

Concept of Operations: Architecture Model Standard

1. System Overview

The **Architecture Model Standard** system provides tooling to create, validate, enrich, and maintain structured architecture models that stay synchronized with source code. It solves the persistent problem of architecture documentation drifting from implementation by establishing a bidirectional link between models and code, enabling continuous verification of architectural intent against reality.

The system operates as an MCP-compatible toolset consumable by AI agents, developers, and CI/CD pipelines.

2. Stakeholders & Actors

Actor	Type	Goals
AI Agent (MCP Client)	Automated	Consume models for context, generate documentation, execute extraction pipelines
Developer	Human	Author models from requirements, query/slice models, assess readiness, fix drift
CI/CD Pipeline	Automated	Gate deployments on architecture compliance, detect drift, validate models

3. Operational Scenarios

Scenario A: Greenfield Model Authoring

A Developer receives a requirements document for a new subsystem. They invoke **Author Model from Requirements** (CAP-6) to parse the document into a concept-phase architecture model. They then use **Decompose Models Hierarchically** (CAP-9) to break it into per-system sub-models. The model is validated via **Validate Architecture Models** (CAP-1) before commit.

Scenario B: Code-First Extraction

An AI Agent is pointed at an existing codebase. It runs **Generate Reality Manifest** (CAP-4) to produce structural facts, then executes the **Run Modular Extraction Pipeline** (CAP-3) through all 10 stages (observe→infer→allocate→relate→specify→contract→validate→decompose→synthesize→emit) to produce a complete architecture model from code.

Scenario C: Continuous Compliance Gate

On every pull request, the CI/CD Pipeline runs **Check Development Gate** (CAP-12) to verify code changes track toward authored architecture intent. If drift is detected, **Detect and Fix Model Drift** (CAP-13) reports coverage gaps. The pipeline fails if thresholds are not met.

Scenario D: AI-Assisted Documentation Generation

An AI Agent queries the model using **Slice and Query Models** (CAP-7) for a specific subsystem, then invokes **Generate SE Documentation** (CAP-5) to produce functional analysis, logical architecture, use cases, requirements, V&V, and operations documents. Output is formatted via **Export for AI Consumption** (CAP-15) for token-limited environments.

Scenario E: Model Evolution Tracking

A Developer uses **Diff Model Versions** (CAP-8) to review what changed between releases. They invoke **Enrich Models with Code Intelligence** (CAP-10) to auto-populate updated signatures and test contracts, then **Assess Regen Readiness** (CAP-11) to confirm the model captures sufficient detail for code regeneration.

4. System Capabilities

Group	Capabilities
Model Creation	Author Model from Requirements (CAP-6), Extract Architecture from Code (CAP-2), Run Modular Extraction Pipeline (CAP-3)
Model Maintenance	Enrich Models with Code Intelligence (CAP-10), Decompose Models Hierarchically (CAP-9), Diff Model Versions (CAP-8)
Validation & Compliance	Validate Architecture Models (CAP-1), Check Development Gate (CAP-12), Detect and Fix Model Drift (CAP-13), Assess Regen Readiness (CAP-11)
Query & Export	Slice and Query Models (CAP-7), Export for AI Consumption (CAP-15), Generate SE Documentation (CAP-5)
Intelligence	Generate Reality Manifest (CAP-4), Manage Global Learnings (CAP-14)

5. Operational Constraints

ID	Constraint	Type
CON-1	Python >=3.11 required	Technology
CON-2	CI/CD runs on GitHub Actions	Technology

Assumptions: - Source code is accessible to the tooling at execution time - Models are stored in version-controlled repositories alongside code - AI Agents communicate via MCP protocol

6. System Context

The system sits between source code repositories and consumers of architectural knowledge. It reads code to produce models, reads models to produce documentation, and integrates into GitHub Actions workflows to enforce compliance. The **Manage Global Learnings** (CAP-14) capability persists heuristics across runs, enabling the system to improve extraction quality over time.

