

Core Subsystem — Concept of Operations

1. System Overview

Intent

The Core subsystem is the **single source of truth for what an architecture model IS and how to manipulate it**. Every other subsystem in the architecture-model-standard—CLI, pipelines, enrichment, documentation generation, export—depends on Core to define data structures, parse YAML, validate correctness, and perform analytical operations like slicing, diffing, and coverage analysis.

Without Core, there is no shared type system, no validation, no way to load or save models, and no analytical operations. Every subsystem would reinvent these independently, leading to divergent interpretations of what constitutes a valid model.

Philosophy

Core follows a **foundation library** pattern: it is stateless, has no external orchestration logic, and exposes pure functions and dataclasses. Design choices prioritize:

- **Round-trip fidelity over convenience** — loading and saving a model must not lose data or reorder fields (REQ-27), because models are version-controlled and spurious diffs destroy trust.
- **Validation as a first-class operation** — not an afterthought. The validator enforces structural invariants (referential integrity, hierarchy consistency) AND semantic rules (status consistency, completeness). A model that passes validation is safe to consume downstream.
- **Token efficiency for AI consumption** — the slicer exists because full models exceed LLM context windows. The 4000-token budget (REQ-20) is sized for focused single-component reasoning within typical 8K–32K context windows.

Subcomponents

ID	Name	Role
COMP-1.1	Type System	Dataclasses, enums, ArchitectureModel root type
COMP-1.2	Validation	Schema checks, referential integrity, hierarchy, domain rules
COMP-1.3	Parser & Persistence	YAML load/save, round-trip preservation, multi-file merging, snapshot storage
COMP-1.4	Model Operations	Slice, diff, coverage, clustering, source-block assignment
COMP-1.5	Quality Metrics	Confidence scoring, regen readiness, corrections tracking, visualization

2. Stakeholders & Actors

Actor	Type	Goal	Core Interface Used
CLI (COMP-8)	Internal component	Execute user commands (validate, slice, diff, visualize)	All Core APIs
Pipeline stages (COMP-2.x)	Internal components	Coordinate extraction: allocate entities to types, validate, emit YAML	Type System, Validation, Parser
Enrichment (COMP-5.x)	Internal component	Populate model entities with code-grounded data, decompose into subsystems	Type System, Quality Metrics
Doc generators (COMP-4.1)	Internal component	Read typed model to produce architecture documents	Type System
Export (COMP-10)	Internal component	Serialize model data to external formats	Parser & Persistence
Authoring (COMP-7)	Internal component	Create/modify model entities programmatically	Type System
Human architect	External	Understand model health, identify gaps, track drift	Validation results, coverage reports, regen scores

3. Operational Scenarios

Scenario 1: Validate a Model After Editing

Intent: An architect has manually edited `.architecture-model.yaml` and needs to know if the model is still structurally and semantically correct before committing.

1. CLI invokes `validate()` from `validator.py` with a parsed `ArchitectureModel`.
2. Validator checks ID uniqueness, referential integrity (all `from_id/to_id` in relationships resolve to entities), hierarchy consistency (REQ-3: `parent_id` children bidirectional), status consistency, and completeness.
3. Returns `ValidationResult` with `score` (0–100), `is_valid`, `completeness_grade`, and actionable `issues`.
4. If `score < 80` (REQ-1), the model fails the quality gate. Each `ValidationIssue` carries a `severity`, `code`, and `entity_id` so the architect knows exactly what to fix.

Why the 80-point threshold: Score deducts 10 per error and 2 per warning. This means 2 errors pass (marginal), but 3+ errors fail. The threshold balances strictness (catching real problems) against practicality (not blocking on warnings).

Scenario 2: Slice a Model for AI Context

Intent: An LLM needs to reason about a single source block (e.g., “S1”) without exceeding its context window. The full model may be 20K+ tokens.

1. Consumer calls `slice_by_source_block(model, "S1")` from `slicer.py`.
2. Slicer first attempts to load a pre-split sub-model file via `load_block_model()`. If found, returns it directly (fast path).
3. Otherwise, filters capabilities, behaviors, components, and interfaces by `source_block` tag, then collects only relationships where both endpoints are in the slice.
4. Returns a new `ArchitectureModel` containing only S1's entities, fitting within the 4000-token budget (REQ-20).

Trade-off: Slicing discards cross-block context. This is intentional—focused accuracy within a block is more valuable than diluted full-model context for code generation tasks.

Scenario 3: Detect Architectural Drift

Intent: After code changes, determine whether the model still accurately represents the codebase, and surface specific gaps.

1. Pipeline loads the current model and a fresh manifest (code-grounded facts).
2. `compute_coverage(model, manifest)` in `coverage.py` checks component coverage (model components vs. manifest modules), relationship accuracy, and file mapping.
3. `compute_representativeness(model, modules, edges)` in `representativeness.py` produces `file_coverage`, `relationship_accuracy`, `boundary_coherence`, and `behavioral_coverage` scores.
4. Returns `CoverageResult` and `RepresentativenessResult` with specific `missing`, `extra`, and `uncovered_files` lists.

Failure mode: If coverage drops below ~70%, downstream document generation produces misleading architecture descriptions. The system degrades gracefully—it reports gaps rather than silently producing incorrect output.

Scenario 4: Assess Regeneration Readiness

Intent: Before attempting blind code regeneration from the model, determine which components have sufficient detail (body hints, test contracts, signatures) and which would produce broken output.

1. `compute_regen_readiness(model)` in `regen_readiness.py` iterates components, scoring each on `body_hint_coverage`, `test_contract_count`, `constant_coverage`, `signature_coverage`.
2. Each `ComponentReadiness` includes specific `blockers` (e.g., “3 functions missing `body_hint`”) and a 0–100 score.
3. Returns `RegenReadiness` with `overall` score, `letter grade`, and `recommendation`.
4. Per-component scores (REQ-16) let teams prioritize enrichment effort on the lowest-scoring components.

Value function: A component at 90% readiness is dramatically more valuable than one at 60%—the last 10% of coverage eliminates the most subtle regeneration bugs. The relationship is nonlinear.

4. System Context

Why these dependencies exist: - Every consumer needs `ArchitectureModel` and its entity types (COMP-1.1) — this is the shared language. - Pipeline's validate stage (COMP-2.4) dele-

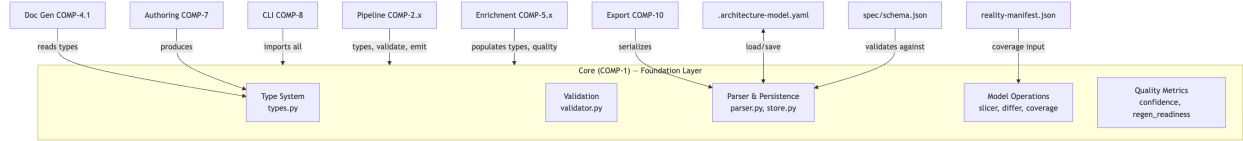


Figure 1: Diagram

gates to Core’s validator rather than reimplementing checks — single source of validation logic. - Pipeline’s emit stage (COMP-2.5) uses Core’s parser for YAML output to guarantee round-trip fidelity. - Enrichment’s decomposer (COMP-5.2) uses quality metrics to decide when a component is complex enough to become a subsystem.

5. Operational Constraints

Constraint	Threshold	Rationale	Violation Impact
Round-trip fidelity	Zero data loss or reordering (REQ-27)	Models are git-tracked; spurious diffs destroy review workflows and trust	Hard failure — users lose data silently
Schema backward compatibility (REQ-28)	New versions must load old schemas	Models persist across tool upgrades; breaking this orphans existing projects	Hard failure — users cannot upgrade
Validation score 80 (REQ-1)	2 errors allowed	Quality gate for downstream consumption; 3+ errors means structural problems	Graceful — reports score, doesn’t crash
Slice token budget (REQ-20)	4000 tokens	Sized for single-block AI reasoning in 8K–32K context windows	Graceful — oversized slices reduce AI output quality
Zero errors on valid models (REQ-2)	Exactly 0 false positives	False validation errors erode trust and cause users to ignore real issues	Graceful but corrosive — trust degradation
Hierarchy bidirectional consistency (REQ-3)	parent_id children must agree	One-directional references cause navigation bugs in slicing and visualization	Hard failure for operations that traverse hierarchy

Failure modes: - **Parser failure** (malformed YAML, missing schema): Hard stop. No model = no operations. Error message with line number. - **Validation failure:** Graceful. Returns **ValidationResult** with issues; consumers decide whether to proceed. - **Slicer on unknown block ID:** Returns empty model. No crash, but consumer gets no useful context. - **Coverage**

with no manifest: Returns zero scores with empty gap lists. Operations continue but metrics are meaningless.

