Managing Complexity for LLM Software Co-Pilots Through Abstraction-Aware Context Modeling

Brad Dennis
Valkyrie Enterprises, LLC
brad.dennis@valkyrie.com

Mason Nixon
Valkyrie Enterprises, LLC
mason.nixon@valkyrie.com

Daniel Claxton
Valkyrie Enterprises, LLC
daniel.claxton@valkyrie.com

Markus Kreitzer
People Tech Group
mkreitzer@peopletech.com

Luke Ebbinghaus
Valkyrie Enterprises, LLC
luke.ebbinghaus@valkyrie.com

***Abstract*—LLM co-pilots accelerate software delivery but often flatten hierarchical artifacts into undifferentiated prompts, hiding complexity and eroding reliability, traceability, and safety. We present Cathedral, an abstraction-aware context model and workflow that scopes prompts, retrieval, and artifacts to five levels (L0–L4) from mission intent to tested implementation. Cathedral applies progressive disclosure and sense discipline to keep level-specific meanings stable, and threads digital identifiers across requirements, design, code, and tests to form a reviewable digital thread. We demonstrate the approach with a reference implementation and show how it maps to DoD systems-engineering processes. We also outline an evaluation plan measuring correctness, security, and traceability, including adversarial probes for cross-level leakage and prompt injection. By first codifying the method as a manual workflow, we provide a validated blueprint for future cognitive assistants. Early evidence and prior studies indicate that level-scoped context reduces hallucinations, improves task alignment, and yields more auditable outputs in LLM-assisted development.**

Keywords—abstraction-aware prompting, progressive disclosure, context modeling, digital thread, systems engineering, DoD, reliability, traceability, prompt security.

I. Introduction

Software engineering has learned to manage complexity by making it visible. From Parnas' modules to modern microservices, abstraction boundaries have required developers to confront dependencies, interfaces, and scope [1]. Large language models disrupt this balance. In "vibe coding," features can be requested in plain English and these materialize as working code [2] [3]. This accelerates prototyping, but it makes complexity disappear not because it is solved, but because it is hidden [4] [5]. Hidden complexity is unacceptable in mission-critical domains like avionics, command systems, and critical infrastructure. These fields demand the highest standards of safety, security, and traceability.

The challenge extends beyond technical concerns to epistemological ones. Traditional software programs are typically deterministic systems with well-defined inputs, outputs, and operational scopes. In contrast, large language models (LLMs) are probabilistic systems that generate outputs by predicting token sequences based on statistical patterns and processing input within fixed-length context windows using attention mechanisms. In this process, structured hierarchies degrade into linear token streams. This context impedance [4] [5] treats all information as equally relevant to the task, allowing noise to obscure critical details. This results in degraded model performance and an increased verification burden on AI agents and developers [6] [7]. These risks are magnified in defense and systems-of-systems engineering.

Our proposal is a systems engineering response: reintroducing abstraction discipline into the interaction itself. We define five discrete abstraction levels (L0–L4), each representing an order-of-magnitude reduction in scope, from domain architecture down to tested implementation. By aligning prompt context with these levels and applying established principles of decomposition and encapsulation, we achieve both progressive disclosure [8] [9] and modular design [1]. Shared systems engineering foundations align with modern lifecycle standards (ISO/IEC/IEEE 15288, 29148, 12207) and digital engineering practices. Today this framework functions as an agent-assisted but mostly manual workflow for developers. In the future, it offers a blueprint for automated cognitive assistants and co-pilots [10] [11]. We refer to this framework as Cathedral, in homage to Raymond's "cathedral" [12]. Even in this disruptive era of LLM-assisted development, our fundamental goal persists: expose hidden complexity to make it manageable.

Next: Section 2 details motivating evidence and gaps; Section 3 defines the decomposition framework we use throughout.

II. Motivation and Related Work

AI co-pilots accelerate prototyping but can amplify hidden complexity. Natural language briefs become code without clarifying assumptions, which makes dependencies implicit and errors harder to audit. In mission-critical domains, this brittleness and lack of traceability are unacceptable.

This is not only about context length; prompt flooding yields "lost-in-the-middle" failures [4][5], and the system suffers from semantic overload. Key terms carry different senses across abstraction layers: "contract" may mean legal obligations, an

application programming interface (API) or interface definition language (IDL) specification, or design-by-contract assertions; "interface" can be an architectural boundary or a language keyword. When prompts and retrieval blend these senses, they inject unnecessary semantics, contaminate context with near-miss evidence, and confuse the model's objective. This semantic incongruence creates a form of context impedance because the model is asked to satisfy conflicting senses at once. Beyond degraded quality, context impedance undermines intent fidelity and raises security risk: as level-specific meanings blur, boundaries erode, inviting prompt injection, unsafe tool calls, and cross-level leakage.

Abstraction-aware sense discipline addresses this. It works at discrete levels, enforces semantic borders where meanings change, and constrains both prompts and retrieval to the level-appropriate sense of each term. This improves signal to noise, reduces hallucination risk, and sustains traceability, while aligning with long-standing DoD software engineering practice. In high-assurance settings, abstraction discipline is a security control: level-scoped language acts as policy for what the model may consider. In our validation proposal, we operationalize this claim with adversarial probes and Common Weakness Enumeration(CWE)-mapped findings.

A. Flat Prompting's Limitations

Most current AI development tools rely on flat prompt injection: concatenating requirements, source code, and metadata into a single request. Studies consistently show this approach is brittle. Pinecone's analysis of context stuffing finds that larger inputs often reduce quality and increase hallucination [5]. Stanford's "Lost in the Middle" study confirms that LLMs underutilize long contexts, especially when relevant details are buried [4]. This evidence motivates the search for structured alternatives.

B. Retrieval-Augmented Generation (RAG)

One family of approaches tackles context overflow with retrieval. RAG pipelines fetch the most relevant documents from external stores instead of flooding the prompt with all possible material. This improves factual accuracy and keeps prompts lean. However, retrieval focuses on what to include, not how to constrain meaning. In software engineering tasks, where terms shift sense across abstraction levels, RAG alone cannot enforce abstraction discipline or prevent retrieval contamination from near-miss documents [13] [14] [15].

C. Summarization and Context Distillation

Another line of work compresses large contexts into summaries. Research on summarizers, windowed memory, and context distillation attempts to maintain salience within LLM limits. These methods mitigate token overflow but risk losing traceability: once details are collapsed, developers cannot easily verify what was retained. Summarization therefore improves efficiency but not alignment with software modularity [16].

D. Hierarchical and Progressive Prompting

Staged prompting addresses complexity by sequencing information. Lo et al. show that hierarchical prompting first consolidates observations and then passes them to an action prompt, which improves results on long-context tasks [8]. PromptChainer similarly demonstrates the value of modular prompt nodes [9]. These echo progressive disclosure, but they do not yet formalize layer-specific terminology or SE decomposition.

E. Structured Reasoning, Decomposition, and Tool Use

A complementary thread externalizes complexity through intermediate reasoning and programmatic tools. Chain-of-thought and related methods improve multi-step reasoning by eliciting intermediate rationales and sampling multiple reasoning paths [17] [18] [19] [20]. Tool-augmented prompting delegates subproblems to calculators, search, or code execution, reducing cognitive load on the model and increasing verifiability [21] [22] [23]. Orchestration frameworks and prompt compilers make these patterns modular and tunable across tasks [24].

