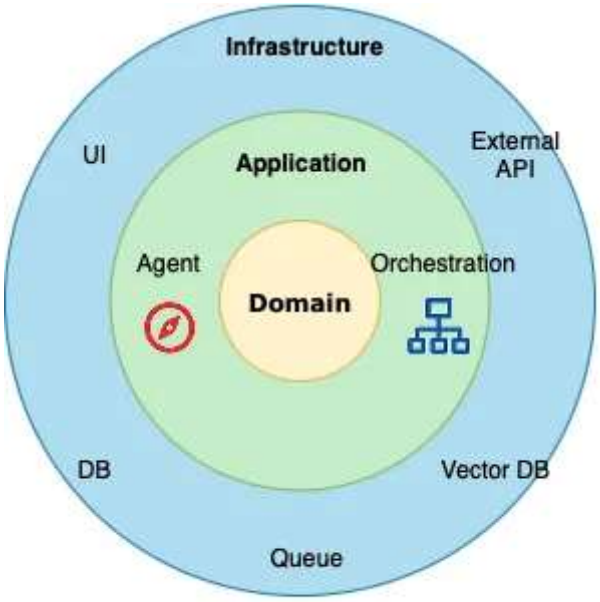
AI Agent Architecture: Mapping Domain, Agent, and Orchestration to Clean Architecture

5 min read · Aug 28, 2025

Naoyuki Sakai Following

Listen Share ••• More



Introduction

AI agents aren't magic black boxes — they fit into Clean Architecture.

AI agents are often portrayed as a *black box*.

An LLM "reads" your request, "decides" which tools to call, and somehow produces the right answer.

This story is attractive but misleading. In real-world enterprise systems, software must be **auditable, testable, and maintainable**. You cannot rely on a monolithic prompt to encode business logic, nor can you let a stochastic model silently control mission-critical workflows.

To bring AI agents into the same engineering discipline as other systems, we need a clear architectural perspective.

In this article, I propose mapping AI agents onto **Clean Architecture / DDD principles**, showing how **Domain, Agent, and Orchestration** fit into the concentric layers of software design.

Example upfront: Think of a logistics company asking an AI agent, *"Generate tomorrow's delivery plan."* The answer should not come from a black box but from a transparent pipeline that interprets intent, gathers data, and validates against strict rules.

Three Roles Defined

When building AI agent systems, three distinct responsibilities emerge:

1. **Domain (Correctness)**

- The core business truth: legal rules, business constraints, KPI formulas.
- Must be implemented in deterministic code and tested explicitly.
- Example: "Delivery trucks must not exceed 8 hours of operation per day."

2. Agent (Tactical Selection)

- Interprets the user's intent and chooses which use case or pipeline to execute.
- Think of it as the "router" that decides *what to do next*.
- Example: For a request *"Generate tomorrow's delivery plan"*, the agent decides to call GeneratePlan and then EvaluatePlan.

3. Orchestration (Information Flow)

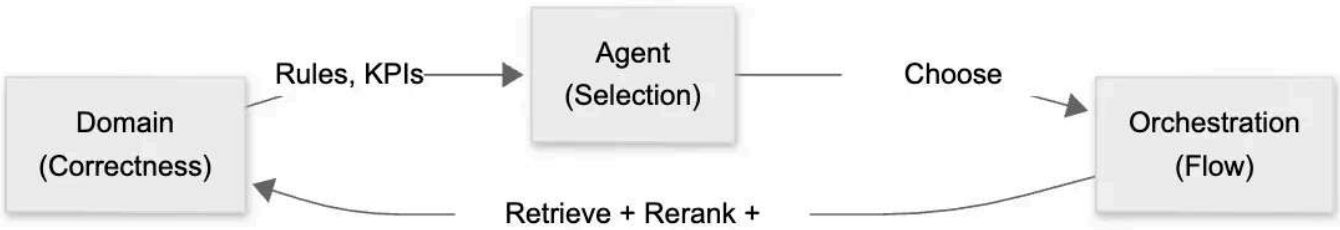
- Manages how data and tools are connected.