6. Data Flow

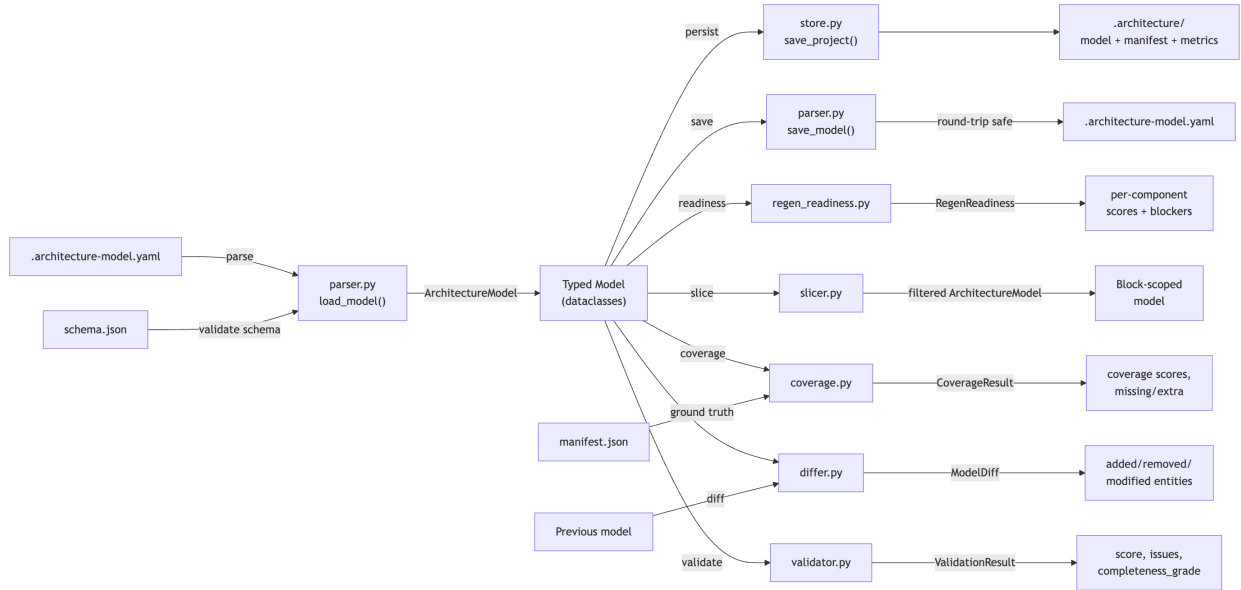


Figure 2: Diagram

7. Measures of Effectiveness

MoE	What “good” looks like	What “bad” looks like	How to measure
Validation precision	Zero false positives on valid models; every reported error is a real problem	Users ignore validation output because of noise	<code>error_count == 0</code> on known-good test models (REQ-2)
Validation recall	Catches hierarchy inconsistencies, dangling references, orphans before they cause downstream failures	Broken models pass validation and cause silent errors in doc gen or code regen	Fault injection: introduce known defects, verify detection
Round-trip fidelity	<code>load(save(model)) == model</code> byte-for-byte on YAML output	Git diffs show reordered keys, lost comments, changed quoting	Automated round-trip tests comparing input/output YAML

MoE	What “good” looks like	What “bad” looks like	How to measure
Slice relevance	Sliced model contains all entities needed for the target block’s reasoning, nothing extraneous	Slice includes unrelated components (noise) or misses critical dependencies (gaps)	Measure entity count vs. known-good slices; token count 4000
Coverage accuracy	<code>overall_score</code> correlates with actual model-vs-code alignment	High coverage score but model is clearly wrong; or low score on a well-modeled repo	Compare <code>CoverageResult.overall_score</code> against manual assessment
Regen readiness correlation (REQ-15)	Components scoring 90+ regenerate successfully; components scoring <50 fail predictably	Score says “ready” but generated code is broken	Track regen success rate per component against readiness score
Diff completeness	Diff catches all meaningful changes between model versions; no silent omissions	Structural changes (new component, removed relationship) go unreported	Compare <code>ModelDiff</code> output against known edit sets
Schema evolution safety	Models from schema v1.0 load correctly under v1.3 parser	Upgrade breaks existing projects	Load archived models from each prior schema version

Value beyond thresholds: Validation score above 80 merely passes the gate. A score of 95+ means the model is rich enough for high-confidence downstream operations. Each point above 80 reduces the probability of subtle errors in generated documentation and code. The relationship between validation score and downstream output quality is the ultimate measure of Core’s effectiveness.

Core Subsystem — Functional Analysis

1. Intent & Purpose

The Core subsystem is the **foundation layer** of the architecture-model system. Every other subsystem—pipeline orchestration, CLI, documentation generation, enrichment, export—depends on Core for its type definitions, validation logic, parsing, and model operations.

Why it exists: Without a single, authoritative source of truth for what an architecture model *is* (its types), how to verify it (validation), how to read/write it (parser), and how to query it (slicer, differ, coverage), every consumer would reinvent these primitives inconsistently. Core eliminates that divergence.

What would break without it: Everything. Core is the gravitational center—COMP-2.1, COMP-2.3, COMP-2.4, COMP-2.5, COMP-4.1, COMP-5.1, COMP-5.2, COMP-7, COMP-8, and COMP-10 all depend on it.

2. Capability Inventory

ID	Capability	Intent	Optimal Delivery (MoE)
CAP-1	Validate Architecture Models	Catch structural errors <i>before</i> they propagate to downstream consumers (generators, pipelines). A model with broken references silently produces wrong artifacts.	Zero false negatives on referential integrity; score correlates with actual downstream failure rate; validation completes in <1s for models with 500+ entities
CAP-7	Slice and Query Models	LLM context windows are finite. Delivering a full model wastes tokens and degrades AI reasoning quality. Slicing gives focused, relevant context.	Sliced output fits 4000 tokens (REQ-20); slice retains all entities needed to reason about a single block with zero dangling references
CAP-8	Diff Model Versions	Detect architectural drift between regeneration cycles. Without diff, staleness is invisible.	Diff detects 100% of added/removed entities; modified-field detection covers all typed fields; diff runs in O(n) entity count

ID	Capability	Intent	Optimal Delivery (MoE)
CAP-11	Assess Regen Readiness	Predict whether a model has enough detail to regenerate code <i>before</i> attempting expensive LLM regeneration. Saves tokens and developer time.	Score correlates with actual regeneration success rate (REQ-15); per-component blockers are actionable (REQ-16)
CAP-13	Detect and Fix Model Drift	Models rot as code evolves. Coverage analysis compares model against manifest reality to surface gaps before they become architectural debt.	File coverage 90% for well-modeled repos; relationship accuracy identifies spurious edges; overall score is a single number a CI gate can use

3. Functional Decomposition

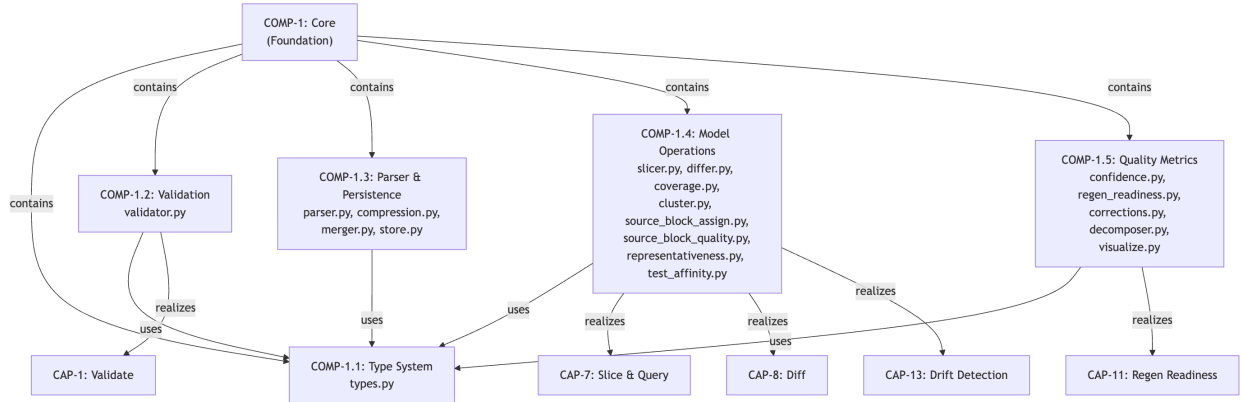


Figure 3: Diagram

4. Capability-Component Mapping

Capability	Realizing Component	Rationale
CAP-1 → COMP-1.2	Validation (<code>validator.py</code>)	Validation is complex enough to warrant isolation: referential integrity, hierarchy checks, status consistency, completeness grading, and regen readiness checks are distinct rule families. Separating from types keeps the type system pure (data only, no policy).
CAP-7 → COMP-1.4	Model Operations (<code>slicer.py</code>)	Slicing is a query operation over the model graph. Collocated with <code>differ.py</code> and <code>coverage.py</code> because all three traverse the same entity/relationship graph with similar access patterns.
CAP-8 → COMP-1.4	Model Operations (<code>differ.py</code>)	Diff compares two <code>ArchitectureModel</code> instances structurally. Shares the entity-indexing pattern with slicer and coverage.
CAP-11 → COMP-1.5	Quality Metrics (<code>regen_readiness.py</code> , <code>confidence.py</code>)	Regen readiness is a <i>quality judgment</i> , not a structural operation. It scores <code>body_hint</code> coverage, test contracts, constants—all qualitative assessments. Grouped with <code>confidence.py</code> which uses the same field-completeness scoring philosophy.
CAP-13 → COMP-1.4	Model Operations (<code>coverage.py</code> , <code>representativeness.py</code>)	Drift detection compares model against manifest (code reality). This is a cross-referencing operation similar to diff but between heterogeneous data sources.

Trade-off: Why not separate slicer/differ/coverage into their own components? They share the same dependency (COMP-1.1 types) and the same traversal patterns. Splitting would increase import complexity without improving cohesion. The current grouping in COMP-1.4 keeps related graph-query operations together.

5. Key Behavioral Flows

5.1 Model Validation Flow

Intent: Give the user a single, authoritative answer to “Is my model correct and complete enough to use?” before any downstream processing occurs. Early detection prevents cascading failures in pipeline stages.

