Managing Complexity in LLM-Assisted Software Development: The Cathedral Framework

Brad Dennis, Ph.D.

AI4SE Workshop - Cognitive Assistants Track
September 17-18, 2025

"Systems Engineering AI that Works"

Valkyrie Enterprises, LLC
An ISO 9001:2015 Certified Company
Huntsville, AL 35802

The Problem

The Problem With Vibe Coding

“Build an interactive personal finance dashboard with animated charts showing monthly expenses across different categories. Include a budget tracker with progress bars, an expense input form with category selection, and a savings goal visualizer. Use modern glassmorphism design with animated gradients and smooth transitions between different time periods (daily/weekly/monthly views).”

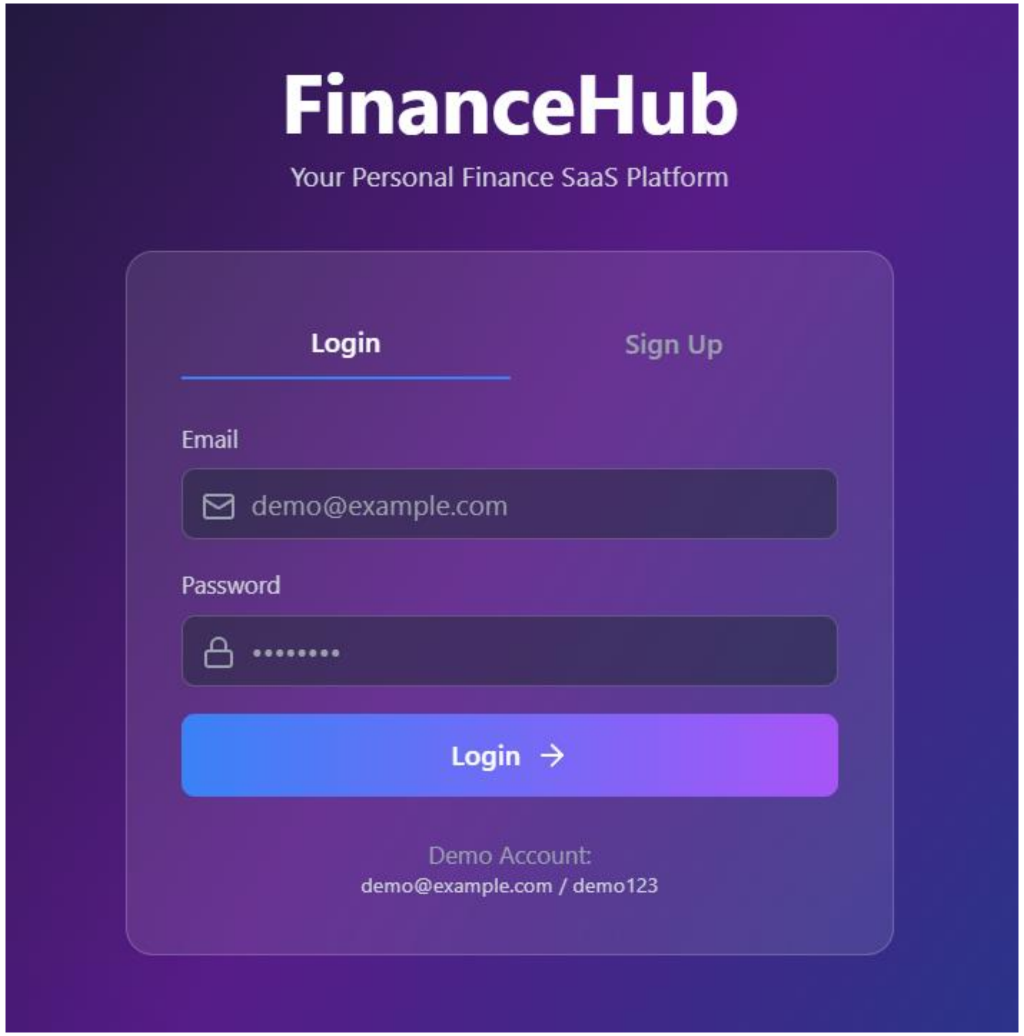
Available at: https://claude.ai/share/ce3c951b-dbd4-4e6c-99e8-c63d030d8ca0

