Turn Your VIBE Code into PRO Code Instantly? - Master This One Principle

Wilson Vu / NashTech
09 2025

Session 0:
Mindset Shift

00

ADD V3 vs DDD
Độ phù hợp AI & Chất lượng dự án

Phương pháp	Điểm AI Fit	Điểm Quality	Điểm tổng
ADD V3	8.6	8.0	**8.3**
DDD	6.0	5.8	5.9

Theo bạn, phương pháp nào phù hợp hơn cho dự án AI-driven?

01

ADD V3 vs DDD – Chi Tiết Đánh Giá

A. Phù hợp với quy trình AI

Tiêu chí	ADD	DDD
Cấu trúc chuẩn hoá	9	6
DIP-first	9	7
Swap công nghệ	9	6
Guardrails	8	5
Prompt-friendly	8	6
Trung bình	**8.6**	**6.0**

B. Dự báo chất lượng

Tiêu chí	ADD	DDD
Đơn giản/KISS	8	5
Cô lập nghiệp vụ	9	7
Tiến hoá an toàn	8	7
Chi phí học	6	4
Minh bạch review	9	6
Trung bình	**8.0**	**5.8**

Điểm tổng

Phương pháp	AI Fit	Quality	Tổng
ADD V3	8.6	8.0	**8.3**
DDD	6.0	5.8	5.9

Nhận xét ngắn gọn

- ADD V3 ưu tiên DIP, thư mục rõ ràng → AI co-gen, refactor an toàn.
- Tài liệu & guardrails giảm *hallucination*, review tự động.
- DDD mạnh ở chiến lược nhưng tactical pattern nặng, AI khó sinh mã chính xác.
- ADD rút ngắn ~25% thời gian release đầu, giảm drift kiến trúc.
- Domain phức tạp: kết hợp DDD (chiến lược) & ADD-Extended (triển khai).

Prompt: Đọc `ADD Theory V3.en.md` và chấm điểm ADD vs DDD theo AI Fit & chất lượng.

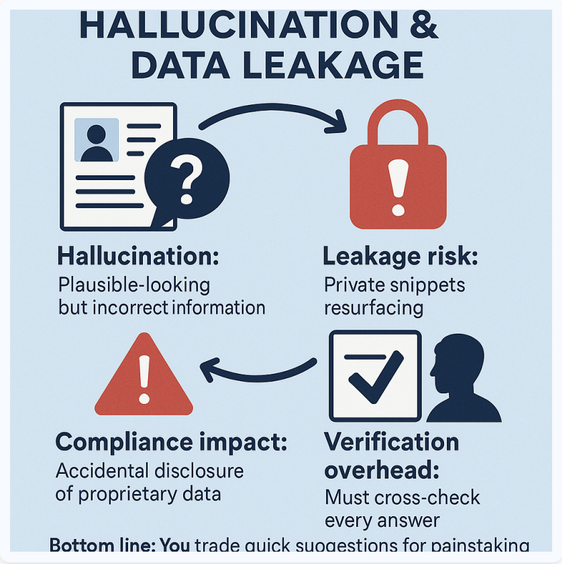
Agenda

- **Session 1 – Pains and "Remedies"?**
- **Session 2 – How ADD Fixes Them**
- **Session 3 – What the Fun is ADD?**
- **Session 4 – Playground**

Session 1:
Problem Statement

01

LLM Pain – Hallucination & Data Leakage



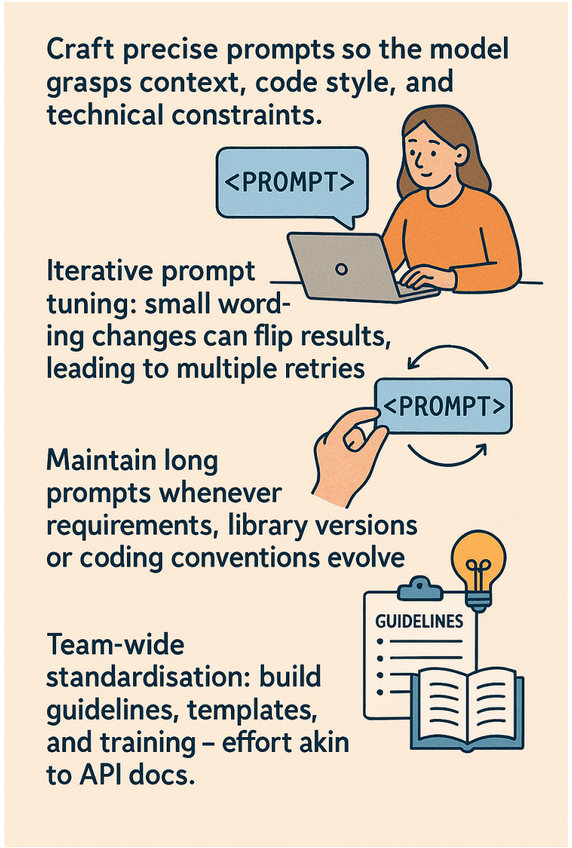
"Hallucination & Data Leakage" covers two intertwined issues that erode trust:

- Hallucination: the model asserts plausible-looking but incorrect information.
- Leakage risk: private snippets placed in prompts may resurface in other sessions.
- Compliance impact: accidental disclosure of proprietary data violates policies.
- Verification overhead: engineers must cross-check every answer, negating speed gains.

Bottom line: you trade quick suggestions for painstaking validation and potential breaches.

Hình này được tạo bởi AI trong lúc đang ảo giác của nó; về cơ bản, AI đã tạo ra một hình không có nghĩa.

LLM Pain – Prompt Engineering Overhead



"Prompt Engineering Overhead" represents the extra effort and time required to:

- Craft precise prompts so the model grasps context, code style, and technical constraints.
- Iterative tuning: small wording changes can flip results, leading to multiple retries.
- Maintain long prompts whenever requirements, library versions, or coding conventions evolve.
- Team-wide standardisation: build guidelines, templates, and training — effort akin to API docs.

*Bottom line: instead of **writing code directly**, developers must craft prompts to steer the LLM — adding extra time and mental overhead.*

LLM Pain – Limited Long-term Context Awareness

Large Language Models have a finite context window, causing:

- Forgetting earlier parts of a conversation when prompts exceed limits.
- Shallow understanding of sprawling codebases or documents.
- Workarounds: chunking, summarising, re-feeding context — manual and error-prone.
- Performance trade-offs: bigger windows cost more tokens and latency.

Bottom line: the assistant lacks persistent memory, so you carry the burden of context management.