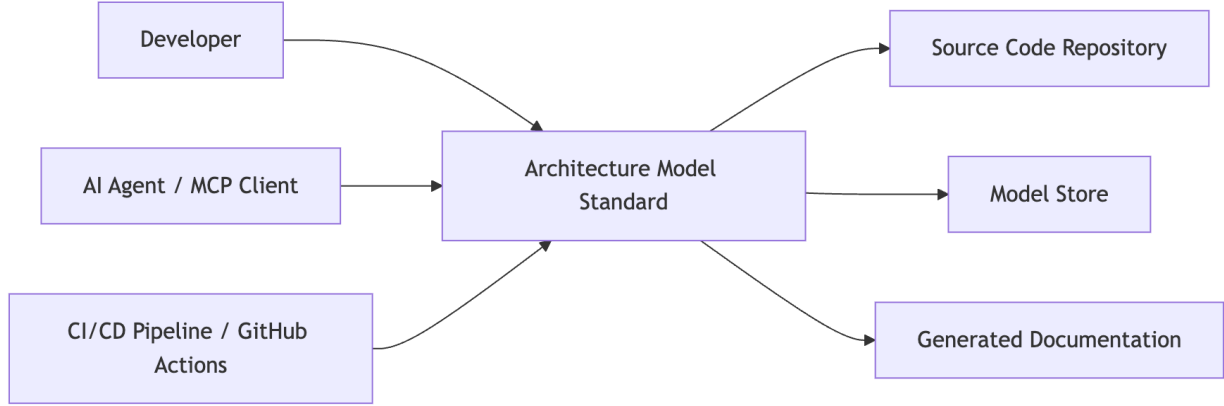


Figure 1: Diagram

Functional Analysis Document

1. Capability Inventory

ID	Capability	Description	Rationale
CAP-1	Validate Architecture Models	Checks model correctness via JSON schema validation, referential integrity between elements, hierarchy consistency (parent/child), cycle detection, and domain-specific rules from profiles	Ensures models remain internally consistent and conformant before downstream consumption
CAP-2	Extract Architecture from Code	Scans source code ASTs (Python, TypeScript, Kotlin) to automatically derive components, relationships, behaviors, routes, and constraints	Enables bottom-up architecture discovery from existing codebases
CAP-3	Run Modular Extraction Pipeline	Executes a 10-stage pipeline: observe → infer → allocate → relate → specify → contract → validate → decompose → synthesize → emit	Provides a repeatable, cacheable, stage-gated process for producing complete architecture models
CAP-4	Generate Reality Manifest	Scans source files to produce structural facts including functions, imports, classes, routes, and test coverage	Creates ground-truth inventory of what actually exists in code

ID	Capability	Description	Rationale
CAP-5	Generate SE Documentation	Produces a full systems engineering document suite: functional analysis, logical architecture, use cases, requirements, V&V, ConOps, and operations docs	Delivers formal engineering artifacts from model data without manual authoring
CAP-6	Author Model from Requirements	Parses a requirements document and synthesizes a concept-phase architecture model with components, capabilities, and relationships	Supports top-down, requirements-driven architecture development
CAP-7	Slice and Query Models	Filters models by block, layer, status, or artifact type for focused context delivery	Reduces cognitive load and token usage when working with large models
CAP-8	Diff Model Versions	Compares two model versions, reporting additions, removals, and changes at element level	Supports change tracking and review workflows
CAP-9	Decompose Models Hierarchically	Breaks coarse system-level models into per-subsystem sub-models maintaining parent/child traceability	Enables scalable modeling of complex systems
CAP-10	Enrich Models with Code Intelligence	Auto-populates function signatures, constants, test contracts, and behaviors from source analysis	Bridges the gap between abstract models and concrete implementation detail
CAP-11	Assess Regen Readiness	Scores how completely a model captures enough detail (signatures, contracts, behaviors) to regenerate code	Quantifies model completeness for code-generation workflows
CAP-12	Check Development Gate	Verifies that code reality is tracking toward the authored architecture intent, flagging divergence	Acts as a governance checkpoint in development workflows
CAP-13	Detect and Fix Model Drift	Compares model against current code, reports coverage gaps and suggests corrections	Maintains model-code synchronization over time
CAP-14	Manage Global Learnings	Stores, retrieves, and applies heuristics, archetypes, and workflow lessons across pipeline runs	Enables continuous improvement of extraction quality
CAP-15	Export for AI Consumption	Produces flat-file exports optimized for token-limited AI environments	Makes architecture knowledge accessible to LLM-based tools

2. Functional Decomposition

CAP-1: Validate Architecture Models

- Schema validation against JSON schema definitions
- Referential integrity checks (all references resolve)
- Hierarchy consistency (parent/child relationships valid)
- Cycle detection in dependency graphs
- Domain rule enforcement via profiles

CAP-2: Extract Architecture from Code

- AST parsing per language (Python, TS, Kotlin)
- Route detection (HTTP endpoints, CLI commands)
- Constraint detection (decorators, annotations)
- Relationship inference from imports/calls
- Artifact parsing (configs, tables)

CAP-3: Run Modular Extraction Pipeline

- **Observe:** Scan codebase, produce raw observations
- **Infer:** Derive components and behaviors from observations
- **Allocate:** Assign modules to architectural blocks
- **Relate:** Discover inter-component relationships
- **Specify:** Add interface specifications
- **Contract:** Attach test contracts
- **Validate:** Check intermediate model quality
- **Decompose:** Break into sub-models
- **Synthesize:** Merge and reconcile
- **Emit:** Output final model YAML

CAP-4: Generate Reality Manifest

- Multi-language file scanning
- Function/class/import extraction
- Call graph construction
- Test file analysis
- Component grouping

CAP-5: Generate SE Documentation

- Document type detection and selection
- Frontmatter generation
- Per-document-type rendering (ConOps, functional analysis, logical architecture, use cases, requirements, V&V, etc.)