These techniques mitigate flat prompting's brittleness, but they typically optimize within a single blended context. Without explicit level scoping, they can still mix senses (e.g., "interface" as contract vs. UI) and propagate near-miss evidence. Constrained decoding demonstrates that typed outputs reduce error, e.g., grammars for Text-to-SQL, but constraints alone do not align meanings across abstraction layers [25]. Cathedral's contribution is to pair these advances with abstraction-aware boundaries so that reasoning traces, tool calls, and outputs remain consistent with level-appropriate semantics.

F. Cognitive Load and Human Factors

Prompting is a human problem as much as a technical one. Subramonyam et al. describe the "gulf of envisioning" when turning goals into prompts [6]. GitHub surveys report mixed effects on cognitive load, with increased verification burden for less experienced developers [7]. Structure and level awareness can reduce this burden.

G. Abstraction and Modularity in AI Workflows

Frameworks such as Pocketflow emphasize modular, declarative flows for human-AI co-development [11]. Microsoft's AutoGen and OpenAI tool-calling show modular orchestration with clear boundaries [10] [26]. These validate modularity, yet they stop short of explicit mappings to abstraction levels or terminology control.

H. Spec-Driven Development

Github's Spec Kit operationalizes a gated flow from transforming the user's intent to implementation, validating artifacts before proceeding [27]. This disciplines AI workflows, but it does not formalize layer-specific senses or traceability across system-of-systems boundaries.

I. Traceability and Verification in Regulated Domains

Requirements engineering emphasizes alignment across intent, design, and implementation and the need for explainability in regulated systems [28] [29]. Existing guidance on traceability does not prescribe concrete methods for AI assisted software engineering practices.

J. Semantic Congruency and Level-Scoped Language

Key terms often shift meaning by abstraction level. For example, 'contract' may refer to legal obligations, an application programming interface (API) or interface definition language (IDL) specification, or design-by-contract assertions. 'Interface'

may denote an architectural boundary or a language construct. Mixing these senses injects unnecessary semantics and confuses model objectives. Level-scoped language constraints prompts and retrieval to the level-appropriate sense, which improves precision and auditability [30] [31] [32] [33].

K. Level-Aware Retrieval and Filtering

Retrieval quality strongly affects generation quality. Polysemy, words having several related but distinct meanings, introduces hard negatives and near-miss context that harms faithfulness [14] [15] [34]. Indexing and querying by abstraction level, and filtering candidates with level tags, may reduce contamination and align with software engineering (SE) modularity.

These approaches reveal both progress and limitations. Retrieval and summarization improve efficiency but neglect modular alignment. Hierarchical prompting and artifact-driven workflows like Spec Kit demonstrate the power of staged disclosure but have not been explicitly tied to SE decomposition principles and are limited by system complexity. Traceability research emphasizes the need for assurance, but current AI co-pilots lack the mechanisms to enforce it.

Thus, there remains a critical gap: no framework systematically maps systems engineering abstractions, from domain architecture down to implementation, to LLM context modeling. This gap is especially acute for defense and mission-critical domains, where verification, modularity, and artifact traceability are as important as functional performance.

Our framework introduces abstraction-aware context modeling, a principled method we call Cathedral in homage to the 'cathedral' model: the top-down, centralized foil to the bazaar in Eric Raymond's 1997 essay The Cathedral and the Bazaar. It applies SE decomposition directly to LLM prompt and context design. Unlike ad hoc retrieval or summarization, it enforces boundaries across five abstraction levels, ensuring prompts are scoped neither too broadly nor too narrowly. This level discipline also functions as a soft security perimeter that enables treating security as a boundary property and not an afterthought. The result is a workflow that provides manual best practices today and a blueprint for automated, context-aware AI co-pilots tomorrow.

We next formalize this SE-mapped, abstraction-aware framework, Cathedral, in Section 3.

III. The Cathedral Framework: An SE Approach to LLM Context Management

The Cathedral framework is our central contribution: a systems engineering (SE) approach to structuring LLM interactions by enforcing abstraction boundaries and level-scoped language. Section 2 showed how semantic overload from polysemy, where common software engineering terms like 'contract', 'policy', 'resource' or 'component' shift sense across abstraction levels, injects near-miss evidence and confuses model objectives. Cathedral restores modular decomposition and constrains each interaction to the level-appropriate sense, so only task-relevant, semantically congruent information reaches the model. This reduces context impedance and cognitive overload, lowers hallucinations, and reestablishes traceability across levels [1] [3] [11].

Classic SE disciplines remain effective even when the "system under design" is an LLM context window. Treating prompt construction as architectural decomposition aligns with DoD Digital Engineering and MOSA principles at each layer, emphasizing clear interfaces, model-like traceability, and testable increments. At the same time, this preserves the Goldilocks Principle of context engineering: neither under-specified nor over-detailed, when guiding the model [8] [9].

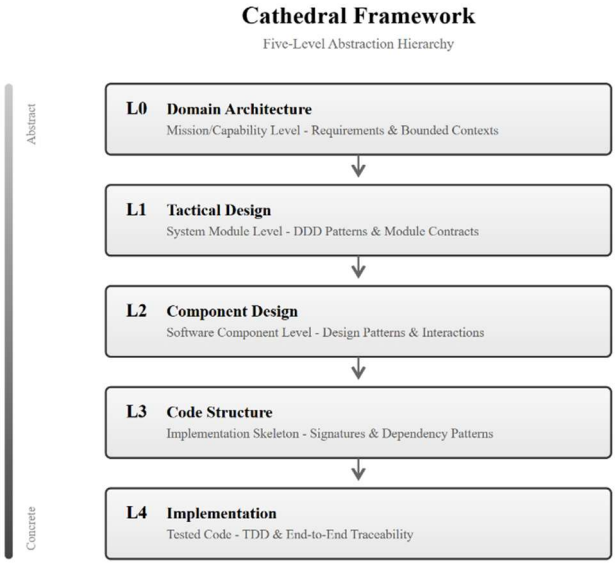

Fig. 1. Cathedral's Abstraction Layers

Cathedral's five-level decomposition flows down through the conceptual, i.e. what are we building, to the logical, i.e. how are we going to build it, and finally to concrete software implementation:

- **L0: Domain Architecture (Mission/Capability Level)** Smooths the requirements language and defines bounded contexts from job or user stories and mission goals. Extracts ubiquitous language, capability boundaries, and integration seams. Provides the architectural map against which all subsequent levels trace.
- **L1: Tactical Design (System Module Level)** Translates domain architecture into module specifications using Domain Driven Design (DDD) and other tactical encapsulation patterns (aggregates, repositories, domain events, Command Query Responsibility Segregation (CQRS) and others. Establishes contracts and interfaces that isolate and structure downstream component work [1].
- **L2: Component Design (Software Component Level)** Groups related elements into cohesive components, applies design patterns, and defines inter-component interactions. Ensures modularity and

architectural consistency before code is generated [11].