- Fetches relevant context, runs retrieval-augmented generation (RAG), merges results, and builds prompts.
- Example: Query demand forecast DB → retrieve past exceptions from VectorDB → fetch regulations from FAQ → fuse into context for the LLM.

👉 Put simply:

- **Domain = correctness**
- **Agent = tactical selection**
- **Orchestration = information flow**

Visualizing the Separation



Mapping to Clean Architecture

Clean Architecture, popularized by Robert C. Martin (Uncle Bob), organizes software into concentric layers:

- **Domain** (Entities, Use Cases) at the core
- **Application** controlling flow and orchestration
- **Infrastructure** = **Interface Adapters** + **Frameworks & Drivers**
  - *Interface Adapters*: bridges that connect the application to external systems (DB/VectorDB adapters, API adapters, UI adapters, retrievers, rerankers, prompt builders, etc.)
  - *Frameworks & Drivers*: the actual external technologies (LLM APIs, databases, vector stores, web frameworks, schedulers, monitoring systems, etc.)

Two timeless principles apply:

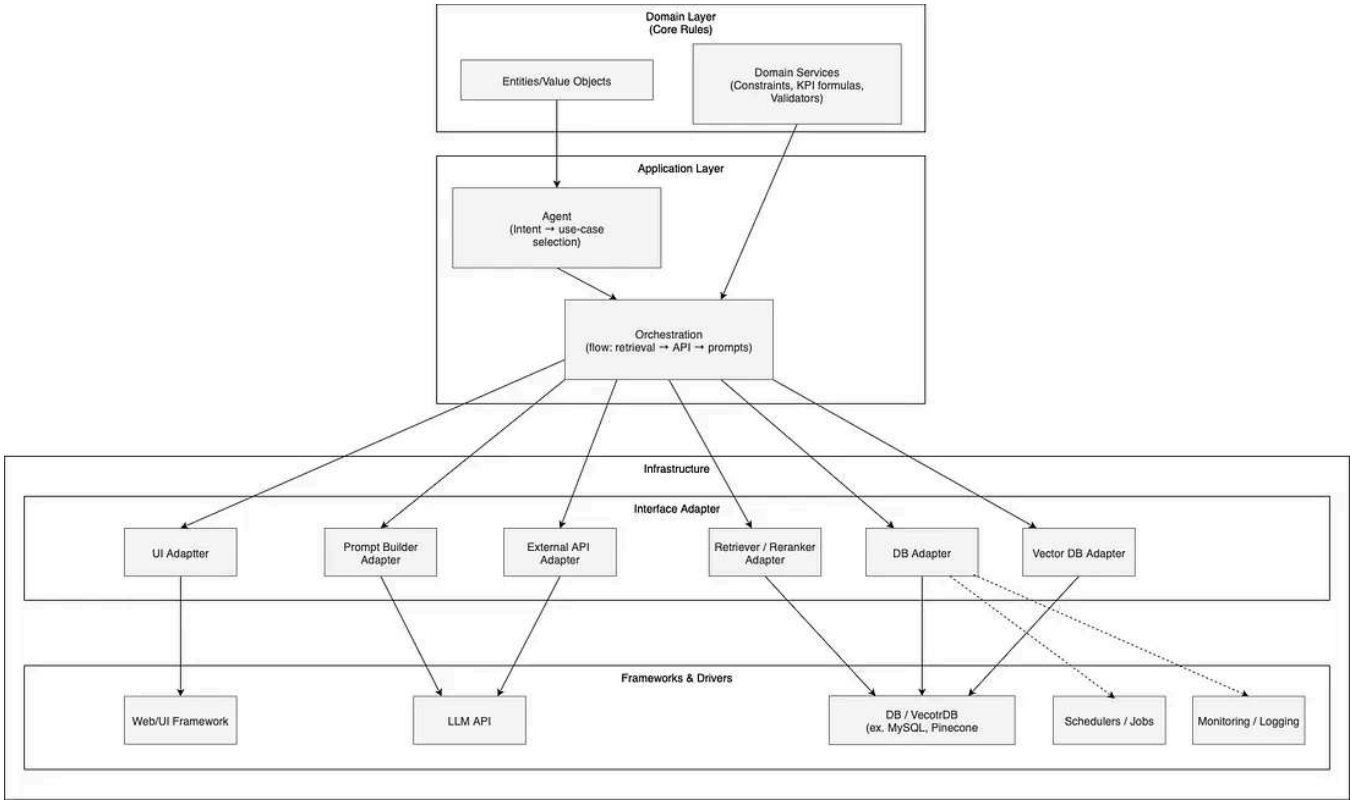
1. Inner layers must not depend on outer layers.