CAP-6: Author Model from Requirements

- Requirements document parsing
- Concept-phase model synthesis

- Component/capability/relationship generation

CAP-7: Slice and Query Models

- Filter by block, layer, status, artifact
- Context-appropriate subset extraction

CAP-8: Diff Model Versions

- Element-level comparison
- Addition/removal/change classification

CAP-9: Decompose Models Hierarchically

- Parent/child relationship establishment
- Per-system sub-model generation
- Deep decomposition with behavior flows

CAP-10: Enrich Models with Code Intelligence

- Signature extraction and attachment
- Constant discovery
- Test contract inference
- Capability inference from code patterns
- Trigger/use-case detection

CAP-11: Assess Regen Readiness

- Completeness scoring across dimensions
- Gap identification

CAP-12: Check Development Gate

- Intent-vs-reality comparison
- Pass/fail gate determination

CAP-13: Detect and Fix Model Drift

- Coverage gap reporting
- Correction suggestion
- Drift documentation

CAP-14: Manage Global Learnings

- Heuristic storage and retrieval
- Lesson extraction from pipeline runs
- Archetype management

CAP-15: Export for AI Consumption

- Flat-file formatting
- Reference document generation
- Token-budget optimization

3. Capability-Component Mapping

Capability	Primary Component(s)	Explanation
CAP-1	COMP-1.2 (Validation)	<code>validator.py</code> implements schema, integrity, hierarchy, and domain rule checks
CAP-2	COMP-6 (Extract)	<code>from_code.py</code> , <code>route_detector.py</code> , <code>constraint_detector.py</code> perform AST-based extraction
CAP-3	COMP-2 (Pipeline), COMP-2.1–2.5	Coordinator orchestrates all 10 stages implemented across pipeline sub-components
CAP-4	COMP-3 (Manifest), COMP-3.1–3.3	Scanners produce facts; graph/analysis derives relationships; grouping organizes output
CAP-5	COMP-4 (Documentation), COMP-4.2 (SE Suite)	<code>se/generator.py</code> and per-document modules render the full SE document set
CAP-6	COMP-7 (Authoring)	<code>authoring/parser.py</code> handles requirements-to-model transformation
CAP-7	COMP-1.4 (Model Operations)	<code>slicer.py</code> implements filtering logic
CAP-8	COMP-1.4 (Model Operations)	<code>differ.py</code> implements version comparison
CAP-9	COMP-5.2 (Decomposition), COMP-1.5	<code>decompose.py</code> , <code>deep_decompose.py</code> break models hierarchically
CAP-10	COMP-5.1 (Enrichment)	<code>enrich.py</code> , <code>auto_enrich.py</code> populate models from code intelligence
CAP-11	COMP-1.5 (Quality Metrics)	<code>regen_readiness.py</code> computes readiness scores

Capability	Primary Component(s)	Explanation
CAP-12	COMP-7 (Authoring)	<code>gate.py</code> implements development gate checks
CAP-13	COMP-4.1 (Core Doc Generators), COMP-1.4	<code>drift.py</code> and <code>coverage.py</code> detect and report drift
CAP-14	COMP-11 (Pipeline Learning)	<code>global_learning.py</code> , <code>learning.py</code> , <code>lessons.py</code> manage heuristic state
CAP-15	COMP-10 (Export)	<code>flatfiles.py</code> , <code>reference.py</code> produce AI-optimized outputs