- **L3: Code Structure (Implementation Skeleton)** Produces class signatures, method stubs, and dependency injection patterns. Maintains design best practices and specifies public/internal contracts. At this level, LLM prompts are scoped to structural artifacts, not business abstractions [3].
- **L4: Implementation (Tested Code)** Expands structure into fully tested code via incremental, context managed, test-driven development (TDD) and refactoring. Keeps implementations aligned with upstream abstractions and preserves end-to-end traceability from mission needs to unit tests.

At every level, outputs form both artifacts for the developer and context inputs for the LLM. This layering ensures:

1. **Traceability**: Model-based links from capabilities (L0) to implementation (L4), maintaining lineage across artifacts [28].
2. **Verification**: Each layer defines measurable acceptance, verification and other exit criteria appropriate to its scope [11].
3. **Cognitive load reduction**: Developers and models operate only on context scoped to the current layer, reducing near-miss interference [6] [7].

In its current form, Cathedral functions as a semi-automated but mostly manual workflow: a guide for developers to craft prompts that respect abstraction boundaries. In the long term, it offers a blueprint for automation. Future coding assistants and autonomous agents can operate within well-defined, semantically congruent, bounded encapsulations according to the same decomposition logic [4] [5] to produce complex software systems with the rigor needed for sensitive domains. Next: Section 4 details how we intend to validate the framework empirically, and Section 5 discusses Cathedral's application in DoD related work.

At every level, outputs form both artifacts for the developer and context inputs for the LLM. This layering ensures:
Traceability: Model-based links from capabilities (L0) to implementation (L4), maintaining lineage across artifacts [28].
Verification: Each layer defines measurable acceptance, verification and other exit criteria appropriate to its scope [11].
Cognitive load reduction: Developers and models operate only on context scoped to the current layer, reducing near-miss interference [6][7].
In its current form, Cathedral functions as a semi-automated but mostly manual workflow: a guide for developers to craft prompts that respect abstraction boundaries. In the long term, it offers a blueprint for automation. Future coding assistants and autonomous agents can operate within well-defined, semantically congruent, bounded encapsulations according to the same decomposition logic [4] [5] to produce complex software systems with the rigor needed for sensitive domains. Next: Section 4 details how we intend to validate the framework empirically, and Section 5 discusses Cathedral's application in DoD related work.

IV. Proposed Evaluation Method

We propose to evaluate Cathedral against a parity flat-prompt baseline in a controlled study on defense-relevant software engineering tasks. Primary endpoints are functional correctness, security (Common Weakness Enumeration-mapped findings), and requirements traceability; secondary measures capture iteration count and perceived cognitive load. Red-team and team-level cognitive studies are deferred to future work, this study isolates core technical effects under lab conditions

A. Hypotheses

We test the following hypotheses:

- **H1**: Cathedral (L0–L4) yields higher functional correctness than flat prompting.
- **H2**: Cathedral reduces security-relevant findings.
- **H3**: Cathedral increases requirements traceability and coverage.
- **H4** (secondary): Cathedral reduces iteration count and perceived cognitive load.

B. Participants and Design

We plan to recruit software engineers (junior–senior) and administer a brief pre-test to estimate baseline skill. We will use a matched-pairs, between-subjects design: within each pair, one participant is randomized to Cathedral and the other to the flat-prompt baseline. Each participant will complete multiple tasks within their assigned condition only (no cross-condition exposure). Task order will be balanced via a Latin square or balanced incomplete block design to control for order and difficulty effects while avoiding carryover.

Training will be brief and standardized per condition, followed by a single practice task to normalize familiarity without exposing participants to the alternate condition.

C. Baseline, Tasks, and Materials

Participants in the baseline arm receive the same task briefs, acceptance tests, and available context as the Cathedral arm. No level-scoped semantics or boundary rules are provided. We supply standardized baseline prompt templates and examples to avoid handicapping. We are considering a possible extension, time permitting. We wish to evaluate a hierarchical-prompting condition without explicit L-level boundary rules to isolate the contribution of boundary enforcement from generic staging.

We propose to evaluate Cathedral using three defense-relevant tasks, each provisioned with necessary artifacts and acceptance tests carrying requirement IDs to enable traceability. These tasks are summarized as:

1. Secure telemetry ingest: Role-based Access Control (RBAC), audit, data retention; tests for authorization, Personally Identifying Information (PII) redaction, audit integrity, and error handling.

2. Communications module with protocol negotiation: negotiation/fallback contracts; tests for negotiation paths, timeouts/retries, error injection, and recovery.
3. Policy rule engine: domain events and policy contracts; tests for precedence, overrides, and deny-by-default behavior.

Study artifacts will include standardized Cathedral templates, parity baseline templates, fixture datasets, and a seed-locked prompt runner to ensure reproducibility.

D. Procedure

Participants will complete training and a practice task in their assigned condition, then perform time-boxed tasks with all prompts, model outputs, and edits logged. Artifacts submitted include code, tests, configuration, and brief rationale notes. After each task, participants will be asked to complete a short NASA Task Load Index (NASA-TLX) survey and to record any perceived clarity/issues encountered.

E. Measures

We will use three primary outcomes: correctness, security, and traceability. Correctness is quantified by unit and integration test pass rates and rubric-based acceptance of deliverables. Security is assessed via static analysis findings mapped to CWE categories with severity weighting, complemented by targeted checks for secret handling and potential injection sinks. Traceability is evaluated through precision and recall of requirement-to-test and requirement-to-code links, together with scored conformance to L1 contracts, L2 module boundaries, and L3 component interfaces; we will also report coverage gaps.

Secondary outcomes will capture efficiency, maintainability, robustness, and human factors. Efficiency includes time to completion, tokens consumed, and the number of prompt–edit iterations required to reach acceptance. Maintainability is measured by changes in code complexity and defect density under small requirement changes. Robustness is assessed by the correctness of patches following minor requirement modifications. Human factors are measured using the NASA-TLX composite and its subscales.

F. Instrumentation and Reproducibility

We plan to use a unified logging pipeline for prompts/tokens/outputs and artifact diffs, an automated test harness, static analysis/software composition analysis (SCA) where applicable, and automated link checks for traceability. Requirement IDs are propagated across artifacts (L0→L4). A seed-locked prompt runner, fixture datasets, and anonymization scripts support reproducibility. We will preregister hypotheses and the analysis plan when appropriate and release templates, scripts, and anonymized logs according to controlling policies.

G. Threats to Validity

This evaluation may be affected by selection bias, construct drift, stochasticity, baseline parity confounds, task leakage, and limits on generalizability. Volunteer participants may not represent the target system engineering population; we stratify recruitment by experience and domain. Definitions of "traceability" and "conformance" are themselves polysemic and may vary; we will publish rubrics with worked examples and calibrate raters before scoring. Stochastic LLM outputs and API changes threaten reproducibility; here we can fix temperature=0, capture retries in logs, pin versions and endpoints, and enable replayable runs. Flat-prompt baselines can be unintentionally handicapped by token caps or summarization; we intend to enforce explicit information-parity rules and document token budgets. Prior familiarity with specific libraries or patterns can advantage participants; we may rotate stacks across tasks or screen for prior exposure. Because tasks are defense-flavored and time-boxed, generalization to broader enterprise settings and larger systems may be limited; we will report these limits and expand the task bank in follow-on phases.

