

Concept of Operations: Pipeline Subsystem

1. System Overview

Intent

The Pipeline subsystem exists to solve a fundamental problem: **extracting structured architecture models from arbitrary source code without manual intervention**. Codebases contain implicit architecture—component boundaries, dependency relationships, interface contracts—but this knowledge lives scattered across files, imports, naming conventions, and documentation. The Pipeline transforms this implicit knowledge into an explicit, validated YAML model (`.architecture-model.yaml`) plus supporting SE documentation.

The design philosophy is a **10-stage linear pipeline with DAG-based dependency resolution**, where each stage performs one conceptual transformation with typed inputs and outputs. This decomposition enables caching, independent re-runs, and graceful degradation—any stage can fail without losing upstream results.

The 10 Stages

#	Stage	Module	Transformation
1	observe	<code>observe.py</code>	Source files → raw Inventory (AST facts, imports, constants)
2	infer	<code>infer.py</code>	Inventory → InferResult (components, capabilities, behaviors)
3	allocate	<code>allocate.py</code>	InferResult → AllocationResult (files assigned to components)
4	relate	<code>relate.py</code>	AllocationResult → RelateResult (typed relationships: depends-on, realizes, contains)
5	specify	<code>specify.py</code>	RelateResult → SpecifyResult (interfaces, signatures)
6	contract	<code>contract.py</code>	SpecifyResult → ContractResult (test contracts, behavioral expectations)
7	validate	<code>validate.py</code>	ContractResult → ValidateResult (schema/consistency checks)
8	decompose	<code>decompose.py</code>	ValidateResult → DecomposeResult (system boundaries, sub-components)
9	synthesize	<code>synthesize.py</code>	DecomposeResult → SynthesizeResult (merged SoSModel / SystemModel)
10	emit	<code>emit.py</code>	SynthesizeResult → files on disk



Figure 1: Diagram

2. Stakeholders & Actors

Actor	Mechanism	Goal
Developer/Architect	CLI (<code>architecture-model pipeline</code>) or MCP tool (<code>architect_pipeline</code>)	Obtain accurate architecture model with minimal effort
PipelineCoordinator	<code>coordinator.py</code> — DAG resolution via <code>resolve_order()</code> , Kahn’s algorithm	Run minimum necessary stages to reach target, enforce determinism
PipelineCache	<code>cache.py</code>	Skip completed stages on re-runs, enable stage independence
LLM (optional)	copilot-relay enrichment via <code>EnrichmentRecord</code>	Improve naming, descriptions, and ambiguity resolution when available
CI/CD systems	Automated pipeline runs	Validate architecture model stays consistent with code changes

3. Operational Scenarios

Scenario 1: Full Extraction from Scratch

Trigger: Developer runs `architecture-model pipeline --repo ./my-project` on a repo with no cached state.

Flow: `PipelineCoordinator.resolve_order("emit")` returns all 10 stages in topological order. Each stage’s `run(ctx: PipelineContext)` executes sequentially. `ObserveStage` scans all `.py` files (excluding test dirs via `_is_excluded`), produces `Inventory` with `ModuleRecord`, `ImportEdge`, `ClassRecord`, etc. Each `StageResult` is stored in `PipelineContext` and cached. Final `EmitStage` writes `.architecture-model.yaml` and per-component docs via `_write_file()`.

Success measure: Complete model with validation score $>85/100$, all source files accounted for.

Scenario 2: Incremental Re-run from Cache

Trigger: Developer modifies 3 files and re-runs pipeline.

Flow: `PipelineCache` detects that stages observe through relate have valid cached `StageResults` (input hashes unchanged for unmodified files). Coordinator skips to the first invalidated stage, loads cached predecessors via `ctx.get("observe")` etc. Only stages from the invalidation point forward re-execute.

Rationale (REQ-7): Stage independence via caching is critical for developer iteration speed. Without it, every change forces a full 10-stage run.

Scenario 3: LLM-Enriched Extraction

Trigger: Pipeline runs with LLM access configured (copilot-relay available).

Flow: Stages that support enrichment (primarily infer, specify) issue `LLMCallRecord` requests for improved component naming, description generation, and ambiguity resolution. `SOURCE_WEIGHTS` in `protocol.py` assigns `llm_analysis` confidence of 0.6—lower than AST (1.0) or test (0.95) evidence. LLM suggestions become `Claim` values with `uncertain: True` when confidence is below threshold.

Key constraint (REQ-24): LLM inferences are flagged as `Uncertainty` objects, never silently accepted as ground truth.

Scenario 4: Scoped Sub-pipeline (Recursive Decomposition)

Trigger: `ctx.scope_files` is set to a subset of repository files (e.g., a single subsystem).

Flow: `ObserveStage.run()` respects `ctx.scope_files`, scanning only specified paths. Downstream stages operate on the reduced `Inventory`. `DecomposeStage` applies `_cluster_by_directory()` and `_decompose_component()` with `HIERARCHY_FILE_THRESHOLD = 8` to create `SubComponent` hierarchies. If a component has files across multiple directories with 2 files each, it gets decomposed.

Use case: Generating focused architecture docs for a single package without processing the entire monorepo.

4. System Context

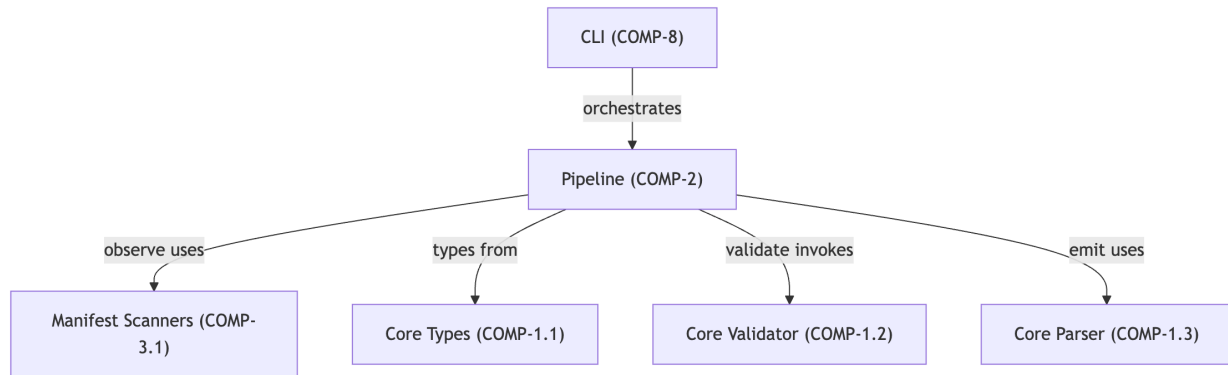


Figure 2: Diagram

Dependency	Why
Manifest Scanners (COMP-3.1)	<code>ObserveStage</code> delegates actual file scanning/AST parsing to shared scanner infrastructure rather than reimplementing it. Avoids duplication of Python AST walking logic.

Dependency	Why
Core Types (COMP-1.1)	<code>PipelineCoordinator</code> and allocation stages need canonical entity types (<code>Component</code> , <code>Capability</code> , <code>Relationship</code>) to build the model. Using shared types ensures pipeline output is compatible with the rest of the system.
Core Validator (COMP-1.2)	<code>ValidateStage</code> invokes the same validation logic used everywhere else. A pipeline-specific validator would drift from the canonical rules.
Core Parser (COMP-1.3)	<code>EmitStage</code> uses the parser’s YAML serialization to produce <code>.architecture-model.yaml</code> . Single serialization path prevents format inconsistencies.

5. Operational Constraints

Constraint	Requirement	Rationale
Performance	<60s for 200+ file repos without LLM (REQ-25)	Developer adoption requires near-interactive speed. LLM latency is excluded because it’s optional and externally bounded.
Determinism	Same input → same output (REQ-6)	Architecture models are version-controlled. Non-deterministic output creates phantom diffs that erode trust. <code>PipelineCoordinator</code> enforces deterministic stage ordering via topological sort.
Stage independence	Each stage independently runnable with cached predecessors (REQ-7)	Enables incremental runs and debugging of individual stages. <code>PipelineCache</code> serializes each <code>StageResult</code> .

Constraint	Requirement	Rationale
Graceful degradation	Partial results on failure, not crash (REQ-23)	A failed <code>contract</code> stage should not destroy a valid <code>relate</code> result. Coordinator catches stage failures and preserves completed upstream results.
Uncertainty surfacing	Ambiguous inferences flagged, not silently guessed (REQ-24)	<code>Claim.uncertain</code> and <code>Uncertainty</code> objects in <code>StageResult</code> make confidence explicit. Downstream consumers can filter by confidence threshold.
Test preservation	All existing tests pass after changes (REQ-17)	<code>ValidateStage</code> enforces this as a pipeline-internal check.