4. Behavioral Flows

CAP-3: Run Modular Extraction Pipeline

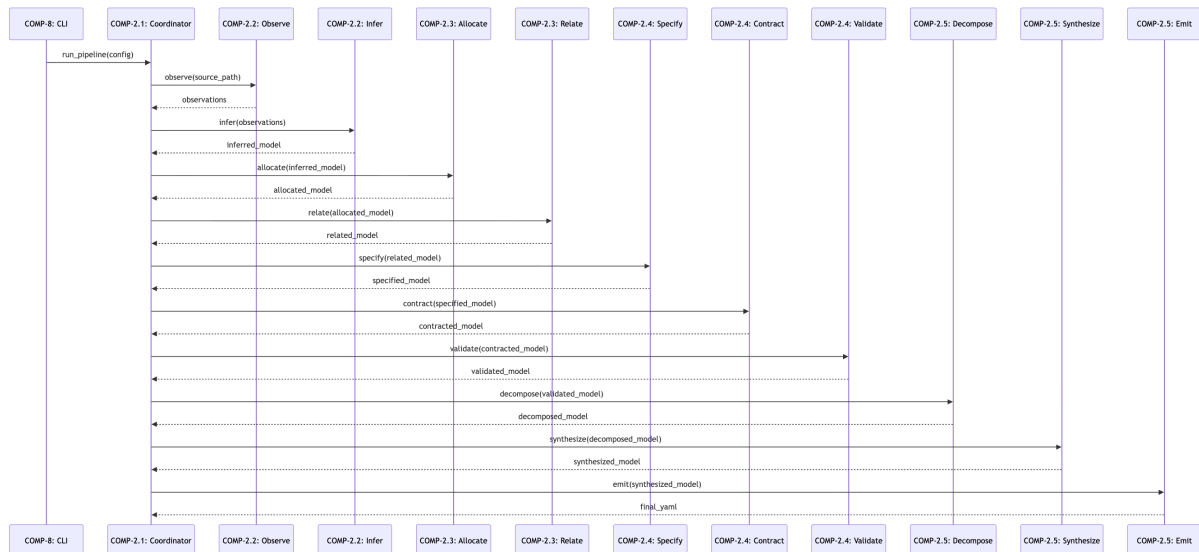


Figure 2: Diagram

CAP-4: Generate Reality Manifest

1. **Entry:** CLI or orchestration invokes `generator.py`
2. **Scan:** `multi_scanner.py` dispatches to language-specific scanners (`scanner.py` for Python, `ts_scanner.py`, `kt_scanner.py`)
3. **Analyze:** `call_graph.py` resolves imports; `interfaces.py` extracts public APIs; `behavior.py` detects behavioral patterns; `test_analyzer.py` maps test coverage
4. **Group:** `grouping.py` clusters functions/classes into logical components
5. **Emit:** `generator.py` produces the manifest structure with all facts

CAP-5: Generate SE Documentation

1. **Entry:** CLI invokes `se/generator.py` with model path and document type selection
2. **Detect:** `se/detect.py` determines which document types are applicable
3. **Frontmatter:** `se/frontmatter.py` generates metadata headers
4. **Render:** Per-type modules (`functional_analysis.py`, `logical_architecture.py`, `use_cases.py`, etc.) render markdown from model data
5. **Output:** Documents written to target directory

CAP-10: Enrich Models with Code Intelligence

1. **Entry:** `orchestration/enrich.py` or `auto_enrich.py` invoked with model + source path
2. **Context:** `enrichment_context.py` gathers available code intelligence
3. **Signatures:** Function signatures extracted and attached to components
4. **Capabilities:** `capability_inference.py` infers capabilities from code patterns
5. **Triggers:** `trigger_detection.py` identifies entry points and events
6. **Use Cases:** `use_case_inference.py` derives use cases from routes/CLI commands
7. **Output:** Enriched model returned

5. Functional Dependencies

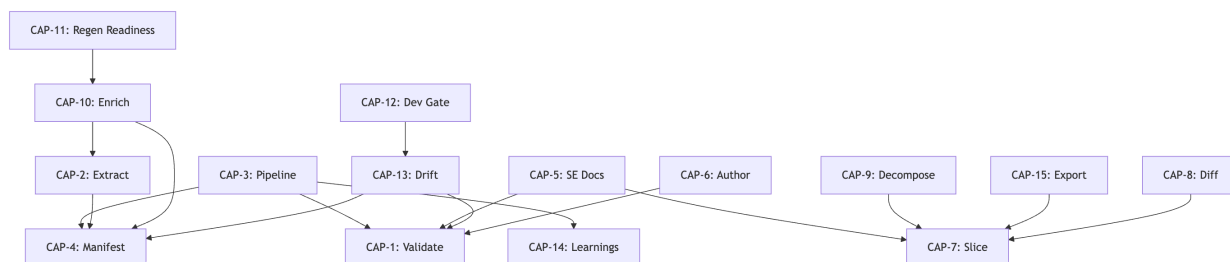


Figure 3: Diagram

Capability	Depends On
CAP-1	COMP-9 (Configuration) for schema/profile definitions
CAP-2	CAP-4 (Manifest scanning for AST data)
CAP-3	CAP-1 (validation stage), CAP-4 (observation stage), CAP-14 (heuristics)
CAP-5	CAP-1 (valid model required), CAP-7 (slicing for focused sections)
CAP-9	CAP-7 (slicing to extract sub-models)
CAP-10	CAP-4 (source facts), CAP-2 (code extraction)
CAP-11	CAP-10 (enriched model needed for scoring)
CAP-12	CAP-13 (drift detection underlies gate checks)
CAP-13	CAP-4 (current code facts), CAP-1 (model integrity)
CAP-15	CAP-7 (slicing for context reduction)