H. Analysis and Decision Criteria

Mixed-effects models will be fitted with our condition as a fixed effect and random intercepts for pair and task; pre-test skill may be included as a covariate. This approach separates the effect we care about from possible background differences while reducing noise and simultaneously improving fairness and precision. We expect to report effect sizes and 95% confidence intervals and apply multiplicity control across outcomes.

We will use predefined thresholds to guide decisions and expect to see observable improvements in functional correctness, reductions in severity-weighted security findings, and increases in traceability/conformance scores. If at least three of four primary metrics meet thresholds without material regressions, we proceed to plan a standardized red-team study; otherwise, we refine boundary rules or training and repeat the controlled study.

I. Power, Ethics, and Timeline

We plan to conduct a small pilot to estimate variance and effect sizes and to finalize per-arm sample size (targeting 80–90% power) and to validate task timing, rubrics, and logging. If required, we will obtain institutional review board (IRB) approval, anonymize logs, and store data securely as controlling policies govern. Our proposed timeline is Pilot → Controlled Study → Synthesis → Red-Team Design (future) → Team Cognitive Study (future).

Next, we discuss Cathedral's application to the defense industry in Section 5. To demonstrate Cathedral in practice, Section 6 presents the highlights of a reference implementation that illustrates key principles and learnings.

V. Application to DoD Systems Engineering

Cathedral's L0–L4 structure fits how Department of Defense (DoD) software is planned, designed, built, and approved. The core ideas are simple: make boundaries explicit, keep each level focused, and carry identifiers through everything so work stays traceable and reviewable.

The domain architecture layer (L0) captures mission goals and capabilities and shapes the system boundaries. The tactical design layer (L1) defines the modules and contracts necessary to realize the system described in L0. The component design layer (L2) decomposes the modules into cohesive units of work, components, and their interfaces. The code structure layer (L3) shapes the specifics of the low-level code structure. The implementation layer (L4) provides concrete implementations and tests. Each level has entry and exit criteria and produces

artifacts that map naturally to design reviews and deliverables. Freezing L1 contracts before L3 scaffolding, and mapping L4 tests back to L0 goals, reduces scope creep and creates reviewable evidence.

Digital identifiers flow from L0 through L4 so you can follow a requirement into specifications, code, and tests. Specifications, the repository, and test systems exchange the same identifiers. This creates an end-to-end digital thread without flooding engineering teams with additional documentation burdens.

Evidence is generated as part of the workflow: L0/L1 record intent and constraints, L2 captures trust boundaries, L3 enforces structure, and L4 records test and analysis results. This supports faster, more consistent security and approval decisions, including Risk Management Framework (RMF) and Authorization to Operate (ATO) determinations, because each claim has a matching artifact.

Cathedral turns abstraction boundaries into working agreements. The result is clearer reviews, a usable digital thread, smoother approvals, and tighter control over dependency risk—without adding noise. We illustrate highlights of these mappings with a demonstration in Section 6.

VI. Reference Implementation

This section traces a single requirement end to end to illustrate how abstraction-aware context keeps each interaction "just right," avoids cross-level leakage, and yields predictable, auditable outcomes. We follow JS010, manifest validation, from domain intent (L0) to tested implementation (L4). At each level we include compact excerpts that a developer or reviewer can understand without consulting external artifacts. The example is extracted from a reference system which is a command line interface (CLI) based co-pilot like popular co-pilots such as Anthropic's Claude Code [35] or OpenAI's Codex [36].

A. L0 — Domain Scope and Bounded Context

We begin with a job story, JS010 'agent manifests', Figure 2., that anchors the demonstration thread:

```
- 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
```

Fig. 2. Example job stories extracted from the entry input into Cathedral.

The job stories are the first input into the thirteen step L0 process, shown in Figure 3. First up, we smooth and analyze the input job stories to identify a "walking skeleton"—the tiniest end-to-end slice that stitches the key architectural parts together [37] [38]. The walking skeleton allows Cathedral to reduce the scope, and thus complexity, for the first iteration without introducing unnecessary risk by missing key architectural elements. Then we systematically model the domain language, consider relevant non-functional requirements, and identify the bounded contexts from the requirements language. After executing the full L0 workflow, Cathedral identified an "agent configuration" bounded context whose purpose is to accept, validate, and ready configuration manifests for downstream execution. Adjacent contexts (e.g., execution, monitoring) are treated as stable neighbors with explicit integration points. The goal at L0 is minimal but sufficient precision: decide where the capability lives, enumerate its immediate neighbors, and avoid over-specifying lower-level details, and mitigate architectural risk.

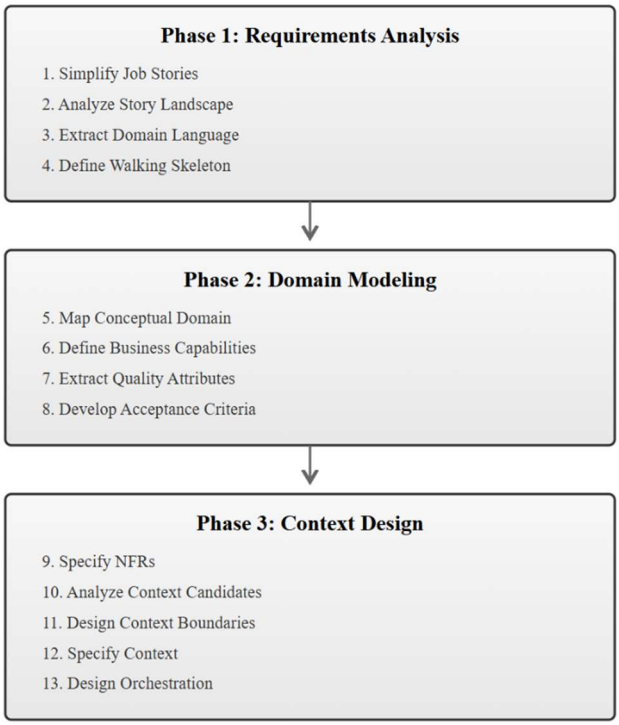

Fig. 3. Cathedral's L0 Workflow

Outcome: a domain context specification, Figure 4, covering the JS010 job story with explicit boundaries and a promise that validation success is a precondition for "ready."

```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:
```