2. The core domain should remain stable and testable, even if technologies change outside.

How AI Agents Fit

- **Domain Layer**: Immutable rules (constraints, KPI formulas, validators). Always implemented in code, never in prompts.
- **Application Layer**: Where AI agents live.
  - *Agent*: interprets intent and selects use cases.
  - *Orchestration*: arranges flows (retrieval, API calls, prompts).
- **Infrastructure Layer** (Interface Adapters + Frameworks & Drivers):
  - *Interface Adapters*: DB adapters, VectorDB adapters, API adapters, UI adapters, retriever/reranker adapters, prompt builders.
  - *Frameworks & Drivers*: LLM APIs, DB/VectorDB engines, web/UI frameworks, schedulers, monitoring/logging systems.

Diagram (Mermaid)



✅ This keeps the original structure but clarifies:

- **Infrastructure = Interface Adapters + Frameworks & Drivers**
- **Application = Agents + Orchestration**

- **Domain = Immutable business rules**

Where ChatGPT and RAG Fit

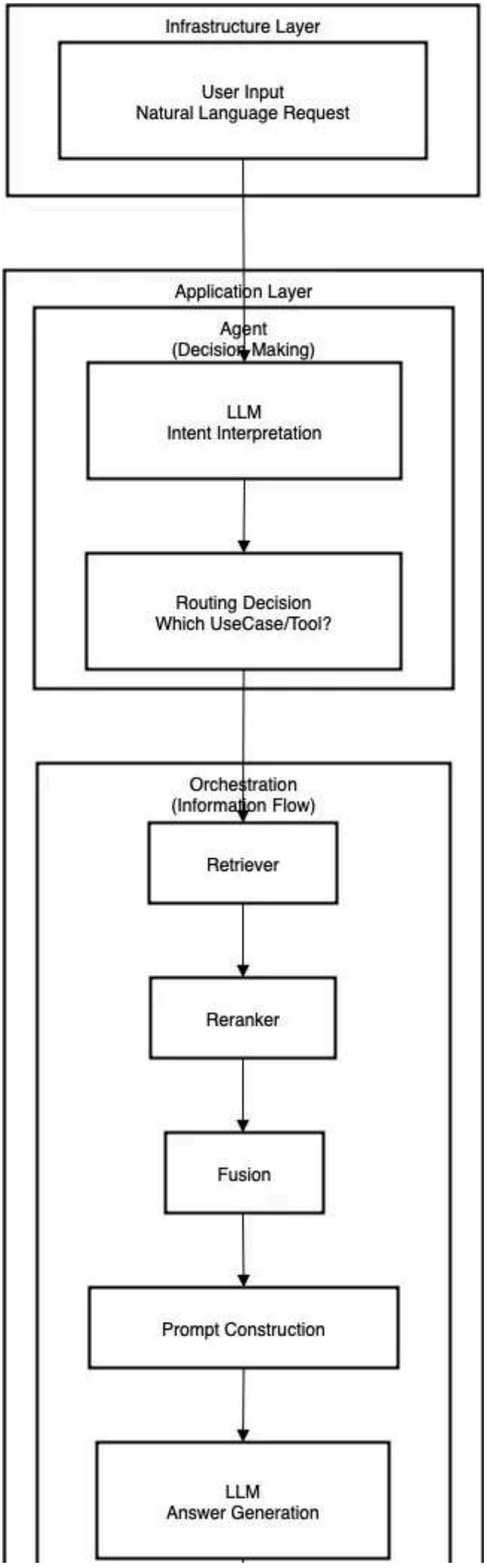
LLMs like ChatGPT are often mistaken as “the whole agent.” In fact:

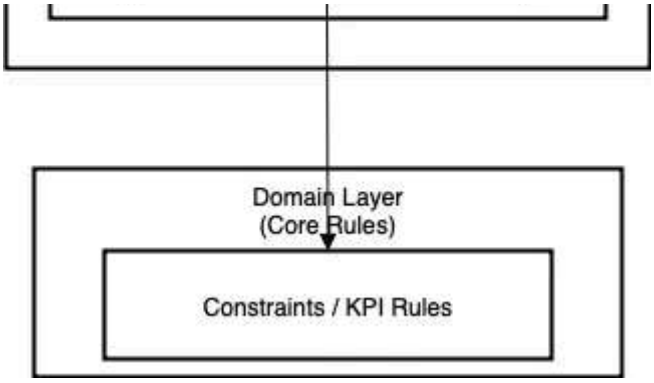
ChatGPT / LLMs

- Not the Domain.
- Instead:
  - **Agent-LLM** interprets intent and decides what to do.
  - **Orchestration-LLM** fuses data and generates natural-language output.
- ⚠️ LLMs are **powerful but stochastic**. They excel at interpretation and expression, but **should not hold business-critical truth**.

RAG

- Not a new layer, but an **orchestration pattern**:
  - Retriever → Reranker → Fusion → Prompt → LLM


Infrastructure Layer
User Input
Natural Language Request
Application Layer
Agent
(Decision Making)
LLM
Intent Interpretation
Routing Decision
Which UseCase/Tool?
Orchestration
(Information Flow)
Retriever
Reranker
Fusion
Prompt Construction
LLM
Answer Generation



Design Guidelines (Do)

To keep AI agents maintainable, auditable, and trustworthy:

Must-have

1. **Externalize prompts** — version-controlled templates.
2. **Separate Planner and Executor** — LLM proposes, code executes.
3. **Use JSON Schema contracts** — validate tool I/O.
4. **Ensure observability** — log input, output, rationale, model ID.

- Example log entry:

```
{
  "input": "Generate plan for Tokai",
  "output": {"planId": 123, "score": 0.87},
  "rationale": "Best load balance",
  "model": "gpt-4",
  "timestamp": "2025-08-25T10:00Z"
}
```

Recommended

5. **Separate Router and Orchestrator** — avoid mixing intent routing with flow design.

6. **Declare orchestration pipelines** — YAML/DAG for RAG flows.

```
pipeline:
  - stage: retriever
```

```
    - stage: reranker
    - stage: fusion
    - stage: prompt_construction
    - stage: llm_answer
```

7. **Keep the UI thin** — presentation only.

Anti-patterns (Don't)

Avoid these traps:

- **God Prompt** — one giant prompt that mixes everything.
- **Domain in Prompt** — hiding business rules in LLM instructions.
- **Tool Zoo** — unstructured tools with no capability map.
- **Fat UI/Application** — logic and prompts leaking into UI.
- **Implicit State** — hidden history, no explicit session store.