6. Data Flow

Stage	Input	Output	Key Types
observe	Source .py files	Inventory	ModuleRecord, ImportEdge, ClassRecord, FunctionRecord, ConstantRecord
infer	Inventory	InferResult	Claim[Component], Claim[Capability], Uncertainty
allocate	InferResult	AllocationResult	File-to-component assignments
relate	AllocationResult	RelateResult	Typed relationships (realizes, depends-on, contains)
specify	RelateResult	SpecifyResult	Interfaces, function signatures
contract	SpecifyResult	ContractResult	Test contracts, behavioral expectations
validate	ContractResult	ValidateResult	Validation score, diagnostics
decompose	ValidateResult	DecomposeResult	SystemBoundary, SubComponent
synthesize	DecomposeResult	SynthesizeResult	SystemModel, SoSModel
emit	SynthesizeResult	EmitResult	Written file paths, total bytes

Every stage returns `StageResult[T]` which wraps the typed output with `diagnostics: list[Diagnostic]`, `uncertainties: list[Uncertainty]`, and `quality: QualityMetrics`. Evidence provenance flows through `Claim` objects with confidence scores weighted by `SOURCE_WEIGHTS`.

7. Measures of Effectiveness

Metric	Target	Measurement Point	Trade-off
Validation score	>85/100	<code>ValidateResult</code>	Higher thresholds reject more models; 85 balances completeness vs. strictness
Entity coverage	>90% of entity types present	<code>InferResult</code> — capabilities, components, signatures, constants, symbols, files (REQ-4)	Aggressive inference increases coverage but may reduce precision
File coverage	100% of non-excluded files	<code>AllocationResult</code> — every file allocated to a component	Unallocated files indicate gaps in component detection
Relationship accuracy	>80%	<code>RelateResult</code> — realizes, depends-on, contains present (REQ-5)	Recall vs. precision: missing relationships are worse than spurious ones for architecture understanding
Boundary coherence	>50%	<code>DecomposeResult</code> — <code>SystemBoundary</code> clusters share more internal than external edges	Low coherence suggests arbitrary decomposition; threshold is modest because some repos have flat structure
Pipeline latency	<60s (200+ files, no LLM)	End-to-end wall clock	Caching amortizes cost over incremental runs
Uncertainty rate	Tracked, not minimized	<code>StageResult.uncertainties</code> count	Goal is surfacing, not suppressing. A pipeline that reports zero uncertainties on a complex repo is likely hiding information.

Functional Analysis: Pipeline Subsystem

1. Intent & Purpose

The Pipeline subsystem exists to **transform raw source code into a structured architecture model through a sequence of deterministic, composable stages**. Without it, architecture extraction would be a monolithic, opaque process impossible to debug, cache, resume, or incrementally improve.

The core philosophy: **separate observation from inference from synthesis**. Each stage has a single epistemic responsibility — observe facts, infer meaning, allocate structure, relate entities, specify interfaces, validate correctness, then emit artifacts. This separation means each stage’s output can be inspected, cached, corrected, and re-run independently.

2. Capability Inventory

CAP-3: Run Modular Extraction Pipeline

Intent: Enable automated architecture model extraction that is transparent, resumable, and correctable — not a black box.

Goal (optimal): Every stage produces high-confidence outputs with full provenance chains; the final model scores >90/100 on validation; total wall-clock time scales linearly with codebase size.

Sub-Capability	Intent	Optimal Delivery
Codebase Observation	Establish ground truth from AST parsing — zero inference, pure facts	100% parseable file coverage, all im-ports/classes/functions/routes captured
Semantic Inference	Derive capabilities, actors, behaviors from observed patterns	Every meaningful capability identified with evidence; ambiguities explicitly flagged
Structural Allocation	Map files → components with coherent boundaries	>95% file coverage, boundary coherence >80%, no component >12 files
Relationship Derivation	Discover realizes/depends-on/contains/exposes edges	All import-based dependencies captured; no false positives from transitive deps
Interface Specification	Extract REST/CLI/library interfaces from routes and exports	Every public API surface captured with methods listed
Contract Mapping	Link test files to components as behavioral contracts	High test-component coverage ratio
Structural Validation	Check model completeness and consistency	Score reflects actual model quality; actionable issue descriptions
System Decomposition	Detect system boundaries for large codebases	Meaningful splits based on coupling, not arbitrary thresholds

Sub-Capability	Intent	Optimal Delivery
Model Synthesis	Assemble per-system models into unified SoS model	Single coherent YAML with namespaced IDs, no collisions
Artifact Emission	Write final files to disk in canonical structure	All paths written, reproducible output
Pipeline Coordination	Orchestrate stage execution order, caching, error handling	Minimum stages executed; failures produce partial results

3. Functional Decomposition

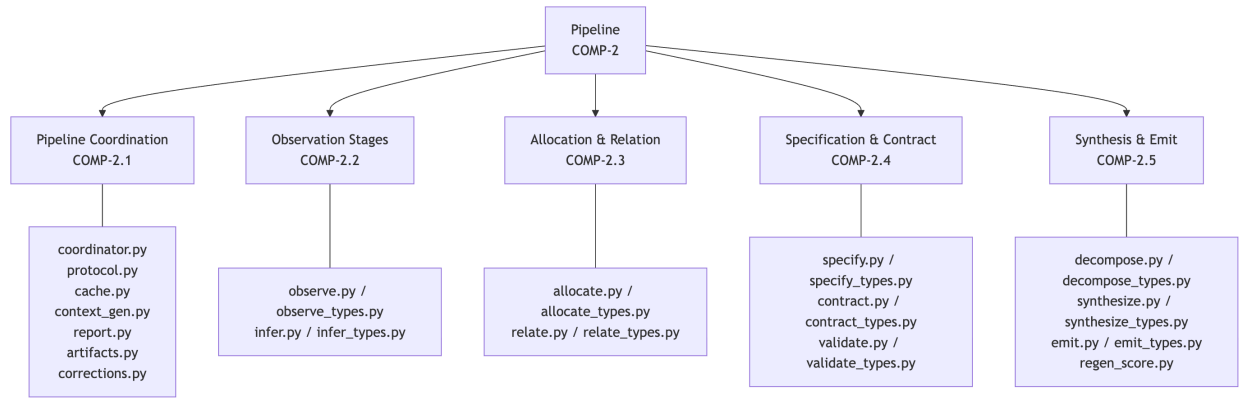


Figure 3: Diagram

Stage Dependency DAG

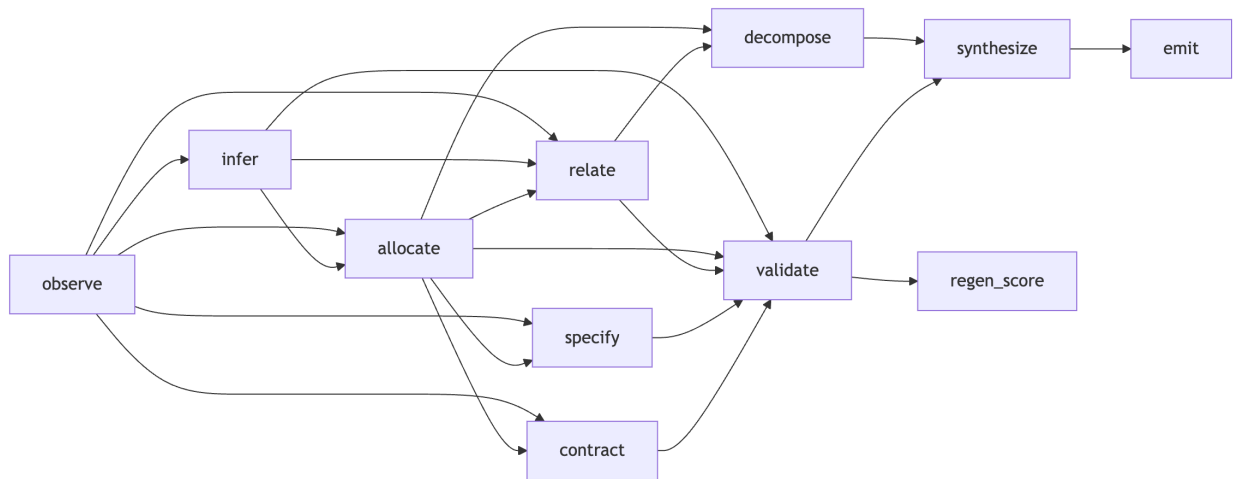


Figure 4: Diagram

4. Capability-Component Mapping

Component	Realizes	Rationale
COMP-2.1 (Coordination)	Stage orchestration, caching, graceful degradation	Separates “how to run stages” from “what stages do” — enables independent stage execution (REQ-7) and partial results on failure (REQ-23)
COMP-2.2 (Observation)	Codebase observation + semantic inference	ObserveStage produces Inventory (facts only); InferStage produces InferenceResult (capabilities/actors/behaviors). Split enforces the observation/inference boundary — observe never guesses
COMP-2.3 (Allocation & Relation)	File→component mapping + relationship derivation	AllocateStage seeds components from capabilities then assigns by import affinity. RelateStage derives typed edges. Both depend on observe+infer outputs
COMP-2.4 (Spec & Contract)	Interface specs + test contracts + validation	SpecifyStage extracts interfaces from routes/exports. ContractStage maps tests→components. ValidateStage scores the complete model. Grouped because all three refine/verify the structural model
COMP-2.5 (Synthesis & Emit)	Decomposition + assembly + disk output	DecomposeStage detects system boundaries. SynthesizeStage runs scoped sub-pipelines and merges. EmitStage writes canonical file structure. RegenScoreStage computes regeneration readiness

5. Behavioral Flows

5.1 Full Pipeline Execution (Primary Flow)

Intent: Transform a repository path into a complete architecture model on disk, executing only the minimum stages needed, with caching to avoid redundant work.