Fig. 4. Cathedral's L0 Workflow

B. *L1 - Tactical Design*

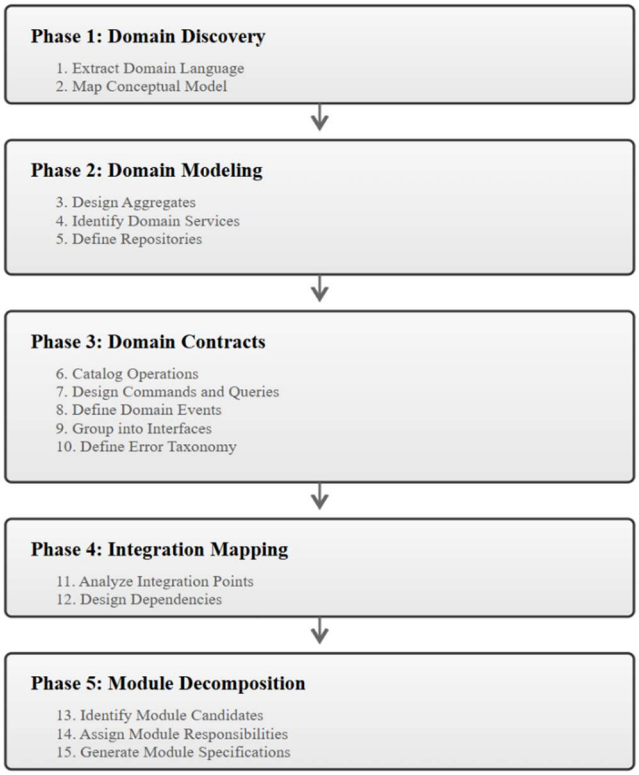

Fig. 5. Cathedral's L0 Workflow

At L1 we transform the domain boundaries and their requirements into module specifications, applying level appropriate design patterns and guidance. This fifteen step workflow, Figure 4, starts by extracting domain concepts from the behavioral scenarios described in the L0 output, applies tactical Domain Driven Design (DDD) patterns, designs the public interface, maps cross-domain integration, and then decomposes the domain into implementable modules. In Figure 5 we show an intermediate step where Cathedral agents are applying the CQRS pattern to segregate and encapsulate the reads and writes necessary to interact with elements in the domain.


```yaml
domain: "agent-configuration"
id: "CTX001"

commands:
  validate-configuration:
    description: "Validates agent manifest content against schema and business rules, updating configuration
    input:
      manifest-content:
        type: "manifest-content-vo"
        description: "Complete manifest specification with name, model, parameters, and raw content"
        required: true
      configuration-id:
        type: "string"
        description: "Unique identifier for the configuration being validated"
        required: false
        default: "generated-uuid"
    output:
      configuration-id:
        type: "string"
        description: "Identifier of the validated configuration"
      validation-result:
        type: "validation-result-vo"
        description: "Validation outcome with success status and error details"
      timestamp:
        type: "timestamp"
        description: "When validation completed"
    validation-rules:
      - "manifest-content.name must be non-empty string"
      - "manifest-content.model must be valid LLM model identifier"
      - "manifest-content.raw-content must be valid YAML or JSON"
      - "manifest-content.parameters must contain valid configuration keys"
    errors:
      - "ManifestContentInvalid: Invalid or malformed manifest content provided"
      - "SchemaValidationFailed: Manifest does not conform to required schema"
      - "RequiredFieldsMissing: Essential fields missing from manifest"
      - "BusinessRulesViolated: Manifest violates domain business rules"
      - "ParsingError: Unable to parse manifest content format"
```


Fig. 6. An intermediate work product of the L1 workflow demonstrating traceability and early encapsulation decisions

The CQRS analysis in Figure 6, demonstrates one step of the fifteen L1 steps that produce the module specifications necessary to realize the responsibilities of the inputted domain context.Number footnotes separately in superscripts. Place the actual footnote at the bottom of the column in which it was cited. Do not put footnotes in the abstract or reference list. Use letters for table footnotes.


```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"
```


Fig. 7. An excerpt of an L1 output, a module specification

The invariants tie readiness to successful validation while event sequences make state changes observable and testable. These contracts are the only promises other modules may rely on at this level to interoperate with the agent configuration core module. This tight encapsulation delivers two benefits. First, a

simple surface that a co-pilot can consume to understand and interact with the "agent configuration core" module. Second, a bounding box for co-pilots that isolates their work, wherein they may operate freely to fulfill the external contract without having to understand the rest of the system. In short, after our L1 is completed we've produced the first level of decomposition and have identified tighter encapsulations for our framework to operate in.

The L1 to L2 boundary is also a key boundary where the framework transitions from operating in a conceptual context (the 'what') to a logical one (the 'how'). This distinction allows us to focus our agent instruction while reducing off-task noise in the prompts. Onion (domain-centric) architecture organizes code into concentric layers with inward-only dependencies to protect business logic from framework and UI concerns [43] [44]. Cathedral adopts an Onion+ profile and applies it to guide agents from context specification to module design, leveraging four encapsulation boundaries—Domain, Application, Infrastructure, and Presentation—to drive decomposition and prevent cross-layer leakage.. An example of a candidate domain module is shown in Figure 3. As Figure 3 suggests, the JS010 job story can be traced from the CTX001 context through the intermediate work products, Figure 2, to the final specification shown in Figure 3, The MOD001 agent configuration core domain layer module. This traceability is preserved in the L2 workflow, described next, that designs and specifies the components necessary to realize the responsibilities of MOD001.

C. L2 - Module Design and Component Boundaries

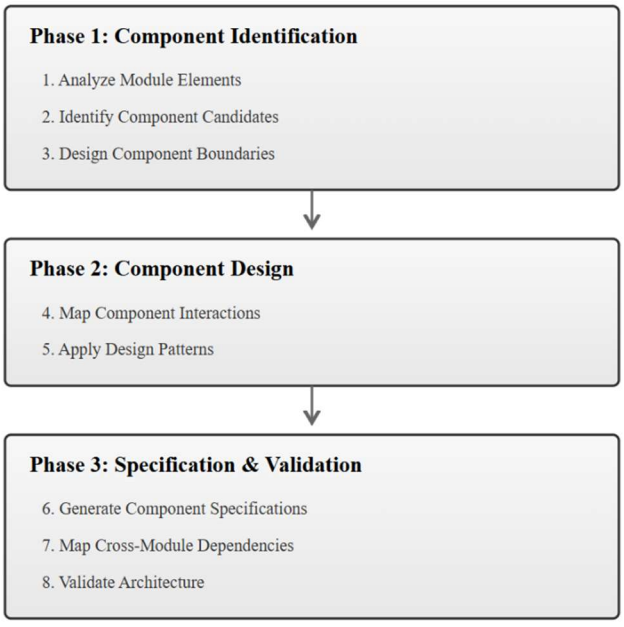

Fig. 8. Cathedral's L2 Workflow

L2 workflow, Figure 8, decomposes a module specified by L1 into cohesive components used by L3 where the implementation starts to take shape. Cathedral starts by analyzing the module elements to identify candidate components, then defines interactions and applies design patterns to preserve boundary discipline, from this it then generates specifications and attempts to validate the overall design. Figure 9 shows an intermediate step where a Cathedral agent has identified candidate components. Candidates are evaluated and component specifications are created, an example component specification relevant to JS010 can be seen in Figure 10.

```yaml
module: MOD001
type: core
layer: domain

component-candidates:
  - id: CND001
    name: agent-configuration-aggregate
    cohesion-type: functional
    primary-responsibility: "Manages complete agent configuration lifecycle including manifest validation
     state and readiness transitions"

    elements:
      - name: AgentConfiguration
        type: entity
        role: primary
      - name: ManifestContent
        type: value-object
        role: supporting
      - name: ValidationResult
        type: value-object
        role: supporting
      - name: AgentConfigurationRepository
        type: repository
        role: coordinating

    element-count: 4

    dependencies:
      internal: ["ManifestContent", "ValidationResult"]
      external: []
      external-count: 0
```

Fig. 9. An excerpt of an L1 output, a module specification

```yaml
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"
```

Fig. 10. An excerpt from the COMP005 specification of a repository component produced by Cathedral.