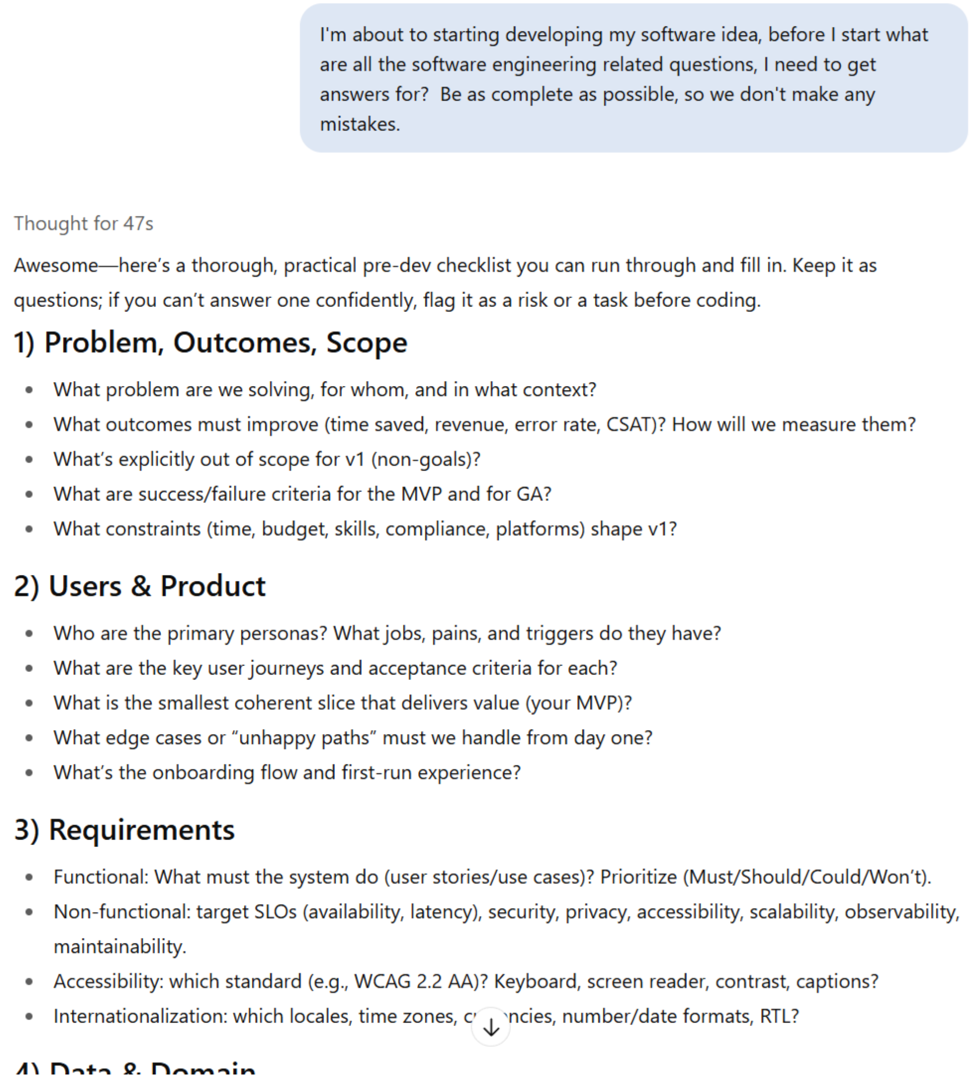
The Problem With Vibe Coding

“I'm about to starting developing my software idea, before I start what are all the software engineering related questions, I need to get answers for? Be as complete as possible, so we don't make any mistakes.”

Available at: https://chatgpt.com/share/68c93fa7-b8c8-800f-92fb-0c80ed756aa3

It's Not a Code Problem. It's a Control Problem.





I'm about to starting developing my software idea, before I start what are all the software engineering related questions, I need to get answers for? Be as complete as possible, so we don't make any mistakes.

Thought for 47s

Awesome—here's a thorough, practical pre-dev checklist you can run through and fill in. Keep it as questions; if you can't answer one confidently, flag it as a risk or a task before coding.

1) Problem, Outcomes, Scope

- What problem are we solving, for whom, and in what context?
- What outcomes must improve (time saved, revenue, error rate, CSAT)? How will we measure them?
- What's explicitly out of scope for v1 (non-goals)?
- What are success/failure criteria for the MVP and for GA?
- What constraints (time, budget, skills, compliance, platforms) shape v1?

2) Users & Product

- Who are the primary personas? What jobs, pains, and triggers do they have?
- What are the key user journeys and acceptance criteria for each?
- What is the smallest coherent slice that delivers value (your MVP)?
- What edge cases or "unhappy paths" must we handle from day one?
- What's the onboarding flow and first-run experience?

3) Requirements

- Functional: What must the system do (user stories/use cases)? Prioritize (Must/Should/Could/Won't).
- Non-functional: target SLOs (availability, latency), security, privacy, accessibility, scalability, observability, maintainability.
- Accessibility: which standard (e.g., WCAG 2.2 AA)? Keyboard, screen reader, contrast, captions?
- Internationalization: which locales, time zones, c ncies, number/date formats, RTL?

4) Data & Domain

Flattened Context = One Giant Lever

- **Feels magical** – but quickly turns messy
- **Flat prompting –** all the software engineering collapses into a stream of tokens, flattening the hierarchy and *hiding complexity*
- **Flattened hierarchies –** semantic cohesion is lost, word sense is mixed, model confusion is created
- **No evidence** – Without structure, there's nowhere to attach proofs or trace links to understanding change

We give an LLM one giant lever and expect it to be precise

Software Engineering Levers

- **Software abstractions are control levers -** requirements, interfaces, specifications, design patterns, tests, etc. are the levers humans use to control how we create software systems
- **Authority lives in the interface** – Interfaces at every level of abstraction define how a consumer interacts with the software and creates a binding encapsulation of responsibility within it
- **Predictability needs boundaries** – abstractions encode what matters at a level and hide the rest, in effect, it creates a binding of meaning
- **Traceability needs attachment points** – flattened hierarchies lead to invisible decision making