6. Gaps & Unrealized Capabilities

Issue	Description
Capability Realization links empty	The architecture model's realizes relationships are unpopulated (all entries show empty source/target), meaning formal traceability from behaviors to capabilities is not yet encoded
Behaviors misaligned with capabilities	BEH-1 through BEH-18 describe Django HTTP endpoints (login, password reset, bookmarklets, templates) that do not clearly map to any of the 15 declared capabilities — these appear to be application behaviors from a different system or a web admin interface not reflected in the capability model
BEH-19 through BEH-25 (CLI behaviors)	These CLI test/benchmark behaviors lack explicit capability realization links, though they functionally exercise CAP-3 (pipeline) and CAP-10 (enrichment)
CAP-14 component isolation	COMP-11 (Pipeline Learning) is nested under the pipeline namespace but serves cross-cutting concerns; no explicit interface contract defines how other capabilities consume learnings
No explicit behavior definitions for CAP-6, CAP-8, CAP-11, CAP-12	These capabilities lack corresponding behavior entries in the model, indicating incomplete behavioral specification

*Document generated from architecture model. All element identifiers (CAP-, BEH-, COMP-) reference the canonical model.**

Logical Architecture

1. Architecture Overview

The system is an **architecture model extraction and documentation platform** — it scans source code, infers structural components and behaviors, validates the resulting model, and generates systems engineering documentation. The architecture follows a layered approach from foundation types through domain logic to application-level workflows and a CLI interface.

2. Layer Structure

Layer	Responsibility	Components
Infrastructure Foundation	Configuration, utilities, shared services Core types, validation, parsing, model operations	COMP-9, COMP-12 COMP-1.x
Domain	Business logic — pipeline extraction, manifest scanning, learning	COMP-2.x, COMP-3.x, COMP-6, COMP-11
Application	High-level workflows — docs, orchestration, authoring, export	COMP-4.x, COMP-5.x, COMP-7, COMP-10
Interface	User-facing CLI	COMP-8

Data flows **upward**: infrastructure supports foundation, foundation supports domain, domain supports application, and the CLI exposes application capabilities to users.

3. Component Inventory

ID	Component	Purpose	Key Responsibilities	Key Files
COMP-1.1	Type System	Canonical data model	Defines all dataclasses, enums (Component, Relationship, Behavior, etc.) used throughout	<code>core/types.py</code>
COMP-1.2	Validation	Model correctness	JSON schema validation, referential integrity checks, cycle detection, domain rule enforcement	<code>core/validator.py</code>
COMP-1.3	Parser & Persistence	I/O layer	YAML parse/serialize with round-trip preservation, model compression, multi-file merging, persistent storage	<code>core/parser.py</code> , <code>persistence/store.py</code>
COMP-1.4	Model Operations	Analytical transforms	Slice models by scope, diff two models, compute coverage metrics, impact analysis, clustering, source-block assignment	<code>core/slicer.py</code> , <code>core/differ.py</code> , <code>core/coverage.py</code>

ID	Component	Purpose	Key Responsibilities	Key Files
COMP-1.5	Quality Metrics	Model health scoring	Confidence scoring per element, regen-readiness assessment, corrections tracking, decomposition quality, visualization	core/confidence.py, core/regen_readiness.py
COMP-2.1	Pipeline Coordination	Stage orchestration	Manages 10-stage pipeline execution order, inter-stage context, caching, progress reporting, artifact collection	pipeline/coordinator.py, pipeline/cache.py
COMP-2.2	Observation Stages	Code discovery	Scans codebase (observe) and infers components/behaviors from observations (infer)	pipeline/observe.py, pipeline/infer.py
COMP-2.3	Allocation & Relation	Structure mapping	Assigns source modules to architectural blocks (allocate), discovers inter-component relationships (relate)	pipeline/allocate.py, pipeline/relate.py
COMP-2.4	Specification & Contract	Interface definition	Adds interface specifications, binds test contracts to components, runs validation passes	pipeline/specify.py, pipeline/contract.py, pipeline/validate.py
COMP-2.5	Synthesis & Emit	Final output	Decomposes large blocks, synthesizes partial results into coherent model, emits final YAML	pipeline/synthesize.py, pipeline/emit.py
COMP-3.1	Scanners	Source parsing	Language-specific AST scanning for Python, TypeScript, Kotlin; extracts symbols, metrics, body hints	manifest/scanner.py, manifest/ts_scanner.py, manifest/kt_scanner.py
COMP-3.2	Graph & Analysis	Dependency analysis	Builds call graphs, resolves imports, extracts interfaces, detects behavioral patterns, analyzes tests	manifest/call_graph.py, manifest/interfaces.py, manifest/behavior.py
COMP-3.3	Grouping & Generation	Component inference	Groups source files into logical components, generates manifest, recursive deep scanning	manifest/grouping.py, manifest/generator.py
COMP-4.1	Core Doc Generators	Technical docs	Generates component specs, ICDs, dependency matrices, health reports, drift analysis, diagrams	docs/generator.py, docs/icd.py, docs/diagrams.py
COMP-4.2	SE Document Suite	Formal SE docs	Produces 15 document types: ConOps, functional analysis, logical architecture, requirements, use cases, V&V, etc.	docs/se/generator.py, docs/se/conops.py, docs/se/logical_architectur

