

Concept of Operations: Orchestration Subsystem

1. System Overview

Intent

The Orchestration subsystem exists because raw architecture models are hollow shells — they declare components exist but say nothing about what those components actually contain, how they relate at a code level, or how they decompose into manageable units. Without Orchestration, an architecture model is a manually-maintained lie that drifts from reality with every commit.

Orchestration solves two fundamental problems:

1. **Enrichment:** Automatically populating architecture models with ground-truth data extracted from source code — function signatures, constants, test contracts, behaviors, capabilities, and patterns. This transforms a model from a declaration of intent into a verified reflection of implementation.
2. **Decomposition:** Breaking coarse-grained models into hierarchical sub-models that match actual code structure, using import-graph clustering rather than human intuition. This makes large systems navigable without requiring architects to manually trace every dependency.

The subsystem is the bridge between static analysis (manifests, AST scanning) and the semantic architecture model. It consumes raw code intelligence and produces enriched, structured models.

Philosophy

The design prioritizes **automated accuracy over manual curation**. Every field that can be derived from code should be — human input is reserved for intent and rationale that code cannot express. The subsystem is intentionally pipeline-shaped: each stage transforms the model incrementally, and stages compose without tight coupling.

2. Stakeholders & Actors

Actor	Type	Goal
CLI (COMP-8)	Internal component	Trigger enrichment/decomposition workflows on user command. Needs deterministic, idempotent operations.
Architecture Model (COMP-1.1)	Internal dependency	Provides core types (<code>ArchitectureModel</code> , <code>Component</code> , <code>Behavior</code> , etc.) that Orchestration populates.
Manifest subsystem (COMP-3)	Internal dependency	Supplies AST-extracted module data (<code>Manifest</code> , <code>ModuleInfo</code> , <code>CallGraph</code>) that Orchestration consumes as ground truth.

Actor	Type	Goal
Quality Metrics (COMP-1.5)	Internal dependency	Provides quality scoring used by Decomposition to validate output coherence.
Human architect	External user	Wants a model that reflects reality without manually inspecting every source file. Needs decomposition that matches their mental model of system structure.
LLM agents	External consumer	Consume formatted prompts (<code>format_enrichment_prompt</code> , <code>format_naming_context</code>) to provide semantic annotations that code analysis alone cannot infer.

3. Operational Scenarios

Scenario 1: First-Time Model Enrichment from Source

Intent: A team has a skeleton `.architecture-model.yaml` with component IDs and file lists but no signatures, constants, or test contracts. They want the model to reflect what the code actually contains.

1. CLI invokes `enrich_model(model, project_root)` via COMP-8.
2. `enrich_model` iterates ACTIVE components, skipping those without files.
3. For each component, `_enrich_signatures` calls `extract_file_hints` (from manifest) to parse AST and extract public function signatures, deduplicating against existing entries.
4. `_enrich_constants` parses module-level assignments via `ast.parse`.
5. `_enrich_test_contracts` discovers and analyzes matching test files.
6. The enriched `ArchitectureModel` is returned with populated `signatures`, `constants`, and `test_contracts` on each component.

What would break without it: Models would contain only hand-written descriptions. Signature drift would be invisible. Test coverage gaps would go undetected.

Scenario 2: Manifest-Driven Deep Enrichment

Intent: After manifest generation, leverage richer data (class hierarchies, decorators, call graphs) to populate symbols, behaviors, patterns, and responsibilities — going beyond what simple AST scanning provides.

1. CLI triggers `enrich_from_manifest(model, manifest)`.
2. For each component, files are matched to `ModuleInfo` entries in the manifest.
3. Functions become `FunctionSignature` objects via `_parse_signature`; classes become `Symbol` objects via `_class_to_symbol` with kind detection (dataclass, protocol, enum, etc.).
4. `enrich_behaviors_from_manifest` detects trigger decorators (routes, event handlers) via `_TRIGGER_DECORATORS` regex and creates `Behavior` entities.

5. `detect_behavior_triggers` traces the call graph to find inter-behavior trigger relationships.
6. `infer_capabilities` groups behaviors by URL prefix into **Capability** entities.
7. `infer_composite_behaviors` finds trigger chains 2 long and creates UC-* composite behaviors.

Scenario 3: Hierarchical Model Decomposition

Intent: A monolithic model with 50+ components is too large to reason about. The architect wants per-subsystem sub-models that preserve relationship fidelity.

1. CLI invokes `run_pipeline(project_root, deep=True)`.
2. `generate_recursive_manifests` scans each functional block's directories.
3. `decompose_model` traces relationships outward from each block's components — **realizes** → Capabilities, **exposes** → Interfaces, **traces-to** → Behaviors — producing faithful model slices.
4. `deep_decompose_block` clusters modules by import-graph affinity via `cluster_modules`, producing **SubComponent** objects with file/class/function inventories.
5. `iterative_decompose` recurses: clusters above `max_modules` are re-decomposed.
6. `write_sub_models` persists each sub-model to `.architecture-models/<block_id>/`.
7. If `compact=True`, `compact_for_storage` offloads leaf behaviors into summary groups, keeping use cases and structurally-referenced behaviors inline.

Scenario 4: Agent-Assisted Pattern Classification

Intent: Code analysis can identify structure but not architectural intent. An LLM agent needs compact, structured context to classify each component's pattern and write contracts.

1. After decomposition produces **DecomposeResult** objects, `format_enrichment_prompt` is called.
2. It loads the pattern catalog, formats each leaf component's files/classes/functions, and emits structured instructions requesting YAML annotations.
3. `format_naming_context` separately formats cluster data for semantic naming — showing file stems, top classes, and inter-cluster import counts.
4. Agent responses are parsed back into model annotations.

Trade-off: Formatting is optimized for token efficiency (stems, truncation at 5 files / 6 classes / 4 functions) because LLM context windows are finite and costly. This trades completeness for practical usability.

4. System Context

Why these dependencies: - **Manifest (COMP-3):** Enrichment cannot infer signatures or behaviors without AST-extracted data. The manifest is the single source of code truth. - **Core Types (COMP-1.1):** Orchestration's entire purpose is populating these types. Without them, there's nothing to enrich. - **Quality Metrics (COMP-1.5):** Decomposition needs feedback on whether its clustering produces coherent sub-models. Without quality checks, decomposition could produce arbitrary groupings.

5. Operational Constraints

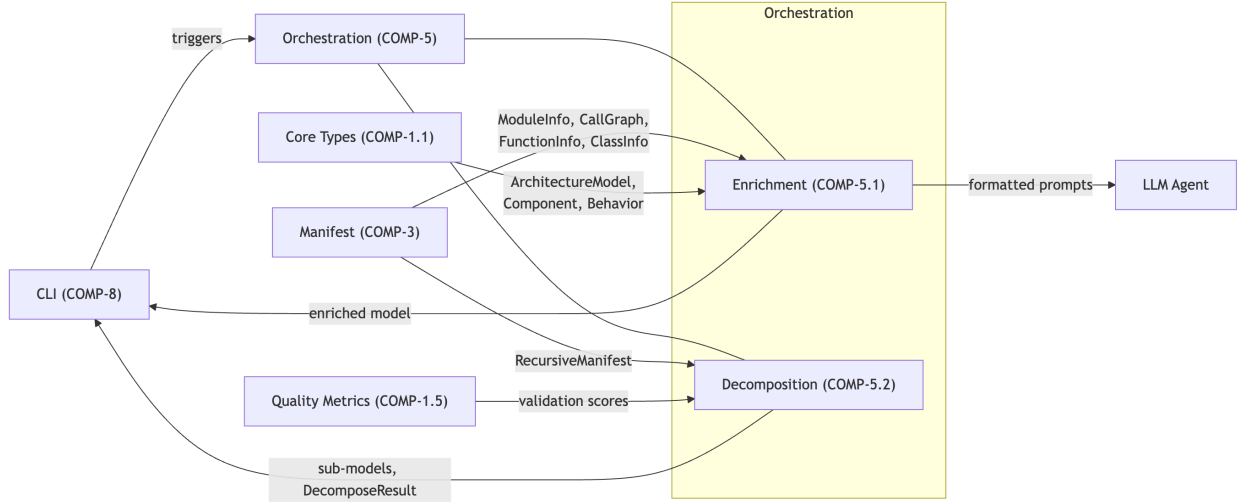


Figure 1: Diagram

Constraint	Threshold	Rationale	Failure Mode
Behavior cap per component (REQ-18)	40 behaviors	Without a cap, REST-heavy components generate hundreds of behaviors from route decorators, creating “orphan explosion” — behaviors with no meaningful relationships that bloat the model and overwhelm consumers. 40 was chosen as the point where CRUD grouping and summarization remain tractable.	Graceful: excess behaviors are filtered/summarized, not lost. <code>compact_for_storage</code> offloads them to per-component groups.
Decomposition minimum cluster size	<code>min_cluster_size=3</code>	Clusters smaller than 3 modules lack sufficient internal cohesion to justify separate sub-component status. Single-file clusters are noise.	Graceful: small clusters merge into nearest neighbor.