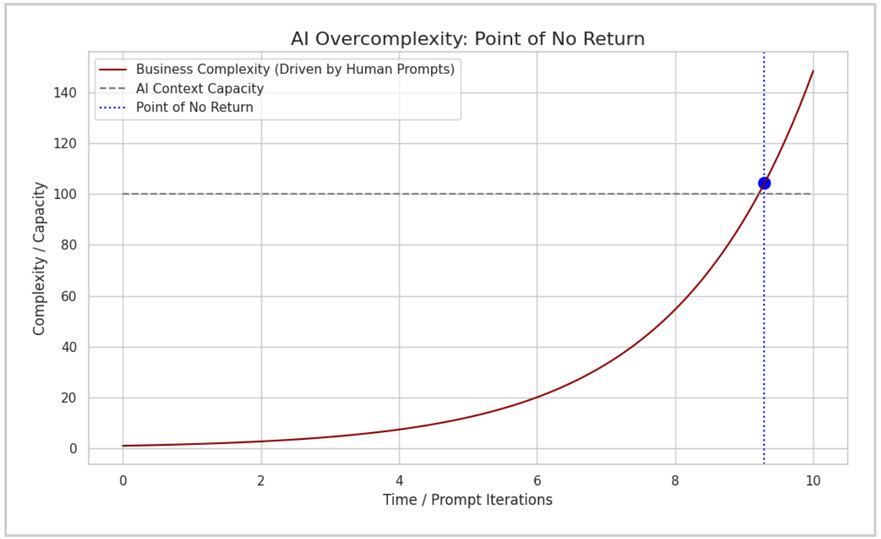
LLM Pain – Cognitive Off-loading Trap

Relying heavily on LLMs can dull engineering judgement:

- Surface-level understanding: accepting answers without grasping fundamentals.
- Skill atrophy: less practice writing code or designing solutions.
- Blind spots: model biases and gaps propagate unchecked.
- Team dependence: juniors may default to the AI, widening knowledge gaps.

Bottom line: off-loading thinking to an LLM risks long-term erosion of expertise.

LLM Pain – When Business Complexity Outgrows AI Context



At first, AI can generate **very accurate** code for a simple requirement – it feels “awesome”. But…

- Business keeps expanding: more features, rules, and edge-cases – especially in e-commerce.
- Context ceiling: eventually the rule set outgrows the LLM’s context window.
- Point of no return: the more detailed the prompt, the more it exceeds limits; AI answers wrong or incoherently.
- Humans lose control: after “outsourcing” everything to AI, no one has the full picture.

*At that point the project must **rally a developer team** to untangle the mess – far costlier than staying hands-on from day one.*

Legacy Architecture Pain Points

Conventional project structures repeatedly stumble over familiar hurdles that slow teams and confuse automated tools.

- Pattern ambiguity & implementation drift

  Same concept coded multiple contradictory ways.

- Infrastructure concerns bleeding into core logic

  Transactions & DB code leak into business layer.

- Expensive and risky technology swaps

  Changing tech stacks triggers invasive refactors.

- Complex business rules scattered across layers

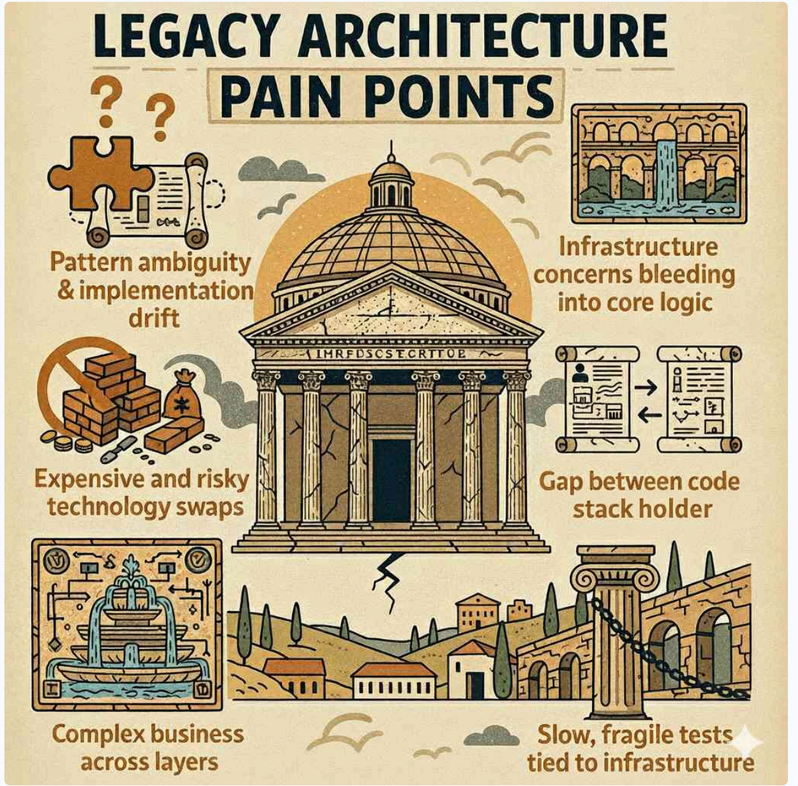
  Rules split between controllers, services, helpers.

- Gap between code and stakeholder language

  Models diverge from terms business actually uses.

- Slow, fragile tests tied to infrastructure

  Tests need databases and mocks to even run.



Pain Point 1 – Pattern Ambiguity & Inconsistency

Same prompt, two AI generations. The concept "*OrderAggregate*" diverges, forcing the team to merge incompatible designs.

Version A – "Fat" Aggregate

```
class OrderAggregate {
  List<OrderLine> _lines = new();

  // Heavy business logic inside
  public void AddLine(Product p, int qty) {
    if(qty < 0) throw new InvalidOperationException();
    _lines.Add(new OrderLine(p, qty));
    RecalculateTotals();
  }

  private void RecalculateTotals() { /* ... */ }
}
```

Version B – "Anemic" Aggregate

```
class OrderAggregate {
  public List<OrderLine> Lines { get; init; } = new();
  public decimal Total { get; set; }
}

class OrderDomainService {
  public void AddLine(OrderAggregate order, Product p, int qty) {
    // Business rules here instead
  }
}
```

- **Version A**: Rich Aggregate contains validation, calculations and enforces invariants internally.
- **Version B**: Aggregate is just a data bag; business rules move to a separate `OrderDomainService`.
- Both styles can be valid, but mixing them in one code-base causes confusion and extra refactoring work.
- Without a clear guideline, each AI generation may pick a different style → *architectural drift*.
- **SRP Perspective**: A well-designed rich aggregate *can* respect SRP if all code only enforces its own invariants; however, in practice extra concerns (tax, messaging, mapping) creep in → violation. ▼
- An anemic aggregate keeps data clean but scatters business rules across services → harder to trace invariants and maintain.