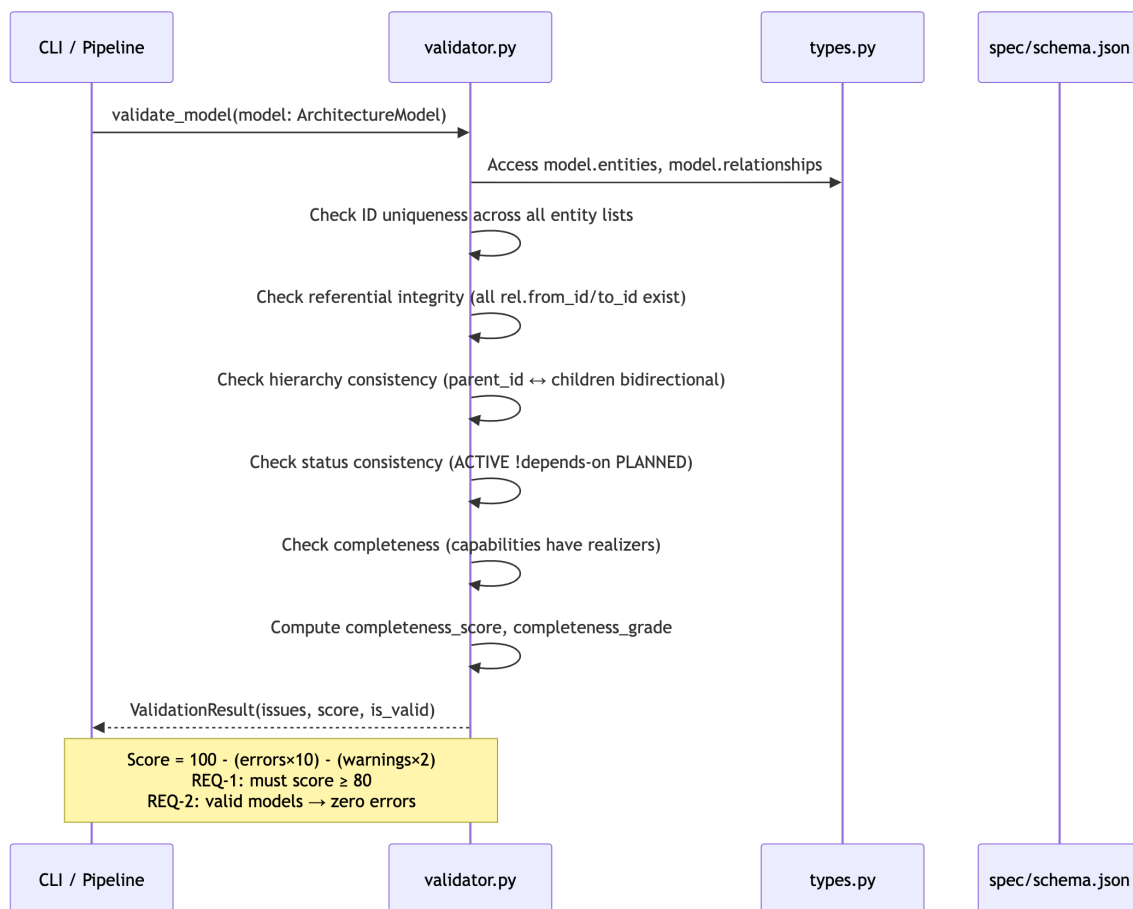


Figure 4: Diagram

The `ValidationResult` dataclass exposes `score`, `is_valid`, `error_count`, `warning_count`, and `completeness_grade`. The scoring formula ($100 - 10 \times \text{errors} - 2 \times \text{warnings}$) makes the trade-off explicit: errors are 5× more costly than warnings, reflecting that errors indicate broken invariants while warnings indicate missing-but-not-broken information.

5.2 Model Slicing Flow

Intent: Deliver the minimal, self-consistent subset of a model that an LLM needs to reason about a single source block. Every unnecessary entity wastes tokens and dilutes attention.

Trade-off: The slicer first checks for a pre-decomposed sub-model file (`load_block_model`). This is an optimization—pre-split files are faster and guarantee token budget compliance. The fallback dynamic slicing is more flexible but requires token-budget enforcement at output time.

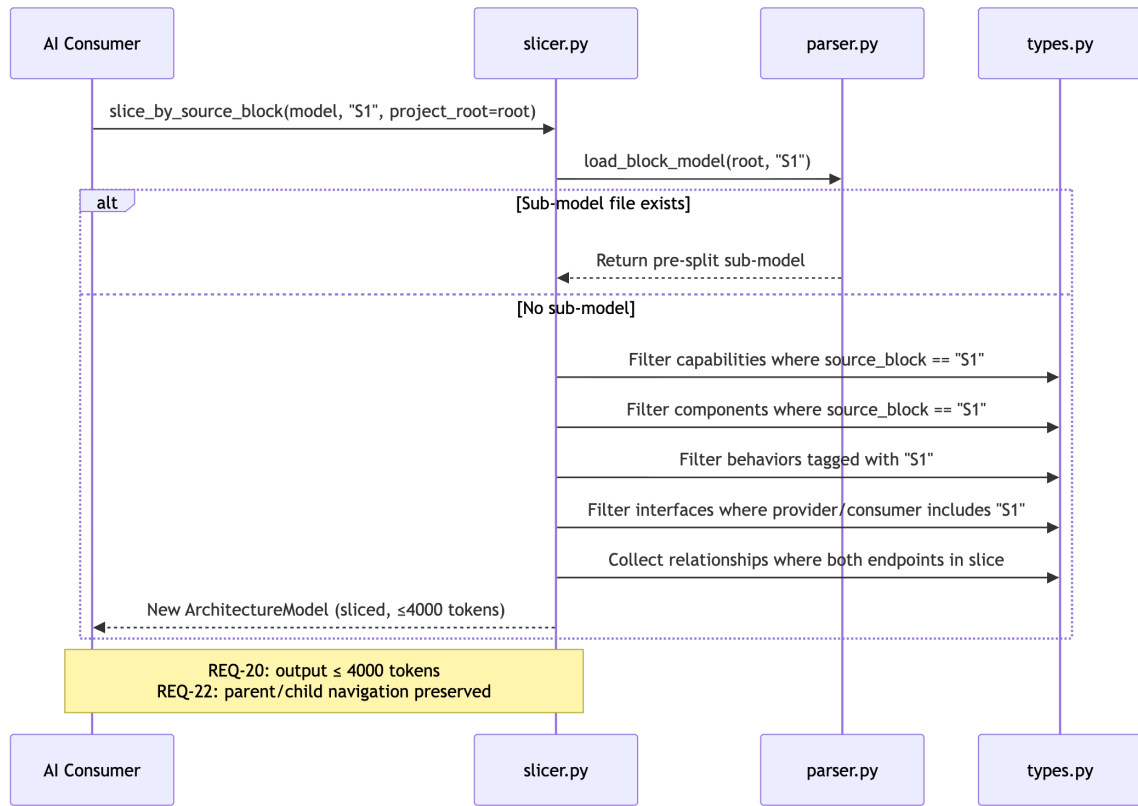


Figure 5: Diagram

5.3 Regen Readiness Assessment Flow

Intent: Before spending LLM tokens on code regeneration, predict the likelihood of success. A model with 40% `body_hint` coverage will produce code that needs heavy manual editing—better to enrich first.