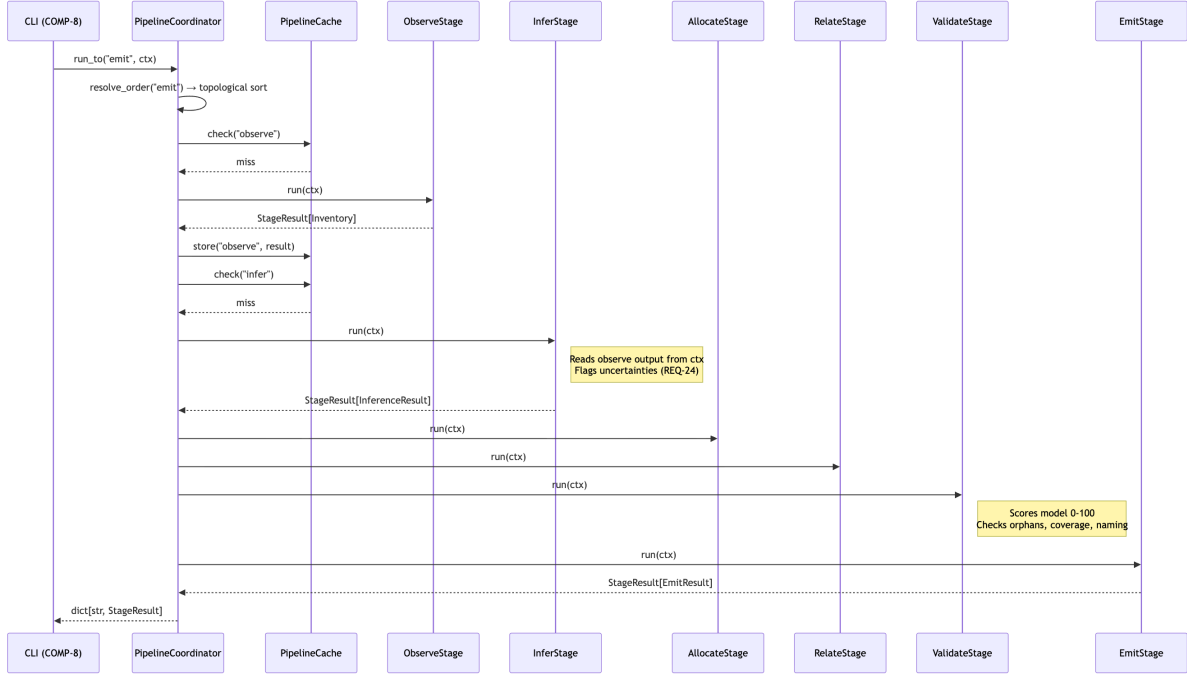


Figure 5: Diagram

5.2 Graceful Degradation on Stage Failure

Intent: Ensure that a failure in one stage (e.g., relate crashes) doesn't destroy all work done by prior stages. The system should produce the best partial model it can.

5.3 Cached Stage Resume (MCP Orchestration)

Intent: Enable the MCP orchestrator to run one stage per invocation across separate process lifetimes, resuming from disk-serialized `StageResult` objects.