ID	Component	Purpose	Key Responsibilities	Key Files
COMP- 5.1	Enrichment	Model augmentation	Auto-enriches models with function signatures, constants, test contracts, capability inference, use-case inference	orchestration/enrich.py, orchestration/capability_in
COMP- 5.2	Decomposition	Model refinement	Deep-decomposes large components, extracts behavior flows, compacts redundant structures	orchestration/deep_decompos orchestration/behavior_flow
COMP- 6	Extract	Code-to-model	Detects routes, constraints, parses artifacts (tables, configs) into model elements	extract/from_code.py, extract/route_detector.py
COMP- 7	Authoring	Forward modeling	Authors models from requirements text, enforces development gate checks	authoring/parser.py, authoring/gate.py
COMP- 8	CLI	User interface	Exposes all commands (scan, validate, generate docs, export) via command-line	cli/main.py
COMP- 9	Configuration	Settings	Loads config files, manages domain profiles, defines config schemas	config/loader.py, profiles/schema.py
COMP- 10	Export	Output formats	Produces flat-file exports for AI consumption, reference documentation	export/flatfiles.py, export/reference.py
COMP- 11	Pipeline Learning	Adaptive improvement	Persists lessons across runs, extracts heuristics, applies learned patterns to future extractions	pipeline/global_learning.py, pipeline/lessons.py
COMP- 12	Utilities	Shared services	File discovery, monitoring/health checks, pattern matching, data loading	utils/discovery.py, monitoring.py

4. Component Interactions

Key data flows:

1. **CLI** → **Orchestration/Pipeline**: User commands trigger orchestration workflows or pipeline runs
2. **Pipeline Coordination** → **Pipeline Stages**: Coordinator sequences observe → infer → allocate → relate → specify → contract → validate → decompose → synthesize → emit
3. **Pipeline Stages** → **Manifest**: Observation/allocation stages invoke scanners and graph analysis to understand source code
4. **Pipeline Stages** → **Core**: All stages read/write the canonical type system, use validation, model operations
5. **Orchestration** → **Core** + **Manifest**: Enrichment and decomposition workflows combine model operations with source analysis
6. **Documentation** → **Core**: Doc generators read the validated model to produce output

7. **Extract** → **Core**: Code extractors produce model elements conforming to the type system
8. **Configuration** → **All**: Config/profiles are consumed across layers
9. **Pipeline Learning** → **Pipeline**: Lessons feed back into pipeline stage decisions

5. Dependency Analysis

Key dependency chains: - CLI → Orchestration → Pipeline → Manifest → Core Types - CLI → Documentation → Core Types + Validation - Pipeline Stages → Core Operations (slicer, differ, coverage)

Coupling concerns: - **Core Types is a universal dependency** — any change to `types.py` propagates everywhere. This is acceptable for a canonical model but requires stability. - **Pipeline stages share context** via the coordinator, creating implicit coupling through the context object. - **Manifest scanners** are language-specific but share a common protocol, keeping them decoupled from each other.

6. Design Rationale

- **Layered separation** ensures foundation stability — types and validation change rarely while application workflows evolve.
- **10-stage pipeline** provides modularity: stages can be cached, skipped, or replaced independently.
- **Manifest as separate domain component** isolates language-specific parsing complexity from architectural reasoning.
- **SE Document Suite** as a dedicated sub-component acknowledges the breadth of formal documentation needs without polluting core logic.
- **Learning component** enables the system to improve extraction quality over repeated runs without modifying stage logic directly.