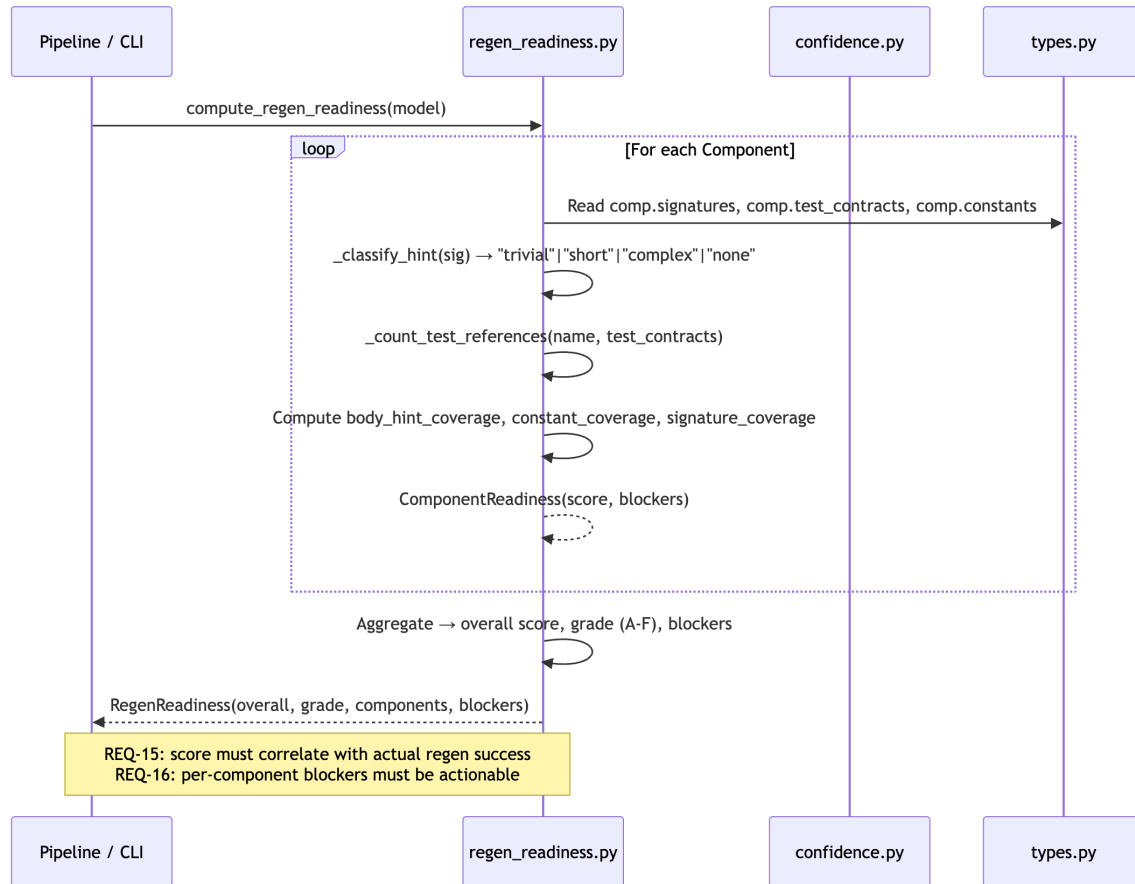


Figure 6: Diagram

6. Requirements Satisfaction

REQ	Text	Satisfied By	Rationale & Consequences of Violation
REQ-1	Model must score 80 on validation	COMP-1.2: <code>ValidationResult.score</code>	Why 80? At score < 80, the model has 2 errors or 10 warnings. Two structural errors (e.g., dangling references) reliably cause downstream artifact generation to produce incorrect output. The threshold is set at the empirical point where pipeline consumers can still function. Violation: Pipeline stages consume a broken model → generated docs reference nonexistent entities → user trust erodes.
REQ-2	Valid models produce zero validation errors	COMP-1.2: <code>ValidationResult.is_valid</code>	Why zero? Errors represent invariant violations (broken references, duplicate IDs). Unlike warnings, there is no “acceptable” number of broken invariants—one dangling reference can cascade. Violation: A “valid” model with errors becomes a landmine for any consumer that indexes by ID.

REQ	Text	Satisfied By	Rationale & Consequences of Violation
REQ-3	parent_id/children bidirectionally consistent	COMP-1.2: hierarchy validator	Why bidirectional? If component A lists B as a child but B's parent_id is null, tree traversal breaks depending on direction. Navigation becomes inconsistent—upward traversal and downward traversal disagree. Violation: slice_by_source_block misses child components; decomposer creates orphaned sub-models.
REQ-15	Regen score correlates with actual success rate	COMP-1.5: regen_readiness.py	Why correlation, not just threshold? A score that doesn't predict outcomes is worse than no score—it creates false confidence. The scoring weights (body_hint_coverage, test_contract_count, constant_coverage) were chosen to match the information an LLM actually needs to regenerate code. Violation: Teams attempt regeneration on low-readiness models, waste tokens, and lose trust in the tool.

REQ	Text	Satisfied By	Rationale & Consequences of Violation
REQ-16	Per-component readiness with actionable blockers	COMP-1.5: <code>ComponentReadiness.blpercomponent</code>	Why percomponent? An overall score of 60% is useless without knowing <i>which</i> components drag it down. Blockers like “missing body_hint on 12 functions” tell the user exactly what to enrich. Violation: Users see a low score but don’t know what to fix → they give up or fix the wrong things.
REQ-20	Slice output 4000 tokens	COMP-1.4: <code>slicer.py</code>	Why 4000? This leaves room within typical 8K–16K context windows for the system prompt, user query, and response. A slice consuming the entire window leaves no room for reasoning. Value function: Smaller is better down to ~1000 tokens; below that, the slice likely dropped essential context. Sweet spot is 2000–3500 tokens. Violation: Oversized slices cause LLM truncation or degraded reasoning quality.

REQ	Text	Satisfied By	Rationale & Consequences of Violation
REQ-22	Parent/child component navigation	COMP-1.1: <code>Component.parent_id</code> , <code>Component.children</code>	Why? Architecture is hierarchical—a Core component contains Type System , Validation , etc. Without navigation, consumers must reconstruct hierarchy from relationships, which is error-prone and expensive. Violation: Slicer cannot determine component containment; decomposer cannot identify sub-model boundaries.
REQ-27	YAML round-trip fidelity	COMP-1.3: <code>parser.py</code>	Why? Models are version-controlled. If load→save changes ordering or drops fields, every save creates a noisy diff that obscures real changes. Violation: Git diffs become unreadable; merge conflicts multiply; users stop trusting the tool to preserve their work.
REQ-28	Schema backward compatibility	COMP-1.3: <code>parser.py</code>	Why? Models are long-lived artifacts. A schema upgrade that can't read old models forces migration effort and breaks CI pipelines. Violation: Users on older model versions are locked out after tool upgrade; adoption stalls.

7. Trade-offs & Design Decisions

7.1 Validation Scoring: Deductive vs. Additive

Chosen: Deductive scoring ($100 - 10 \cdot \text{errors} - 2 \cdot \text{warnings}$). **Alternative:** Additive scoring (sum points for things present). **Why:** Deductive scoring means a model starts “perfect” and loses points for defects. This matches the mental model of validation—you’re looking for what’s *wrong*. Additive scoring would reward complexity (more entities = higher score), which is perverse. **If constraints changed:** If models were generated (not human-authored), additive scoring for completeness might make more sense since the baseline wouldn’t be “presumably correct.”

7.2 Slicer: Pre-split Files vs. Dynamic Slicing

Chosen: Try pre-split sub-model files first, fall back to dynamic slicing. **Why:** Pre-split files are $O(1)$ disk reads; dynamic slicing is $O(n)$ entity traversal. For large models (500+ entities), the difference matters. Pre-split also guarantees token budget compliance at write time rather than requiring runtime enforcement. **Trade-off cost:** Pre-split files can become stale if the master model is edited without re-decomposing. The `differ.py` capability partially mitigates this.

7.3 Quality Metrics Separation from Validation

Chosen: Confidence scoring and regen readiness are in COMP-1.5, not COMP-1.2. **Why:** Validation answers “is the model structurally correct?” (binary per-rule). Quality metrics answer “how good is this model for a specific purpose?” (continuous score). Mixing them would conflate structural validity with fitness-for-purpose. A model can be perfectly valid (zero errors) but have terrible regen readiness (no `body_hint` fields).

7.4 Clustering in Core vs. Orchestration

Chosen: `cluster.py`, `source_block_assign.py`, `test_affinity.py` live in Core (COMP-1.4). **Why:** These are pure graph algorithms over model/manifest data. They have no LLM dependencies, no pipeline state, no side effects. Placing them in Core makes them testable in isolation and reusable by both the pipeline and CLI.

8. Measures of Effectiveness

Capability	Minimum (pass/fail)	Good	Optimal	How to Measure
CAP-1: Validate	Zero false negatives on referential integrity	Catches status inconsistencies, orphans	Completeness grading accurately predicts downstream artifact quality	Run validator on known-good and known-bad models; measure precision/recall

Capability	Minimum (pass/fail)	Good	Optimal	How to Measure
CAP-7: Slice	Output contains all entities for the requested block	Output 4000 tokens with zero dangling references	Slice contains <i>exactly</i> the minimal set needed—nothing extra	Token count of serialized slice; count dangling refs in sliced model
CAP-8: Diff	Detects all added/removed entities	Detects field-level modifications	Diff output is directly actionable (maps to specific artifact staleness)	Diff two known models with controlled changes; verify completeness
CAP-11: Regen Readiness	Score exists per component	Score correlates with regen success ($r > 0.6$)	Blockers directly map to enrichment actions that improve score	Correlate scores with actual LLM regeneration outcomes across repos
CAP-13: Drift Detection	File coverage percentage computed	Relationship accuracy identifies false edges	Overall score usable as CI gate (threshold predicts model staleness)	Compare coverage scores pre/post code changes; validate against manual review

9. Failure Modes

Component	Failure Mode	Impact	Degradation
COMP-1.1 Type System	Dataclass field added without schema update	Silent data loss on round-trip	Hard fail — all consumers see inconsistent data
COMP-1.2 Validation	Validator misses a broken reference	Downstream generators produce artifacts referencing nonexistent entities	Silent corruption — no immediate error, delayed failure
COMP-1.3 Parser	YAML ordering changes on save	Noisy git diffs, merge conflicts	Graceful degradation — functionally correct but operationally painful
COMP-1.3 Parser	Old schema version not handled	<code>KeyError</code> or <code>TypeError</code> on load	Hard fail — user cannot load their model after tool upgrade

Component	Failure Mode	Impact	Degradation
COMP-1.4 Slicer	Slice exceeds token budget	LLM context overflow, truncated reasoning	Graceful degradation — output is correct but too large
COMP-1.4 Coverage	Manifest format changes	<code>KeyError</code> in <code>_check_component_coverage</code>	Hard fail — coverage analysis crashes
COMP-1.5 Regen Readiness	Score doesn't correlate with outcomes	Teams waste tokens on doomed regenerations	Silent failure — tool provides false confidence
COMP-1.5 Corrections	Correction applied twice	Entity renamed twice or relationship duplicated	Guarded — applied flag prevents re-application

Logical Architecture — Core Subsystem

1. Intent

The Core subsystem exists to provide a **single, authoritative in-memory representation** of an architecture model and all operations that can be performed on it without external side effects. It is the gravity center of the system: every other subsystem (Pipeline, CLI, Orchestration, Export) depends on Core but Core depends on nothing outside itself.

The design philosophy is **model-as-value**: the architecture model is a typed, immutable data structure that functions transform and return. This makes operations composable, testable, and safe for concurrent use by AI agents.