6. Requirements Satisfaction

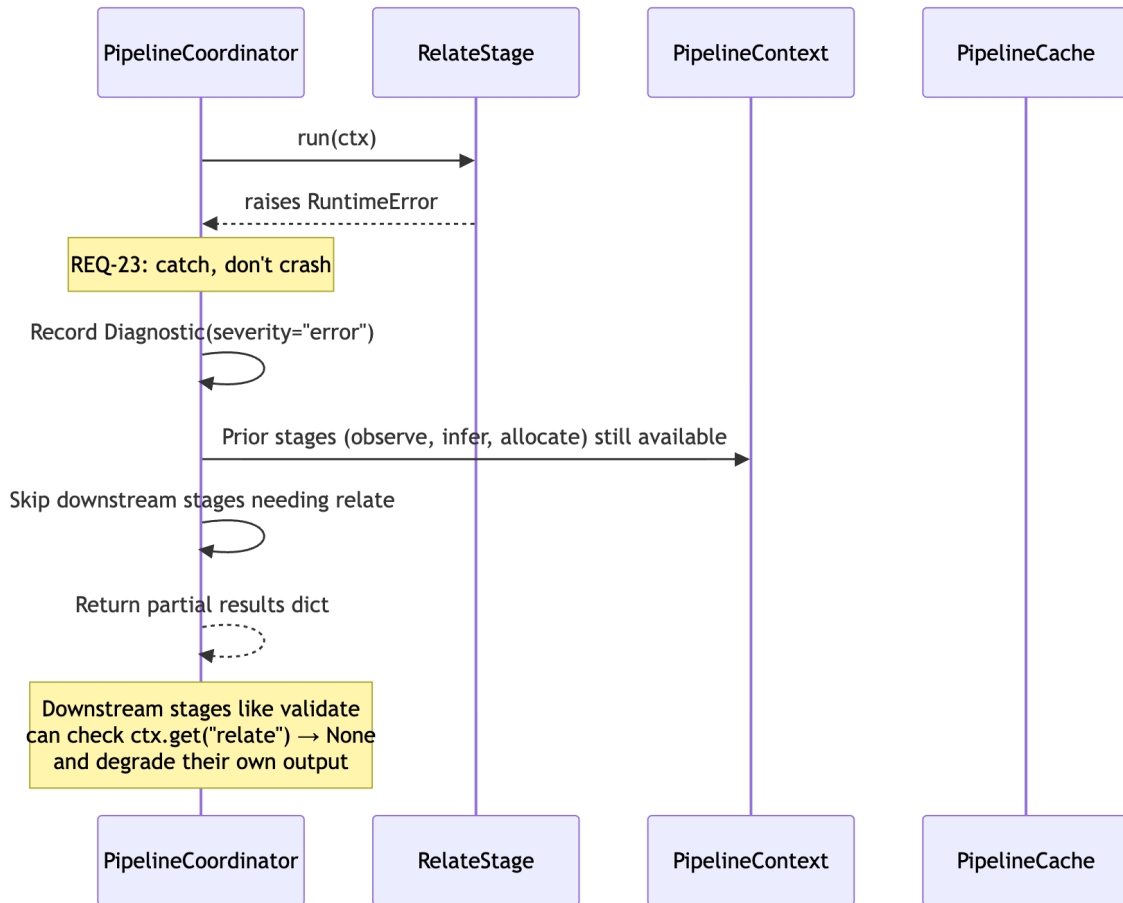


Figure 6: Diagram

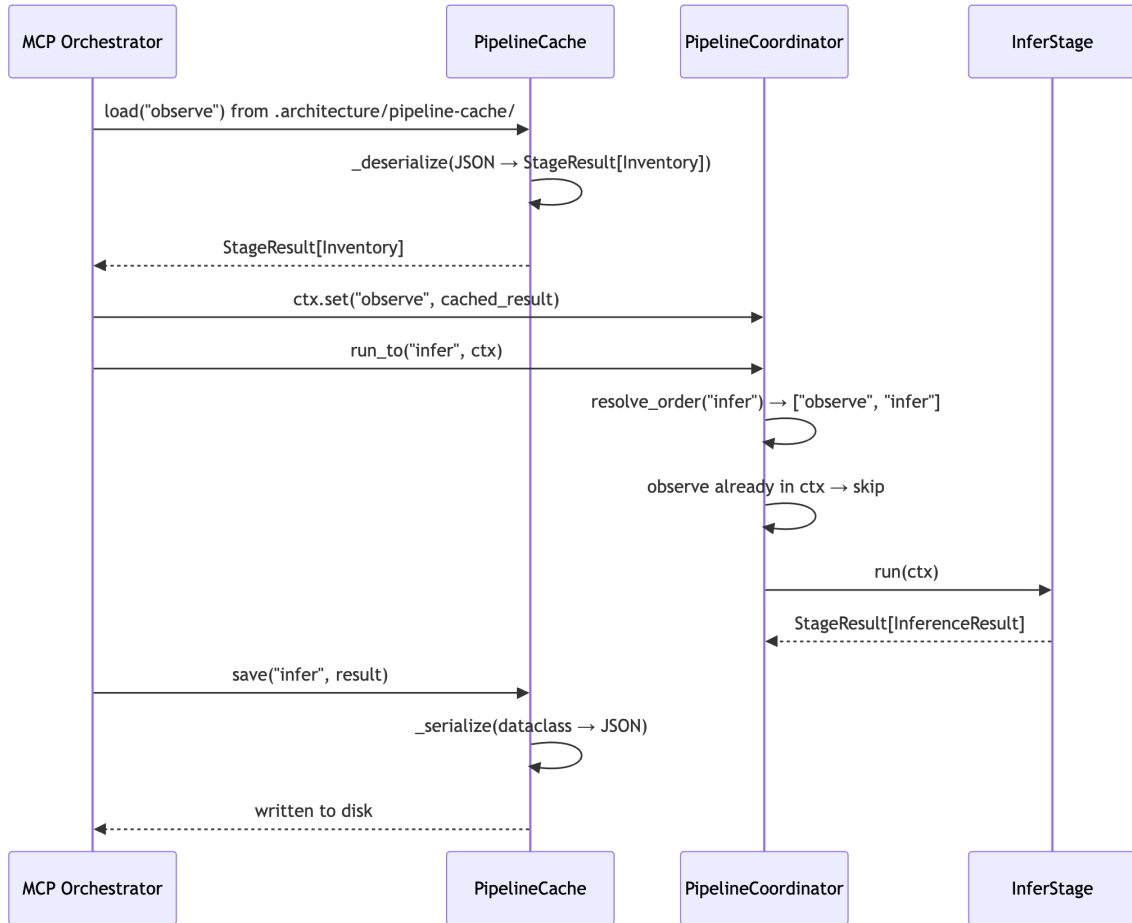


Figure 7: Diagram

Req	Text	Satisfied By	Rationale & Consequences
REQ-4 Entity type coverage	Include capabilities, components with signatures, constants, symbols, files	COMP-2.2 — ObserveStage captures ModuleRecord , ClassRecord , FunctionRecord , ConstantRecord ; InferStage produces InferredCapability	Why: An architecture model missing entity types is incomplete — downstream consumers (documentation generators, code generators) need typed entities to produce useful output. Violation: Missing entities → incomplete model → silent gaps in generated docs.
REQ-5 Relationship population	Include realizes, depends-on, contains	COMP-2.3 — RelateStage produces DerivedRelationship with rel_type in {realizes, depends-on, contains, exposes, uses}	Why: Relationships ARE the architecture — entities without connections are just a file listing. The minimum set {realizes, depends-on, contains} captures capability traceability, coupling, and hierarchy. Violation: Missing realizes → can't trace capabilities to implementation; missing depends-on → invisible coupling.

Req	Text	Satisfied By	Rationale & Consequences
REQ-6 Deterministic pipeline	Same input $\rightarrow$ same output	COMP-2.1 — <code>PipelineCoordinator</code> uses <code>resolve_order()</code> with Kahn’s algorithm (sorted tie-breaking); no randomness in stage logic	Why: Non-determinism makes diffs meaningless, CI flaky, and debugging impossible. The sorted tie-breaking in <code>_topo_sort</code> ensures even stages with equal priority execute in consistent order. Violation: Random output $\rightarrow$ spurious diffs in version-controlled models $\rightarrow$ developer distrust $\rightarrow$ abandonment.
REQ-7 Stage independence	Each stage independently runnable with cached predecessors	COMP-2.1 — <code>PipelineCache</code> serializes/deserializes <code>StageResult</code> to JSON; <code>PipelineContext.get()</code> retrieves cached results	Why: Enables MCP-style one-stage-per-call orchestration, debugging individual stages, and human review between stages. Violation: Monolithic execution $\rightarrow$ no resume, no inspection, no iterative correction.
REQ-17 Test preservation	All existing tests must pass after changes	COMP-2.4 — <code>ValidateStage</code> checks structural consistency	Why: The pipeline modifies output artifacts; if it breaks existing test contracts, it has violated the system’s behavioral guarantees. Violation: Silent test breakage $\rightarrow$ shipped regressions.

Req	Text	Satisfied By	Rationale & Consequences
REQ-23 Graceful degradation	Produce partial results on failure	COMP-2.1 — Coordinator catches stage exceptions, records diagnostics, returns completed stages	Why: A 7-stage partial model is infinitely more valuable than a crash with no output. Architecture extraction is heuristic — some stages will fail on unusual codebases. Violation: All-or-nothing → pipeline unusable on any codebase with edge cases.
REQ-24 Uncertainty surfacing	Flag ambiguous inferences	COMP-2.2 — InferStage emits Uncertainty objects with category , description , suggested_fallback	Why: Silent guessing produces models that LOOK correct but ARE wrong — the worst possible outcome. Explicit uncertainty lets humans or LLMs make informed decisions. Violation: False confidence → wrong architecture decisions based on wrong models.
REQ-25 Large repo handling	>200 files in <60s without LLM	COMP-2 — AST-only observe stage, import-affinity allocation (no LLM calls in deterministic path)	Why the threshold: 200 files medium production codebase; 60s tolerable CI pipeline addition. LLM calls are excluded because they add 5-30s each and are non-deterministic. Violation: Slow pipeline → not used in CI → architecture model drifts from code.

7. Trade-offs & Design Decisions

10-Stage Pipeline vs. Fewer Stages

Chosen: 10 stages with explicit types per stage (`Inventory`, `InferenceResult`, `AllocationResult`, etc.)

Traded: Simplicity of a 3-stage pipeline (`scan` → `analyze` → `emit`) for debuggability and cacheability. Each stage boundary is an inspection point. The cost is more dataclass definitions and serialization logic in `cache.py`.

Would change if: The pipeline only ran in batch mode (no MCP), fewer stages would reduce overhead.

Capability-Seeded Allocation vs. Import Clustering

Chosen: `AllocateStage` seeds components from inferred capabilities, then assigns remaining files by import affinity (`_assign_by_import_affinity`). Oversized components split at `MAX_COMPONENT_FILES = 12`.

Traded: Pure import clustering (which discovers natural code modules) for capability-driven grouping (which produces architecturally meaningful components). Import clustering can produce components like “all files that import `utils`” — meaningless architecturally.

Threshold rationale: 12 files max is a heuristic for cognitive load — a component with 30 files is not a component, it’s a subsystem. The `MIN_COMPONENT_FILES = 2` prevents single-file “components” that add noise.

Evidence-Based Claims vs. Boolean Facts

Chosen: `Claim[T]` wraps values with `Evidence` list and weighted confidence via `SOURCE_WEIGHTS`. AST evidence gets weight 1.0; LLM analysis gets 0.6.

Traded: Simplicity for traceability. Every model assertion can be traced to its source and confidence assessed. The weight hierarchy (`ast=1.0 > test=0.95 > documentation=0.8 > llm=0.6`) reflects epistemic reliability.

File-Based Cache vs. In-Memory Only

Chosen: `PipelineCache` writes JSON to `.architecture/pipeline-cache/` with full dataclass serialization/deserialization.

Traded: Speed (in-memory is faster) for cross-process resumability. The MCP orchestrator runs stages in separate invocations, so disk persistence is essential. The `_serialize/_deserialize` functions handle nested dataclasses, `Path` objects, and dynamic type resolution via `_get_output_class`.

Deterministic Sort Tie-Breaking

Chosen: `_topo_sort` uses `sorted()` for both initial queue and successor iteration.

Traded: Nothing meaningful — sorted tie-breaking has negligible cost but guarantees REQ-6. Without it, Python’s set iteration order (implementation-dependent) would produce non-deterministic stage ordering.

8. Measures of Effectiveness

Capability	MoE	Minimum	Optimal	Value Function
Observation	File parse success rate	>95%	100%	Each unparsed file is a blind spot; value increases linearly with coverage
Observation	Import edge completeness	Captures direct imports	Captures re-exports and dynamic imports	More edges → better allocation and relationship derivation
Inference	Capability count vs. manual extraction	>50% of human-identified capabilities	>90% match	Under-extraction is worse than over-extraction (false negatives harder to catch)
Inference	Uncertainty flagging rate	Flags obvious ambiguities	Flags all below-threshold inferences with <code>Uncertainty.priority</code>	Unflagged wrong inferences are the costliest failure mode
Allocation	<code>AllocationResult.imports_coverage</code>	>95% coverage	100%	Unallocated files are invisible to the architecture model
Allocation	<code>AllocationResult.boundary_coherence</code>	>70%	>90%	Low coherence = components with many cross-boundary imports = wrong decomposition
Validation	<code>ValidateResult.score</code>	>60 (usable)	>90 (publication-ready)	Score is a composite; value is non-linear — below 50 the model misleads rather than helps
Performance	Wall-clock for 200-file repo	<60s	<10s	Sub-10s enables interactive use; >60s breaks CI integration
Coordination	Stages skipped via cache	0	All unchanged stages skipped	Each cache hit saves seconds; compound savings across iterative development

Capability	MoE	Minimum	Optimal	Value Function
Emit	<code>EmitResult.total_bytes</code> > 0	Files written	Canonical structure with per-component docs	More structured output → more downstream tool compatibility