Constraint	Threshold	Rationale	Failure Mode
Decomposition module threshold	<code>max_modules=15</code>	Blocks with fewer than 15 modules are already comprehensible; decomposing them produces trivial 1-2 file clusters.	Graceful: returns empty <code>sub_components</code> , block stays monolithic.
Call graph trace depth	<code>max_depth=4</code>	Deeper traces produce false trigger relationships through utility functions. 4 hops captures direct behavioral chains without transitivity pollution.	Hard boundary: relationships beyond depth 4 are invisible.
Idempotency	Enrichment deduplicates by name	Re-running enrichment must not create duplicate signatures/constants. <code>existing_names</code> sets in <code>_enrich_signatures</code> and <code>_enrich_constants</code> enforce this.	Silent: duplicates are dropped.

6. Data Flow

7. Measures of Effectiveness

MoE	What “good” looks like	What “bad” looks like	How to measure
Enrichment coverage	>90% of ACTIVE components have 1 signature and 1 constant populated	Components remain empty despite having source files	<code>component_count</code> metric from <code>@monitored</code> on <code>enrich_model</code> ; ratio of components with non-empty <code>signatures</code>
Signature accuracy	Every enriched signature matches the actual function in source	Stale signatures from deleted functions persist	Delta between enriched signatures and fresh AST parse

MoE	What “good” looks like	What “bad” looks like	How to measure
Decomposition coherence	Sub-components correspond to recognizable code domains; internal import edges » cross-cluster edges	Random-seeming groupings that split tightly-coupled modules	<code>avg_cluster_size</code> from <code>deep_decompose_block</code> monitoring; ratio of internal to external import edges per cluster
Behavior signal-to-noise	Cross-component behaviors surface meaningful flows; CRUD groups are summarized, not enumerated	200 behaviors per component, most trivial single-step CRUD	Behavior count per component (target: 40 per REQ-18); ratio of <code>cross_component</code> to <code>trivial</code> in <code>BehaviorClassification</code>
Prompt token efficiency	Agent prompts fit within context window with room for response; token estimate tracked via <code>@monitored</code>	Prompts exceed context limits, truncating critical information	<code>token_estimate</code> and <code>char_count</code> from <code>format_enrichment_prompt</code> monitoring
Pipeline idempotency	Running the pipeline twice produces identical output	Second run creates duplicates or changes ordering	Diff of output artifacts across consecutive runs
Compaction ratio	Compacted models retain all use cases and structurally-referenced behaviors while reducing total behavior count by >50%	Compaction drops behaviors that other entities reference via <code>triggers</code> or <code>traces-to</code>	<code>len(kept_behaviors) / len(original_behaviors)</code> ; zero dangling relationship references post-compaction

Value Functions

- **Enrichment coverage** has diminishing returns above 95% — the last 5% are typically `__init__.py` files and configuration modules with no meaningful signatures. Effort to reach 100% is not justified.
- **Decomposition coherence** has increasing value as cluster count approaches `target_k` (default 5). Too few clusters (1-2) means no decomposition occurred; too many (>10) means over-fragmentation where every file is its own component.
- **Behavior cap (40)** is a knee in the value curve: below 40, each additional behavior adds information; above 40, each additional behavior adds noise faster than signal, degrading model readability and agent prompt quality.

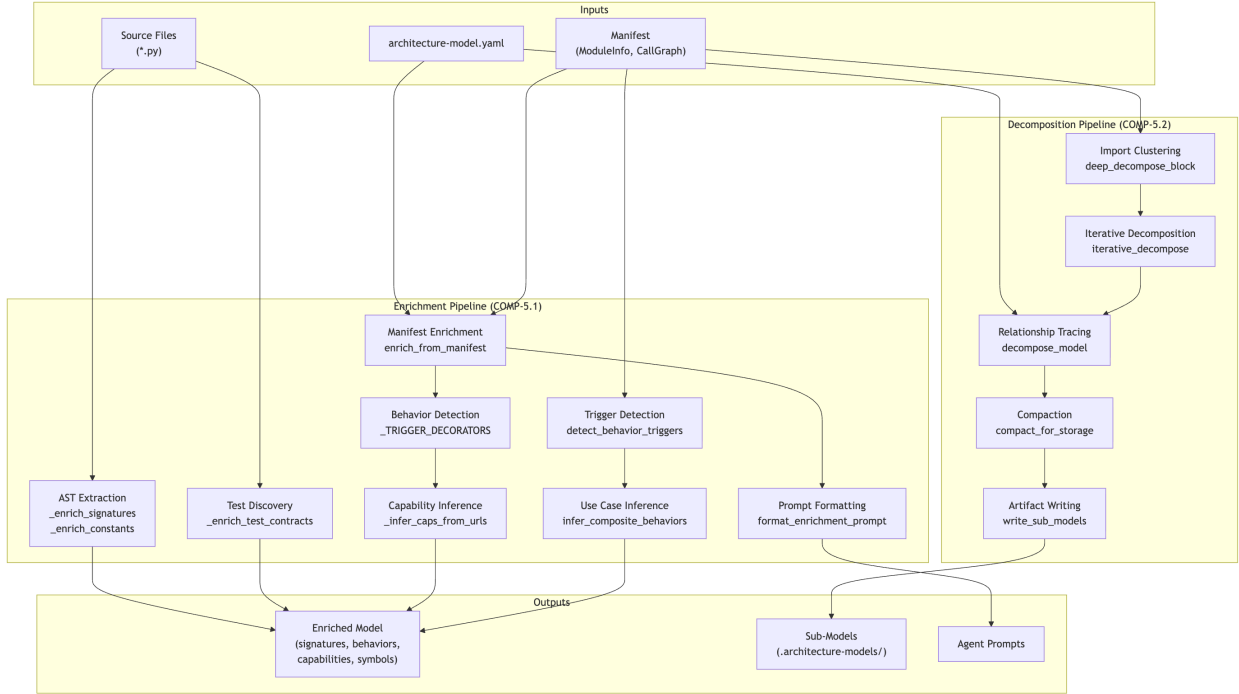


Figure 2: Diagram

Functional Analysis: Orchestration Subsystem

1. Intent & Purpose

The Orchestration subsystem exists because **an architecture model without grounding in actual source code is fiction**. Raw YAML models capture intent but drift from reality. Orchestration bridges this gap by automatically enriching models with code-derived facts (signatures, constants, test contracts, behaviors) and decomposing coarse models into navigable hierarchies.

Without this subsystem, architecture models would require manual maintenance — a process that fails at scale and guarantees staleness.

2. Capability Inventory

CAP-10: Enrich Models with Code Intelligence

Intent: Eliminate the gap between what the model *claims* and what the code *does*. A component listing files but lacking signatures or test contracts is an unverifiable assertion.

Goal (optimal): Every active component has complete signatures, constants, test contracts, detected patterns, inferred capabilities, and behavior trigger chains — all derived automatically from source, not manually authored.

Sub-capabilities:

Sub-capability	Realized By	Intent
AST-based signature extraction	<code>enrich.py::_enrich_signature</code>	Populate function signatures from actual source AST so the model reflects real APIs
Constant extraction	<code>enrich.py::_enrich_constants</code>	Capture module-level constants that define system configuration boundaries
Test contract discovery	<code>enrich.py::_enrich_test_contracts</code>	Link components to their test files, making testability visible in the model
Manifest-driven enrichment	<code>auto_enrich.py::enrich_from_manifest</code>	Unify canned manifest data (richer than raw AST) for bulk enrichment of signatures, symbols, patterns
Behavior enrichment	<code>auto_enrich.py::enrich_behavior</code>	Detect behaviors (HTTP routes, event handlers) from decorator/trigger patterns
Capability inference	<code>capability_inference.py</code>	Group behaviors by URL prefix/actor into user-facing capabilities automatically
Trigger detection	<code>trigger_detection.py</code>	Discover behavior-to-behavior trigger chains via call graph traversal
Use case inference	<code>use_case_inference.py</code>	Compose trigger chains into end-to-end use cases (composite behaviors)
Pattern classification context	<code>enrichment_context.py::format_enrichment_prompt</code>	Generate prompts for agent-assisted pattern/contract annotation

CAP-9: Decompose Models Hierarchically

Intent: A flat list of 50+ components is unusable. Hierarchical decomposition makes architecture navigable — each level reveals appropriate detail for its audience.

Goal (optimal): Every coarse block decomposes into cohesive sub-components clustered by import affinity, with internal relationships preserved, producing self-contained sub-models that can be reasoned about independently.