Software abstractions are how we control software development

Why This Matters

- **Critical systems can't afford hidden complexity** - when we flatten hierarchies with unstructured prompting, boundaries vanish, and complexity gets buried and becomes and operational risk.
- **DoD/regulated domains demand traceability** - audits and accreditation expect a defensible chain from intent → design → code → tests. If abstractions aren’t visible, there’s nowhere to *attach* proof
- **The verification burden is shifting, not disappearing** – while LLMs can accelerate code construction, oversight moves upstream, and the verification tax just hits later at a higher cost

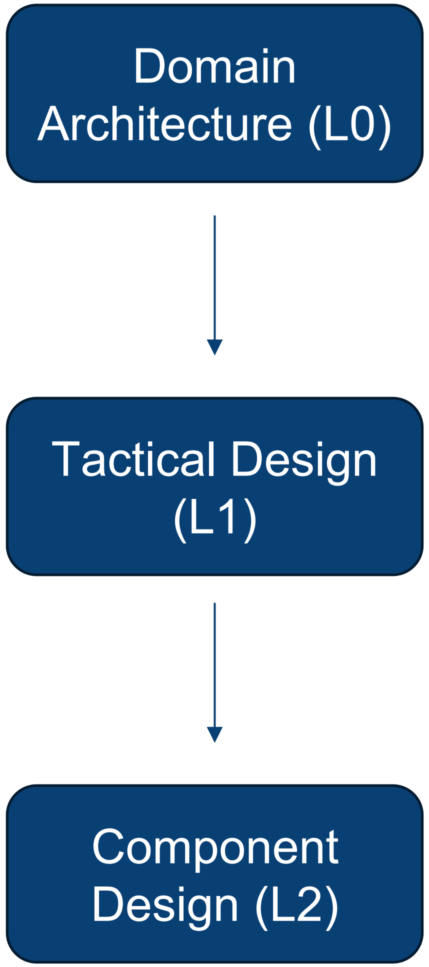
The Cathedral Framework

Core Insight: Put the Hierarchy Back in the Loop

- **We haven’t forgotten – we just aren’t applying it.**
  Flattened prompts dissolve hierarchy; the fix is to *re-surface* the layers and operate them as levers. *Make complexity visible*.
- **Progressive disclosure is controllable authority.**
  Each layer grants clear scope boundaries constraining the model’s behavior
- **Semantic boundaries mean predictable behavior.**
  Level-scoped language and clear encapsulation boundaries mitigates hallucinations and improves model understanding

Abstraction discipline is an engineering response to the failures of vibe coding

Cathedral: From Conceptual to Logical

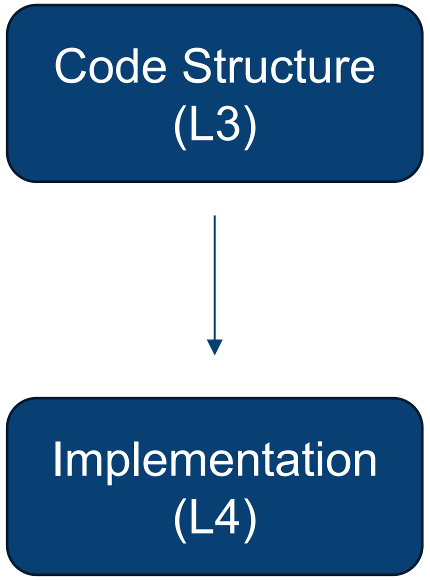


Transform requirements into encapsulated domain contexts with clear integration boundaries using domain-driven design (DDD) style language modeling and analysis. Boundaries are identified and augmented with acceptance criteria and other level relevant features, such as non-functional-requirements.

Takes an L0 specification and transforms it into detailed module specifications. We transition from conceptual work to logical work by analyzing the language again, but this time to organize the responsibilities into tactical, yet technical, patterns mapped across architectural layers.

Transforms an L1 specification and transforms it into component specifications ready for implementation, where a component is 1 to 5 tightly related classes. Appropriate design patterns are applied and component interactions are mapped.

Cathedral: to Concrete

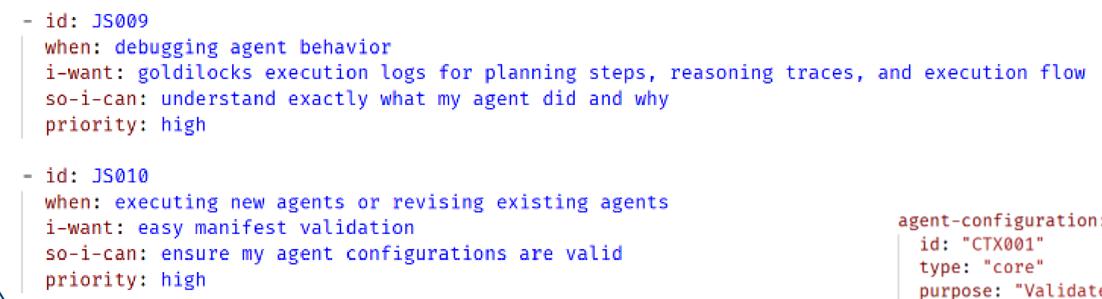


Transforms a component specification from L2 into concrete class signatures and interfaces. Extracts public and internal contracts, applies layer-specific patterns, designs method signatures following SOLID principles, and creates dependency injection patterns. Outputs class signatures and interface definitions ready for L4 implementation.