9. Failure Modes

Component	Failure Mode	Impact	Degradation Behavior
ObserveStage	<code>SyntaxError</code> on a file	Single file missing from <code>Inventory</code>	Graceful — logs <code>Diagnostic(severity="warning", code="parse-failed")</code> , continues with remaining files
InferStage	No capabilities detected	Empty <code>InferenceResult.capabilities</code>	<code>_infer_fallback_capabilities</code> — creates capabilities from packages with >3 public functions
AllocateStage	All files unallocated	Components list empty	Creates catch-all “Infrastructure” component for remaining files
RelateStage	Missing predecessor results	<code>RuntimeError</code>	Coordinator catches; downstream stages (validate, decompose) receive <code>None</code> from <code>ctx.get("relate")</code>
ValidateStage	Low score	<code>ValidateResult.score</code> < 60	Non-fatal — score reported, issues listed with actionable messages
PipelineCache	Corrupted JSON on disk	Deserialization fails	Stage re-executes from scratch (cache miss behavior)
PipelineCoordinator	Circular dependency in stages	<code>RuntimeError("Circular dependency detected")</code>	Hard failure — this is a programming error, not a runtime condition
EmitStage	Disk write failure	<code>OSError</code>	Hard failure — no partial file writes; atomic per-file via <code>Path.write_text</code>

Pipeline Subsystem — Logical Architecture

1. Intent & Purpose

The Pipeline subsystem exists to transform a raw codebase into a structured architecture model through a deterministic, 10-stage extraction process. Without it, architecture extraction would be a monolithic, opaque operation impossible to debug, cache, resume, or extend.

The core philosophy: **decompose architecture extraction into independently testable, cacheable stages with explicit data contracts between them.** Each stage produces a typed result that subsequent stages consume, creating an auditable chain of evidence from source code to architecture model.

2. Component Structure



Figure 8: Diagram

COMP-2.1 — Pipeline Coordination

Intent: This boundary exists because stage orchestration, caching, and reporting are cross-cutting concerns that must not leak into individual stages. Without coordination, every stage would need to know about execution order, caching, and error handling.

File	Responsibility
<code>coordinator.py</code>	DAG resolution via Kahn’s algorithm (<code>_topo_sort</code>), dependency collection (<code>_collect_deps</code>), and minimum-path execution (<code>run_to</code>)
<code>protocol.py</code>	Defines the Stage protocol, <code>StageResult[T]</code> , <code>PipelineContext</code> , <code>Claim</code> , <code>Evidence</code> , <code>Uncertainty</code> , <code>Diagnostic</code> , <code>SOURCE_WEIGHTS</code> — the entire type vocabulary
<code>cache.py</code>	<code>PipelineCache</code> — JSON serialization/deserialization of <code>StageResult</code> objects to <code>.architecture/pipeline-cache/</code> for MCP resume
<code>context_gen.py</code>	Produces token-efficient markdown summary from pipeline results for LLM context windows
<code>report.py</code>	<code>StageReport</code> and <code>generate_pipeline_report</code> — markdown rendering of pipeline run metrics
<code>artifacts.py</code>	<code>write_artifacts</code> — writes structured output (<code>inventory.json</code> , <code>functional.yaml</code> , <code>structure.yaml</code> , etc.)

File	Responsibility
<code>corrections.py</code>	<code>get_corrections_for_stage</code> — loads user corrections from <code>LearningStore</code> filtered by stage name

COMP-2.2 — Observation Stages

Intent: Separates fact-gathering (observe) from interpretation (infer). Observe produces zero-inference facts; infer derives meaning. This separation exists because facts are deterministic and cacheable, while inference may involve heuristics that evolve independently.

File	Responsibility
<code>observe.py</code>	<code>ObserveStage</code> — AST scanning, import edge extraction, route detection, constraint detection. Produces <code>Inventory</code>
<code>observe_types.py</code>	<code>Inventory</code> , <code>ModuleRecord</code> , <code>ImportEdge</code> , <code>RouteRecord</code> , <code>ClassRecord</code> , <code>FunctionRecord</code> , <code>ConstantRecord</code> , <code>TestFileRecord</code> , <code>DocRecord</code> , <code>ConstraintRecord</code>
<code>infer.py</code>	<code>InferStage</code> — derives <code>InferredCapability</code> , <code>InferredActor</code> , <code>InferredBehavior</code> from <code>Inventory</code> via pattern matching (route clustering, domain module detection, fallback heuristics via <code>_infer_fallback_capabilities</code>)
<code>infer_types.py</code>	<code>InferenceResult</code> , <code>InferredCapability</code> , <code>InferredActor</code> , <code>InferredBehavior</code>

COMP-2.3 — Allocation & Relation Stages

Intent: Translates abstract capabilities into concrete component boundaries (allocate) and then discovers structural relationships between them (relate). These are separated because allocation is a clustering problem while relation discovery is a graph analysis problem — different algorithms, different failure modes.

File	Responsibility
<code>allocate.py</code>	<code>AllocateStage</code> — 4-step strategy: seed from capabilities → assign by import affinity → split oversized (<code>>MAX_COMPONENT_FILES=12</code>) → merge undersized (<code><MIN_COMPONENT_FILES=2</code>). Scoped mode uses per-file allocation for 15 files
<code>allocate_types.py</code>	<code>AllocationResult</code> , <code>ComponentAllocation</code> (with <code>file_coverage</code> , <code>boundary_coherence</code> metrics)

File	Responsibility
<code>relate.py</code>	<code>RelateStage</code> — derives <code>realizes</code> , <code>depends-on</code> , <code>contains</code> , <code>exposes</code> relationships. Uses <code>_pick_relationship_type</code> to distinguish <code>uses</code> (utility) from <code>depends-on</code> (domain)
<code>relate_types.py</code>	<code>RelateResult</code> , <code>DerivedRelationship</code>

COMP-2.4 — Specification & Contract Stages

Intent: Adds behavioral and interface contracts on top of the structural model. This boundary exists because interface specs and test contracts are verification artifacts — they validate the model rather than construct it.

File	Responsibility
<code>specify.py</code>	<code>SpecifyStage</code> — derives <code>InterfaceSpec</code> from routes (REST), CLI frameworks (<code>click/typer/argparse</code>), module exports
<code>specify_types.py</code>	<code>SpecifyResult</code> , <code>InterfaceSpec</code>
<code>contract.py</code>	<code>ContractStage</code> — maps test files to components via stem matching, name matching, and directory matching
<code>contract_types.py</code>	<code>ContractResult</code> , <code>TestContract</code>
<code>validate.py</code>	<code>ValidateStage</code> — checks: unrealized capabilities, orphan components, file coverage <95%, naming. Produces 0–100 score
<code>validate_types.py</code>	<code>ValidateResult</code> , <code>ValidationIssue</code>

COMP-2.5 — Synthesis & Emit Stages

Intent: Handles the transition from internal pipeline data structures to external artifacts. Decompose detects system boundaries for multi-system repos; synthesize runs scoped sub-pipelines; emit writes everything to disk. This boundary exists because output formatting and system-of-systems concerns are distinct from extraction logic.

File	Responsibility
<code>decompose.py</code>	<code>DecomposeStage</code> — detects <code>SystemBoundary</code> via directory clustering (<code>_cluster_by_directory</code>), splits large components into <code>SubComponents</code> . Thresholds: <code>FULL_SYSTEM_FILE_THRESHOLD=5</code> , <code>HIERARCHY_FILE_THRESHOLD=8</code> , <code>MAX_SYSTEMS=8</code>
<code>decompose_types.py</code>	<code>DecomposeResult</code> , <code>SystemBoundary</code> , <code>SubComponent</code>

File	Responsibility
<code>synthesize.py</code>	SynthesizeStage — runs scoped sub-pipelines per system boundary, builds <code>SoSModel</code> , generates reports/lessons. Decides full vs abbreviated stages via <code>_decide_stages</code> (8 files → full)
<code>synthesize_types.py</code> <code>emit.py</code>	SynthesizeResult , SystemModel , SoSModel EmitStage — writes final artifacts to disk, builds source→test reverse maps, tracks EmitResult (paths, bytes)
<code>emit_types.py</code> <code>regen_score.py</code>	EmitResult RegenScoreStage — computes regeneration readiness score from enriched model

3. Layer Allocation

All components are in the **domain layer** because they implement the core business logic of architecture extraction. They contain no HTTP handlers, no database access, no UI — they operate purely on in-memory data structures and file I/O. The pipeline is invoked by the CLI layer (COMP-8) and consumes services from the core layer (COMP-1.x) and scanners (COMP-3.1).