Sub-capabilities:

Sub-capability	Realized By	Intent
Relationship-traced sub-models	<code>decompose.py::decompose_model</code>	Slice parent model into per-block sub-models by tracing relationship graphs outward from block components

Sub-capability	Realized By	Intent
Import-graph clustering	<code>deep_decompose.py::deep_decompose_cluster_blocks</code>	Cluster blocks into sub-components using import affinity, producing meaningful groupings
Iterative deep decomposition	<code>deep_decompose.py::iterative_deep_decompose</code>	Recursively decompose blocks that are still too large
Behavior flow classification	<code>behavior_flows.py::classify_behaviors</code>	Separate cross-component flows (architecturally significant) from single-component CRUD (noise)
Behavior step structuring	<code>behavior_decompose.py::decompose_behavior_step</code>	Parse behavior step names into structured <code>Step</code> objects with component references
Model compaction	<code>compaction.py::compact_for_storage</code>	Collapse leaf behaviors to per-component summaries, keeping models readable
Unified pipeline	<code>pipeline.py::run_pipeline</code>	Single entry point chaining manifest generation → decomposition → artifact writing

3. Functional Decomposition

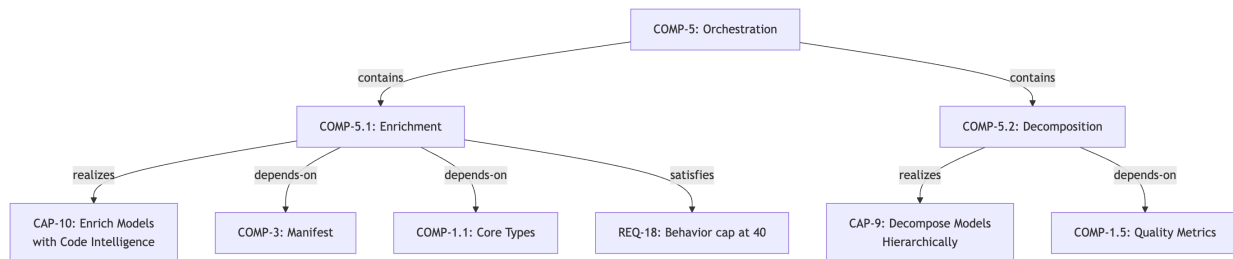


Figure 3: Diagram

Capability-Component Mapping Rationale

COMP-5.1 → CAP-10: Enrichment is separated from decomposition because enrichment is *additive* (populates fields on existing components) while decomposition is *structural* (creates new components/relationships). They have different failure modes: enrichment failure leaves gaps; decomposition failure produces wrong architecture. Separating them allows enrichment to run independently and incrementally.

COMP-5.2 → CAP-9: Decomposition requires graph algorithms (clustering, chain detection, flow tracing) that are fundamentally different from AST parsing. The dependency on quality metrics (COMP-1.5) is unique to decomposition — enrichment doesn't need to evaluate decomposition quality.

4. Behavioral Flows

Flow 1: Full Pipeline Execution (Primary Use Case)

Intent: Transform a bare YAML model + source code into a fully enriched, hierarchically decomposed architecture with per-block sub-models written to disk. This is the “make it all work” entry point invoked by CLI.

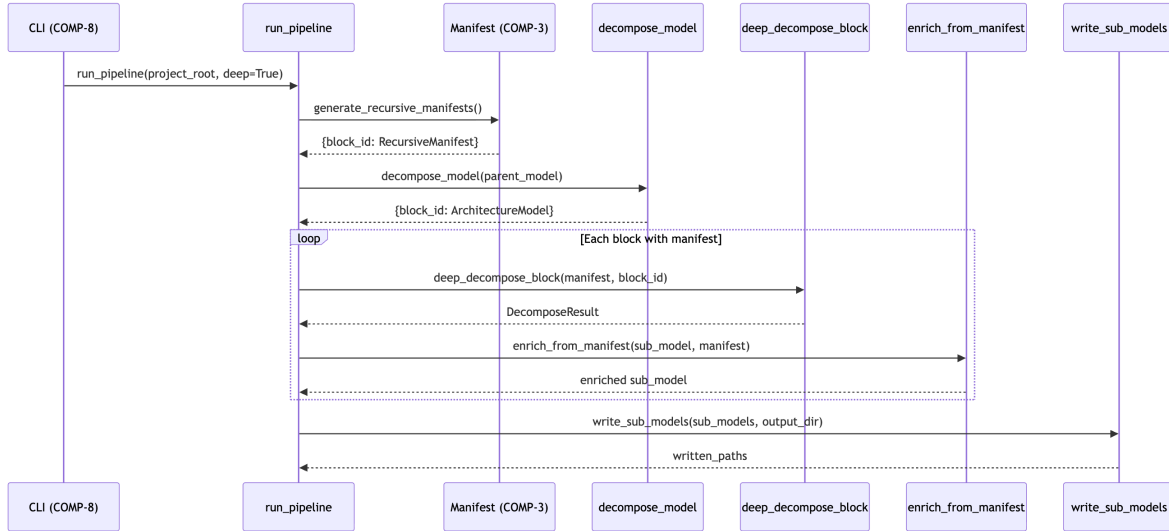


Figure 4: Diagram

Flow 2: Behavior Discovery and Composition

Intent: Automatically discover what the system *does* (behaviors) from code patterns, then compose individual behaviors into end-to-end use cases. This transforms raw code into architecturally meaningful sequences without human annotation.

Flow 3: Model Compaction

Intent: Prevent model bloat. A component with 30 CRUD behaviors adds noise without insight. Compaction preserves architecturally significant behaviors (cross-component, trigger-linked) while summarizing the rest, keeping models human-readable.

5. Requirements Satisfaction

REQ-18: Behavior Filtering Cap (Maximum 40 per component)

Requirement: “Maximum 40 behaviors per component to prevent orphan explosion”

Rationale: Without a cap, a single REST controller with 80 endpoints generates 80 behaviors, each spawning capability-inference entries, trigger-detection traces, and relationship edges. This causes:

- **Combinatorial explosion** in trigger detection ($O(n^2)$ pairwise checks via call graph)
- **Model illegibility** — 80 behaviors drowns the signal of cross-component flows
- **Orphan proliferation** — behaviors that connect to nothing useful but inflate entity counts

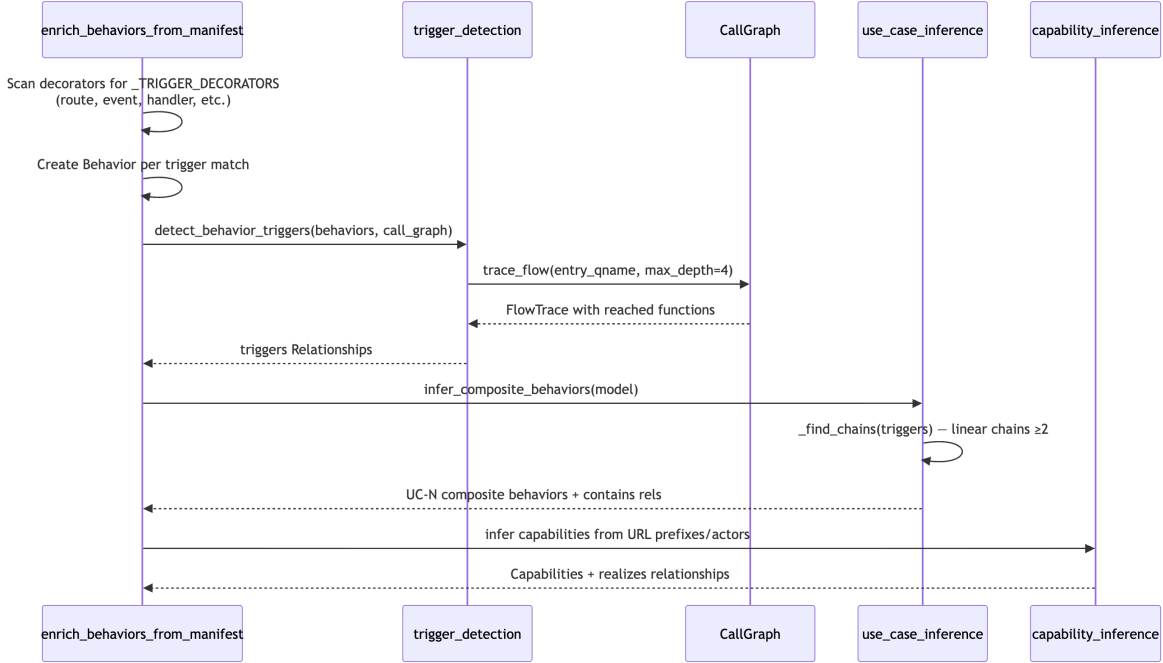


Figure 5: Diagram

Why 40? Empirically, components with >40 behaviors are almost always CRUD-heavy REST controllers where individual endpoint behaviors add no architectural insight. The value function is roughly: insight peaks around 15-20 behaviors, plateaus to 30, and actively degrades past 40 due to noise.

Satisfaction: COMP-5.1 (Enrichment) applies this cap during `enrich_behaviors_from_manifest`. Behaviors beyond the cap are either summarized via compaction or filtered during classification in `behavior_flows.py::classify_behaviors`.

Consequence of violation: Trigger detection becomes slow, capability inference produces dozens of near-identical capabilities (“User Management”, “User Operations”, “User Queries”), and the model becomes unusable without manual curation — defeating the purpose of automation.