7. Mermaid Dependency Graph

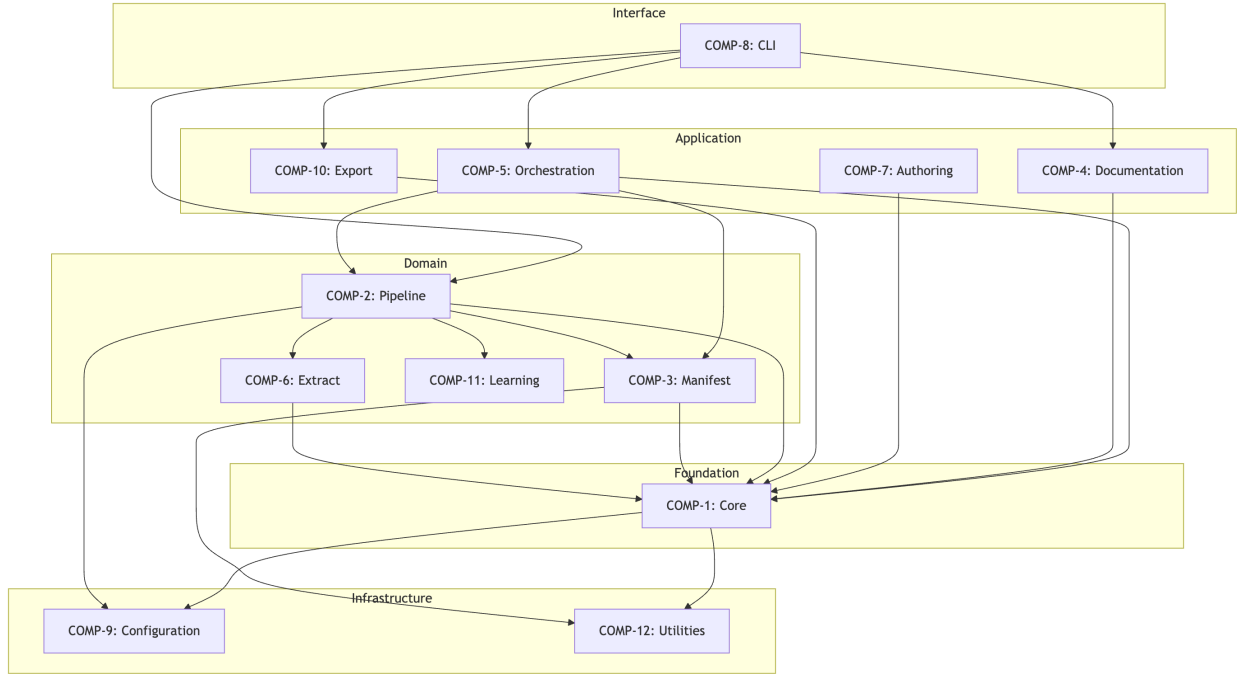


Figure 4: Diagram

Use Cases Document

1. Actor-Goal Matrix

Actor	Primary Goals
ACT-1: AI Agent (MCP Client)	Query models, retrieve documentation, consume exported artifacts, validate architecture
ACT-2: Developer	Extract architecture from code, run pipelines, manage models, generate documentation
ACT-3: CI/CD Pipeline	Validate models, check development gates, detect drift, run benchmarks

2. Use Case Catalog

UC-1: Extract Architecture from Source Code

Field	Description
Actor	Developer (ACT-2)
Preconditions	Source code repository exists; extraction pipeline is configured

Field	Description
Main Flow	1. Developer invokes CLI (BEH-24: Main) via <code>ArgumentParser → add_subparsers → add_parser → add_argument → parse_args</code> 2. System executes 10-stage pipeline (CAP-3): <code>observe → infer → allocate → relate → specify → contract → validate → decompose → synthesize → emit</code> 3. AST scanning (CAP-2) derives components, relationships, behaviors 4. Reality manifest (CAP-4) is generated with structural facts 5. Model is emitted as flat-file output (CAP-15)
Alternate Flows	A1: Source contains unparseable files → skip and log warning A2: No routes/classes found → emit empty model with warning
Postconditions	Architecture model exists with components, relationships, and behaviors populated

UC-2: Run Benchmark Test Round Trip

Field	Description
Actor	Developer (ACT-2) or CI/CD Pipeline (ACT-3)
Preconditions	Training examples available; CLI installed
Main Flow	1. Actor invokes BEH-22: <code>ArgumentParser → add_argument → parse_args → print → load_training_examples</code> 2. System parses CLI arguments 3. Training examples are loaded 4. Round-trip extraction and regeneration is performed 5. Results are printed to output
Alternate Flows	A1: Training examples missing → exit with error A2: Round-trip produces diff → report discrepancies
Postconditions	Benchmark results reported; pass/fail status determined

UC-3: Query Model via API

Field	Description
Actor	AI Agent (ACT-1)
Preconditions	System is running; model data is loaded; agent is authenticated

Field	Description
Main Flow	1. Agent sends <code>GET models/</code> (BEH-7) to list available models 2. Agent sends <code>GET ^models/(?P<app_label>[^.]+)\.(?P<model_name>[/]+)/\$</code> (BEH-8) for specific model 3. System returns model details (components, relationships) 4. Agent optionally queries <code>GET views/</code> (BEH-5) or <code>GET views/<view>/</code> (BEH-6) for filtered views 5. Agent consumes response within token budget (CAP-15)
Alternate Flows	A1: Model not found → return 404 A2: Invalid <code>app_label.model_name</code> format → return 400
Postconditions	Agent has received model data for downstream reasoning