Creates tested code using test-driven design(TDD) practices. This is done incrementally starting with a class stub and a single ‘golden path’ test and ending with a full test suite of happy path, sad path, and edge case tests and fully realized class.

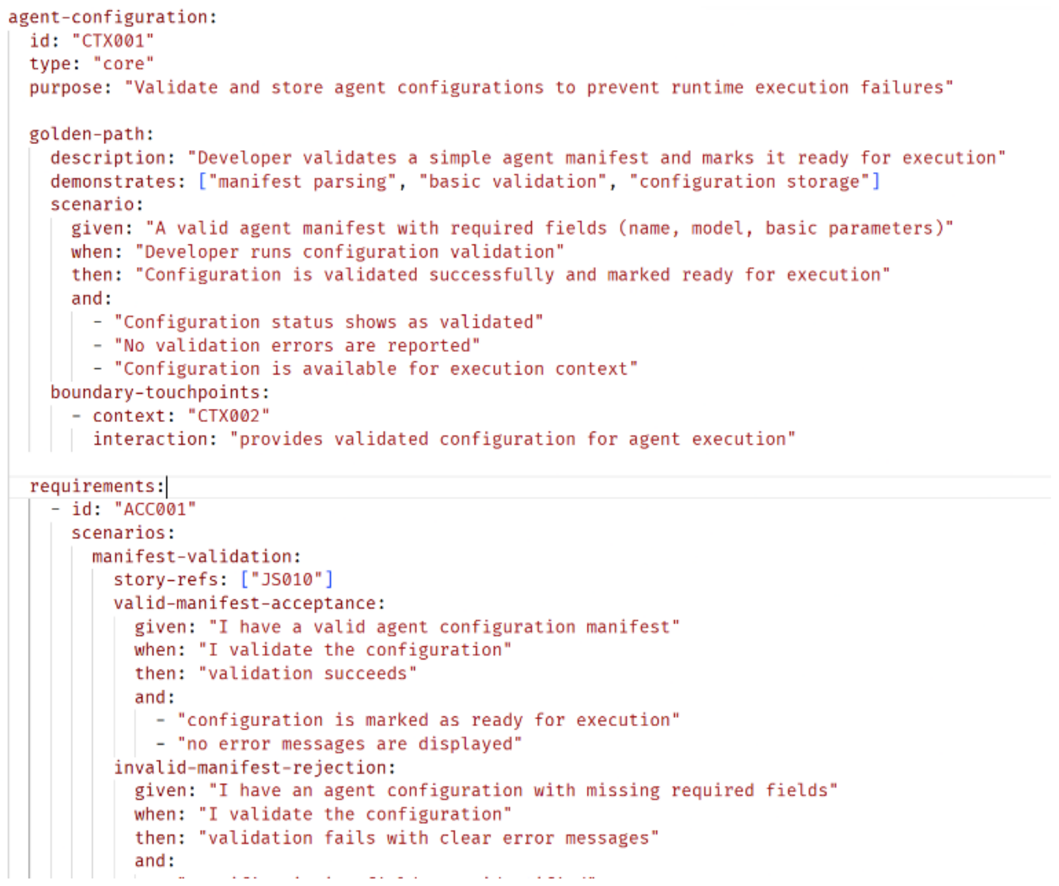
Cathedral: Walkthrough (L0 to L1)

0 We start with requirements as the input into L0:



```yaml
- id: JS009
  when: debugging agent behavior
  i-want: goldilocks execution logs for planning steps, reasoning traces, and execution flow
  so-i-can: understand exactly what my agent did and why
  priority: high

- id: JS010
  when: executing new agents or revising existing agents
  i-want: easy manifest validation
  so-i-can: ensure my agent configurations are valid
  priority: high
```