6. Trade-offs & Design Decisions

Decision: Separate Enrichment from Decomposition

Considered: Unified pass that decomposes and enriches simultaneously. **Chosen:** Separate components with independent entry points. **Why:** Enrichment is idempotent and incremental (re-running adds missing data). Decomposition is structural and destructive (re-running may change component boundaries). Users need to enrich without decomposing (e.g., updating signatures after code changes) and decompose without re-enriching. **If constraints changed:** If enrichment depended on decomposition results (e.g., enrichment needed sub-component boundaries), merging would be justified.

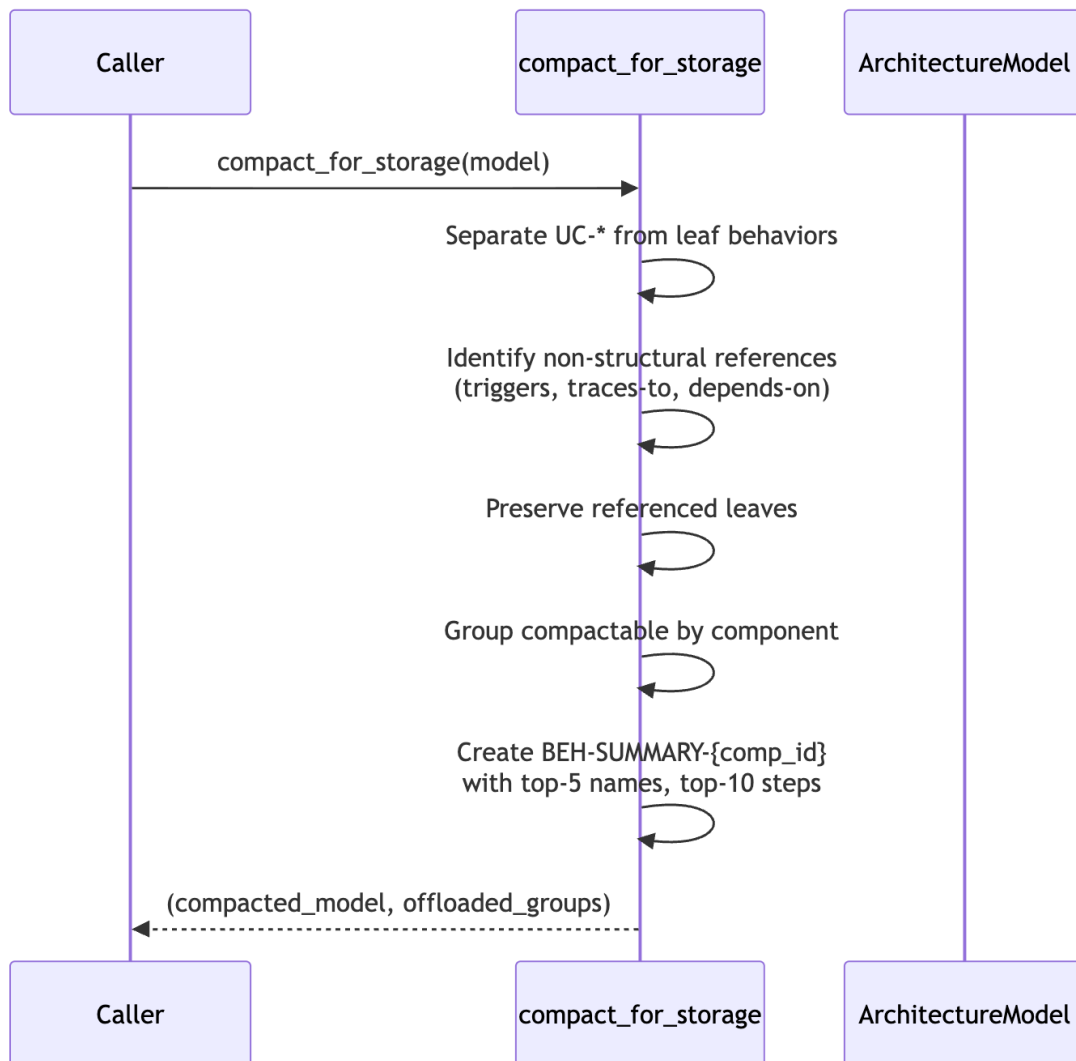


Figure 6: Diagram

Decision: Import-Graph Clustering for Decomposition

Considered: Directory-based grouping, class-hierarchy grouping, semantic similarity. **Chosen:** Import-graph affinity via `cluster_modules` (spectral/community detection on import edges). **Why:** Imports are the strongest signal of coupling in Python. Directory structure reflects organizational choices, not runtime dependencies. Import clustering produces sub-components that minimize cross-cluster dependencies, which is the definition of good modular decomposition. **Trade-off:** Misses runtime coupling (e.g., shared database tables, message queues). Accepted because static analysis is fast and deterministic.

Decision: Call-Graph Trigger Detection with `max_depth=4`

Considered: Unlimited depth, AST-only (no call graph), manual annotation. **Why 4?** Balances discovery (most meaningful trigger chains are 2-3 hops) against false positives (at depth 8+, nearly everything connects to everything via utility functions). Value function: discovery rate increases sharply from depth 1→3, plateaus at 4, false positive rate accelerates past 5.

Decision: Compaction via Summary Behaviors

Considered: Simply deleting low-value behaviors, collapsing into component metadata. **Chosen:** `BEH-SUMMARY-{comp_id}` entities that preserve top-N names as steps. **Why:** Deletion loses information permanently. Summary behaviors maintain traceability (the offloaded groups are returned separately) while keeping the primary model navigable.

7. Measures of Effectiveness

Capability	MoE	Minimum	Optimal	Measurement
Signature Extraction	% of public functions captured	80%	>95%	Compare <code>comp.signatures</code> count to actual public functions in <code>comp.files</code>
Test Contract Discovery	% of components with tests linked	70%	>90%	Components with non-empty <code>test_contracts</code> / active components with test files
Behavior Detection	Precision of trigger-decorated functions identified	>85% precision	>95% precision, <5% false positives	Manual audit sample
Capability Inference	Capabilities map to user-recognizable features	No nonsense capabilities	1:1 with actual product features	Review against product documentation

Capability	MoE	Minimum	Optimal	Measurement
Decomposition Cohesion	Intra-cluster import density vs inter-cluster	Ratio > 2:1	Ratio > 5:1	<code>InternalRelationship.edge_c</code> analysis
Behavior Cap Compliance	No component exceeds 40 behaviors	100% compliance	N/A (hard constraint)	Post-enrichment audit
Compaction Ratio	Behavior count reduction in root model	>50% reduction for CRUD-heavy models	>70% with zero loss of cross-component flows	<code>len(compacted.behaviors)</code> / <code>len(original.behaviors)</code>
Pipeline Completion	End-to-end success rate	>90% of blocks produce artifacts	100% with graceful degradation for unparseable files	<code>PipelineResult.errors</code> count

Failure Modes

Component	Failure Mode	Impact	Graceful?
<code>enrich.py</code>	Source file not found / parse error	Component lacks signatures — model incomplete but valid	Yes — logs warning, skips file
<code>auto_enrich.py</code>	Manifest missing modules	No behaviors/symbols detected — model is a skeleton	Yes — returns model unchanged
<code>deep_decompose.py</code>	Too few modules (< <code>max_modules</code>)	Block not decomposed — remains monolithic	Yes — returns empty <code>DecomposeResult</code>
<code>trigger_detection.py</code>	Call graph incomplete	Missing trigger edges — use cases not inferred	Yes — partial chains still detected
<code>capability_inference.py</code>	No URL prefixes or actors	Falls back to “Internal Operations” catch-all	Yes — degraded but functional
<code>compaction.py</code>	All behaviors have non-structural references	Nothing compacted — model stays large	Yes — returns original model unchanged
<code>pipeline.py</code>	Parent model file missing	Hard failure — cannot proceed without root model (unless <code>from_scratch=True</code>)	Partial — <code>from_scratch</code> bootstraps a minimal model

Logical Architecture: Orchestration Subsystem

1. Intent & Purpose

The Orchestration subsystem exists because architecture models cannot be manually kept in sync with evolving codebases. It automates two fundamental transformations: **enrichment** (populating models with intelligence extracted from source code) and **decomposition** (breaking coarse models into hierarchical sub-models). Without this subsystem, architecture models would drift from reality within days of code changes, becoming fiction rather than documentation.

The philosophical design choice is **AST-first, not LLM-first**: enrichment derives facts (signatures, constants, test contracts) from deterministic code analysis, reserving agent/LLM involvement only for semantic classification (pattern matching, naming). This makes the system reproducible and auditable.

2. Component Structure

COMP-5: Orchestration (Container)

Intent: Provide a single import surface for all orchestration workflows. This boundary exists to decouple CLI and external consumers from the internal split between enrichment and decomposition concerns.

Layer: Application — it coordinates domain operations (core types, clustering, manifests) into user-facing workflows. It sits above domain logic but below CLI/presentation.

