameworks clés : NestJS (inspiré Angular), Fastify, Prisma (ORM type-safe), tRPC, Zod. Points forts : full-stack unifié, NestJS impose une structure modulaire proche de Clean Architecture, serverless mature.

Python / FastAPI — Interprété, dominant en ML/IA. Frameworks clés : FastAPI (async haute performance), SQLAlchemy, Pydantic, Celery, Alembic. Variante MongoDB : FastAPI + Motor async + Pydantic, idéal pour les APIs manipulant des documents JSON, l'analytics et les pipelines ML. Points forts : écosystème ML inégalé (PyTorch, TensorFlow), génération OpenAPI automatique.

Rust / Actix-Web — Compilé, sécurité mémoire garantie sans GC, performances comparables au C. Frameworks clés : Actix-Web, Axum, SQLx (queries SQL compile-time checked), Serde, Tokio. Points forts : performances brutes imbattables, latence prévisible, idéal pour l'edge computing et les systèmes critiques.

Go / Gin — Compilé par Google, conçu pour la concurrence (goroutines). Binaires statiques autonomes. Frameworks clés : Gin/Echo/Chi, GORM, Wire (DI), NATS/Kafka. Points forts : concurrence native, binaires légers, écosystème cloud-native dominant (Docker, Kubernetes, Terraform sont en Go).

Java / Spring Boot — Language enterprise sur JVM. Frameworks clés : Spring Boot, Spring Data JPA, Spring Cloud, Spring Modulith, Axon Framework (CQRS/ES). Points forts : écosystème le plus complet de l'industrie, JVM robuste, Spring Modulith pour VSA natif.

Kotlin / Ktor — Moderne sur JVM (JetBrains), 100% interopérable Java. Coroutines natives, null-safety, data classes. Frameworks clés : Ktor, Exposed (ORM type-safe), Koin, Axon. Points forts : async élégant, migration progressive depuis Java, data classes idéales pour les événements CQRS/ES.

Elixir / Phoenix — Sur BEAM VM (Erlang), conçue pour les systèmes distribués. Frameworks clés : Phoenix, Ecto, LiveView, Broadway. Points forts : tolérance aux fautes inégalée, concurrence massive, LiveView élimine le besoin de frontend JS pour le temps réel.

PHP / Laravel — Langage le plus déployé sur le web. Frameworks clés : Laravel, Eloquent ORM, Livewire, Horizon, Vapor (serverless AWS). Points forts : rapidité de développement, hébergement économique, communauté massive, excellent pour CMS et e-commerce.

> ### Points clés à retenir
>
> - **VSA + Clean Lite est le standard 2026** pour les projets 3–12 développeurs, 2–8 ans. Style recommandé par défaut.
> - **Règle d'or** : 90% d'une tâche doit se réaliser dans un seul dossier feature. Si ce n'est pas le cas, la découpe est incorrecte.
> - **5 règles inviolables** : no cross-feature import, duplication volontaire, Result pattern, no logique dans controllers, persistence ignorance.
> - **Le prompt maître** permet une génération IA de haute précision avec Cursor/Claude/Windsurf — utilisez-le systématiquement.
> - **C#/.NET, Java/Spring Boot et Python/FastAPI** sont les références absolues pour VSA en 2026.
> - **Microservices = évolution** d'une slice mature, pas un point de départ. Ne sur-engineerez pas.

Conclusion et Prochaines Étapes

VSA + Clean Architecture Lite n'est pas une mode architecturale — c'est une réponse pragmatique aux défis réels des équipes de développement en 2026 : livrer vite, maintenir longtemps, et rester efficace avec les outils IA modernes. En adoptant ce style, vous choisissez la localisation des changements plutôt que l'élégance théorique, l'indépendance des features plutôt que la factorisation prématurée.

Pour démarrer immédiatement : créez votre premier dossier `features/`, appliquez les 5 règles d'or inviolables, et utilisez le prompt maître avec votre assistant IA de codage préféré. La courbe d'apprentissage initiale est récompensée dès le premier refactoring majeur, lorsque la suppression d'une feature n'impacte pas les autres. Si vous gérez des projets d'infrastructure avec des besoins d'architecture SOC et sécurité avancée, VSA s'y applique tout aussi bien.

Questions Fréquentes

Qu'est-ce que la Vertical Slice Architecture (VSA) ?

La Vertical Slice Architecture découpe l'application par fonctionnalité métier (feature) plutôt que par couche technique. Chaque "tranche" est autonome et contient ses propres couches Domain, Application, Infrastructure et API. Cela réduit drastiquement le couplage entre modules. Règle d'or : un développeur doit pouvoir réaliser 90% d'une tâche en restant dans un seul dossier de fonctionnalité.

Quelle est la différence entre VSA + Clean Lite et la Clean Architecture pure ?

La Clean Architecture pure structure l'application en cercles concentriques (Domain au centre). Elle est verbale, avec des changements dispersés dans de multiples couches. VSA + Clean Lite combine la localisation des changements de VSA avec les principes d'isolation du domaine de Clean Architecture, appliqués à l'intérieur de chaque tranche — moins de verbosité, meilleure vélocité, domaine toujours protégé et testable.

Comment structurer un projet avec Vertical Slice Architecture ?

Structure standard 2026 : `src/features/[nom-feature]/` avec `domain/` (entités, value objects), `application/` (commands, queries, ports, events), `infrastructure/` (persistence, clients API) et `api/` (controllers, validation). Un `src/core/` léger pour les cross-cutting concerns (Logger, `Result<T>`, interfaces génériques). Les `entrypoints/` gèrent le setup global et la configuration.

Quel langage est le plus adapté à VSA + Clean Architecture Lite en 2026 ?

C# / .NET 8+ reste la référence absolue (★★★★★) grâce à MediatR et FluentValidation. Java / Spring Boot avec Spring Modulith atteint le même niveau. Python / FastAPI (★★★★★) est idéal pour les projets IA/ML. Go / Gin (★★★★☆) excelle grâce à ses interfaces implicites. Rust / Actix-Web (★★★★☆) offre une robustesse structurelle compilée pour les systèmes critiques.

Pourquoi préférer VSA à l'architecture microservices pour une équipe de 3 à 12 développeurs ?

Les microservices introduisent une complexité opérationnelle massive (réseau, transactions distribuées, debugging asynchrone) injustifiée pour moins de 15 développeurs. Pour les projets de grande envergure nécessitant une orchestration avancée, consultez notre guide sur Kubernetes et la gestion des politiques réseau. VSA + Clean Lite offre les bénéfices d'isolation sans la surcharge d'infrastructure. Chaque slice peut évoluer vers un microservice si elle grossit — mais ce n'est pas un point de départ par défaut.

Sources et références : MITRE ATT&CK · CERT-FR

Sources et Références

1. Ayi NEDJIMI MCT MCSD. *Guide Architecture 2026 : Vertical Slice + Clean Architecture Lite.* Document de référence, 2026.

2. Martin, R. C. (2017). *Clean Architecture: A Craftsman's Guide to Software Structure and Design.* Prentice Hall.
3. Bogard, J. (2019). *CQRS with MediatR and Vertical Slices.* NDC Sydney.
4. Microsoft. *Spring Modulith — Module boundaries and testing.* Documentation officielle Spring, 2025.
5. FastAPI Documentation. *Dependency Injection & Router Architecture.* Tiangolo, 2025.
6. Go Blog. *Package Design and Dependency Management in Go.* Google, 2024.

Ayi NEDJIMI Consultants — Expert cybersécurité offensive & intelligence artificielle

ayinedjimi-consultants.fr · ayi@ayinedjimi-consultants.fr