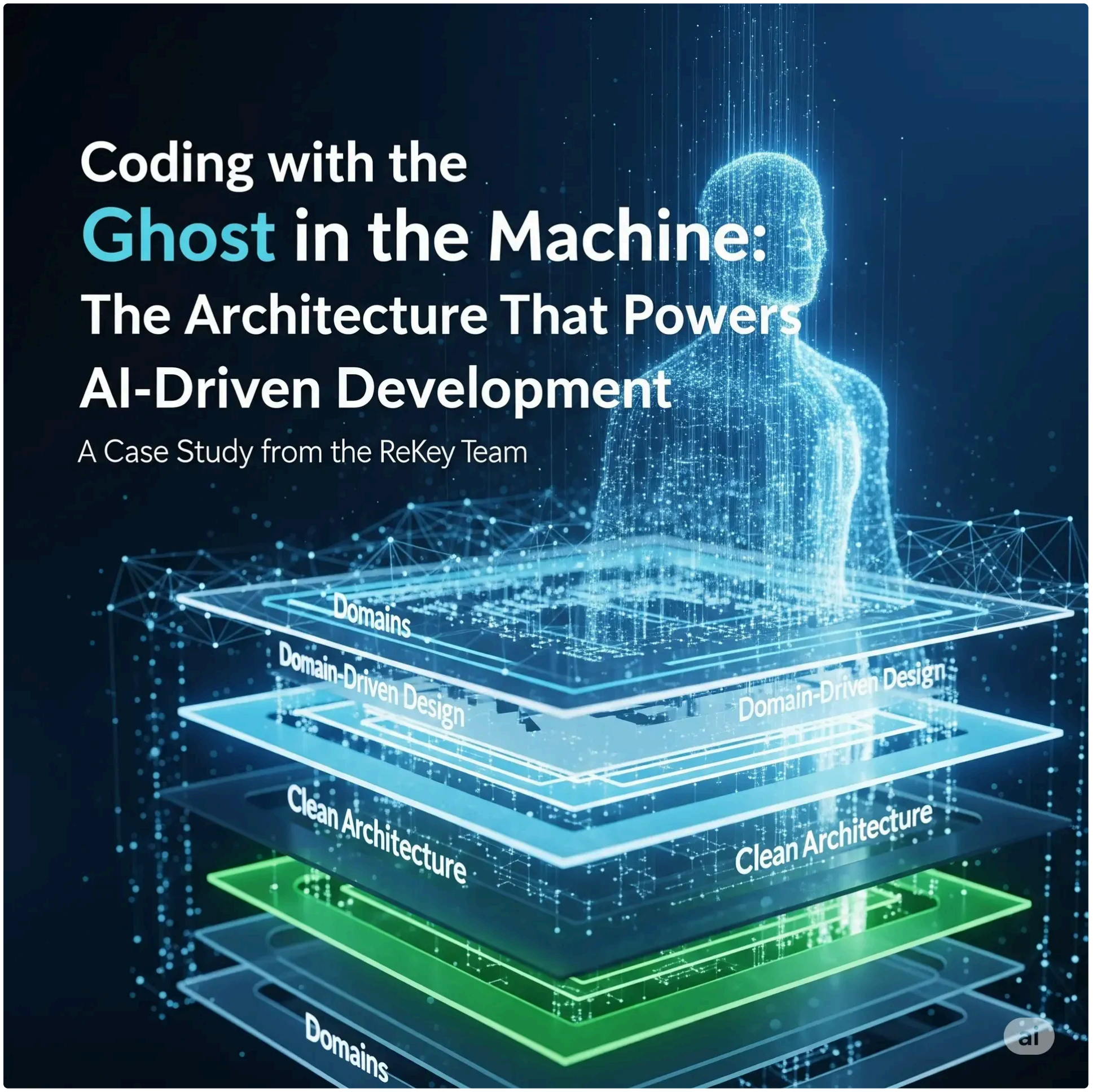
ARTICLE

Coding with the Ghost in the Machine: The Architecture That Powers AI-Driven Development

Huan Li Follow

Aug 09, 2025 · 6 mins read



A Case Study from the ReKey Team

We've all heard the promise of AI-powered development, or "vibe coding": describe a feature in plain English, and a tireless AI agent builds it. But anyone who has tried this on a real-world project knows the reality is often messy. The AI gets lost in a tangled codebase, makes changes in the wrong places, and breaks things you didn't expect.

The dream of seamless AI collaboration isn't a fantasy, but it depends on a secret prerequisite: **a disciplined architecture.**

At ReKey, we're building an AI-powered keyboard. It's a complex product where speed and stability are critical. From day one, we knew we wanted to leverage AI not just as a product feature, but as a core part of our development process. This meant we couldn't afford a "spaghetti code" architecture. We had to build a home that AI agents could navigate as easily as a human developer.

This is the story of that architecture—how we combined Domain-Driven Design (DDD) and Clean Architecture to solve common development problems and, more importantly, create a codebase that's truly ready for the future of AI-assisted programming.