2. Component Structure

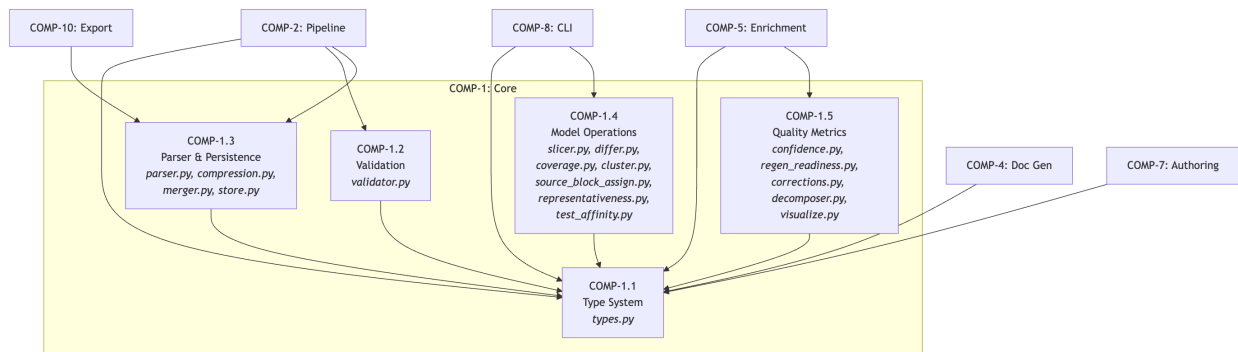


Figure 7: Diagram

COMP-1.1 — Type System

Intent: Establish a single source of truth for every entity shape in the model. By centralizing `ArchitectureModel`, `Component`, `Capability`, `Behavior`, `Interface`, `Requirement`, `Actor`, `Constraint`, `Relationship`, and all enums (`Status`, `RelationType`, `Layer`, `ComponentKind`) in `types.py`, we guarantee that every subsystem speaks the same structural language. No subsystem invents its own component representation.

Files: `src/architecture_model/core/types.py`

COMP-1.2 — Validation

Intent: Enforce model invariants that cannot be expressed by the type system alone—referential integrity, hierarchy consistency, semantic completeness, and domain rules. Validation is separated from parsing because a model can be syntactically valid YAML and structurally valid per the type system yet still be architecturally broken (dangling references, orphaned entities, unreachable states).

Files: `src/architecture_model/core/validator.py`, `src/architecture_model/spec/__init__.py`

COMP-1.3 — Parser & Persistence

Intent: Own the serialization boundary. Models live as YAML on disk and as `ArchitectureModel` in memory; this component guarantees lossless translation between the two. Round-trip fidelity (REQ-27) and backward compatibility (REQ-28) live here because they are serialization concerns, not validation concerns.

Files: `parser.py`, `compression.py`, `merger.py`, `persistence/store.py`

COMP-1.4 — Model Operations

Intent: Provide pure-function transformations over `ArchitectureModel` that answer questions without mutating state. Slicing, diffing, coverage analysis, and clustering are all read-only projections. Grouping them together reflects their shared contract: `ArchitectureModel` $\rightarrow$ `derived value`.

Files: `slicer.py`, `differ.py`, `coverage.py`, `cluster.py`, `source_block_assign.py`, `source_block_quality.py`, `representativeness.py`, `test_affinity.py`

COMP-1.5 — Quality Metrics

Intent: Quantify model quality along dimensions that matter for code regeneration. Confidence scoring (`compute_component_confidence`, `compute_behavior_confidence`) and regen readiness answer “can we regenerate from this model?” — a question orthogonal to “is this model valid?”

Files: `confidence.py`, `regen_readiness.py`, `corrections.py`, `decomposer.py`, `visualize.py`

3. Layer Allocation

All Core components are allocated to the **foundation (domain) layer**.

Why domain, not infrastructure: Core defines the problem-space concepts (capabilities, components, behaviors). It has zero I/O dependencies beyond reading YAML via the standard library. Infrastructure concerns (file watchers, HTTP, databases) belong elsewhere.

Why domain, not application: Application-layer components orchestrate user-facing workflows (CLI commands, pipeline stages). Core provides the building blocks those workflows compose. Placing Core in domain ensures it has no upward dependencies on orchestration or presentation.

4. Dependency Graph

Internal Dependencies

All sub-components depend on **COMP-1.1 (Type System)** and nothing else within Core:

From	To	Rationale
COMP-1.2 Validation	COMP-1.1	Validates <code>ArchitectureModel</code> instances, reads <code>RelationType</code> , <code>Status</code>

From	To	Rationale
COMP-1.3 Parser	COMP-1.1	Deserializes YAML into typed dataclasses (<code>ArchitectureModel</code> , <code>Component</code> , etc.)
COMP-1.4 Operations	COMP-1.1	Operates on <code>ArchitectureModel</code> , <code>Entities</code> , <code>Relationship</code>
COMP-1.5 Quality	COMP-1.1	Scores <code>Component</code> , <code>Behavior</code> , <code>Capability</code> instances

COMP-1.2 through COMP-1.5 are **peers** — no dependency between them. This is deliberate: validation should not require quality metrics, and slicing should not require validation.

External Dependencies (inbound)

Core has **zero outbound dependencies** on any other subsystem.

5. Interface Specification

Interface	Exposed By	Contract
IF-auto-COMP-1 Core API	COMP-1	Top-level re-exports from <code>core/__init__.py</code> . Entry point for external subsystems.
IF-auto-COMP-1.1 Type System API	COMP-1.1	Exports all dataclasses and enums. Contract: constructing any dataclass with valid field types produces a valid entity. Enums are <code>str</code> subclasses for YAML round-trip.
IF-auto-COMP-1.2 Validation API	COMP-1.2	<code>validate(model: ArchitectureModel) → ValidationResult</code> . Contract: returns <code>ValidationResult</code> with <code>error_count == 0</code> for valid models (REQ-2), <code>completeness_score >= 80</code> threshold (REQ-1).
IF-auto-COMP-1.3 Parser & Persistence API	COMP-1.3	<code>load(path) → ArchitectureModel</code> , <code>dump(model, path)</code> . Contract: <code>load(dump(m)) == m</code> (REQ-27). Accepts any <code>schema_version</code> current (REQ-28).

Interface	Exposed By	Contract
IF-auto-COMP-1.4 Model Operations API	COMP-1.4	<code>slice_by_source_block(model, block) → ArchitectureModel, diff_models(old, new) → ModelDiff</code> , plus <code>coverage</code> , <code>cluster</code> , <code>representativeness</code> functions. Contract: slice output fits within 4000 tokens (REQ-20).
IF-auto-COMP-1.5 Quality Metrics API	COMP-1.5	<code>compute_component_confidence(comp) → float</code> , <code>compute_behavior_confidence(beh) → float</code> , regen readiness scoring. Contract: scores are 0.0–1.0, per-component with actionable blockers (REQ-16).

6. Key Data Types

7. Design Decisions & Rationale

Decision	Choice	Rationale
Immutable dataclasses	<code>@dataclass</code> with no mutation methods	Enables safe sharing across pipeline stages and AI agent contexts. Transformations return new instances.
Single <code>ArchitectureModel</code> root	All entities + relationships in one object	Simplifies serialization, validation, and slicing — every operation takes and returns one type.
Relationships as flat list	<code>list[Relationship]</code> rather than edges on entities	Allows cross-cutting queries (all depends-on , all realizes) without traversing entity trees. Mirrors YAML structure for round-trip fidelity.
Entities container	Entities grouped by type in a container dataclass	Provides O(1) access by entity type while keeping the model serialization-friendly. Avoids a single heterogeneous list that would require type-switching everywhere.

Decision	Choice	Rationale
Score-based validation	<code>completeness_score: float</code> + letter grade rather than pass/fail	Models exist on a quality continuum. A score lets pipeline stages make nuanced decisions (e.g., warn at 70, block at 50) rather than binary gate (REQ-1).
str enum subclassing	<code>class Status(str, Enum)</code>	Enum values serialize directly to YAML strings without custom representers, preserving round-trip fidelity.

8. Requirements Traceability

Requirement	Satisfied By	Rationale
REQ-1 Score 80	COMP-1.2	Validator computes <code>completeness_score</code> and flags models below threshold
REQ-2 Zero errors on valid models	COMP-1.2	<code>ValidationResult.error_count == 0</code> is the invariant for valid models
REQ-3 Hierarchy consistency	COMP-1.2	Validator checks <code>parent_id/children</code> bidirectionality
REQ-15 Regen score accuracy	COMP-1.5	<code>confidence.py</code> weights fields by regeneration impact
REQ-16 Per-component readiness	COMP-1.5	<code>compute_component_confidence</code> returns per-entity score with field-level gaps as blockers
REQ-20 Token-efficient output	COMP-1.4	<code>slice_by_source_block</code> constrains output to 4000-token budget
REQ-22 Hierarchical navigation	COMP-1.1	<code>Component.parent_id</code> and <code>Relationship(type=contains)</code> enable tree traversal
REQ-27 YAML round-trip fidelity	COMP-1.3	Parser preserves key ordering and comments; <code>load(dump(m)) == m</code>
REQ-28 Schema backward compat	COMP-1.3	Parser migrates older <code>schema_version</code> values on load