Files: `src/architecture_model/orchestration/__init__.py`

Responsibilities: Re-export public API (`enrich_model`, `decompose_model`, `iterative_decompose`, `enrich_from_manifest`, etc.). No logic lives here.

COMP-5.1: Enrichment

Intent: Bridge the gap between raw architecture models (which only declare component boundaries) and rich models (which carry function signatures, constants, test contracts, symbols, behaviors, and patterns). This component exists because a model without code intelligence is just a box-and-arrow diagram — it can't answer “what does this component actually do?”

Layer: Application — orchestrates calls to manifest scanning (COMP-3) and populates core types (COMP-1.1).

Files & Responsibilities:

File	Responsibility	Why It's Separate
<code>enrich.py</code>	AST-based enrichment: signatures, constants, test contracts directly from source files	Pure deterministic extraction, no manifest dependency

File	Responsibility	Why It's Separate
<code>auto_enrich.py</code>	Manifest-based enrichment: signatures, symbols, constants, behaviors, patterns from pre-scanned manifest data	Richer data source (manifest includes call graphs, class hierarchies)
<code>enrichment_context.py</code>	Format decomposition results as prompts for agent-based pattern/contract annotation	Separates prompt engineering from data extraction
<code>capability_inference.py</code>	Infer Capability entities from behavior trigger patterns (URL prefixes, actors)	Automates capability discovery instead of manual authoring
<code>trigger_detection.py</code>	Detect behavior-to-behavior trigger relationships via call graph traversal	Discovers runtime coupling invisible in static imports
<code>use_case_inference.py</code>	Compose linear trigger chains into composite behaviors (use cases)	Elevates implementation-level behaviors to user-facing workflows
<code>naming_context.py</code>	Format decomposition clusters for agent-based semantic naming	Separates naming concerns from clustering logic

COMP-5.2: Decomposition

Intent: A monolithic architecture model becomes unwieldy beyond ~10 components. Decomposition enables hierarchical drill-down by producing per-block sub-models that preserve relationship fidelity to the parent. Without this, teams working on different subsystems must navigate one massive model.

Layer: Application — coordinates manifest scanning, clustering algorithms (COMP-1.5/core.cluster), and model I/O.

Files & Responsibilities:

File	Responsibility	Why It's Separate
<code>decompose.py</code>	Relationship-tracing decomposition: slice parent model per functional block	Produces faithful sub-models by following realizes , exposes , traces-to edges
<code>deep_decompose.py</code>	Import-graph clustering: split a block's modules into sub-components	Uses cluster_modules for affinity-based grouping — distinct algorithm from relationship tracing
<code>behavior_decompose.py</code>	Promote raw step names to structured Step objects with component refs	Bridges unstructured behavior data to typed architecture entities

File	Responsibility	Why It's Separate
<code>behavior_flows.py</code>	Classify behaviors (cross-component vs CRUD vs trivial), trace through call graph	Enables intelligent filtering — only cross-component behaviors merit detailed modeling
<code>compaction.py</code>	Offload leaf behaviors to per-component summaries for storage efficiency	Prevents behavior explosion (addresses REQ-18 indirectly)
<code>pipeline.py</code>	End-to-end pipeline: manifest → decompose → enrich → write	Single entry point eliminates manual step orchestration

3. Dependency Graph

Why Each Dependency Exists

Dependency	Why	What Breaks Without It
COMP-5.1 → COMP-3 (Manifest)	Enrichment reads scanned module data (functions, classes, imports, call graphs)	No signatures, symbols, or behaviors can be auto-populated; models stay empty shells
COMP-5.1 → COMP-1.1 (Core Types)	Enrichment instantiates <code>FunctionSignature</code> , <code>Constant</code> , <code>Symbol</code> , <code>Behavior</code> , <code>Capability</code>	Cannot produce typed model entities; would need raw dicts, losing validation
COMP-5.2 → COMP-1.5 (Quality Metrics)	Decomposition uses clustering quality metrics to evaluate sub-component groupings	Clustering runs blind — no way to assess if decomposition is meaningful
COMP-8 → COMP-5 (CLI → Orchestration)	CLI commands trigger enrichment and decomposition workflows	No user-facing entry point; orchestration becomes library-only

4. Interface Specification

IF-auto-COMP-5: Orchestration API

Contract: Stable import surface for all orchestration workflows. Consumers import from `architecture_model.orchestration` and get `enrich_model`, `decompose_model`, `iterative_decompose`, `enrich_from_manifest`, `enrich_behaviors_from_manifest`, `enrich_with_block_content`, `format_enrichment_prompt`.

Why internal: Only consumed by CLI (COMP-8) and tests within the same process.

IF-auto-COMP-5.1: Enrichment API

Key functions:

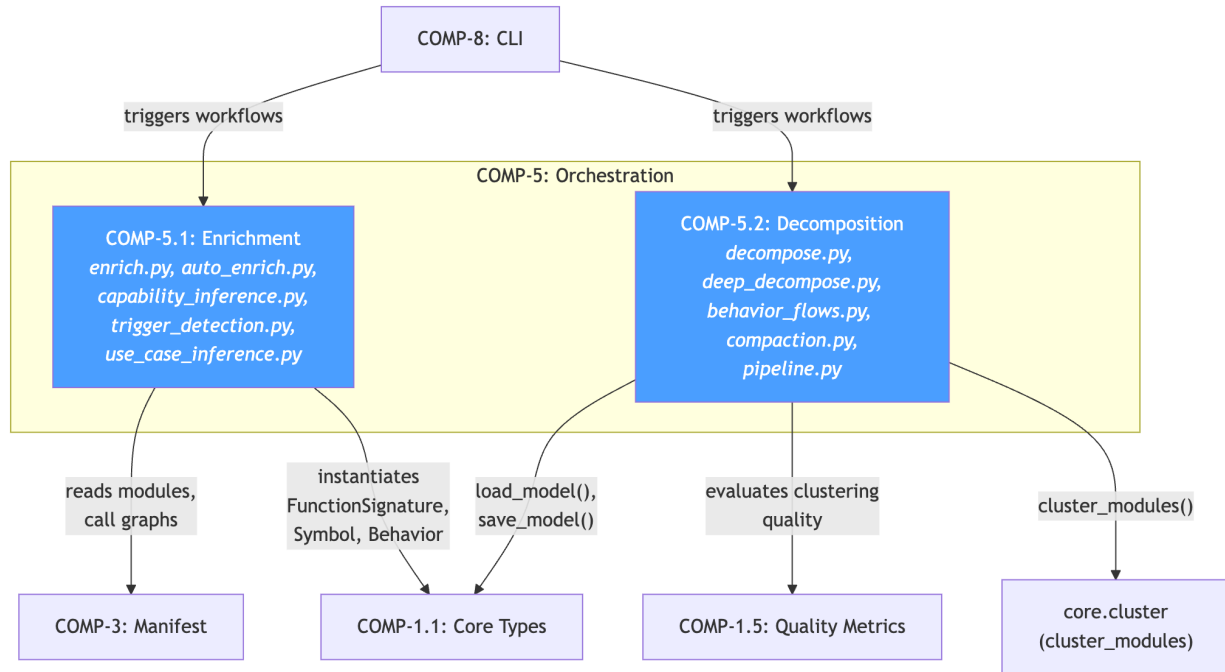


Figure 7: Diagram

Function	Signature	Contract
<code>enrich_model</code>	<code>(model: ArchitectureModel, project_root: Path) -> ArchitectureModel</code>	Idempotent: skips existing signatures/constants by name. Only processes ACTIVE components with files.
<code>enrich_from_manifest</code>	<code>(model, manifest, ...) -> ArchitectureModel</code>	Richer enrichment using pre-scanned manifest. Populates symbols, behaviors, patterns in addition to signatures/constants.
<code>enrich_behaviors_from_manifest</code>	Behaviors-only enrichment	Focused variant for incremental enrichment.
<code>format_enrichment_prompt</code>	<code>(decompositions: list[DecomposeResult]) -> str</code>	Produces agent prompt. Output monitored for token estimate $(\text{len}(r) // 4)$.

IF-auto-COMP-5.2: Decomposition API

Function	Signature	Contract
<code>decompose_model</code>	<code>(model, block_dirs, ...) -> ArchitectureModel</code>	Returns a sub-model that is a faithful slice — no invented entities.

Function	Signature	Contract
<code>deep_decompose_block</code>	<code>(manifest, block_id, block_name, ...) -> DecomposeResult</code>	Returns empty <code>sub_components</code> if block has fewer modules than <code>max_modules</code> (default 15).
<code>run_pipeline</code>	<code>(project_root, ...) -> PipelineResult</code>	End-to-end: <code>manifest</code> $\rightarrow$ <code>decompose</code> $\rightarrow$ <code>write</code> . Errors collected in <code>result.errors</code> , not raised.
<code>compact_for_storage</code>	<code>(model) -> tuple[ArchitectureModel, dict[str, list[Behavior]]]</code>	Returns compacted model + offloaded behaviors grouped by component.