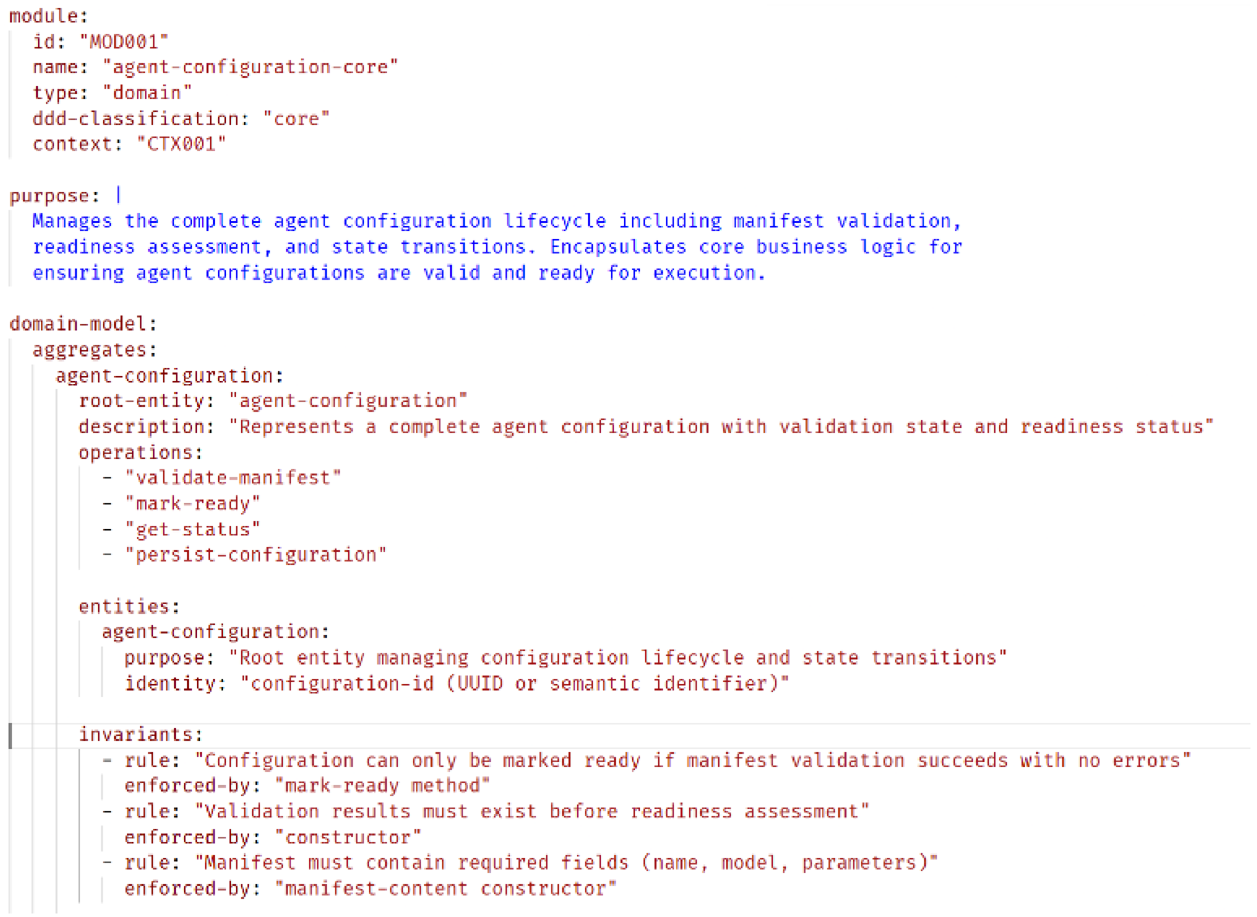
1 And produce domain context specifications that become the inputs into L1:



```yaml
agent-configuration:
  id: "CTX001"
  type: "core"
  purpose: "Validate and store agent configurations to prevent runtime execution failures"

  golden-path:
    description: "Developer validates a simple agent manifest and marks it ready for execution"
    demonstrates: ["manifest parsing", "basic validation", "configuration storage"]
    scenario:
      given: "A valid agent manifest with required fields (name, model, basic parameters)"
      when: "Developer runs configuration validation"
      then: "Configuration is validated successfully and marked ready for execution"
      and:
        - "Configuration status shows as validated"
        - "No validation errors are reported"
        - "Configuration is available for execution context"
    boundary-touchpoints:
      - context: "CTX002"
        interaction: "provides validated configuration for agent execution"

  requirements:
    - id: "ACC001"
      scenarios:
        manifest-validation:
          story-refs: ["JS010"]
          valid-manifest-acceptance:
            given: "I have a valid agent configuration manifest"
            when: "I validate the configuration"
            then: "validation succeeds"
            and:
              - "configuration is marked as ready for execution"
              - "no error messages are displayed"
          invalid-manifest-rejection:
            given: "I have an agent configuration with missing required fields"
            when: "I validate the configuration"
            then: "validation fails with clear error messages"
            and:
```

Cathedral: Walkthrough (L1 to L2)