4. Dependency Graph

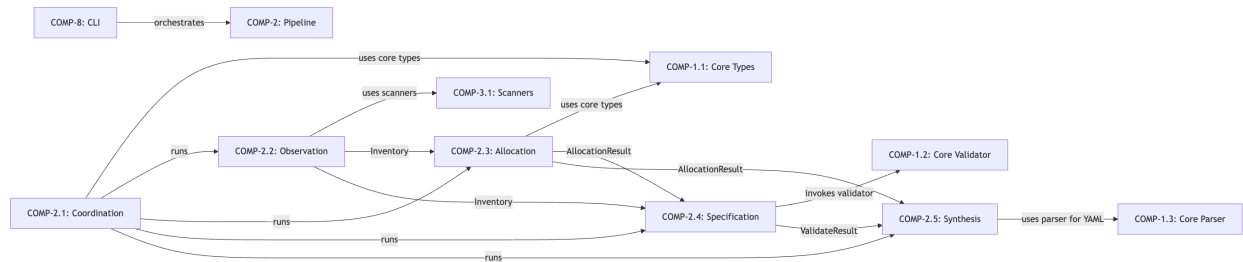


Figure 9: Diagram

Why each external dependency exists:

Dependency	Rationale	What breaks without it
COMP-2.1 → COMP-1.1	Coordinator needs Model , entity types to build final output	Cannot produce valid architecture models
COMP-2.2 → COMP-3.1	Observe stage delegates AST scanning to reusable scanners	No code facts — entire pipeline produces empty results
COMP-2.3 → COMP-1.1	Allocation needs component/capability type definitions	Cannot create typed components

Dependency	Rationale	What breaks without it
COMP-2.4 → COMP-1.2	Validate stage reuses core validation rules	Validation becomes incomplete, misses structural errors
COMP-2.5 → COMP-1.3	Emit stage needs YAML serialization via core parser	Cannot write <code>.architecture-model.yaml</code>

5. Interface Specification

IF-auto-COMP-2.1 — Pipeline Coordination API

Contract: `PipelineCoordinator.run_to(target, ctx)` guarantees: (1) all transitive dependencies of `target` execute first, (2) cached stages are skipped, (3) circular dependencies raise `RuntimeError`, (4) unknown stages raise `KeyError`.

IF-auto-COMP-2.2 — Observation Stages API

Contract: `ObserveStage.run(ctx) → StageResult[Inventory]` guarantees a complete `Inventory` with all parseable Python files scanned. Parse failures produce `Diagnostic` warnings, not exceptions. `InferStage` guarantees `InferenceResult` with uncertainties flagged via `Uncertainty` objects (REQ-24).

IF-auto-COMP-2.3 — Allocation & Relation Stages API

Contract: `AllocateStage` guarantees every source file appears in exactly one `ComponentAllocation`. Reports `file_coverage` and `boundary_coherence` as quality metrics. `RelateStage` guarantees at minimum `realizes`, `depends-on`, and `contains` relationship types (REQ-5).

IF-auto-COMP-2.4 — Specification & Contract Stages API

Contract: `ValidateStage` produces a 0–100 `score` and `is_valid` boolean. Issues are categorized by `severity` (error/warning/info) and `rule` name.

IF-auto-COMP-2.5 — Synthesis & Emit Stages API

Contract: `EmitStage` writes to `ctx.output_dir`, tracks all written paths in `EmitResult.written_paths`, and reports `total_bytes`. Idempotent — safe to re-run.

6. Key Data Types

The Evidence Chain: Evidence → Claim → Uncertainty

Why these exist: Architecture extraction is inherently uncertain. Rather than silently guessing, every assertion carries provenance.

- **Evidence** — source attribution with confidence (0.0–1.0) and location. Weighted by `SOURCE_WEIGHTS` (e.g., `ast: 1.0, llm_analysis: 0.6, search_result: 0.5`).
- **Claim[T]** — generic wrapper making any value auditable. `confidence` property computes weighted average across evidence.
- **Uncertainty** — explicit “I don’t know” with `category`, `suggested_fallback`, and `priority`.

StageResult[T]

Why generic: Every stage produces different output types but shares quality metrics, diagnostics, uncertainties, duration, and version. The generic parameter `T` ensures type safety while the wrapper provides uniform observability.

PipelineContext

Why a context object: Stages need shared state (repo path, output dir, cached results, scope files, learning store) without direct coupling. The context is the dependency injection mechanism — stages call `ctx.get("observe")` rather than importing each other.

Stage Output Types

Each `*Result` dataclass is deliberately separate from its stage implementation to allow serialization (cache), testing, and evolution independently.

7. Design Decisions & Rationale

Decision	Alternatives Considered	Chosen	Rationale	What Would Change If...
10 fixed stages with DAG ordering	Free-form plugin stages; monolithic extractor	Fixed stages, topological sort via Kahn's	Predictable, debuggable, cacheable. Each stage boundary is a checkpoint.	More stages needed → add to DAG, existing stages unaffected
Typed <code>StageResult[T]</code> per stage	Untyped dict passing; single result type	Generic dataclass per stage	Compile-time safety, serialization clarity, independent evolution	Common fields change → update one base type
File-based <code>PipelineCache</code> (JSON)	In-memory only; database; pickle	JSON files in <code>.architecture/pipelinecache/</code>	Human-readable, cacheable, no external deps. Trade-off: serialization complexity in <code>_serialize/_deserialize</code>	Large repos with huge inventories → JSON becomes slow, would need binary format
<code>SOURCE_WEIGHTS</code> as static dict	Per-project configurable weights; ML-learned weights	Hardcoded weights in <code>protocol.py</code>	Simplicity, determinism (REQ-6). Trade-off: can't adapt to domains where e.g. documentation is more reliable than AST	Domain-specific accuracy needed → make weights configurable per project

Decision	Alternatives Considered	Chosen	Rationale	What Would Change If...
Capability-seeded allocation	Import-graph clustering; directory-based; LLM-based	Seed from capabilities, then import affinity	Produces architecturally meaningful boundaries, not just code structure. Trade-off: depends on infer quality	Infer produces poor capabilities → allocation degrades to fallback “Infrastructure” bucket
Separate observe/infer	Single “analyze” stage	Two stages	Facts (cacheable, deterministic) vs interpretation (heuristic, evolvable). Trade-off: extra stage boundary overhead	Heuristics stabilize → could merge, but cache granularity loss
Uncertainty as first-class type	Log warnings; drop uncertain items; guess silently	Explicit Uncertainty objects surfaced in results	REQ-24 compliance. Enables downstream LLM enrichment to target unknowns. Trade-off: more complex result types	Uncertainties ignored → silent model errors, harder debugging

8. Failure Modes

Component	Failure Mode	Behavior	Graceful?
ObserveStage	Python file has SyntaxError	Skipped with <code>Diagnostic(severity="warning", code="parse-failed")</code> . Other files still scanned	Graceful — partial
ObserveStage	Repo path doesn't exist	StageResult with empty Inventory	Graceful — empty but valid
InferStage	No capabilities detected	<code>_infer_fallback_capabilities</code> activates — creates capabilities from packages with >3 public functions	Graceful — degraded but functional

Component	Failure Mode	Behavior	Graceful?
AllocateStage	No capabilities to seed from	All files land in “Infrastructure” catch-all component	Degraded — model is valid but architecturally meaningless
AllocateStage	Missing predecessor results	<code>RuntimeError("allocate requires observe and infer")</code>	Hard fail — coordinator should prevent this
RelateStage	No import edges	Only <code>realizes</code> and <code>contains</code> relationships produced; no <code>depends-on</code>	Graceful — sparse but valid
ValidateStage	Low file coverage	<code>ValidationIssue(severity="warning")</code> with coverage percentage; <code>score</code> reduced but pipeline continues	Graceful
PipelineCache	Corrupt JSON on disk	<code>_deserialize</code> fails → stage re-runs from scratch	Graceful — cache miss, not crash
PipelineCoordinator	Circular dependency in stage graph	<code>RuntimeError("Circular dependency detected")</code> in <code>_collect_deps</code> or <code>_topo_sort</code>	Hard fail — configuration error
PipelineCoordinator	Unknown target stage	<code>KeyError(f"Unknown stage: {name}")</code>	Hard fail — caller error
SynthesizeStage	Sub-pipeline for a system boundary fails	Per REQ-23, partial results from successful sub-pipelines are preserved	Graceful
EmitStage	Disk write fails (permissions)	Standard <code>OSError</code> propagates	Hard fail — no output
RegenScoreStage	Model file missing	<code>can_run</code> returns <code>False</code> ; stage skipped	Graceful

9. Measures of Effectiveness

MoE	Threshold (minimum)	Target (optimal)	How measured
File coverage	80%	>95%	<code>AllocationResult.file_coverage</code>
Boundary coherence	50%	>80%	<code>AllocationResult.boundary_coherence</code> (low cross-boundary imports)
Validation score	60/100	>85/100	<code>ValidateResult.score</code>

MoE	Threshold (minimum)	Target (optimal)	How measured
Large repo latency	<60s for 200+ files (REQ-25)	<30s	<code>StageResult.duration_ms</code> summed
Capability realization	All capabilities have 1 component	1:1 mapping	Validate stage check
Uncertainty ratio	<30% of claims uncertain	<10%	<code>len(uncertainties)</code> / <code>total_claims</code>
Stage cache hit rate	N/A	>50% on re-runs	Cache hits / total stage invocations

Pipeline Subsystem — Use Cases

Use Case Diagram

UC-1: Full Pipeline Extraction

Actor: Developer via CLI (COMP-8)

Intent: Extract a complete architecture model from a codebase in one shot, producing a YAML model file and supporting artifacts. The developer needs a machine-readable architecture description without manually documenting every component and relationship.