9. Failure Modes

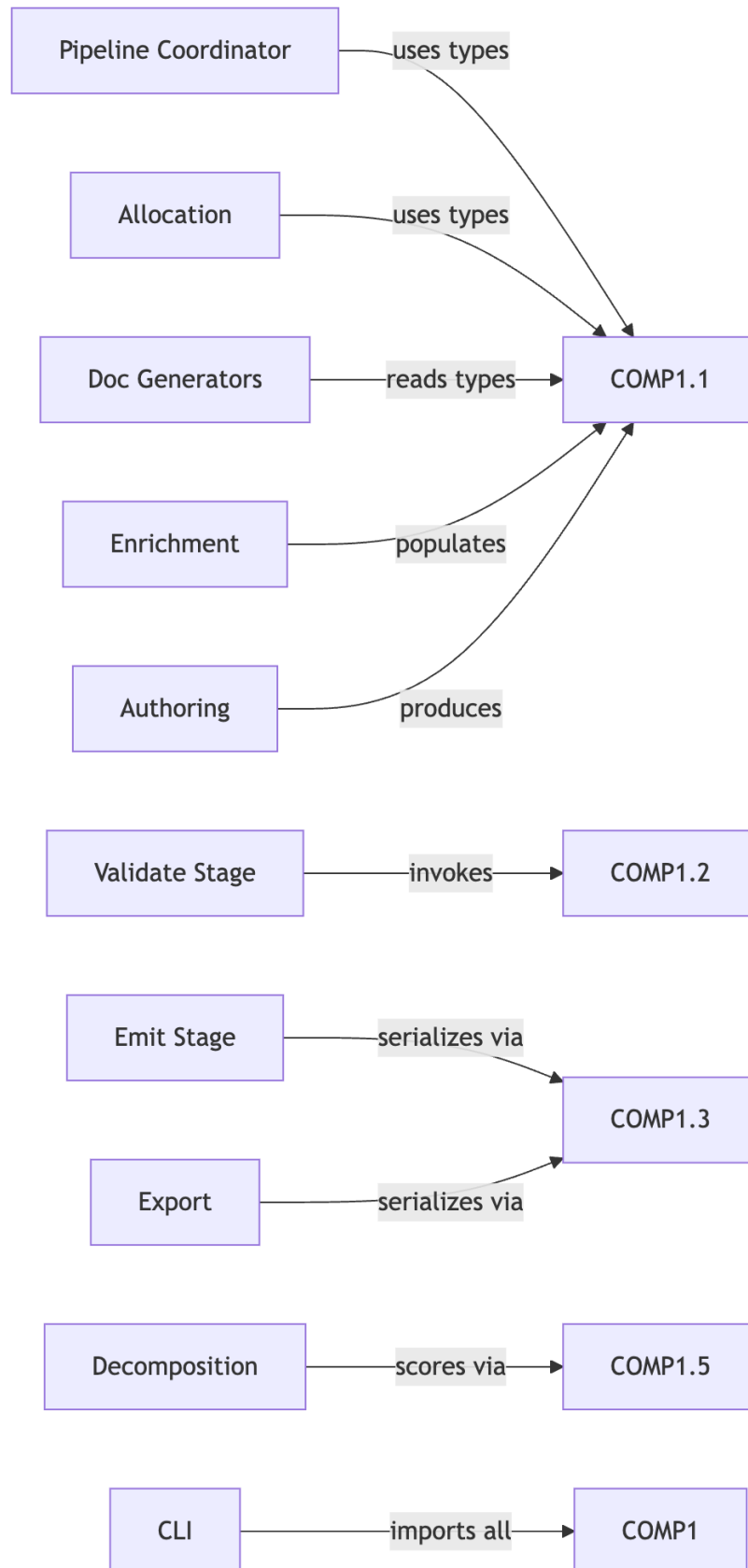


Figure 8.1 Diagram

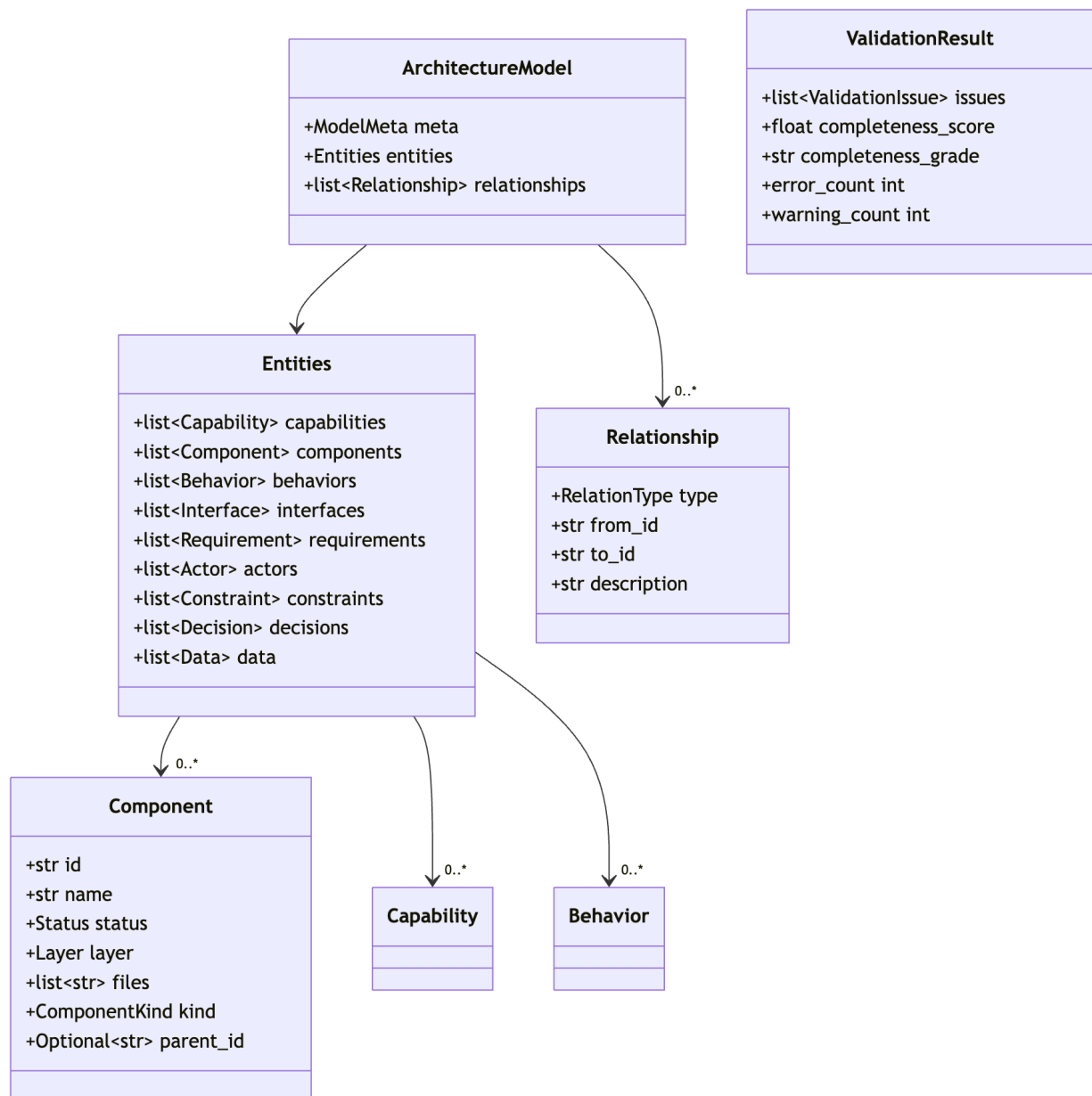


Figure 9: Diagram

Failure	Component	Behavior	Recovery
Validation fails (score < 80)	COMP-1.2	Returns <code>ValidationResult</code> with <code>issues</code> populated; does not raise. Caller decides whether to abort.	Fix model per <code>ValidationIssue.message</code> guidance.
Invalid YAML syntax	COMP-1.3	<code>yaml.YAMLError</code> propagates. Parser does not silently return partial models.	User fixes YAML syntax. Parser provides line/column from PyYAML.
Schema validation failure	COMP-1.3	<code>jsonschema.ValidationError</code> raised when <code>HAS_JSONSCHEMA</code> is true; skipped gracefully when <code>jsonschema</code> is not installed.	Error from model to match <code>spec/schema.json</code> .
Unknown F-block in slicer	COMP-1.4	<code>slice_by_source_block</code> returns an empty <code>ArchitectureModel</code> (zero entities, zero relationships). No exception.	Caller checks <code>len(result.entities.components) == 0</code> and reports to user.
Diff with incompatible schema versions	COMP-1.4	<code>diff_models</code> compares entity IDs; missing entity types in older model appear as additions.	Normalize both models to current schema via parser before diffing.
Confidence score on empty entity	COMP-1.5	Returns 0.0. All field checks are <code>if</code> guards; no exceptions on missing fields.	Score of 0.0 surfaces as a blocker in regen readiness report.

Core Subsystem — Use Cases

Why This Document Exists

The Core subsystem is the foundation that every other subsystem depends on. These use cases define *how* external actors (humans, pipelines, CLI) interact with Core’s capabilities, what “good” looks like beyond pass/fail, and what breaks when things go wrong.

UC-1: Validate Architecture Model

Capability: CAP-1 — Validate Architecture Models

Actor: Developer or Pipeline Engine

Intent: Confirm a model is structurally sound and semantically complete *before* using it for code generation, documentation, or AI context — catching errors early prevents costly downstream failures.