The Pain We Knew We Had to Avoid

Before we laid down the first line of code for ReKey, we mapped out the problems we'd all faced on previous projects:

1. **The God Object:** Whether a `Massive ViewController` in UIKit or a sprawling `SwiftUI View`, all projects have a tendency to centralize logic in one place. This makes the code brittle, impossible to test in isolation, and a nightmare to onboard new developers (human or AI).
2. **Business Logic Scattered to the Winds:** When you need to know the rules for "transforming text," do you look in the UI, a network model, or a random helper file? Without discipline, critical business logic gets smeared across the entire codebase.
3. **The Fear of Change:** In a tangled architecture, every change is risky. Fixing a bug in the settings screen might break the keyboard extension. This fear slows down development and kills innovation.
4. **Vendor Lock-In:** Tightly coupling your core logic to external services like Firebase or a specific API means you can never easily swap them out. Your business logic becomes a prisoner of your infrastructure.

These problems don't just frustrate humans; they are kryptonite for AI coding agents, which thrive on clarity, context, and predictability.

Our Blueprint for Clarity: Clean Architecture Meets DDD

To solve these problems, we adopted a layered architecture based on two powerful philosophies:

- **Clean Architecture:** This provides the structure. It divides the app into four distinct layers, creating a one-way street for dependencies.
- **Domain-Driven Design (DDD):** This provides the soul. It insists that our code should speak the language of the business domain (e.g., "Transforms," "Tones," "Commands"), not technical jargon.

Here are the four layers of ReKey:

1. **Domain (The Heart):** This is the center of our universe. It contains our core business entities (`Transform`), rules, and repository *interfaces*. It knows nothing about databases, APIs, or UI. It's pure, distilled business logic. This is where we define *what* our app does.
2. **Application (The Orchestra Conductor):** This layer defines the specific use cases of our app, like `CreateTransformCommand` or `GetTransformsQuery`. It orchestrates the Domain objects to perform tasks. It doesn't know *how* the data is stored or displayed, only what needs to happen.
3. **Infrastructure (The Gritty Details):** This is where the outside world lives. It contains the concrete implementations of the interfaces defined in the Domain layer, like our `FirestoreTransformRepository`. It handles networking, databases, and other services. If it's a third-party SDK, it belongs here.
4. **Presentation (The Face):** This is the UI. For us, it's SwiftUI views and ViewModels. Its job is to present data and capture user input, delegating all the real work to the Application layer.

The golden rule is the **Dependency Rule**: dependencies can only point inwards. The UI knows about the Application layer, but the Domain layer knows nothing about the UI. This separation is our superpower. It means we can test our entire business logic without a UI, and we can change our database without touching a single business rule.

How This Architecture Becomes "AI-Native"

This is where the investment in architecture pays off for the future. A clean, layered architecture isn't just good for humans; it's fundamentally better for AI coding agents.

1. Focused Context Windows: When we ask an AI agent to work on a task, the architecture gives it natural blinders. If the task is "add a 'createdAt' timestamp to every Transform," the agent only needs to look at `Domain/Entities/Transform.swift`. It doesn't need to load the entire project's context. This high-signal, low-noise environment dramatically improves the AI's accuracy.

2. High-Level, "Vibe-Based" Prompting: Because our architecture mirrors our business, our prompts can be high-level. *Human Vibe:* "We need a way to save a new text transformation." *AI Translation:* "Create a `SaveTransformCommand` in the `Application` layer that uses the `TransformRepository` to store a `Transform` entity."

The architecture provides the "slots" for the AI to fill. It turns a vague "vibe" into a precise, actionable task with a clear location in the codebase.

3. Guardrails for Safety: The dependency rule acts as a set of guardrails for the AI. It's incredibly difficult for an AI working in the Domain layer to accidentally make a UI change. This makes us confident enough to delegate entire features to our AI partner, knowing the blast radius of any potential error is contained.

The Next Frontier: Literate-in-Place (LiP)

A clean architecture is our foundation, but we're already building the next floor. As outlined in our public RFC, **rfcs/01-literate-in-place.md**, we are adopting a "Literate-in-Place" (LiP) documentation style.

The principle is simple: **a source file should be a readable design document, with code serving as the implementation detail.** If you were to strip out all the code, the remaining comments should form a coherent narrative explaining the *what*, *why*, and *how* of the system.

LiP is the final bridge between human intent and machine execution. It allows an AI agent to not just see the code, but to understand the *rationale* behind it before making a single change. This is critical for maintaining the integrity of the system as it grows and evolves under the guidance of both human and AI developers.

Build the Foundation, Then Vibe

The dream of “vibe coding” is alive, but it doesn’t come for free. It’s earned through the discipline of building a clean, understandable, and robust architecture. By investing in this foundation, you’re not slowing yourself down; you’re building a superhighway for future development.

For us at ReKey, this isn’t just a theory. It’s how we build production software every day. It’s the architecture that lets us move fast, stay stable, and truly collaborate with the ghost in the machine.

ai architecture ricky

Written by Huan Li Follow

Founding Human & Architect | Author @ Wechaty, Silicon Adventist, Serial Entrepreneur, Angel Investor, Burner🔥