Preconditions: - Repository path exists with Python source files - No prior pipeline state required

Main Flow: 1. CLI constructs `PipelineContext` with `repo_path` and `output_dir` 2. CLI instantiates `PipelineCoordinator` with all stages (`ObserveStage`, `InferStage`, `AllocateStage`, `RelateStage`, `SpecifyStage`, `ContractStage`, `ValidateStage`, `DecomposeStage`, `SynthesizeStage`, `EmitStage`) 3. `PipelineCoordinator.run_to("emit", ctx)` is called 4. `resolve_order("emit")` performs topological sort via `_topo_sort()` producing the 10-stage sequence 5. `ObserveStage.run(ctx)` scans all `.py` files, produces `Inventory` with `ModuleRecord`, `ImportEdge`, `RouteRecord`, `ConstantRecord` 6. `InferStage` derives `InferredCapability`, `InferredActor`, `InferredBehavior` from inventory patterns 7. `AllocateStage` seeds components from capabilities, assigns files by import affinity, enforces `MAX_COMPONENT_FILES=12 / MIN_COMPONENT_FILES=2` 8. `RelateStage` produces `DerivedRelationship` entries (realizes, depends-on, contains, exposes) 9. `SpecifyStage / ContractStage` derive `InterfaceSpec` and `TestContract` 10. `ValidateStage` scores model 0–100, emits `ValidationIssue` list 11. `DecomposeStage` detects `SystemBoundary` entities, splits large components via `_decompose_component()` 12. `SynthesizeStage` builds `SoSModel` and `SystemModel` YAML strings 13. `EmitStage` writes all artifacts to disk, returns `EmitResult` with `written_paths`

Postconditions: - `.architecture-model.yaml` exists with capabilities, components, relationships (REQ-4, REQ-5) - Same input produces identical output (REQ-6) - All source files allocated to components (`file_coverage` 95%)

Error Handling: - `SyntaxError` in individual files → `Diagnostic(severity="warning", code="parse-failed")`, file skipped, pipeline continues (REQ-23) - Missing stage dependency → `RuntimeError` raised by `_collect_deps()` with circular dependency detection - If infer finds no capabilities → `_infer_fallback_capabilities()` generates package-based capabilities rather than producing an empty model

Quality Attributes: - Repos with >200 files must complete in <60s without LLM (REQ-25) - Deterministic: no randomness in any stage (REQ-6)

Measures of Effectiveness: - `ValidateResult.score` — higher is better; 80+ indicates a structurally sound model - `AllocationResult.file_coverage` — 100% means every source file has a component home - `AllocationResult.boundary_coherence` — low cross-boundary imports indicate clean component boundaries - Zero Uncertainty objects with `priority="blocking"` — ambiguities were resolved or flagged, not silently guessed (REQ-24) - Relationship type diversity: model contains at minimum realizes, depends-on, and contains (REQ-5)

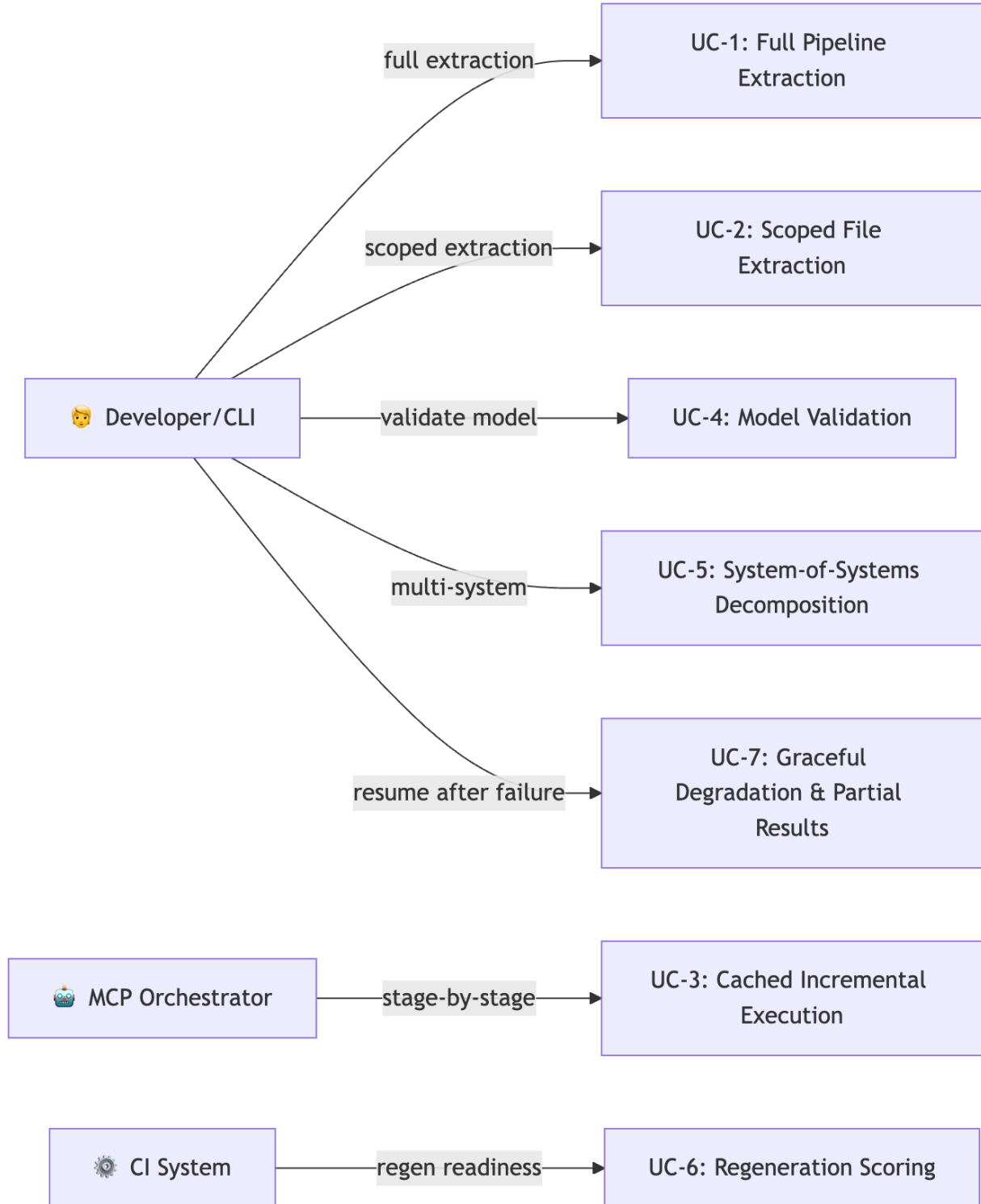


Figure 10: Diagram

UC-2: Scoped File Extraction

Actor: Developer or MCP tool

Intent: Extract architecture for a subset of files (e.g., a single package or PR changeset) without scanning the entire repository. This enables fast feedback loops during development.

Preconditions: - `PipelineContext.scope_files` populated with specific `Path` objects

Main Flow: 1. `ObserveStage.run()` checks `ctx.scope_files` — only scans listed files instead of `rglob("*.py")` 2. `AllocateStage` detects `is_scoped=True` and `len(source_modules) <= _SCOPED_FILE_LIMIT (15)` → uses `_allocate_per_file()` strategy (one component per substantive file) instead of capability-seeded clustering 3. Remaining stages operate on the reduced dataset

Postconditions: - Model produced covering only scoped files - No scanning of unrelated files

Error Handling: - If scoped files don't exist → filtered out silently in `ObserveStage` (`abs_f.exists()` check) - If too few files for meaningful architecture → single-component model produced (graceful, not an error)

Quality Attributes: - Sub-second for 15 files

Measures of Effectiveness: - All scoped files appear in output components (zero unallocated) - Proportional reduction in execution time vs. full scan

UC-3: Cached Incremental Stage Execution

Actor: MCP Orchestrator

Intent: Execute pipeline stages one at a time across separate process invocations, resuming from cached results. This is essential for MCP-based workflows where each tool call is a separate request.