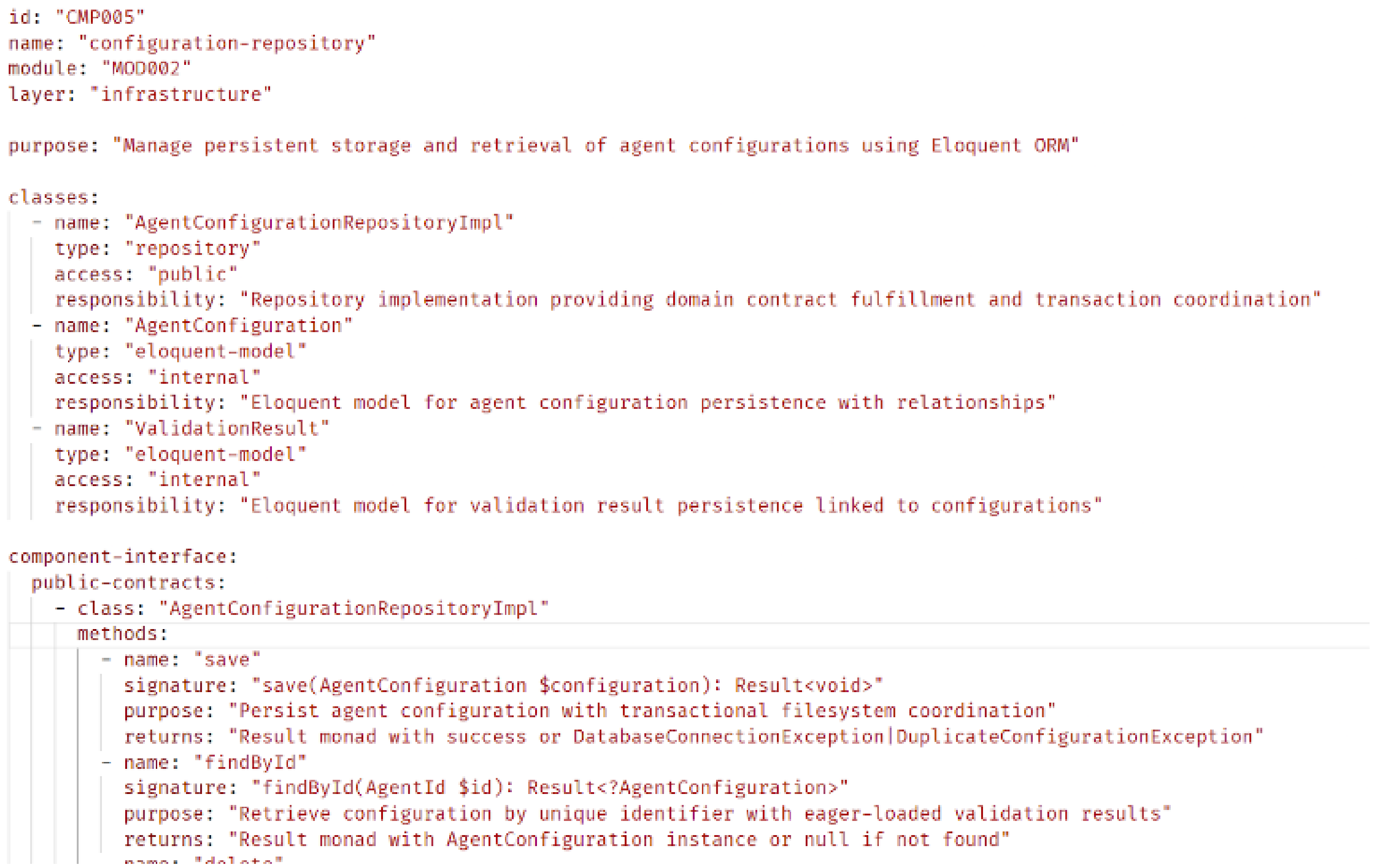
2 The L1 domain specification gets transformed into module specifications and become the L2 input:



```yaml
module:
  id: "MOD001"
  name: "agent-configuration-core"
  type: "domain"
  ddd-classification: "core"
  context: "CTX001"

purpose: |
  Manages the complete agent configuration lifecycle including manifest validation,
  readiness assessment, and state transitions. Encapsulates core business logic for
  ensuring agent configurations are valid and ready for execution.

domain-model:
  aggregates:
    agent-configuration:
      root-entity: "agent-configuration"
      description: "Represents a complete agent configuration with validation state and readiness status"
      operations:
        - "validate-manifest"
        - "mark-ready"
        - "get-status"
        - "persist-configuration"

      entities:
        agent-configuration:
          purpose: "Root entity managing configuration lifecycle and state transitions"
          identity: "configuration-id (UUID or semantic identifier)"

      invariants:
        - rule: "Configuration can only be marked ready if manifest validation succeeds with no errors"
          enforced-by: "mark-ready method"
        - rule: "Validation results must exist before readiness assessment"
          enforced-by: "constructor"
        - rule: "Manifest must contain required fields (name, model, parameters)"
          enforced-by: "manifest-content constructor"
```

Cathedral: Walkthrough (L2 to L3)

3 The module specification gets transformed into component specifications:



```
id: "CMP005"
name: "configuration-repository"
module: "MOD002"
layer: "infrastructure"

purpose: "Manage persistent storage and retrieval of agent configurations using Eloquent ORM"

classes:
  - name: "AgentConfigurationRepositoryImpl"
    type: "repository"
    access: "public"
    responsibility: "Repository implementation providing domain contract fulfillment and transaction coordination"
  - name: "AgentConfiguration"
    type: "eloquent-model"
    access: "internal"
    responsibility: "Eloquent model for agent configuration persistence with relationships"
  - name: "ValidationResult"
    type: "eloquent-model"
    access: "internal"
    responsibility: "Eloquent model for validation result persistence linked to configurations"

component-interface:
  public-contracts:
    - class: "AgentConfigurationRepositoryImpl"
      methods:
        - name: "save"
          signature: "save(AgentConfiguration $configuration): Result<void>"
          purpose: "Persist agent configuration with transactional filesystem coordination"
          returns: "Result monad with success or DatabaseConnectionException|DuplicateConfigurationException"
        - name: "findById"
          signature: "findById(AgentId $id): Result<?AgentConfiguration>"
          purpose: "Retrieve configuration by unique identifier with eager-loaded validation results"
          returns: "Result monad with AgentConfiguration instance or null if not found"
```