** "Fat" Aggregate (rich) – keeps substantial behavior, enforces its own invariants. This is the style DDD encourages, but the word "fat" is merely slang, not an official term.*
** "Anemic" Aggregate/Model – holds data only; business logic lives in a separate service. "Anemic Domain Model" was coined by Martin Fowler as an anti-pattern; DDD regards it as off-track because it weakens the domain model's OO nature.*

Pain Point 2 – Leaky Infrastructure into Business Logic

A request to “make this operation transactional” leads the AI to embed database-specific code directly in the domain layer.

```
  // Domain layer (!)
public class PaymentService {
  private readonly DbConnection _conn;
  public async Task ProcessAsync(PaymentCommand cmd) {
    await using var tx = await _conn.BeginTransactionAsync();
    //     domain rules
    await tx.CommitAsync();   // Infrastructure leaks here
  }
}
```

This couples business logic to a concrete database driver, violating separation of concerns.

Pain Point 3 – High Cost of Technology Swaps

Subtle coupling between domain design and infrastructure makes switching technology slow and expensive.

Example: Migrating from Postgres to MongoDB forces an overhaul of `CustomerAggregate` because relational assumptions (joins, foreign keys) are baked into entities and queries.

Result: weeks of refactor in the **core domain** instead of a clean replacement in infrastructure.

Pain Point 4 – Complex Domain Logic

As systems grow, business rules evolve into *multi-step workflows*, spanning several entities and temporal states. AI generations often capture only a slice, leading to fragile “happy-path” logic.

- **Hidden invariants** – Discounts require customer loyalty *and* stock availability; rule sits in a forgotten service.
- **Temporal coupling** – Order status transitions (Draft → Confirmed → Shipped) enforced in scattered places.
- **Cross-aggregate rules** – Payment must match Shipment total, yet code checks amounts in two different handlers.

Complexity snowballs when each AI prompt patches logic in isolation.

```
// Real-world example
if(order.IsInternational && shipment.Carrier=="DHL" &&
   customer.Tier==CustomerTier.Gold && order.Items.Sum(i=>i.Qty) > 20)
{
    ApplyFreeShipping(order);
}
```

A small change (e.g., new carrier) forces hunting across services – error-prone and slow.

Pain Point 5 – Lack of Ubiquitous Language

Developers, analysts, and AI prompts often use *different terms* for the same concept. “Invoice”, “Bill”, “Receipt” become interchangeable, causing mis-alignment and buggy code.

- **Domain drift**: AI names entity `BillingDocument` while finance team insists on “Invoice”.
- **Duplicate logic**: Two services compute `TotalDue` vs. `AmountOwed` slightly differently.
- **On-boarding overhead**: New team members struggle through a thesaurus of synonyms.

Without a shared vocabulary, every prompt hides a landmine of semantic misunderstanding.

> “If you want your model to work tomorrow, make sure everybody speaks the same language today.”

Pain Point 6 – Difficult Testability

When domain code is tangled with infrastructure and sprawling workflows, writing reliable tests becomes an *expensive chore*.

- **Heavy setup** – Tests require real DB + MQ just to execute a simple command.
- **Flaky execution** – Time-outs and random failures hide real regressions.
- **Slow feedback** – Full end-to-end suite takes 40-60 minutes ⇒ devs skip tests.

Coupling Core logic to tech details breaks the speed loop that modern teams need.

```
// Symptom
[Test]
public async Task CancellingOrder_RefundsPayment()
{
    using var db = new SqlContainer(); // Spin up container
    await SeedAsync(db);
    // ... 120 lines later
}
```

Test harness outweighs assertion – a clear sign of architectural pain.

DDD Remedy – Pattern Clarity via Bounded Contexts

Each Bounded Context provides an explicit semantic boundary so ubiquitous language, entities, and patterns stay consistent and isolated from neighbouring models. Ambiguity drops because every team knows *where* a concept lives and *how* it must behave.

- Define clear context maps to avoid overlapping responsibilities.
- Align micro-services or modules to context boundaries.
- Use context-specific language internally; translate via Anti-Corruption Layers.

DDD Remedy – Infrastructure Isolation through Layers

A classic layered architecture (UI → Application → Domain → Infrastructure) shields the Domain Model from technology churn. Database, messaging and frameworks sit at the edge, accessed only via interfaces so the core logic stays pure.

- Infrastructure code depends on Domain, never the other way around.
- Adapters can be replaced (e.g., swap SQL for NoSQL) without touching business rules.
- Enables easier unit-testing as Domain doesn’t need external resources.

DDD Remedy – Strategic Tech Independence

By modelling the **domain first**, technology choices stay strategic rather than accidental. You can adopt new databases or queues when they provide proven value, not because the framework forces it.

- Core domain APIs express behaviour, not tech jargon.
- Integration details encapsulated in App & Infra layers.
- Lets teams evolve tech stack per bounded context, avoiding big-bang rewrites.

DDD Remedy – Ubiquitous Language Enforcement

Teams speak the same language as the code. Every class, method, and event uses domain terms agreed with business, cutting translation errors.

- Refine glossary during Event Storming & modelling sessions.
- Reject generic names (`DataService`) in code reviews.
- Shared language reduces cognitive load and onboarding time.

DDD Remedy – Domain-centric Testing

With infrastructure peel-off, you can write fast tests against pure domain objects, verifying business rules without databases or web servers.