Preconditions: - `.architecture/pipeline-cache/` directory accessible - Prior stages cached as JSON via `PipelineCache`

Main Flow: 1. MCP tool calls `PipelineCoordinator.run_to(target_stage, ctx)` 2. Coordinator calls `resolve_order()` to determine required predecessors 3. For each predecessor, checks `PipelineCache` — `_deserialize()` reconstructs `StageResult` with correct output type via `_get_output_class(stage_name)` 4. Only uncached stages execute; results serialized via `_serialize()` (handles dataclasses, `Path` objects, nested structures) 5. Target stage runs with predecessor outputs available via `ctx.get(stage_name)`

Postconditions: - Target stage result cached for future use (REQ-7) - Cached results byte-identical to fresh execution (REQ-6)

Error Handling: - Corrupted cache JSON → stage re-executes from scratch - Missing predecessor cache → predecessor re-executes

Quality Attributes: - Cache hit avoids all computation for that stage - Serialization handles all pipeline types including `Path`, `Evidence`, `Claim`, `Uncertainty`

Measures of Effectiveness: - Cache hit rate across repeated runs — ideally 100% for unchanged inputs - Round-trip fidelity: `_deserialize(_serialize(result))` produces identical `StageResult` - Wall-clock time saved vs. full re-execution

UC-4: Architecture Model Validation

Actor: Developer or CI system

Intent: Assess whether an extracted model is structurally sound and complete — not just “does it parse” but “is it a good architecture description.”

Preconditions: - Stages observe through relate (minimum) have completed

Main Flow: 1. `ValidateStage.run(ctx)` retrieves `InferenceResult`, `AllocationResult`, `RelateResult` 2. Check 1: Every `InferredCapability.id` appears as `to_id` in a `realizes` relationship — unrealized capabilities produce `ValidationIssue(rule="capability_realization")` 3. Check 2: Every `ComponentAllocation.id` appears in at least one relationship — orphans flagged with `rule="orphan_detection"` 4. Check 3: `AllocationResult.file_coverage` checked against 95% threshold 5. Score computed 0–100, `ValidateResult.is_valid` set

Postconditions: - `ValidateResult` with actionable `ValidationIssue` list - Score reflects structural quality

Error Handling: - Missing predecessor stages → `RuntimeError("validate requires infer, allocate, relate")` - Individual check failures don’t abort — all checks run, all issues collected

Quality Attributes: - Validation itself must be fast (no I/O beyond reading predecessor results)

Measures of Effectiveness: - Score correlates with model usefulness: 90+ should mean the model can drive code generation - Issue count trending down across iterations indicates model improvement - Zero false-positive issues — every flagged issue represents a real structural deficiency

UC-5: System-of-Systems Decomposition

Actor: Developer working with a large multi-system repository

Intent: Automatically detect autonomous system boundaries within a monorepo, run scoped pipelines per system, and assemble a hierarchical System-of-Systems model. Without this, large repos produce a flat, incomprehensible component list.

Preconditions: - Stages through relate completed - Repository contains multiple distinct subsystems (detected by `FULL_SYSTEM_FILE_THRESHOLD=5`)

Main Flow: 1. `DecomposeStage.run()` analyzes `AllocationResult` and `RelateResult` 2. Components grouped into `SystemBoundary` objects; large components (`HIERARCHY_FILE_THRESHOLD=8` files) decomposed via `_decompose_component()` into `SubComponent` by directory affinity (`_cluster_by_directory()`) 3. Over-fragmented results merged if system count exceeds `MAX_SYSTEMS=8` using `MERGE_COUPLING_THRESHOLD=0.3` 4. `SynthesizeStage` iterates `DecomposeResult.systems`: - For each `SystemBoundary`, `_decide_stages()` selects full or abbreviated pipeline based on file count - Scoped `PipelineContext` created with `scope_files` set to boundary files - Nested pipeline produces `SystemModel` with `model_yaml` 5. `SoSModel` assembled with actors, emergent capabilities, cross-system behaviors, inter-system interfaces

Postconditions: - Hierarchical model: SoS → Systems → Components → SubComponents - Inter-system relationships captured in `DecomposeResult.inter_system_edges`

Error Handling: - Single system failure → that system gets partial model, others unaffected (REQ-23) - If decomposition finds only one system → no SoS layer, direct model output

Quality Attributes: - System boundaries should align with developer mental models (package/directory structure)

Measures of Effectiveness: - `boundary_coherence` per system — high coherence means the boundary cuts few import edges - Number of inter-system edges relative to intra-system edges — lower ratio = better decomposition - Each `SystemBoundary.complexity` score reflects actual coupling, not just file count

UC-6: Regeneration Readiness Scoring

Actor: CI system or developer assessing model maturity

Intent: Determine whether the architecture model is rich enough to drive code regeneration — not just “is it valid” but “could an LLM reconstruct this codebase from this model.”

Preconditions: - `.architecture-model.yaml` exists - Validate stage completed

Main Flow: 1. `RegenScoreStage.run(ctx)` loads model via `load_model(model_path)` 2. Calls `compute_regen_readiness(model)` from core 3. Produces `RegenScoreResult` with `overall` score, letter `grade`, per-component `component_scores` 4. Checks for enrichment (signatures on components) — missing enrichment yields `Diagnostic(code="REGEN_NOT_ENRICHED")` 5. `blockers` list identifies specific gaps; `recommendation` provides actionable next step

Postconditions: - Quantified readiness score with actionable blockers

Error Handling: - Model file missing → `can_run()` returns `False`, stage skipped - Unenriched model → warning diagnostic, score reflects lower readiness (not an error)

Measures of Effectiveness: - Score should predict actual regeneration success rate - `blockers` list length → zero blockers means model is regeneration-ready - Grade progression (D→C→B→A) across enrichment iterations

UC-7: Graceful Degradation on Stage Failure

Actor: Any (automatic behavior)

Intent: Ensure that a failure in any single pipeline stage produces partial results rather than total failure. An incomplete architecture model is infinitely more useful than a stack trace. This is the difference between “we know 8 of 10 things” and “we know nothing.”

Preconditions: - Pipeline execution in progress

Main Flow: 1. `PipelineCoordinator.run_to()` iterates resolved stage order 2. If stage N raises an exception, coordinator catches it (REQ-23) 3. Prior stage results (1 through N-1) remain available in context and cache 4. `Diagnostic(severity="error")` recorded for the failed stage

5. Downstream stages that don't strictly depend on N may still execute 6. **Uncertainty** objects surface what couldn't be determined (REQ-24) — **suggested_fallback** indicates how to resolve (e.g., "llm_analysis")

Postconditions: - Partial **StageResult** objects available for all completed stages - No data loss from stages that succeeded before the failure - Uncertainties explicitly enumerated, not silently dropped

Error Handling (meta): - This IS the error handling use case. The key design choice: catch at stage boundary, not at individual operation level. Each **Stage.run()** is the unit of failure isolation.

Quality Attributes: - Partial output must still be valid (parseable YAML, valid JSON) — just incomplete - Uncertainty objects must have actionable **suggested_fallback** values

Measures of Effectiveness: - Ratio of stages completed vs. total stages attempted — higher is better degradation - All **Uncertainty.category** values are specific (not generic “unknown error”) - Partial model still passes **ValidateStage** with reduced but nonzero score - Time-to-partial-result time of first failed stage (no wasted computation after failure)

Artifact Traceability Map: architecture-model-standard / Pipeline

1. Entity Inventory

Entity Type	Count	Feeds SE Documents
Components	6	Logical Architecture, Maintenance Manual, Operations Manual, Interface Specification, Component Specs
Capabilities	1	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	8	Requirements Analysis, Verification & Validation
Actors	0	ConOps, Use Cases
Layers	0	Logical Architecture

2. Artifact Dependency Graph

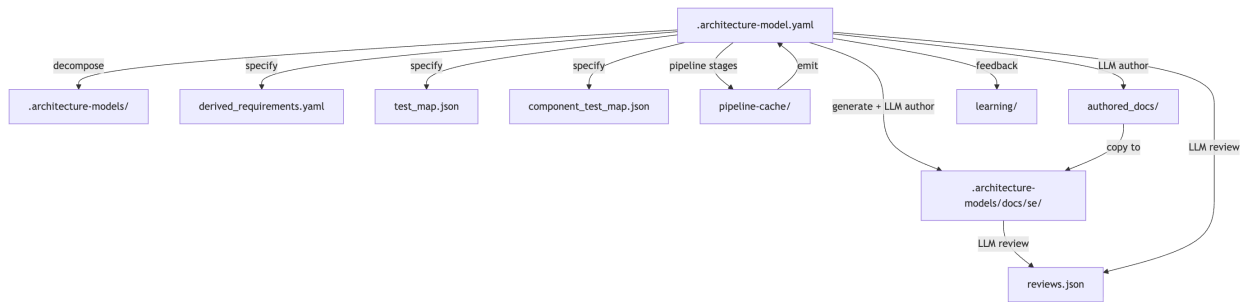


Figure 11: Diagram

3. Entity-to-Artifact Traceability Matrix

Artifact	Component	Capabilities	Behaviors	Interfaces	Constraints	Requirements	Actors	Layers
Behavior			—					
Flows								
Component	6							
Specs								
ConOps		1					—	
Functional		1	—					
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		1			—	8		
Risk Assessment					—			
Use Cases			—				—	
Verification & Validation			—			8		

4. Relationship Distribution

Relationship Type	Count	Connects
satisfies	8	Component → Requirement
depends-on	6	Component → Unknown, Unknown → Component
contains	6	Component → Component, Component → Unknown
exposes	6	Component → Interface
realizes	1	Component → Capability

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