Still within the digital thread of JS010, Cathedral agents worked through the L2 process and produced an infrastructure layer component for the concrete implementation of an agent configuration repository. Even with strict scoping, the LLMs struggled with the multiple senses of the terms 'components' and 'modules'. Our framework organization allows us to mitigate the polysemy problem by making available an L2 specific glossary to all Cathedral agents operating at this level of abstraction. For L2, we reinforce one sense of the terms 'agent', 'context', 'module', and 'component' that is intended at this layer of abstraction. One of Cathedral's guiding principles is the "goldilocks' principle, where the prompt contains just enough context to complete the task. This informs us on our glossary usage too, as the glossaries are made available to the agents that operate at a particular layer of abstraction. Our L2 glossary, Figure 11, started out empty (analogous to a zero-shot prompt) but as problems of interpretation arose we augmented the glossary with problematic terms and concepts.

```yaml
glossary:
  - term: "agent"
    meaning: "an autonomous LLM-backed AI agent"
  - term: "context"
    meaning: "Autonomous domain partitions that encapsulate a
      consistent model, ubiquitous language, and related business capabilities."
  - term: "module"
    meaning: "Cohesive architectural units that encapsulate related
      domain logic and technical responsibilities within the context."
  - term: "component"
    meaning: "A component is a grouping of related functionality behind
      a well-defined interface, often mapping to a coherent set of code in one or more classes"
```

Fig. 11. An excerpt of an L1 output, a module specification

L2 ultimately defines the surface of a component and presents a first pass at organizing the code necessary to provide the features it is responsible for.

D. L3 Component Specification and Workflow Patterns

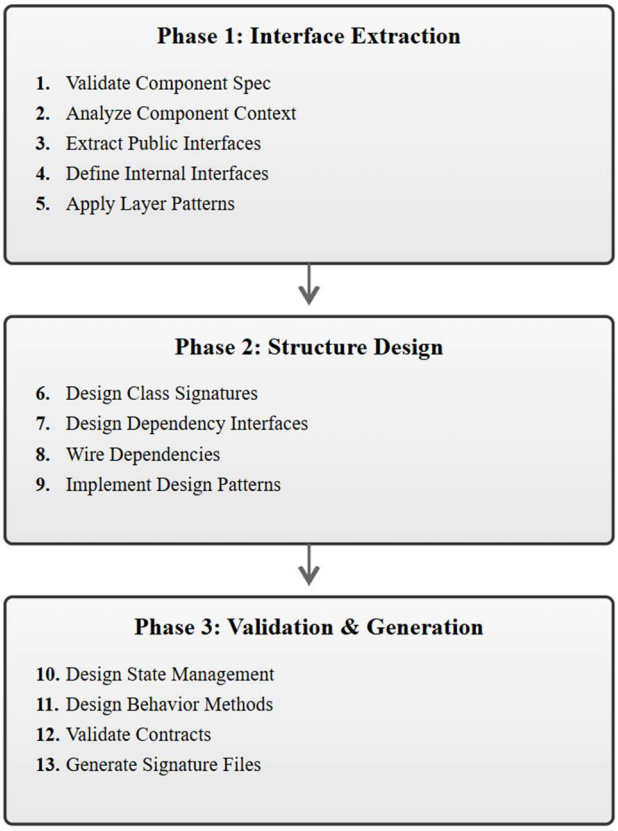

Fig. 12. An excerpt from the COMP005 specification of a repository component produced by Cathedral.

At L3, Figure 12, component specifications are transformed into interface definitions, class signatures and a map of dependencies. L3 is composed of thirteen steps that are layer aware and focus on one component at a time. The work moves from validating the output specification from L2 to extracting and designing component interfaces, to designing the classes using patterns and practices appropriate for the architectural layer, resolving and mapping dependencies, and finally validation of the contracts and creation of the signature files. The signature file is another Cathedral innovation where the public surface of a class is defined with a 'goldilocks' amount of content that an LLM could ingest and understand how to interact with the class. These files are co-located with the implementation and agents are instructed to prefer to ingest a *.signature.llm* file when needing to understand a class. Similar to the L2 glossary, this file also starts out in a "zero-shot" state with no additional context. A "zero-shot" signature file is presented in Figure 13. Figure 14 shows how the *AgentConfigurationRepositoryImpl.signature.llm* file could be enhanced to improve agent comprehension of the class.

```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;

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;
}
```

Fig. 13. The contents of the *AgentConfigurationRepositoryImpl.signature.llm* file.

```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;

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;
}
<examples>
    <example>
        ```php
        // Finding an agent configuration by ID
        $agentId = new AgentId('agent-123');
        $result = $repository->findById($agentId);

        if ($result->isSuccess()) {
        $configuration = $result->getValue();
        // $configuration is an AgentConfiguration entity
        echo "Found agent: " . $configuration->getName();
        } else {
        echo "Error: " . $result->getError()->getMessage();
        }
    </example>
</examples>
```

Fig. 14. The contents of the *AgentConfigurationRepositoryImpl.signature.llm* file.
```

E. L4 Implementation and Tests

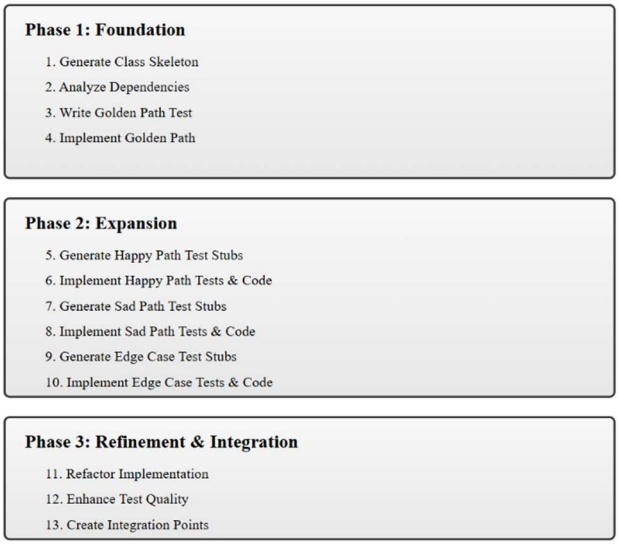

Fig. 15. Cathedral's L4 Workflow

With the class surfaces defined, Cathedral proceeds with a thirteen-step workflow beginning with generating a class skeleton, Figure 16, and a single 'golden path' unit test, Figure 17. The golden path test case is the core, primary use case that must work for the class to deliver its value. This skeletal implementation is an intentional starting point. The class skeletons are created with empty bodies that only throw a runtime error stating the method is not implemented. The golden path unit test serves as the grounding point for resolving any syntax errors and mocking necessary to run the test suite and verify a single failing test. It was observed that at this very minimal level of implementation, agents with tool-calling could reliably, and autonomously, work through any errors or missing dependencies.