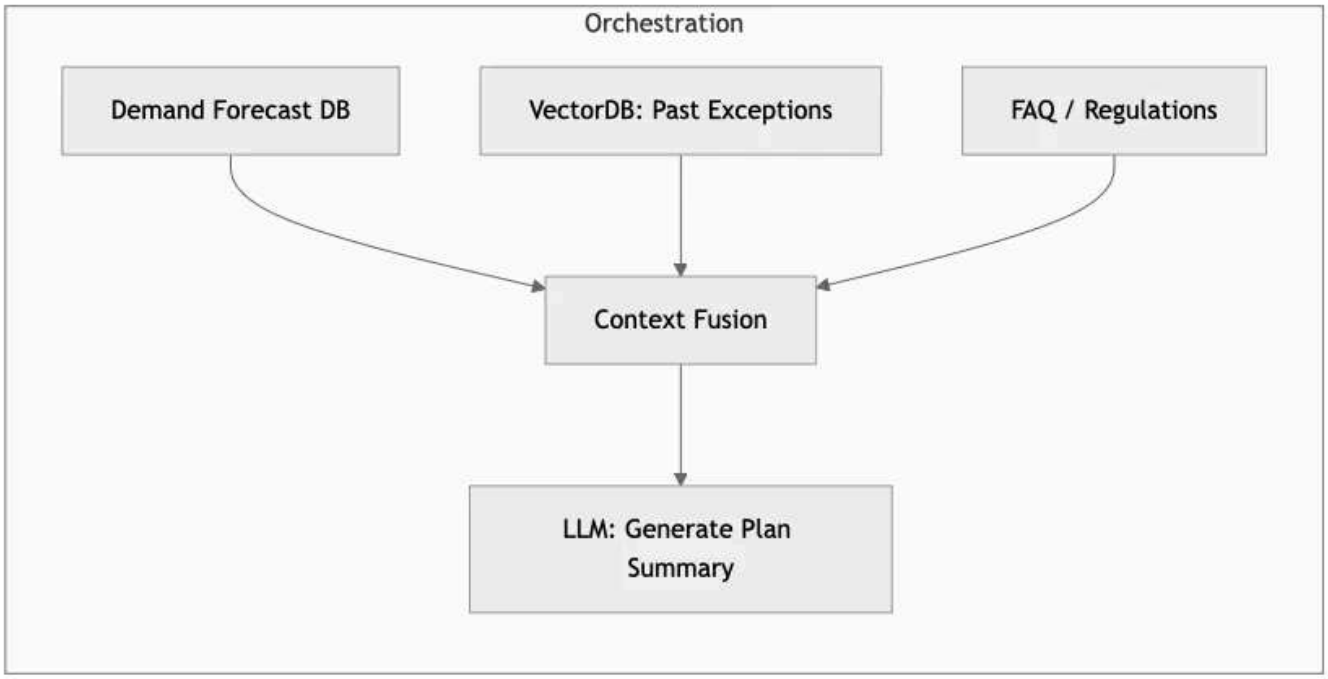
(Common case: embedding prompts directly in UI code → impossible to test or reuse later. Seen frequently in early prototypes. Works short-term, breaks long-term.)

Example: Logistics Dispatch

A dispatch planner asks: *"Generate tomorrow's delivery plan for the Tokai region."*

- **Agent**: Interprets intent, decides GeneratePlan → EvaluatePlan.
- **Orchestration**: Retrieves demand forecast (DB), past exceptions (VectorDB), regulations (FAQ); fuses context and asks LLM to draft plan.
- **Domain**: Enforces rules (driver hours, load capacity, delivery windows). Invalid plans are rejected; valid ones committed via CommitPlan.

Visualizing the Orchestration Flow



👉 Outcome: a transparent, auditable workflow — not a black box.

👉 Business benefit: improves efficiency *and* guarantees compliance with operational rules.

👉 Non-functional benefit: easier auditability and component reuse.

Conclusion

AI agents do not require a brand-new architectural paradigm.

They can be **cleanly mapped into existing Clean Architecture principles**:

- **Domain = correctness**
- **Agent = tactical selection**
- **Orchestration = information flow**

With this separation, AI agents become **auditable, testable, and maintainable components** in enterprise systems — not magical black boxes, but reliable architecture citizens.

Coming next in Part 2: How to handle multi-stage pipelines (retrieval → rerank → generation → validation) and fit them into this architecture.

Artificial Intelligence | Software Architecture | Ai Agent | Enterprise Architecture