Preconditions: - An `ArchitectureModel` instance exists (parsed from YAML or constructed in memory) - Model has `meta`, `entities`, and `relationships` populated

Main Flow: 1. Actor invokes the validator (COMP-1.2, `validator.py`) 2. Validator checks ID uniqueness across all entity types 3. Validator checks referential integrity — every `Relationship.from_id` and `Relationship.to_id` resolves to an existing entity 4. Validator checks hierarchy consistency (REQ-3) — `parent_id/children` are bidirectionally consistent 5. Validator checks status consistency — no `ACTIVE` entity depending on a `PLANNED` entity 6. Validator checks completeness — capabilities have realizing behaviors 7. Validator checks meta completeness — `project` and `schema_version` set 8. A `ValidationResult` is returned with `issues`, `completeness_score`, `completeness_grade` 9. `ValidationResult.score` computed: 100 minus 10 per `Severity.ERROR`, 2 per `Severity.WARNING`

Postconditions: - `ValidationResult.is_valid` is `True` when `error_count == 0` (REQ-2) - `ValidationResult.score >= 80` for models considered acceptable (REQ-1) - `completeness_gaps` lists specific missing elements

Error Handling: - Invalid models degrade gracefully: validator always returns a result, never throws. Issues are collected as `ValidationIssue` objects with `severity`, `code`, `entity_id`, and `context`. - A model with score < 80 is flagged but not rejected — downstream consumers decide whether to proceed.

Quality Attributes: - Validation must be fast enough to run on every pipeline iteration (sub-second for models < 500 entities) - Deterministic: same model always produces identical `ValidationResult`

Measures of Effectiveness: - **Zero false negatives:** every structural inconsistency produces an `ERROR-severity` issue - **Low false positive rate:** warnings should be actionable, not noise. Metric: % of warnings that lead to actual model fixes - **Score granularity:** the 10-per-error / 2-per-warning penalty curve should meaningfully differentiate “almost valid” from “deeply broken” models

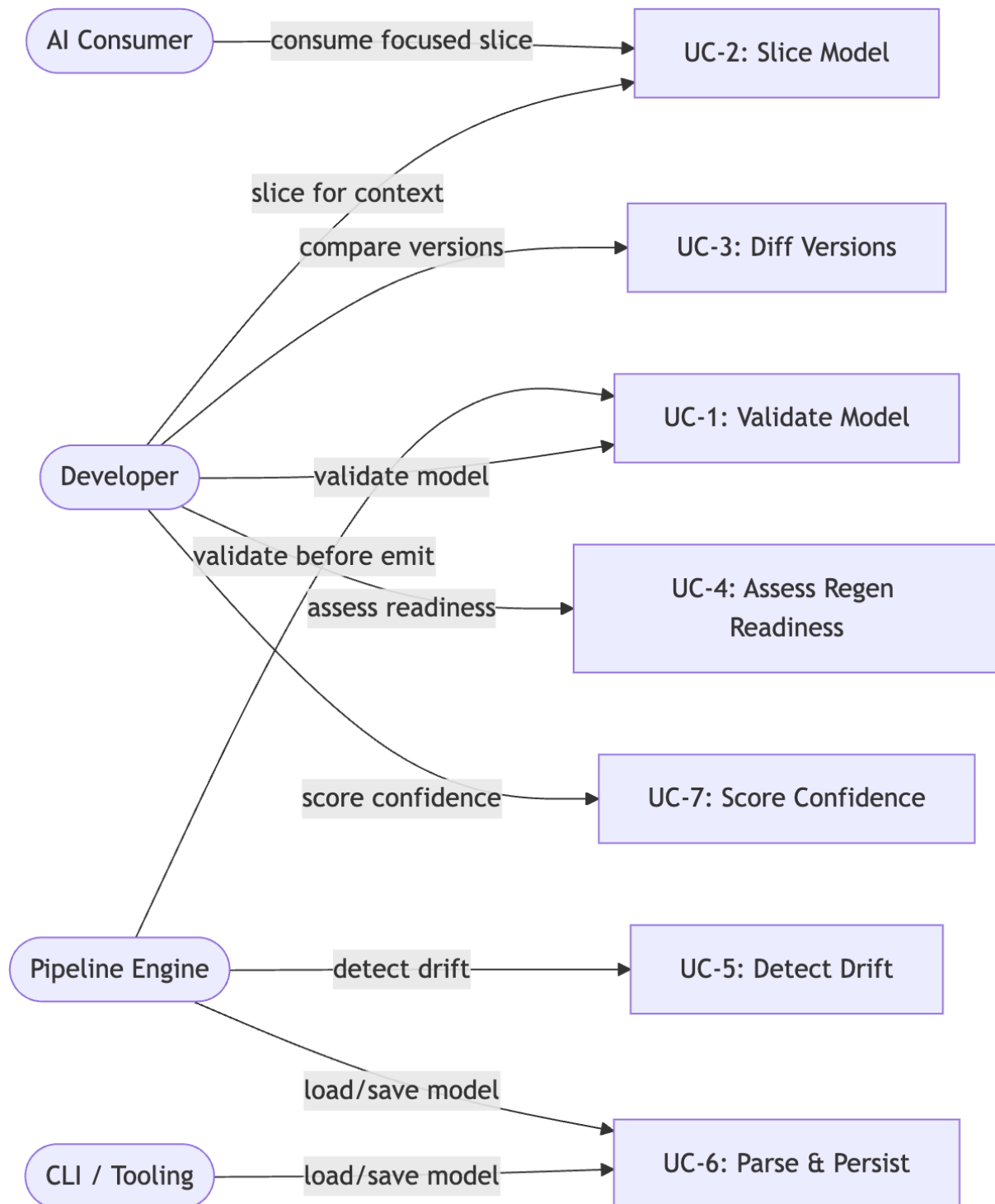


Figure 10: Diagram

UC-2: Slice Model for Focused Context

Capability: CAP-7 — Slice and Query Models

Actor: AI Consumer or Developer

Intent: Extract a minimal, token-efficient subset of the model so an AI agent can reason about one subsystem without exceeding context limits (REQ-20: 4000 tokens).

Preconditions: - Full `ArchitectureModel` loaded - Actor knows the filter criteria (source block ID, layer, status, or artifact path)

Main Flow: 1. Actor calls `slice_by_source_block(model, source_block, project_root=...)` from `slicer.py` 2. Slicer first attempts to load a pre-split sub-model via `load_block_model()` (fast path) 3. If no sub-model exists, slicer filters in-memory: - Collects capabilities, behaviors, components, interfaces tagged with the source block - Collects data entities owned by matching components - Filters relationships to only those between retained entity IDs 4. Returns a new `ArchitectureModel` containing only the slice 5. Alternative: `slice_by_layer`, `slice_by_status` for other filter dimensions

Postconditions: - Returned model contains only entities relevant to the filter - Serialized output fits within 4000 tokens (REQ-20) - Hierarchical navigation preserved — `parent_id/children` intact within slice (REQ-22)

Error Handling: - Unknown source block: returns empty model (no entities, no relationships) — does not throw - If pre-split sub-model file is corrupted, falls back to in-memory slicing

Quality Attributes: - Output token count is the primary optimization target, not just correctness

Measures of Effectiveness: - **Token utilization:** ratio of useful architectural content to total tokens. A slice that uses 3800/4000 tokens with high-value content is better than one using 1000 tokens that omits key relationships - **Completeness within scope:** % of relationships between retained entities that are included (should be 100%) - **Zero dangling references:** no `from_id/to_id` in relationships that point to entities outside the slice

UC-3: Diff Model Versions

Capability: CAP-8 — Diff Model Versions

Actor: Developer

Intent: Understand *what changed* between two model versions to assess architectural evolution, detect unintended drift, and determine which artifacts need regeneration.

Preconditions: - Two `ArchitectureModel` instances (old and new) loaded

Main Flow: 1. Actor calls `diff_models(old_model, new_model)` from `differ.py` 2. Differ builds entity ID maps for both models across all entity types 3. For each entity type, computes added (in new, not in old), removed (in old, not in new), and modified (same ID, different fields) 4. Computes `RelationshipChange` entries for added/removed relationship tuples 5. Returns `ModelDiff` with `entity_changes` and `relationship_changes` 6. Actor calls `ModelDiff.format_report()` for human-readable output or inspects `summary()`

Postconditions: - `ModelDiff.has_changes` accurately reflects whether any differences exist - `added_count`, `removed_count`, `modified_count` are correct

Error Handling: - Models with different `schema_version`: diff proceeds but may flag schema-level changes - Empty models: returns `ModelDiff` with no changes (not an error)

Quality Attributes: - Must detect field-level modifications, not just presence/absence of entities

Measures of Effectiveness: - **Precision of “modified”:** changes flagged as modified should reflect semantically meaningful differences, not serialization noise (e.g., field ordering) - **Actionability:** every `EntityChange` should tell the user *what* to do — `details` field should name the changed fields

UC-4: Assess Regen Readiness

Capability: CAP-11 — Assess Regen Readiness

Actor: Developer or Pipeline Engine

Intent: Before attempting code regeneration, determine whether the model captures enough implementation detail (body hints, test contracts, constants, signatures) to produce correct code — avoiding wasted LLM calls on under-specified components.