```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");
    }
```

Fig. 16. A partial class skeleton generated from a class signature file

```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);

    $this->filesystem
        ->shouldReceive('get')
        ->once()
        ->with('agent-configurations/agent-123.json')
        ->andReturn(json_encode($configData));

    // Act
    $result = $this->repository->findById($this->agentId);

    // Assert
    expect($result)->toBeInstanceOf(Result::class);
    expect($result->isSuccess())->toBeTrue();
    expect($result->getValue())->toBeInstanceOf(AgentConfiguration::class);
```

Fig. 17. A snippet from a golden path test generated by an LLM from the class signature file

Next 'happy path' test cases are identified and stubbed out. 'Happy path' testing tests the normal, expected execution flow with valid inputs. It is not unreasonable to have dozens of assertions to fully verify the happy path scenarios, even for a small class. As a result, Cathedral's agents loop over these stubs, one by one, and implement the test, verify the test suite runs and fails as expected, then implements just enough code to make the test pass, and finally resolves any regressions introduced. It repeats this process through sad path tests and code, where sad paths are anticipated failure scenarios where invalid inputs and error conditions are handled gracefully. Once this is complete, edge cases are then worked through one by one. An edge case is a valid but unusual input or condition that occurs at the boundaries of expected behavior, where implementation assumptions are most likely to break. From here, the agents refactor the class design and review the test code. The output of this process is a well-tested, robust implementation of the inputted class signature file and specification.

By threading identifiers through the specifications, every class and test can be traced back through their component specification, that component's module specification to the module's domain context specification and back to the originating requirement.

VII. Conclusion and Future Work

LLM co-pilots accelerate delivery but frequently hide complexity rather than tame it. Context impedance, polysemous terms crossing abstraction layers, and flattened token streams blur critical boundaries while degrading quality, eroding traceability, and increasing verification burden precisely where

safety and assurance matter most. Cathedral's L0–L4 abstraction-aware context model is a systems engineering response: make boundaries explicit, scope language to the level where meanings are stable, and move work through progressive disclosure with artifacts that carry identifiers end-to-end. The result is not a new ritual but a return to engineering discipline: clearer interfaces at L1, tighter component definitions at L2, structure-first scaffolding at L3, and tested implementations at L4 that trace back to L0 intent.

This framework clarifies what belongs where. It aligns prompts, model context, and task to the appropriate sense of "contract," "interface," "policy," and other overloaded software engineering terms that create ambiguity and misunderstanding in the model. It increases signal-to-noise, reduces near-miss evidence, and curbs cross-level leakage that leads to hallucinations, unsafe tool calls, and brittle integrations. It also reestablishes a usable digital thread by threading identifiers from mission goals to code and tests, enabling evidence that maps naturally to DoD review gates and RMF/ATO expectations. Equally important is what Cathedral does not claim: it is not a prescriptive tool chain or a guarantee of correctness. It does not replace human judgment, domain expertise, or security engineering. It is, instead, a blueprint that surfaces complexity. Cathedral's LLM-centered techniques should carry over to domains where today's models underperform beyond software engineering—law [39], science/medical summarization [40] [41] [42], and long-context enterprise work characterized by messy, evolving information and low tolerance for error [4].

Practically, Section 4 outlines how we intend to validate these claims with correctness, security, and traceability metrics, and Section 5 shows how the abstractions map to DoD workflows. Early signs from structured prompting literature and developer studies are encouraging; our objective is to turn that signal into repeatable practice, measured improvements, and evidence that survives audit. The promise of co-pilots in high-assurance work is real, but only if we pair them with abstraction discipline that keeps systems legible and verifiable.

Our current scope is limited to representative tasks and defense-flavored examples; generalization to broader domains remains an open question. Metric proxies for correctness, security, and traceability can miss qualitative aspects of design quality and team cognition. Human factors, including prompting skill, cognitive load, and prior stack familiarity, confound outcomes even with randomization and training. Tool bias is plausible: relying on specific linters, validators, or agent frameworks may advantage our approach or inadvertently handicap flat-prompt baselines. Finally, retrieval noise and source-spec quality impose independent ceilings on performance; poor inputs can degrade outcomes regardless of abstraction discipline. We address these risks with preregistered thresholds, information-parity baselines, fixed temperature, replayable runs, and transparent evidence releases, but residual threats remain.

While our framework provides a foundation for multi-level abstraction with enforced semantic boundaries, several critical questions remain for future research: What granularity best separates the abstraction levels in practice across domains, and how should level-scoped semantics be specified to minimize ambiguity? How can we quantify and systematically reduce cross-level leakage under adversarial inputs, including CWE-mapped failure modes? Which minimal L1/L2 contracts measurably reduce rework and security findings at L4, and how do those benefits compare to any added process friction? What retrieval and disambiguation strategies most effectively enforce sense discipline across levels without starving the model? Which review and human-in-the-loop patterns remove the most error per minute of added effort, and how can those be automated or guided by agent assistants? How do we incorporate boundary-crossing iterative workflows into the framework?

Cathedral is a practical path to making complexity visible and verifiable again in LLM-assisted development. For the AI4SE audience, it offers a way to couple co-pilot acceleration with the assurance demands of systems engineering, producing artifacts, tests, and attestations that carry through pipelines and approval processes. We invite the community to contribute tasks, adversarial probes, and replication studies, especially across domains and stacks, and to pressure-test the boundary rules in real workflows. Our next paper will report empirical findings from the validation plan, refine level semantics and guardrails, and publish templates, linters, and evidence exporters aligned with DoD DevSecOps. The end state is straightforward: faster delivery, fewer regressions, and evidence you can trust.

REFERENCES

[1] Parnas, D. L. (1972). On the Criteria To Be Used in Decomposing Systems into Modules. Communications of the ACM, 15(12), 1053–1058. https://doi.org/10.1145/361598.361623