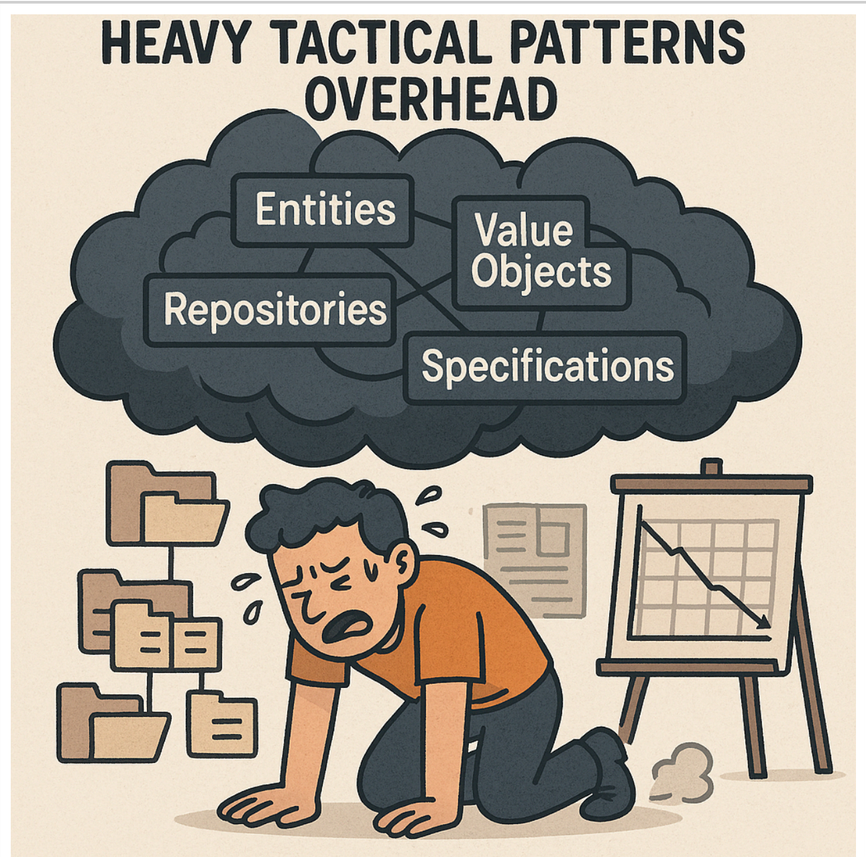
- Given-When-Then scenarios directly exercise aggregates and domain services.
- Mocks/stubs only for external systems at boundaries.
- Feedback cycle shrinks from minutes to milliseconds.

Pain of DDD – Steep Learning Curve

DDD introduces a rich set of concepts (Aggregates, Repositories, Domain Events, Bounded Contexts). Newcomers face terminology overload before they can deliver real features.

- Workshop upfront: requires dedicated domain-modelling sessions.
- Pattern misunderstanding: easily leads to cargo-cult code.
- Training cost: significant learning investment before productivity gains.

Pain of DDD – Heavy Tactical Patterns Overhead



Entities, Value Objects, Repositories, Specifications… Every tactical pattern adds boilerplate. A tiny feature might require numerous artifacts before any real business rule is coded.

- Structure balloons: extra folders, interfaces, and classes just to satisfy the pattern.
- Time spent wiring: developers hook up repositories or factories instead of shipping value.
- Stakeholder concern: burndown charts stall because the scaffolding work is invisible.

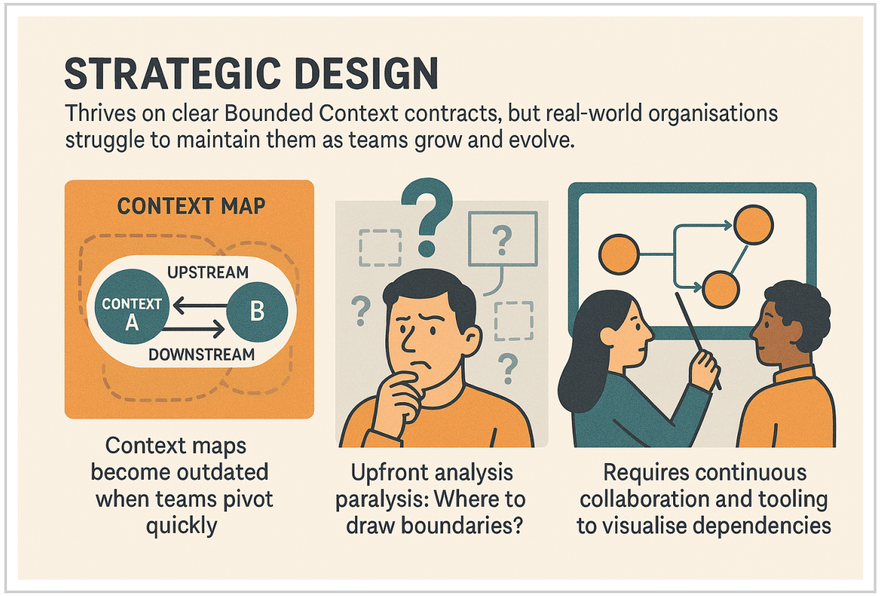
Community Consensus – Hide `IQueryable<T>`

- **Vaughn Vernon – Implementing Domain-Driven Design** (2013)
  Chapter 6 "Repositories" – "A Repository should present a collection-like interface that works the same whether the underlying storage is memory, relational, or NoSQL."
- **Eric Evans – Domain-Driven Design Reference** (2014)
  Recommends returning Aggregate and simple selection criteria; avoid leaking infrastructure.
- **Jimmy Bogard – blog "Why I Hate IQueryable"** (2015)
  Analyzes tight-coupling, testability pain, provider lock-in.
- **Microsoft eShopOnContainers (sample microservice)**
  `IOrderRepository` returns entities / lists. Complex reads done via *Query Objects* / Dapper read-model; no `IQueryable`.
- **Mark Seemann – "IQueryable is a leaky abstraction"** (2014)
  Explains violation of Dependency Inversion when expression trees leak upward.

People have spent *a lot* of ink convincing developers **NOT** to expose `IQueryable<T>` outside infrastructure.

In Microsoft's own *eShopOnContainers*, to stay DDD-friendly they ended up using **Dapper** alongside **Entity Framework**: EF handles writes; Dapper handles read models - a humorous outcome for a mature ORM that promotes `IQueryable`.

Pain of DDD – Complex Bounded Context Governance



Strategic design thrives on clear Bounded Context contracts, but real-world organisations struggle to maintain them as teams grow and evolve.

- Context maps drift: become outdated when teams pivot quickly.
- Analysis paralysis: upfront debate on where to draw boundaries.
- Collaboration overhead: continuous coordination and tooling needed to visualise dependencies.

Session 2:
How ADD Fixes Them

02

ADD Solution 1 – Clear Actor & Abstraction Layers

Pattern ambiguity is removed by assigning responsibilities to explicit actor-oriented layers.

- Actor-first mapping: each layer (Boundary, Core Abstractions, Operators, Implementations, Bootstrap) has a single, well-defined role.
- No misplaced logic: DTOs stay in Boundary, orchestration in Operators, infra details in Implementations.
- High code readability: newcomers navigate code faster and avoid accidental coupling.