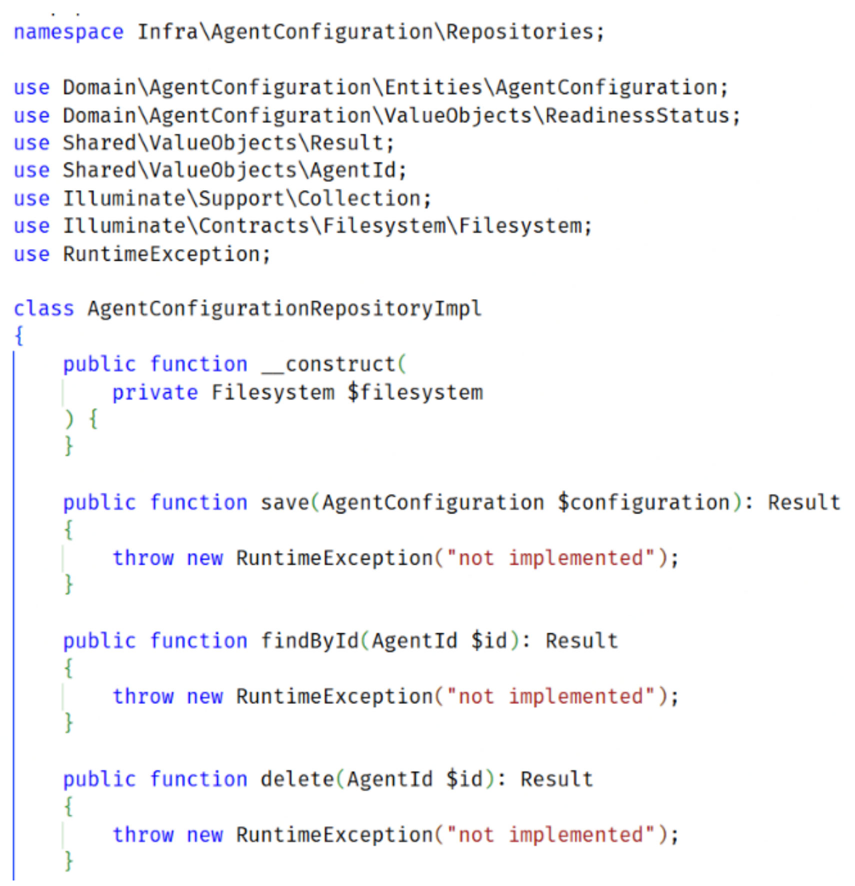
Cathedral: Walkthrough (L3 to L4)

4 Component specifications get transformed into class signatures:

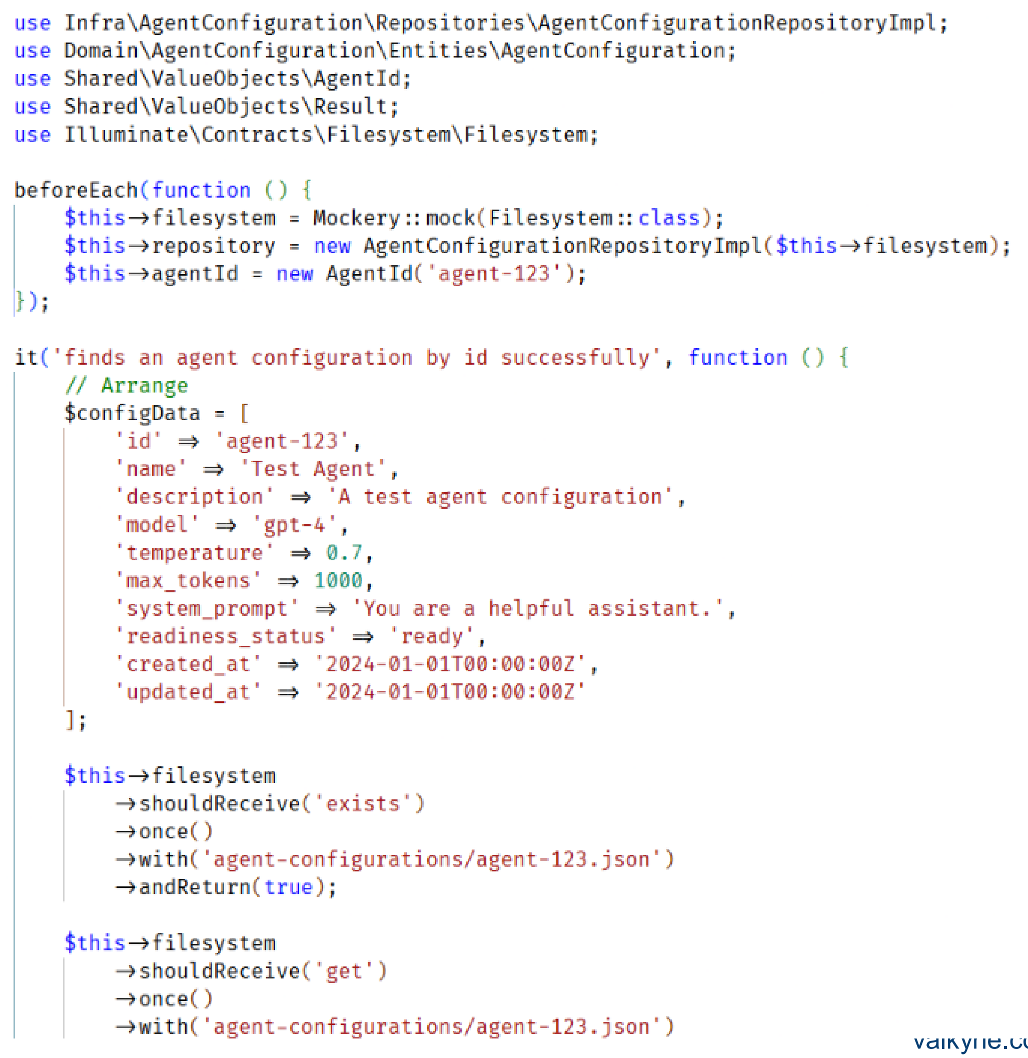
```
namespace Infra\AgentConfiguration\Repositories;

use Domain\AgentConfiguration\Entities\AgentConfiguration;
use Domain\AgentConfiguration\ValueObjects\ReadinessStatus;
use Shared\ValueObjects\Result;
use Shared\ValueObjects\AgentId;
use Illuminate\Support\Collection;

class AgentConfigurationRepositoryImpl
{
    public function save(AgentConfiguration $configuration): Result;
    public function findById(AgentId $id): Result;
    public function delete(AgentId $id): Result;
    public function exists(AgentId $id): Result;
    public function findAll(): Result;
    public function findReadyConfigurations(): Result;
    public function findByReadinessStatus(ReadinessStatus $status): Result;
    public function countByReadinessStatus(ReadinessStatus $status): Result;
}
```