[2] Communications of the ACM (2025). Catching the Vibe of Vibe Coding. News article. https://cacm.acm.org/news/catching-the-vibe-of-vibe-coding/

[3] Welsh, M. (2023). The End of Programming. Communications of the ACM (Viewpoint). https://cacm.acm.org/

[4] Liu, H., et al. (2023). Lost in the Middle: How Language Models Use Long Context. arXiv:2307.03172. https://arxiv.org/abs/2307.03172

[5] Pinecone Engineering (2023). Why Use Retrieval Instead of Larger Context? Engineering blog. https://www.pinecone.io/blog/why-use-retrieval-instead-of-larger-context/

[6] Subramonyam, H., et al. (2024). Designing Interactions with Generative AI: The Gulf of Envisioning. Proceedings of CHI 2024. arXiv:2309.14459. https://arxiv.org/abs/2309.14459

[7] GitHub Research (2023). Quantifying GitHub Copilot's Impact on Developer Productivity and Happiness. GitHub Blog. https://github.blog/news-insights/research/research-quantifying-github-copilots-impact-on-developer-productivity-and-happiness

[8] Lo, R., et al. (2023). Hierarchical Prompting for Long-Context Tasks. Findings of EMNLP 2023. ACL Anthology: https://aclanthology.org/2023.findings-emnlp.685

[9] Wu, T., et al. (2022). PromptChainer: Chaining Prompts for Complex Tasks. CHI EA 2022. https://dl.acm.org/doi/10.1145/3491101.3519729

[10] Microsoft (2023). AutoGen: Enabling Next-Gen Large Language Model Applications. GitHub repository. https://github.com/microsoft/autogen

[11] Zhang, H, et al. (2025). Flow State: Humans Enabling AI Systems to Program Themselves. arXiv:2504.03771. https://arxiv.org/abs/2504.03771

[12] Raymond, E. S. (1997). *The Cathedral and the Bazaar: Musings on Linux and Open Source*. Presented at Linux Kongress, Würzburg, Germany. Retrieved from https://www.catb.org/~esr/writings/cathedral-bazaar/cathedral-bazaar/

[13] Zhang, Q., et al. 2023. A Survey on Large Language Models for Software Engineering. arXiv:2312.15223. https://arxiv.org/abs/2312.15223

[14] Lewis, P., et al. 2020. Retrieval-Augmented Generation for Knowledge-Intensive NLP. In Advances in Neural Information Processing Systems (NeurIPS). https://arxiv.org/abs/2005.11401

[15] Karpukhin, V., et al. 2020. Dense Passage Retrieval for Open-Domain Question Answering. In Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP). arXiv:2004.04906. https://arxiv.org/abs/2004.04906

[16] Snell, C. V., Klein, D., and Zhong, R. 2023. Learning by Distilling Context. In Proceedings of the Eleventh International Conference on Learning Representations (ICLR '23). OpenReview.net. https://openreview.net/forum?id=am22IukDiKf

[17] Wei, J., et al. 2022. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. In *Advances in Neural Information Processing Systems (NeurIPS '22)*, Vol. 35, 24824-24837. arXiv:2201.11903. https://arxiv.org/abs/2201.11903

[18] Wang, X., et al.. 2022. Self-Consistency Improves Chain of Thought Reasoning in Language Models. arXiv:2203.11171. https://arxiv.org/abs/2203.11171

[19] Yao, S., et al. 2023. Tree of Thoughts: Deliberate Problem Solving with Large Language Models. arXiv:2305.10601. https://arxiv.org/abs/2305.10601

[20] Zhou, D., et al. 2022. Least-to-Most Prompting Enables Complex Reasoning in Large Language Models. arXiv:2205.10625. https://arxiv.org/abs/2205.10625

[21] Yao, S., et al. 2022. ReAct: Synergizing Reasoning and Acting in Language Models. arXiv:2210.03629. https://arxiv.org/abs/2210.03629

[22] Schick, T., et al. 2023. Toolformer: Language Models Can Teach Themselves to Use Tools. arXiv:2302.04761. https://arxiv.org/abs/2302.04761

[23] Gao, L., et al. 2023. PAL: Program-Aided Language Models. arXiv:2211.10435. https://arxiv.org/abs/2211.10435

[24] Khattab, O., et al. 2023. DSPy: Compiling Declarative Language Model Pipelines. arXiv:2310.03714. https://arxiv.org/abs/2310.03714

[25] Scholak, T., Schucher, N., & Bahdanau, D. 2021. PICARD: Parsing Incrementally for Constrained Auto-Regressive Decoding for Text-to-SQL. In NeurIPS Workshop on Robustness in Sequence Modeling. arXiv:2109.05093. https://arxiv.org/abs/2109.05093

[26] OpenAI. 2023. Function calling and other API updates. Developer blog. https://openai.com/index/function-calling-and-other-api-updates/

[27] GitHub. 2025. Spec-driven development with AI: Get started with a new open source toolkit. GitHub Blog. https://github.blog/ai-and-ml/generative-ai/spec-driven-development-with-ai-get-started-with-a-new-open-source-toolkit/

[28] U.S. Department of Defense, Office of the Deputy Assistant Secretary of Defense for Systems Engineering. 2018. Digital Engineering Strategy. Washington, DC. https://ac.cto.mil/wp-content/uploads/2019/06/USA001603-18-DSD.pdf

[29] Ruiz, M., Hu, J. Y., and Dalpiaz, F. 2023. Why don't we trace? A study on the barriers to software traceability in practice. Requirements Engineering 28, 619–637. DOI:10.1007/s00766-023-00408-9.

[30] Evans, E. 2003. Domain-Driven Design: Tackling Complexity in the Heart of Software. Addison-Wesley.

[31] ISO/IEC/IEEE. 2018. 29148:2018 Systems and software engineering — Life cycle processes — Requirements engineering. International Standard.

[32] INCOSE Requirements Working Group. 2019. Guide for Writing Requirements. International Council on Systems Engineering (INCOSE). INCOSE-TP-2010-006-03. https://www.incose.org/products-and-publications/se-products/incose-guide-to-writing-requirements

[33] Navigli, R. 2009. Word Sense Disambiguation: A Survey. ACM Computing Surveys 41, 2 (Article 10), 10:1–10:69. https://doi.org/10.1145/1459352.1459355

[34] Asai, A., et al. 2023. Self-RAG: Learning to Retrieve, Generate, and Critique for Language Models. arXiv:2310.12317. https://arxiv.org/abs/2310.12317

[35] Anthropic. n.d.. Claude Code overview. Anthropic Docs. https://docs.anthropic.com/en/docs/claude-code/overview

[36] OpenAI. 2025. Introducing Codex. OpenAI Blog/Announcement. https://openai.com/index/introducing-codex/

[37] Cockburn, A. 2004. Crystal Clear: A Human-Powered Methodology for Small Teams. Addison-Wesley Professional. https://dl.acm.org/doi/10.5555/1406822

[38] Freeman, S., and Pryce, N. 2009. Growing Object-Oriented Software, Guided by Tests. Addison-Wesley Professional. https://dl.acm.org/doi/10.5555/1655852

[39] United States District Court, S.D.N.Y. 2023. Mata v. Avianca, Inc., No. 22-cv-1461 (PKC). Opinion and Order on Sanctions (June 22, 2023). https://law.justia.com/cases/federal/district-courts/new-york/nysdce/1:2022cv01461/575368/54/

[40] Peters, U. and Chin-Yee, B. 2025. Generalization bias in large language model summarization of scientific research. Royal Society Open Science 12, 4 (Apr. 2025), 241776. https://doi.org/10.1098/rsos.241776

[41] Goh, E., Gallo, R., Hom, J., et al. 2024. Large Language Model Influence on Diagnostic Reasoning: A Randomized Clinical Trial. JAMA Network Open 7, 10 (Oct. 2024), e2440969. https://doi.org/10.1001/jamanetworkopen.2024.40969

[42] Feldman, M. J., Hoffer, E. P., Conley, J. J., et al. 2025. Dedicated AI Expert System vs Generative AI With Large Language Model for Clinical Diagnoses. JAMA Network Open 8, 5 (May 2025), e2512994. https://doi.org/10.1001/jamanetworkopen.2025.12994

[43] Palermo, J. 2008. The Onion Architecture: Part 1. Programming with Palermo (blog). July 29, 2008. https://jeffreypalermo.com/2008/07/the-onion-architecture-part-1/

[44] Palermo, J. 2008. The Onion Architecture: Part 2. Programming with Palermo (blog). July 30, 2008. https://jeffreypalermo.com/2008/07/the-onion-architecture-part-2/