ADD Solution 2 – Ports Shield Infrastructure

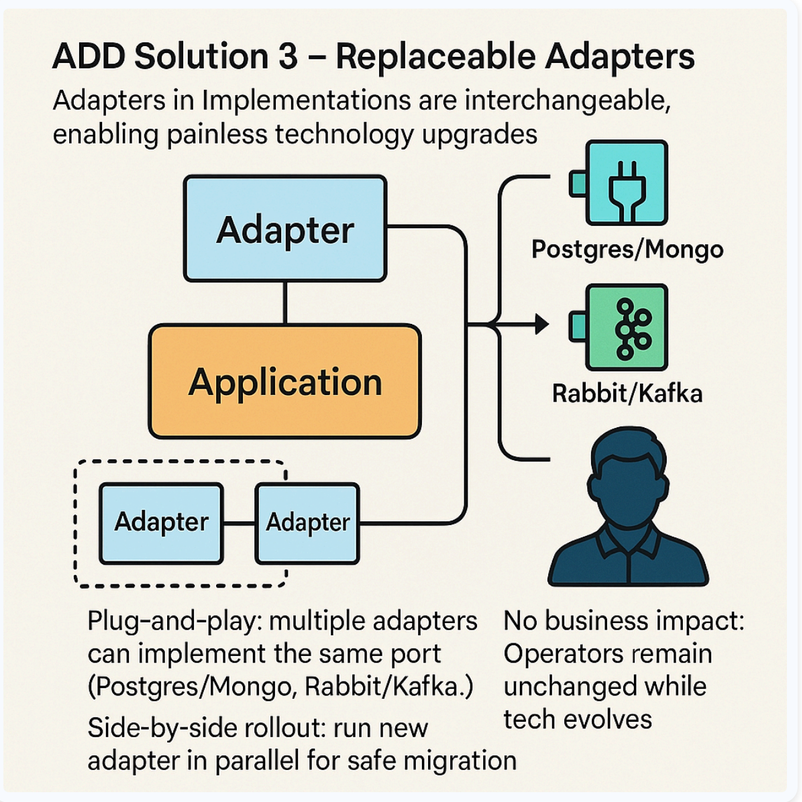
Ports (interfaces) declared in Core Abstractions act as a membrane that prevents infrastructure leakage.

- Strict membrane: Operators call ports, never concrete tech.
- Infrastructure isolation: Implementations implement the ports; DB/MQ/SDK never bleed into business code.
- Easy tech swap: replace or mock adapters without touching business logic.

ADD Solution 3 – Replaceable Adapters

Adapters in Implementations are interchangeable, enabling painless technology upgrades.

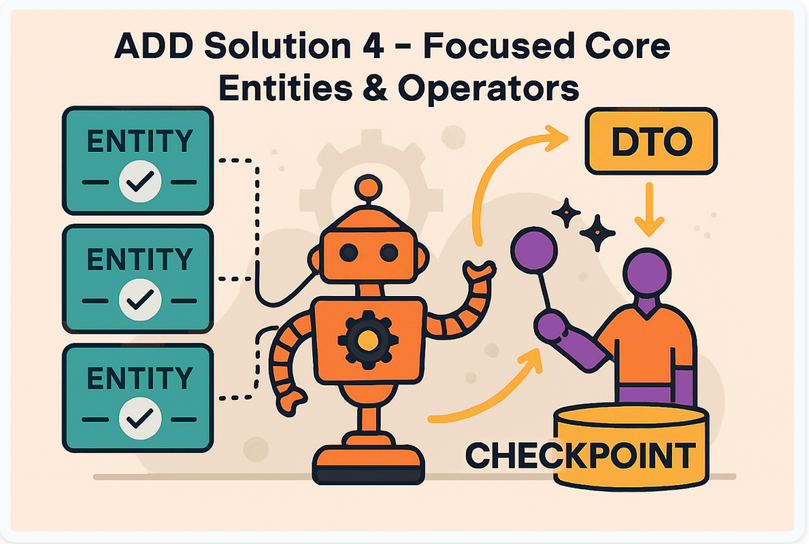
- Plug-and-play: multiple adapters can implement the same port (Postgres/Mongo, Rabbit/Kafka…).
- Side-by-side rollout: run new adapter in parallel for safe migration.
- No business impact: Operators remain unchanged while tech evolves.



ADD Solution 4 – Focused Core Entities & Operators

Complex domain logic is concentrated in thin Entities (validation) and orchestration-centric Operators.

- Entities stay lean: only state + validation, no infrastructure calls.
- Operators orchestrate: map DTO ↔ Entity, coordinate ports/events, execute checkpoints.
- Simpler reasoning: business rules live in one place, easier to test and evolve.



ADD Solution 5 – Design Request as Ubiquitous Language

The term **Design Request** flows consistently across layers, closing the language gap.

- Shared vocabulary: same DTO/Entity/Event naming across Boundary, Operators, Implementations.
- Better collaboration: stakeholders, docs, and code speak the same language.
- Reduced translation cost: fewer mismatches, faster onboarding.

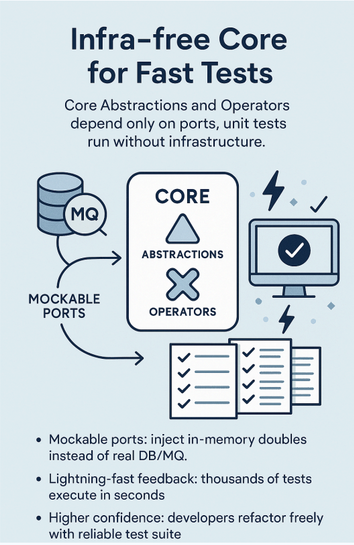
Note: "Design Request" is just an example; the rule is that *every* important business concept (Order, Payment, Ticket, …) must become ubiquitous language across *all* layers in A.D.D.

1. **Step 1**: Work with BA/Product to establish a *glossary* of canonical terms.
2. **Step 2**: When declaring DTOs, Entities, Events, and Ports, name them exactly as in the glossary.
3. **Step 3**: Guard with code review/lint – renaming or translating the term is a violation.

ADD Solution 6 – Infra-free Core for Fast Tests

Because Core Abstractions and Operators depend only on ports, unit tests run without infrastructure.

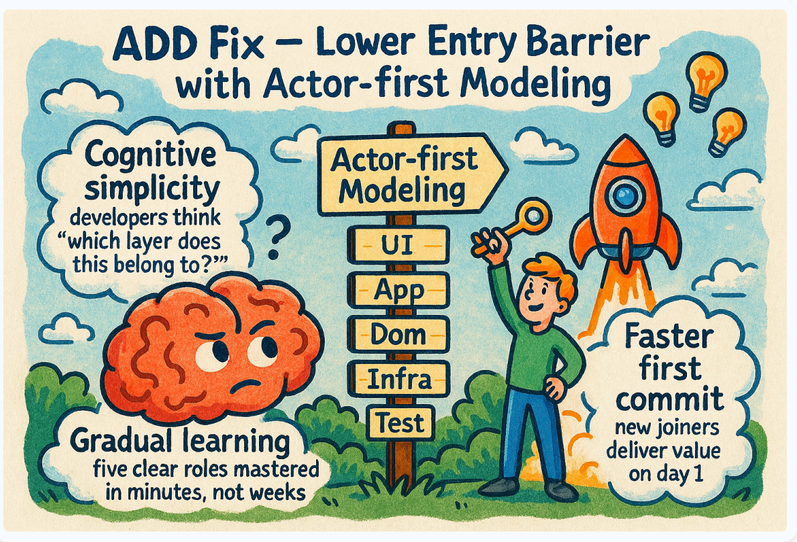
- Mockable ports: inject in-memory doubles instead of real DB/MQ.
- Lightning-fast feedback: thousands of tests execute in seconds.
- Higher confidence: developers refactor freely with reliable test suite.