5. Key Data Types

Type	Location	Intent
<code>DecomposeResult</code>	<code>deep_decompose.py</code>	Captures clustering output: sub-components, internal relationships, depth. Exists because decomposition results need to flow through naming, enrichment context, and pipeline stages.
<code>SubComponent</code>	<code>deep_decompose.py</code>	Intermediate representation before promotion to full <code>Component</code> . Carries <code>files</code> , <code>classes</code> , <code>functions</code> , <code>line_count</code> — the minimal data needed for agent naming and pattern inference.
<code>InternalRelationship</code>	<code>deep_decompose.py</code>	Weighted edge between sub-components (<code>edge_count</code>). Distinct from <code>Relationship</code> because it's pre-model — used for clustering evaluation, not persisted directly.
<code>PipelineResult</code>	<code>pipeline.py</code>	Aggregates all pipeline outputs. <code>errors: list[str]</code> enables partial-success semantics.

Type	Location	Intent
BehaviorClassification	behavior_flows.py	Tri-partition of behaviors into <code>cross_component</code> , <code>crud_groups</code> , <code>trivial</code> . Exists to enable selective modeling — only cross-component behaviors warrant full flow diagrams.
CrudSummary	behavior_flows.py	Verb-counted summary of single-component behaviors. Enables compaction without total information loss.

6. Design Decisions & Rationale

Decision	Alternatives Considered	Chosen	Rationale	What Would Change If...
AST-based enrichment as primary path	LLM-only enrichment; hybrid with LLM fallback	Deterministic AST extraction (<code>ast.parse</code>) with LLM only for semantic tasks	Reproducibility: same code → same signatures every time. LLM calls are expensive and non-deterministic.	If AST parsing couldn't handle the language → would need language-server protocol integration
Two enrichment paths (enrich.py vs auto_enrich.py)	Single unified enricher	Separate: <code>enrich_model</code> for direct AST, <code>enrich_from_manifest</code> for pre-scanned data	<code>enrich_model</code> works without manifest iteration (faster for small projects). <code>enrich_from_manifest</code> is richer but requires COMP-3 scan first.	If manifest were always available → could merge into one path

Decision	Alternatives Considered	Chosen	Rationale	What Would Change If...
Behavior cap at 40 (REQ-18)	No cap; dynamic cap based on component size	Fixed cap of 40	Prevents orphan explosion in models with many endpoints. 40 chosen as $\sim 2\times$ typical REST resource controller size. Above 40, behaviors add noise without insight.	If modeling microservices with 100+ endpoints $\rightarrow$ would need per-component caps or hierarchical behavior grouping
Compaction via summary behaviors	Delete leaf behaviors entirely; keep all behaviors	Offload to per-component groups, replace with summaries	Preserves discoverability (summaries reference top-10 behavior names) while keeping root model navigable. Offloaded behaviors remain accessible.	If storage were unlimited $\rightarrow$ skip compaction entirely
Pipeline collects errors instead of raising	Fail-fast on first error	<code>result.errors: list[str]</code>	Decomposition of 20 blocks shouldn't abort because block 3 has a parse error. Partial results are valuable.	If correctness were paramount over completeness $\rightarrow$ fail-fast with transaction semantics
Import-graph clustering for decomposition	Directory-based grouping; manual component assignment	<code>cluster_modules</code> using import affinity	Files that import each other heavily belong together. Directory structure often reflects historical accidents, not architectural intent.	If codebase used a strict directory-per-component convention $\rightarrow$ directory grouping would be simpler and sufficient

7. Failure Modes

Component	Failure Mode	Impact	Graceful?
enrich.py — file not found	Source file referenced in <code>comp.files</code> doesn't exist	Skipped with <code>logger.debug</code> ; other files still processed	Graceful — partial enrichment
enrich.py — AST parse error	Malformed Python source	Caught by <code>except Exception</code> ; logged as warning; component gets no signatures from that file	Graceful
auto_enrich.py — manifest missing modules	Manifest scan incomplete	Components with no matching modules get no enrichment	Graceful — model remains valid, just sparse
deep_decompose.py — too few modules	Block has < <code>max_modules</code> files	Returns empty <code>sub_components</code> — block stays monolithic	By design — small blocks don't need decomposition
deep_decompose.py — clustering produces degenerate result	All modules in one cluster	Single sub-component returned; no internal relationships	Degraded — decomposition adds no value but doesn't break
trigger_detection.py — call graph missing entries	Entry function not in <code>call_graph.edges</code>	Behavior skipped; no trigger relationships detected for it	Graceful — fewer relationships, not incorrect ones
capability_inference.py — no URL prefixes found	Behaviors lack HTTP triggers	Falls back to actor-based grouping, then “Internal Operations” catch-all	Graceful with degraded specificity
pipeline.py — model file not found	No <code>.architecture-model.yaml</code> exists	Error appended to <code>result.errors</code> ; returns partial result	Graceful
compaction.py — behavior has no <code>realizes</code> relationship	Orphan behavior with no component link	Not grouped into any <code>comp_groups</code> ; effectively dropped from compacted model	Silent data loss — orphan behaviors disappear

Critical Failure Path

The hardest failure is **COMP-3 (Manifest) unavailability**: both enrichment and decomposition depend on manifest data. If manifest generation fails, `enrich_from_manifest` has nothing to work with and `deep_decompose_block` has no modules to cluster. The fallback is `enrich_model` (direct AST), which provides basic signatures but no symbols, behaviors, or call graphs.

8. Measures of Effectiveness

MoE	Minimum	Good	Excellent
Signature coverage	>50% of public functions captured	>80%	>95% with accurate params/returns
Decomposition utility	Sub-components exist	Sub-components map to recognizable concerns	Sub-components match what a human architect would draw
Behavior relevance	Behaviors detected	Cross-component behaviors distinguished from CRUD	Use cases inferred that match user-facing features
Pipeline resilience	Completes without crash	Partial results on errors	Zero silent data loss; all skips logged
Enrichment idempotency	Running twice doesn't duplicate	Perfect dedup by name	Incremental: only processes changed files

Orchestration Subsystem — Use Cases

Why This Document Exists

The Orchestration subsystem is the bridge between raw source code and structured architecture knowledge. Without it, architecture models are static documents that drift from reality. These use cases capture the workflows through which models are automatically enriched, decomposed, and kept faithful to the codebase.

Actors

Actor	Description
Engineer	Invokes orchestration via CLI (COMP-8) to update architecture models
CI Pipeline	Automated trigger for model refresh on code changes
Agent (LLM)	Consumes formatted prompts to assign patterns/contracts to components

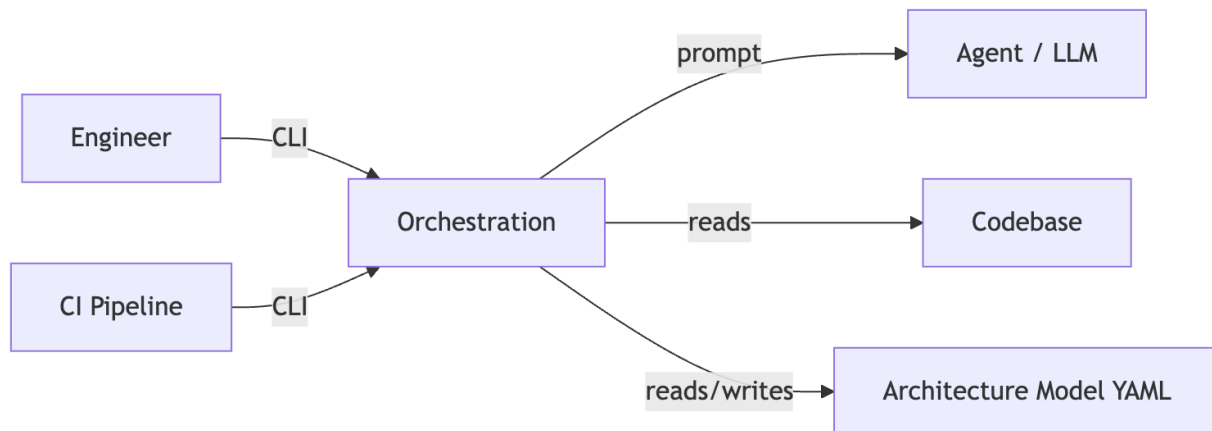


Figure 8: Diagram

UC-1: Enrich Model from AST

Actor: Engineer / CI Pipeline

Intent: Eliminate manual bookkeeping of function signatures, constants, and test contracts in architecture YAML. The model should reflect what the code actually exports, not what someone remembered to document.