Preconditions: - Enriched `ArchitectureModel` with `Component.signatures`, `Component.test_contracts`, `Component.constants` populated

Main Flow: 1. Actor invokes regen readiness scoring (COMP-1.5, `regen_readiness.py`) 2. For each `Component`, computes `ComponentReadiness`: - `body_hint_coverage`: fraction of `FunctionSignature` entries with non-empty `body_hint` - `body_hint_trivial_ratio`: fraction classified as “trivial” via `_classify_hint()` - `test_contract_count`: number of `TestContract` entries - `signature_coverage`, `constant_coverage` - Per-function `FunctionReadiness` with `_count_test_references()` (REQ-16) 3. Computes `RegenReadiness.overall` score (0-100) and grade (A-F) (REQ-15) 4. Populates `blockers` list with actionable items (e.g., “Component X has 0% `body_hint` coverage”)

Postconditions: - Every component has a per-component score with specific blockers (REQ-16) - Overall score correlates with actual regeneration success (REQ-15) - `recommendation` field provides next-step guidance

Error Handling: - Components with no signatures: score = 0, blocker = “no signatures extracted” - Graceful on empty models: returns `RegenReadiness(overall=0, grade="F")`

Quality Attributes: - Score must be a *leading indicator* of regeneration quality, not just a completeness checklist

Measures of Effectiveness: - **Correlation with regen success:** the primary MoE. Grade “A” models should produce >90% correct code; grade “D” should produce <50% - **Blocker actionability:** every blocker should map to a specific enrichment action the user can take - **Discrimination:** the score should meaningfully separate “ready” from “not ready” — a bimodal distribution around 50 is useless

UC-5: Detect Model Drift from Code

Capability: CAP-13 — Detect and Fix Model Drift

Actor: Pipeline Engine or Developer

Intent: Verify that the architecture model still accurately represents the actual codebase — models that drift from reality become actively harmful as AI context.

Preconditions: - `ArchitectureModel` loaded - Manifest (`reality-manifest.json`) available with current code facts - For representativeness: `list[ModuleInfo]` and `list[InterfaceEdge]` from manifest

Main Flow: 1. Actor calls `compute_representativeness(model, modules, edges)` from `representativeness.py` 2. Computes `file_coverage`: fraction of non-trivial manifest modules covered by model component file lists (using `_files_match()` for path normalization) 3. Computes `relationship_accuracy`: fraction of model `depends-on/uses` relationships verified by manifest import edges 4. Computes `boundary_coherence`: whether component file groupings align with manifest module boundaries 5. Computes `behavioral_coverage`: fraction of manifest-observed behaviors captured in model 6. Returns `RepresentativenessResult` with `overall` score and lists: `uncovered_files`, `unverified_relationships`, `low_coherence_components` 7. Alternatively, `coverage.py` provides `CoverageResult` with per-check scoring via `_check_component_coverage()`

Postconditions: - `uncovered_files` lists source files the model doesn't account for - `unverified_relationships` lists model relationships not backed by code evidence

Error Handling: - Missing manifest: cannot compute — should return a result with 0% scores and a clear message, not crash - Trivial modules (`__init__.py` re-exports, `__version__.py`) filtered via `_is_trivial()` to avoid false positives

Quality Attributes: - Must handle path normalization edge cases (leading `./`, `src/` prefix differences) via `_files_match()`

Measures of Effectiveness: - **Drift detection rate:** % of actual code changes since last model update that are surfaced as coverage gaps - **False positive rate:** % of flagged gaps that are actually intentional omissions (e.g., test utilities). Lower is better - **Overall score utility:** a 90%+ representativeness score should mean the model is safe to use as AI context without verification

UC-6: Parse, Persist, and Round-Trip Models

Capability: Implicit in COMP-1.3

Actor: CLI, Pipeline Engine, or Developer

Intent: Load a model from YAML, work with it as typed Python objects, and save it back *without losing data or changing ordering* — round-trip fidelity is essential because models are version-controlled and spurious diffs erode trust.

Preconditions: - `.architecture-model.yaml` file exists (for load) or `ArchitectureModel` instance exists (for save)

Main Flow: 1. Actor calls `load_model(path)` from `parser.py` — parses YAML, optionally validates against JSON schema (`SCHEMA_PATH`), constructs typed dataclasses (`ArchitectureModel`,

Component, Capability, etc.) 2. Actor manipulates model via typed API (all types from `types.py`) 3. Actor calls `save_model(model, path)` — serializes back to YAML 4. For multi-block projects: `load_block_model(project_root, block_id)` loads per-block sub-models 5. For project snapshots: `save_project(root, model, manifest)` via `persistence/store.py` persists model + manifest + metrics to `.architecture/` 6. Schema backward compatibility: parser handles older `schema_version` values (REQ-28)

Postconditions: - Load→save cycle produces byte-identical YAML (REQ-27: no data loss, no ordering change) - All enum values round-trip correctly via `_enum_value()` helper - Old schema versions load without error (REQ-28)

Error Handling: - Malformed YAML: parser raises with clear error message including line number - Missing optional fields: default to `None` or empty lists (dataclass defaults) - Schema validation failure (if `jsonschema` installed): collected as issues, not hard failures

Quality Attributes: - Round-trip fidelity is non-negotiable — any data loss is a critical bug

Measures of Effectiveness: - **Byte-identical round-trip rate:** % of load→save cycles that produce identical output. Target: 100% - **Schema migration success:** % of models from schema version N-1 that load correctly on version N - **Parse performance:** sub-200ms for models with <1000 entities

UC-7: Score Entity Confidence

Capability: Part of CAP-11 (quality assessment)

Actor: Developer or Pipeline Engine

Intent: Prioritize enrichment effort by identifying which entities have the *least* information — confidence scoring tells you where to invest time, not just whether the model is “done.”

Preconditions: - `ArchitectureModel` with entities populated (even partially)

Main Flow: 1. Actor calls `compute_component_confidence(comp)` from `confidence.py` for each Component 2. Scoring weights: `contract` (0.25), `signatures` with returns (0.20), `pattern` (0.15), `test_contracts` (0.15), `symbols` with members (0.10), `constants` (0.05), `responsibilities` (0.05), `files` (0.05), `interfaces` with both provides/requires (0.10) 3. Similarly: `compute_behavior_confidence()`, `compute_capability_confidence()`, `compute_interface_confidence()` 4. All return 0.0–1.0, capped via `min(score, 1.0)` 5. Actor sorts entities by confidence ascending to find enrichment priorities

Postconditions: - Every entity has a confidence score - Scores are deterministic and reproducible

Error Handling: - Empty entities (no fields populated): score = 0.0 — this is correct, not an error - Graceful on `None` fields: each check is guarded with `if comp.field`

Quality Attributes: - Weights should reflect actual regeneration value — `contract` at 0.25 reflects that behavioral contracts are the highest-value enrichment

Measures of Effectiveness: - **Weight calibration:** do the weights match empirical regeneration outcomes? A component with confidence 0.8 should regenerate better than one at 0.3 - **Discrimination:** scores should spread across 0.0–1.0, not cluster at 0.5. A tight distribution means the

weights aren't differentiating well - **Value function:** confidence improvement from 0.2→0.5 should yield more regeneration improvement than 0.7→1.0 (diminishing returns)

Artifact Traceability Map: architecture-model-standard / Core

1. Entity Inventory

Entity Type	Count	Feeds SE Documents
Components	6	Logical Architecture, Maintenance Manual, Operations Manual, Interface Specification, Component Specs
Capabilities	5	ConOps, Functional Analysis, Requirements Analysis
Behaviors	0	Use Cases, Functional Analysis, Verification & Validation, Behavior Flows
Interfaces	6	Interface Specification, Logical Architecture
Constraints	0	Requirements Analysis, Risk Assessment
Requirements	13	Requirements Analysis, Verification & Validation
Actors	0	ConOps, Use Cases
Layers	0	Logical Architecture

2. Artifact Dependency Graph

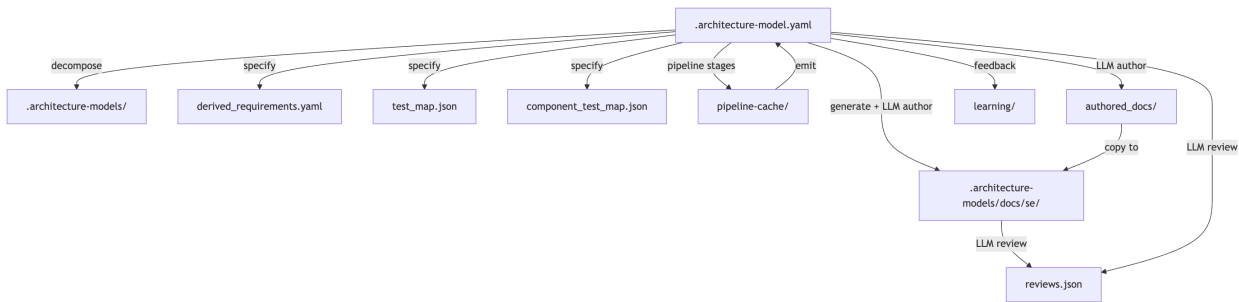


Figure 11: Diagram

3. Entity-to-Artifact Traceability Matrix

Artifact	Component	Capabilities	Behaviors	Interfaces	Constraints	Requirements	Actors	Layers
Behavior			—					
Flows								
Component	6							
Specs								
ConOps		5					—	
Functional		5	—					
Analysis								
Interface	6			6				
Specifi-								
cation								

Artifact	Component	Capabilities	Behaviors	Interfaces	Constraints	Requirements	Actors	Layers
Logical Architecture	6			6				—
Maintenance Manual	6							
Operations Manual	6							
Requirements Analysis		5			—	13		
Risk Assessment					—			
Use Cases			—				—	
Verification & Validation			—			13		

4. Relationship Distribution

Relationship Type	Count	Connects
satisfies	13	Component → Requirement
depends-on	11	Unknown → Component, Unknown → Unknown
exposes	6	Component → Interface
realizes	5	Component → Capability
contains	5	Component → Component

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