ADD Fix – Lower Entry Barrier with Actor-first Modeling

Focusing on actors before patterns makes onboarding much easier than DDD's tactical catalogue.

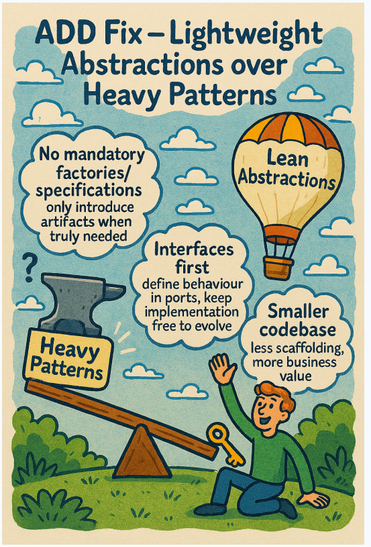
- Cognitive simplicity: developers think "which layer does this belong to?" instead of "which pattern should I pick?".
- Gradual learning: five clear roles mastered in minutes, not weeks.
- Faster first commit: new joiners deliver value on day 1.



ADD Fix – Lightweight Abstractions over Heavy Patterns

ADD replaces heavyweight tactical patterns with lean abstractions that cut boilerplate.

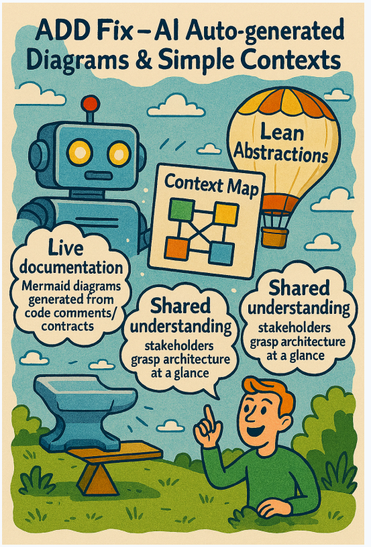
- No mandatory factories/specifications: only introduce artifacts when truly needed.
- Interfaces first: define behaviour in ports, keep implementation free to evolve.
- Smaller codebase: less scaffolding, more business value.



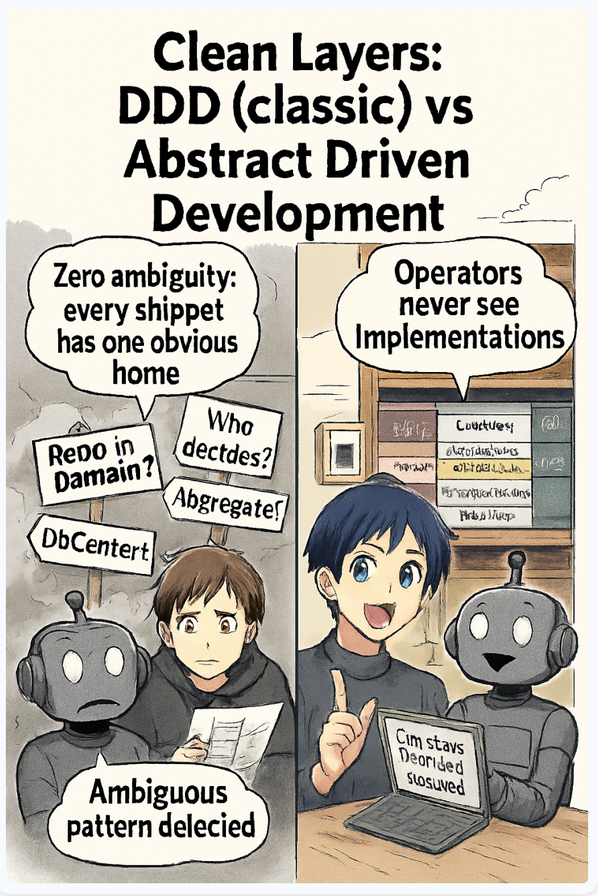
ADD Fix – AI Auto-generated Diagrams & Simple Contexts

ADD leverages AI to render diagrams automatically, keeping context maps simple and always up-to-date.

- Live documentation: Mermaid diagrams generated from code comments/contracts.
- Shared understanding: stakeholders grasp architecture at a glance.
- Governance simplified: no sprawling bounded-context negotiations.



Clean Layers: DDD (classic) vs Abstract Driven Development



Visual comparison of the layer responsibilities in classic Domain-Driven Design versus ADD's strict five-actor structure helps teams quickly see where their code belongs.

Aspect	DDD (Classic)	ADD (V3)
Layer Intent	Domain vs Infrastructure; guidelines but flexible.	Strict 5 actors (Boundary, Core Abstractions, Operators, Implementations, Bootstrap) — explicit physical folders.
Repository Interface	Should return Aggregate, but teams often leak `IQueryable`, `DbContext`.	Port in Core Abstractions — no tech-specific types allowed.
Business		

Session 3:
What the Fun is ADD

O3

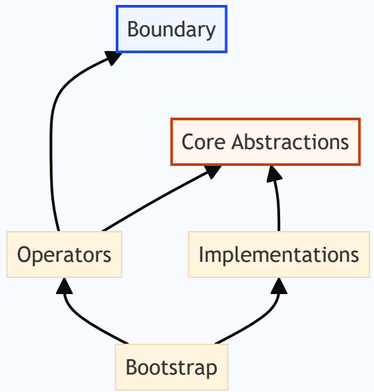
ADD – The Golden Rule

- Absolute separation between **WHAT** (business) and **HOW** (technology).
- Business orchestration depends *only* on abstract Ports (interfaces).
- Concrete technology (DB, MQ, 3rd-party) lives in *Implementations*, wired via DI at Bootstrap.
- Operators and Implementations never reference each other directly.
- This one rule keeps the architecture clean and evolvable even when AI generates code.