Preconditions: - A valid `ArchitectureModel` is loaded with components that have `files` populated - `project_root` points to a repo where those files exist on disk - Components have `status == ACTIVE`

Main Flow: 1. Engineer invokes enrichment (via `enrich_model(model, project_root)` in `enrich.py`) 2. For each ACTIVE component with files, `_enrich_signatures()` calls `extract_file_hints()` to parse AST and extract public function signatures as `FunctionSignature` objects 3. `_enrich_constants()` parses module-level assignments via `ast.parse()` into `Constant` objects, deduplicating by name against `comp.constants` 4. `_enrich_test_contracts()` discovers test files via convention and calls `analyze_test_file()` to produce `TestContract` entries 5. The enriched `ArchitectureModel` is returned

Postconditions: - Every ACTIVE component's `signatures`, `constants`, and `test_contracts` lists reflect current source - No duplicate entries (deduplication by name via `existing_names` sets) - Inactive components and components without files are untouched

Error Handling: - Missing files: logged at DEBUG via `logger.debug("File not found: %s", fpath)` — degraded but not fatal - Unparseable source: caught by broad `except Exception`, logged at WARNING — that file is skipped, other files still enriched - This is a **graceful degradation** design: partial enrichment is always better than none

Quality Attributes: - Idempotent: running twice produces the same result (dedup guards) - Non-destructive: existing manually-added entries are preserved

Measures of Effectiveness: - **Coverage ratio:** % of component files successfully parsed / total component files — target >95% - **Signature freshness:** delta between signatures in model vs. actual public API — should be zero after enrichment - **Zero false additions:** no private functions (guarded by `include_private=False`) leak into signatures

UC-2: Enrich from Manifest Data

Actor: Engineer / CI Pipeline

Intent: When a full `Manifest` (from AST scanning) is already available, use its richer data (class hierarchies, decorator info, docstrings) to populate symbols, patterns, responsibilities, and behaviors — going beyond what simple AST extraction provides.

Preconditions: - A `Manifest` object with populated `modules: list[ModuleInfo]` exists - Components have files that match `ModuleInfo.file` paths

Main Flow: 1. `enrich_from_manifest(model, manifest)` is called (in `auto_enrich.py`) 2. For each component, matched modules are found by file path 3. Functions are converted via `_parse_signature(name, func)` into `FunctionSignature` with params, returns, and `body_hint` from docstrings 4. Classes are converted via `_class_to_symbol(cls)` into `Symbol` objects, with `_detect_symbol_kind()` inferring kind from decorators/bases (dataclass, protocol, enum, exception, etc.) 5. Trigger detection via `_TRIGGER_DECORATORS` regex identifies behaviors from decorated functions (route, event, handler, etc.) 6. Pattern matching via `load_patterns()` assigns architectural patterns based on indicator matching

Postconditions: - Components have `symbols` with correct `SymbolKind` classification - Behaviors are detected from trigger decorators, not just manually authored - Patterns are inferred from code indicators

Error Handling: - Unrecognized decorators/bases: default to `SymbolKind.CLASS` — safe fallback - No manifest modules matching a component: component is skipped silently

Quality Attributes: - Symbol kind accuracy depends on decorator naming conventions — works well for standard Python patterns

Measures of Effectiveness: - **Symbol classification accuracy:** % of symbols with correct `SymbolKind` vs. manual review - **Behavior detection recall:** % of actual entry points detected as behaviors via trigger decorators - **Pattern inference precision:** % of auto-assigned patterns that an architect would agree with

UC-3: Deep Decompose a Block

Actor: Engineer

Intent: Large monolithic components (many files) are architecturally opaque. Decomposition clusters files by import affinity to reveal internal structure — sub-components that cohesively group related code.

Preconditions: - A **Manifest** for the block with modules and their import edges - Block has more modules than `max_modules` threshold (default 15) — blocks below this are too small to meaningfully decompose

Main Flow: 1. `deep_decompose_block(manifest, block_id, block_name)` is called (in `deep_decompose.py`) 2. `__init__.py` files are filtered out (not meaningful standalone modules) 3. `cluster_modules()` groups modules by import-graph affinity into `target_k` clusters (default 5) 4. Clusters smaller than `min_cluster_size` (default 3) are merged 5. Each cluster becomes a `SubComponent` with aggregated files, classes, functions, and `line_count` 6. Import edges between clusters become `InternalRelationship` entries with `edge_count` 7. `DecomposeResult` is returned

Postconditions: - Each source file appears in exactly one `SubComponent` - `InternalRelationship` edges capture the coupling strength between sub-components - Result includes `depth` for tracking recursive decomposition levels

Error Handling: - Too few modules: returns empty `sub_components` — the block is simply not decomposed - Clustering algorithm produces degenerate results (one giant cluster): acceptable — it reflects genuine tight coupling

Quality Attributes: - Deterministic given same input manifest - REQ-18 satisfaction: enrichment applies a **maximum 40 behaviors per component** cap to prevent orphan explosion during decomposition

Measures of Effectiveness: - **Cohesion:** avg intra-cluster import density vs. inter-cluster import density — higher ratio = better decomposition - **Cluster balance:** std deviation of `line_count` across sub-components — lower = more balanced - **Actionability:** an architect can name each sub-component from its files/classes without ambiguity

Trade-off Rationale: `target_k=5` balances granularity (too many clusters = noise) against lumping (too few = no insight). `min_cluster_size=3` prevents single-file clusters that add overhead without value.

UC-4: Run Full Decomposition Pipeline

Actor: Engineer / CI Pipeline

Intent: Execute the entire decomposition workflow in one command — from scanning source to writing per-block sub-models — so that architecture artifacts stay current with minimal manual effort.

Preconditions: - `project_root` contains `.architecture-model.yaml` with `functional_blocks` configuration - Source code is available at paths referenced by the model

Main Flow: 1. `run_pipeline(project_root, deep=True)` is called (in `pipeline.py`) 2. `generate_recursive_manifests()` performs per-block AST scanning 3. `decompose_model()` traces relationships from block components to extract sub-models (capabilities, interfaces, behaviors, constraints) 4. If `deep=True`, `iterative_decompose()` recursively decomposes large blocks 5. `enrich_from_manifest()` populates signatures, symbols, and interfaces on sub-model components 6. `_enrich_test_contracts()` adds test coverage data 7. `write_sub_models()` writes artifacts to `.architecture-models/<block_id>/` 8. If `compact=True`, root model is compacted 9. `PipelineResult` is returned with manifests, `sub_models`, `deep_decompositions`, `written_paths`, and errors

Postconditions: - `.architecture-models/` directory contains per-block YAML sub-models - `PipelineResult.errors` lists any blocks that failed (partial success is valid)

Error Handling: - Per-block failures are caught and appended to `result.errors` — other blocks still process - Missing model file with `from_scratch=True`: bootstraps a model from manifest using module grouping - This is **fail-soft by design**: the pipeline always produces as much output as possible

Quality Attributes: - Monitored via `@monitored` decorator tracking `blocks_scanned`, `blocks_decomposed`, `errors` - Idempotent: safe to re-run

Measures of Effectiveness: - **Completion ratio:** `blocks_decomposed / blocks_scanned` — target 100% - **Error rate:** `len(errors) / len(manifests)` — target 0% - **Freshness latency:** time from code change to updated sub-model — should be single CI cycle

UC-5: Infer Capabilities from Behaviors

Actor: CI Pipeline (automated post-enrichment step)

Intent: Manually curating capabilities is tedious and drifts. By grouping behaviors by URL prefix or actor, capabilities emerge naturally from the code's actual entry points.

Preconditions: - Model has behaviors with populated `trigger` fields (e.g., `"POST /users/{id}"`) or `actor` fields

Main Flow: 1. `_infer_caps_from_urls(behaviors, existing_caps, existing_rels)` is called (in `capability_inference.py`) 2. `_extract_url_prefix(trigger)` extracts the first path segment (e.g., `"users"`) 3. Behaviors are grouped by prefix; those without URL triggers fall back to actor grouping 4. `_name_from_prefix()` applies `_singularize()` and title-casing to produce capability names (e.g., `"User Management"`) 5. New `Capability` entities and `REALIZES` relationships are created 6. Ungrouped behaviors get an `"Internal Operations"` capability

Postconditions: - Every behavior is linked to at least one capability via **REALIZES** - Capability IDs are sequential starting after existing capabilities

Error Handling: - Behaviors without trigger or actor: grouped under “Internal Operations” — nothing is lost - Singularization heuristic fails: produces slightly awkward names but no functional impact

Measures of Effectiveness: - **Grouping coherence:** behaviors within a capability should share a logical domain — measurable by manual review - **Coverage:** 100% of behaviors linked to a capability - **Name quality:** generated names should be immediately understandable without looking at constituent behaviors

UC-6: Infer Composite Behaviors (Use Cases) from Trigger Chains

Actor: CI Pipeline

Intent: Individual behaviors (single API endpoints, event handlers) don’t tell the story of end-to-end user journeys. By following **triggers** relationship chains, the system synthesizes composite behaviors that represent complete workflows.