Cathedral: Walkthrough (L4 to Working Code)

4 The class signatures are used to anchor the L4 TDD incremental workflow to produce working code and tests:



```php
namespace Infra\AgentConfiguration\Repositories;

use Domain\AgentConfiguration\Entities\AgentConfiguration;
use Domain\AgentConfiguration\ValueObjects\ReadinessStatus;
use Shared\ValueObjects\Result;
use Shared\ValueObjects\AgentId;
use Illuminate\Support\Collection;
use Illuminate\Contracts\Filesystem\Filesystem;
use RuntimeException;

class AgentConfigurationRepositoryImpl
{
    public function __construct(
        private Filesystem $filesystem
    ) {
    }

    public function save(AgentConfiguration $configuration): Result
    {
        throw new RuntimeException("not implemented");
    }

    public function findById(AgentId $id): Result
    {
        throw new RuntimeException("not implemented");
    }

    public function delete(AgentId $id): Result
    {
        throw new RuntimeException("not implemented");
    }
```



```php
use Infra\AgentConfiguration\Repositories\AgentConfigurationRepositoryImpl;
use Domain\AgentConfiguration\Entities\AgentConfiguration;
use Shared\ValueObjects\AgentId;
use Shared\ValueObjects\Result;
use Illuminate\Contracts\Filesystem\Filesystem;

beforeEach(function () {
    $this→filesystem = Mockery::mock(Filesystem::class);
    $this→repository = new AgentConfigurationRepositoryImpl($this→filesystem);
    $this→agentId = new AgentId('agent-123');
});

it('finds an agent configuration by id successfully', function () {
    // Arrange
    $configData = [
        'id' ⇒ 'agent-123',
        'name' ⇒ 'Test Agent',
        'description' ⇒ 'A test agent configuration',
        'model' ⇒ 'gpt-4',
        'temperature' ⇒ 0.7,
        'max_tokens' ⇒ 1000,
        'system_prompt' ⇒ 'You are a helpful assistant.',
        'readiness_status' ⇒ 'ready',
        'created_at' ⇒ '2024-01-01T00:00:00Z',
        'updated_at' ⇒ '2024-01-01T00:00:00Z'
    ];

    $this→filesystem
        →shouldReceive('exists')
        →once()
        →with('agent-configurations/agent-123.json')
        →andReturn(true);

    $this→filesystem
        →shouldReceive('get')
        →once()
        →with('agent-configurations/agent-123.json')
```

Cathedral: Key Mechanisms

- **Abstraction discipline**: level-scoped language that’s congruent with and contextual boundaries aligned with the software engineering task
- **Digital threads**: a structured process with intermediate artifacts enables traceability and transparency.
- **Goldilocks principle:** since tasks are well defined and appropriately scoped, we can engineer just the right amount of context for the LLM to understand our intent and steer its work
- **Failing Fast**: early defects cascade and surface quickly

Principles & Heuristics You Can Use Today

- **Semantic cohesion is paramount** – everything in your prompt influences the LLMs behavior, the instructions must be aligned with the task.
- **Polysemy can confuse LLMs** – word sense matters, if an LLM stubbornly misbehaves, look for explicit or implicit mixed word sense.
- **Too big to comply** – if the LLM consistently ignores your output format or key instructions, the context is probably too large
- **Too small to contextualize** – if the LLM adheres to your output format, but repeatedly misses something important, your context probably needs augmentation.
- **Too helpful** – when the LLM consistently over-engineers or over-produces, your task is probably not bounded enough, or your language carries hidden implications: e.g., "You are an expert back-end developer…"