The 5 Actors & Layering

- **Boundary** – System ingress/egress (DTO, API, Boundary Event)
- **Core Abstractions** – Technology-agnostic language (Entities, Ports, Core Events)
- **Operators** – Business orchestrators; map DTO ↔ Entity; depend only on Boundary + Core
- **Implementations** – Concrete tech (DB, MQ, 3rd-party) implementing Ports
- **Bootstrap** – Composition root; wires Ports ↔ Implementations via DI

Clear physical folders mirror these actors, ensuring every file has exactly one rightful home.



Data & Event Flow

Boundary Event (external) → **Operator** → **Port** → **Implementation** → optional **Core Event** back to Operator.
Operators map *DTO* ↔ *Entity*; Implementations may emit Core Events or return Entities via Port, keeping tech details hidden.

```
External → BoundaryDTO → Operator
        ↘ via Port ↙              ↘ CoreEvent (optional)
        Implementation (DB/MQ)
```

Dependency Inversion Principle in ADD

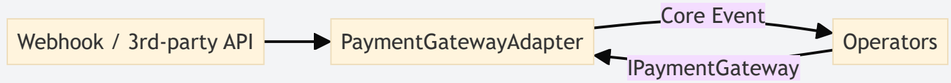
Ports (interfaces) live in **Core Abstractions**; both **Operators** and **Implementations** depend on them.

```
interface IOrderRepository { ... }  // Core Abstractions
class OrderOperator {               // depends on Port
    IOrderRepository repo;
}
class PostgresOrderRepository : IOrderRepository { ... } // Implementations
```

Bootstrap wires Port ⇢ Implementation via DI. Changing Postgres → Mongo touches only Implementations + Bootstrap.

Adapters & Anti-corruption Layer (ACL)

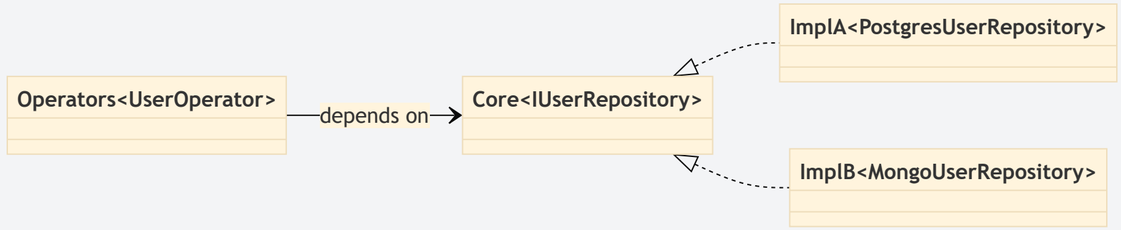
- **Adapter** implements a *Port* declared in Core Abstractions.
- It translates between external protocol/payload ↔ clean DTO/Core Event.
- All mapping, idempotency, versioning, outbox logic stays inside the adapter.
- Business **Operators** remain unaware of external tech details.



Evolvable Structure & Module Swapping

As long as **Operators** stay untouched, we can replace technology by swapping **Implementations** and adjusting DI in **Bootstrap**.

This enables side-by-side rollout (blue/green) and fast rollback without changing business code.



Modules can evolve independently: a messaging provider upgrade stays inside the *Messaging* module inside Implementations. Tests for Operators remain green.

ADD-Extended for Complex Domains

1. Scope Module & Docs

- Each **module** lives in `modules/<name>/` and follows the 5 ADD layers.
- Comes with `README.md`, `shared-terms.md`, optional *decision-log.md* to capture rules & vocabulary.

2. RuleSet / Checkpoint

- Pure business rules go to `rulesets/` as functions.
- Combine several RuleSets into a `checkpoints/` pipeline inside the Operator.

3. TGO & Coordinator Operators

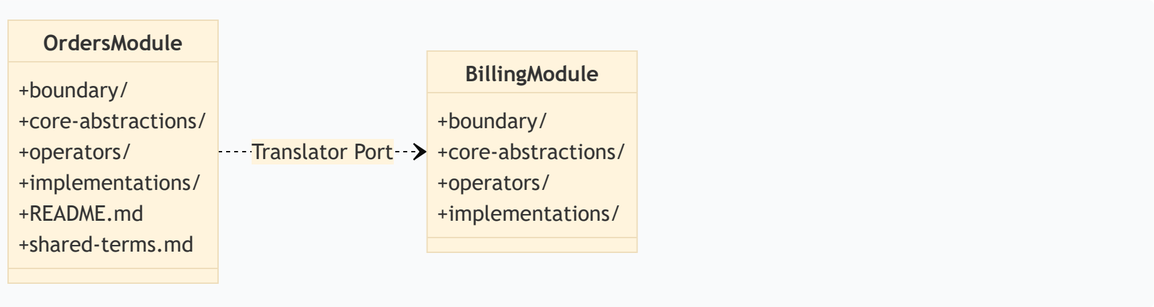
- **TGO** – Transaction Group Operator for strong consistency.
- **Coordinator** orchestrates multi-step flows (eventual) and invokes Compensators.

4. Interaction Map

- Pure *Translator* Port between two modules, declared under `core-abstractions/ports/translators/`.
- An Adapter implements the Translator inside `implementations/`, isolating mapping & infrastructure.

5. Documentation-first Flow

- When adding a new RuleSet/Port, update the adjacent markdown so AI & devs share the same context.
- CI can detect drift between code and docs (DTO/Port/Signal names).



Session 4: Playground

04

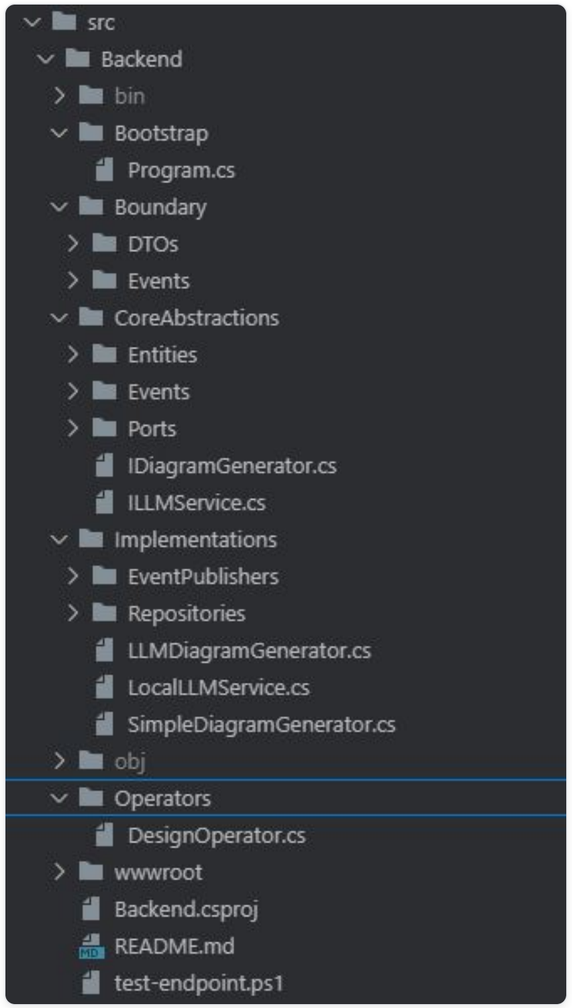
Experiment: Can a weak "Auto" model design an architecture?