Preconditions: - Model has **triggers** relationships between behaviors (produced by `detect_behavior_triggers()`) - At least one chain of 2 behaviors exists

Main Flow: 1. `infer_composite_behaviors(model)` is called (in `use_case_inference.py`) 2. `_find_chains(triggers)` builds a directed graph and identifies chain heads (behaviors not targeted by any trigger) 3. Each chain is followed forward via BFS until no successors remain 4. For chains of length 2, a composite **Behavior** with `id=f"UC-{i}"` is created 5. Composite inherits **trigger** and **actor** from the chain head 6. **steps** are populated with names of constituent behaviors 7. **pattern** is set to `BehaviorPattern.SEQUENTIAL` 8. **CONTAINS** relationships link composite to chain members

Postconditions: - New UC-* behaviors represent end-to-end flows - Original leaf behaviors remain unchanged - Relationship graph gains **CONTAINS** edges from composites to leaves

Error Handling: - No triggers in model: returns model unchanged (no-op) - Cycles in trigger graph: **visited** set prevents infinite loops — cycle members are included up to the first revisit - Chain head behavior not found in index: that chain is skipped

Measures of Effectiveness: - **Chain detection recall:** % of actual multi-step workflows captured as composites - **Composite naming clarity:** “{head.name} (end-to-end)” should be descriptive - **Step completeness:** all chain members appear in **steps**

UC-7: Format Enrichment Context for Agent Annotation

Actor: Agent (LLM), triggered by Engineer

Intent: After decomposition produces sub-components with files/classes/functions, an LLM needs compact, structured context to assign architectural patterns and contracts. This use case produces that prompt — optimized for token efficiency and annotation accuracy.

Preconditions: - One or more `DecomposeResult` objects with populated `sub_components` - Pattern catalog is loadable via `load_patterns()`

Main Flow: 1. `format_enrichment_prompt(decompositions)` is called (in `enrichment_context.py`)
 2. Section 1: Pattern catalog is loaded and rendered with top-3 indicators per pattern
 3. Section 2: Each `DecomposeResult`'s sub-components are listed with files (capped at 5), classes (capped at 4), functions (capped at 4), and line count 4. Section 3: YAML response format instructions with rules (one-sentence contracts, pattern selection)
 4. Section 3: YAML response format instructions with rules (one-sentence contracts, pattern selection)
 5. Joined string is returned

Postconditions: - Output is a self-contained prompt string the agent can process without additional context - Monitored: `token_estimate` (len/4) and `char_count` are tracked

Error Handling: - Empty decompositions list: produces a prompt with pattern catalog and instructions but no components — agent returns empty annotations - Files exceeding display cap: shown as "+N more" to prevent token bloat

Quality Attributes: - Token efficiency: caps on files/classes/functions prevent prompt explosion for large blocks - Deterministic output given same input

Measures of Effectiveness: - **Token economy:** prompt size relative to number of components — lower tokens-per-component = better - **Agent annotation accuracy:** % of patterns/contracts the agent assigns correctly given this prompt — the true end-to-end measure - **Completeness:** every leaf component appears in the prompt — no silent drops

Trade-off Rationale: Capping displayed files at 5 and classes at 4 trades completeness for token budget. The assumption is that top files/classes are sufficient for pattern inference; if not, the agent can use "custom".

Summary Flow

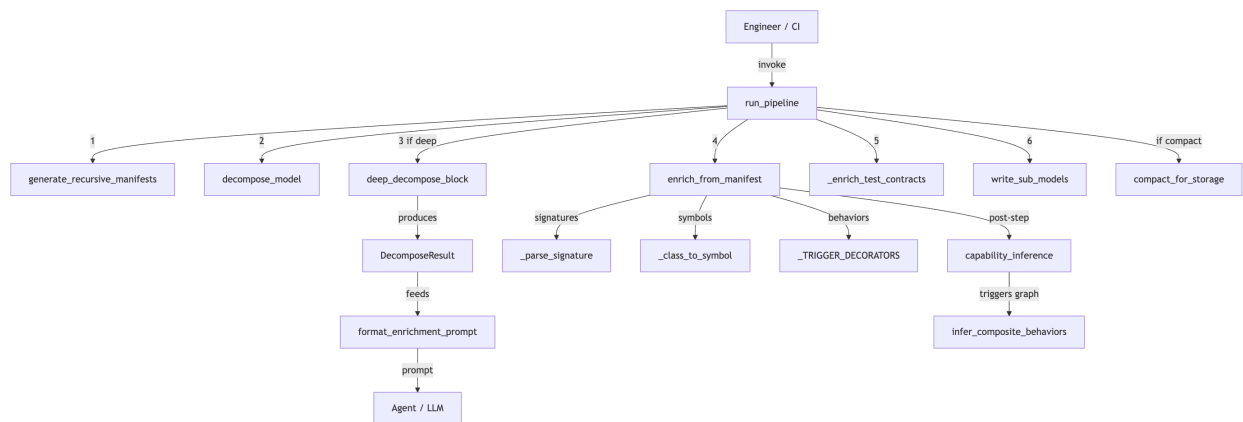


Figure 9: Diagram

Failure Mode Summary

Failure	Impact	Degradation
Source file missing	Signatures/constants not extracted for that file	Graceful — other files still enriched
AST parse error	Single file skipped	Graceful — logged at WARNING
Manifest empty	No enrichment occurs	Graceful — model returned unchanged
Clustering degenerate	One giant sub-component	Functional but uninformative — reflects real coupling
All blocks fail in pipeline	PipelineResult.errors populated, no sub-models written	Graceful — errors reported, no data corruption
Behavior cap (REQ-18) exceeded	Behaviors truncated at 40 per component	By design — prevents orphan explosion

Model Completeness: D (42%) Some sections may be empty due to missing model entities. - 3/3 components have no behavioral specification - Only 1/3 components have linked requirements - No actors defined → conops stakeholder section empty - No constraints defined → operations manual empty Run the extraction pipeline or manually add behaviors/interfaces/constraints.

Artifact Traceability Map: architecture-model-standard / Orchestration

1. Entity Inventory

Entity Type	Count	Feeds SE Documents
Components	3	Logical Architecture, Maintenance Manual, Operations Manual, Interface Specification, Component Specs
Capabilities	2	ConOps, Functional Analysis, Requirements Analysis
Behaviors	0	Use Cases, Functional Analysis, Verification & Validation, Behavior Flows
Interfaces	3	Interface Specification, Logical Architecture
Constraints	0	Requirements Analysis, Risk Assessment
Requirements	1	Requirements Analysis, Verification & Validation
Actors	0	ConOps, Use Cases
Layers	0	Logical Architecture

2. Artifact Dependency Graph

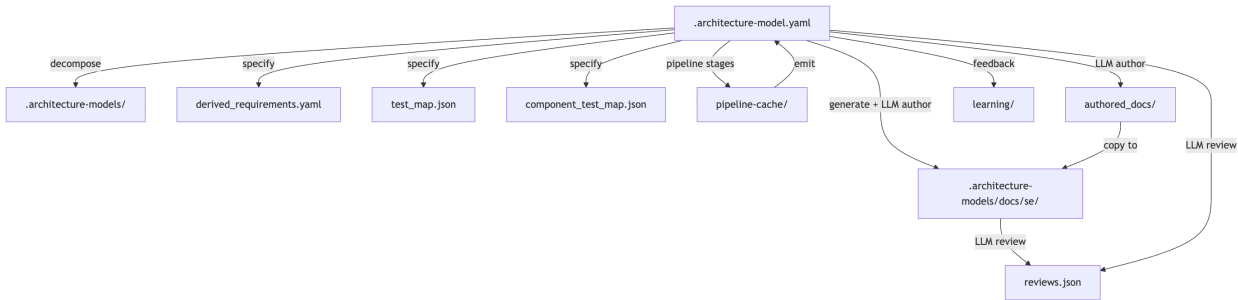


Figure 10: Diagram

3. Entity-to-Artifact Traceability Matrix

Artifact	Component	Capabilities	Behaviors	Interfaces	Constraints	Requirements	Actors	Layers
Behavior Flows			—					

Artifact	Component	Capabilities	Behaviors	Interfaces	Constraints	Requirements	Actors	Layers
Component Specs	3							
ConOps	2						—	
Functional Analysis	2		—					
Interface Specification	3			3				
Logical Architecture	3			3				—
Maintenance Manual	3							
Operations Manual	3							
Requirements Analysis	2				—	1		
Risk Assessment					—			
Use Cases			—				—	
Verification & Validation			—			1		

4. Relationship Distribution

Relationship Type	Count	Connects
depends-on	5	Component → Unknown, Unknown → Component, Unknown → Unknown
exposes	3	Component → Interface
realizes	2	Component → Capability
contains	2	Component → Component
satisfies	1	Component → Requirement

5. Traceability Gaps

- **Behaviors** — 0 entities; leaves gaps in: Use Cases, Functional Analysis, Verification & Validation, Behavior Flows
- **Constraints** — 0 entities; leaves gaps in: Requirements Analysis, Risk Assessment
- **Actors** — 0 entities; leaves gaps in: ConOps, Use Cases

- **Layers** — 0 entities; leaves gaps in: Logical Architecture
- **allocated-to** relationship type missing — weakens cross-entity traceability
- **constrained-by** relationship type missing — weakens cross-entity traceability