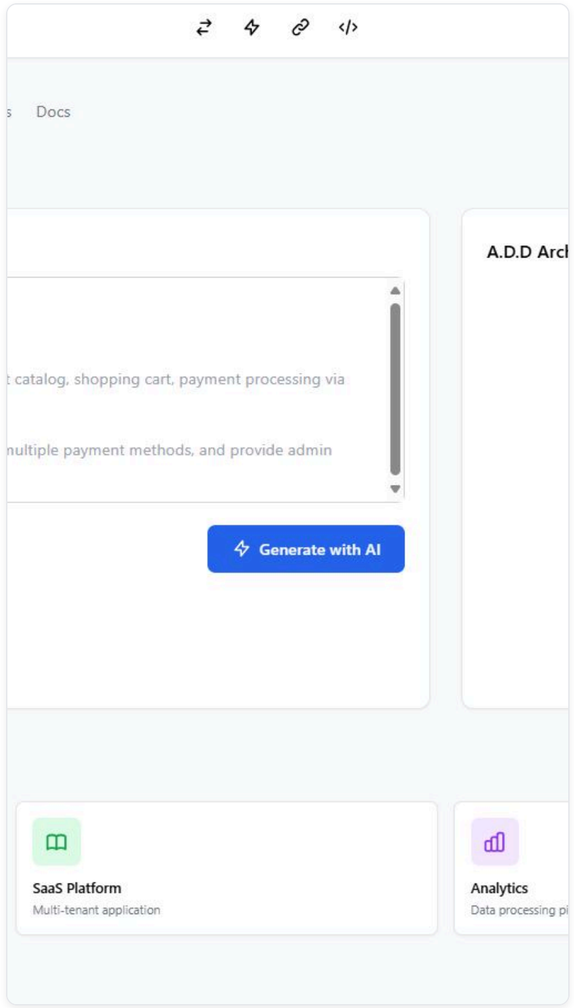
To test the impact of *codebase scaffolding*, we asked Cursor's default **Auto** model – known to be quite limited – to create a web page that plays the role of an *architect* generating an overall architecture for a given business requirement.

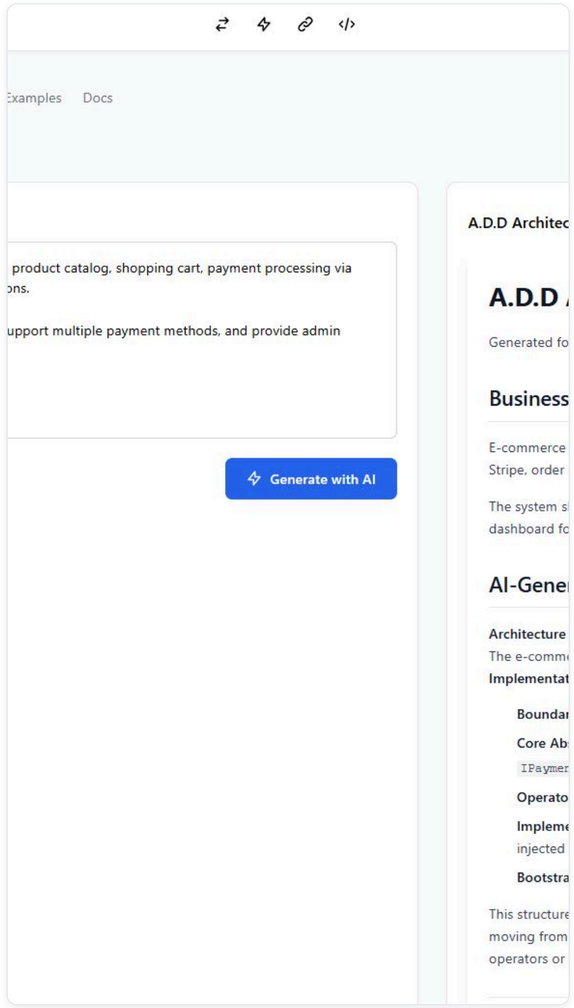
With the A.D.D folder structure and clear naming in place, we want to observe whether the model understands the structure and produces higher-quality output than usual.

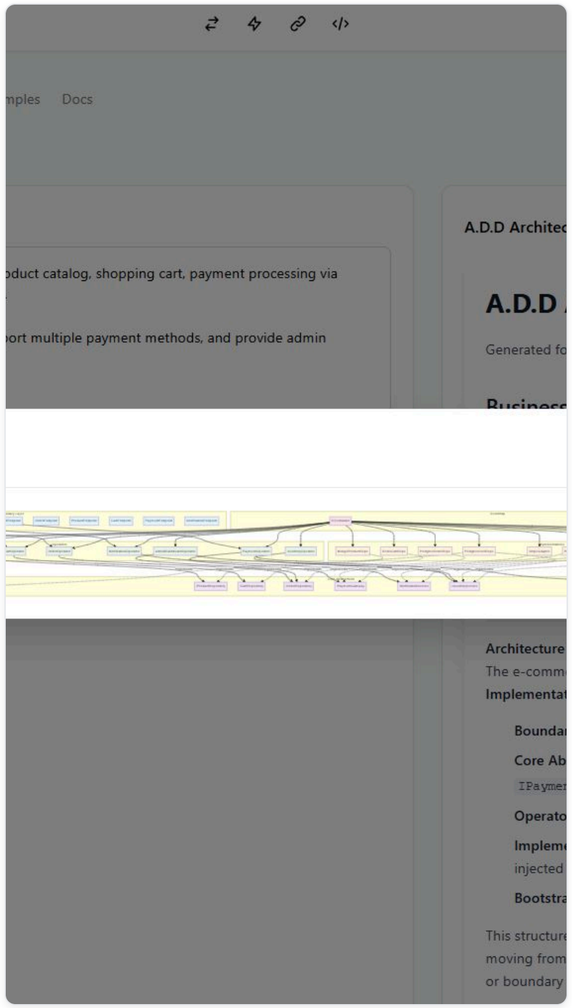
The generated screenshots on the previous slide show the result. Discuss: Does the output quality improve? Which parts look reasonable, which parts still need human review?

AI-Generated Project – Screenshots









Thank You!
Questions?