UC-4: Validate Architecture Model

Field	Description
Actor	CI/CD Pipeline (ACT-3)
Preconditions	Architecture model exists in repository
Main Flow	1. Pipeline triggers validation (CAP-1) 2. System checks model correctness (required fields, valid references) 3. System checks completeness (all components have behaviors) 4. System checks hierarchy consistency (parent/child links valid) 5. System checks domain rules 6. System returns pass/fail with detailed report
Alternate Flows	A1: Model file corrupt → fail with parse error A2: Partial failures → report warnings, pass with caveats
Postconditions	Validation report generated; pipeline gate passes or blocks

UC-5: Check Development Gate

Field	Description
Actor	CI/CD Pipeline (ACT-3)
Preconditions	Authored architecture model exists; current code is available
Main Flow	1. Pipeline invokes gate check (CAP-12) 2. System scans current code reality (CAP-4) 3. System compares code against architecture intent (CAP-13) 4. System scores coverage and drift 5. System returns gate status (pass/warn/fail)
Alternate Flows	A1: New code has no corresponding model element → flag as undocumented A2: Model specifies component not yet implemented → report as pending

Field	Description
Postconditions	Gate result recorded; drift report available

UC-6: Generate SE Documentation

Field	Description
Actor	Developer (ACT-2)
Preconditions	Architecture model is validated and complete
Main Flow	1. Developer requests documentation generation (CAP-5) 2. System produces functional analysis from behaviors 3. System produces logical architecture from components/relationships 4. System produces use cases from actors and behaviors 5. System produces requirements traceability 6. System emits markdown documents
Alternate Flows	A1: Model incomplete → generate partial docs with gaps noted
Postconditions	Full SE documentation suite generated

UC-7: Browse Bookmarklets and Tags

Field	Description
Actor	AI Agent (ACT-1)
Preconditions	System running; data populated
Main Flow	1. Agent sends GET bookmarklets/ (BEH-2) 2. System returns available bookmarklets 3. Agent sends GET tags/ (BEH-3) 4. System returns tag taxonomy 5. Agent sends GET filters/ (BEH-4) for available filter options
Alternate Flows	A1: No bookmarklets configured → return empty list
Postconditions	Agent has metadata for navigation and filtering

UC-8: Run Guided Benchmark

Field	Description
Actor	Developer (ACT-2)
Preconditions	Benchmark suite configured; test fixtures available

Field	Description
Main Flow	1. Developer invokes BEH-19: <code>ArgumentParser</code> → <code>add_argument</code> → <code>parse_args</code> → <code>run</code> → <code>run_test_guided</code> 2. System parses arguments 3. System calls <code>run()</code> orchestrator 4. System executes <code>run_test_guided</code> with guided extraction 5. System reports results
Alternate Flows	A1: Missing arguments → display usage and exit A2: Test failure → report failure details
Postconditions	Guided round-trip benchmark completed with results logged

3. Use Case Relationships

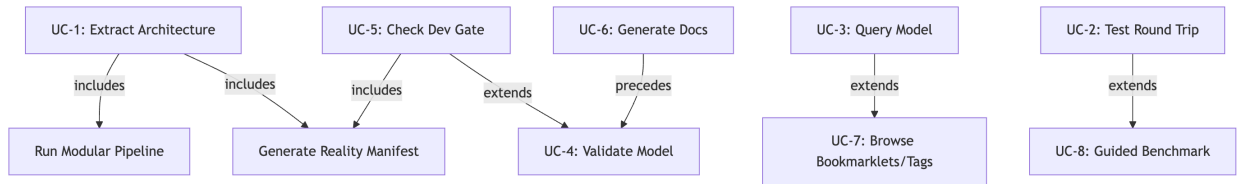


Figure 5: Diagram

Relationship	From	To	Type
UC-1 includes CAP-3	Extract Architecture	Run Modular Pipeline	Include
UC-1 includes CAP-4	Extract Architecture	Generate Reality Manifest	Include
UC-5 includes UC-4	Check Dev Gate	Validate Model	Include
UC-5 includes CAP-13	Check Dev Gate	Detect Drift	Include
UC-6 requires UC-4	Generate Docs	Validate Model	Precondition
UC-8 extends UC-2	Guided Benchmark	Test Round Trip	Extend

Artifact Traceability Map: architecture-model-standard

1. Entity Inventory

Entity Type	Count	Feeds SE Documents
Components	29	Logical Architecture, Maintenance Manual, Operations Manual, Interface Specification, Component Specs
Capabilities	15	ConOps, Functional Analysis, Requirements Analysis
Behaviors	25	Use Cases, Functional Analysis, Verification & Validation, Behavior Flows
Interfaces	43	Interface Specification, Logical Architecture
Constraints	2	Requirements Analysis, Risk Assessment
Requirements	45	Requirements Analysis, Verification & Validation
Actors	3	ConOps, Use Cases
Layers	0	Logical Architecture

2. Artifact Dependency Graph

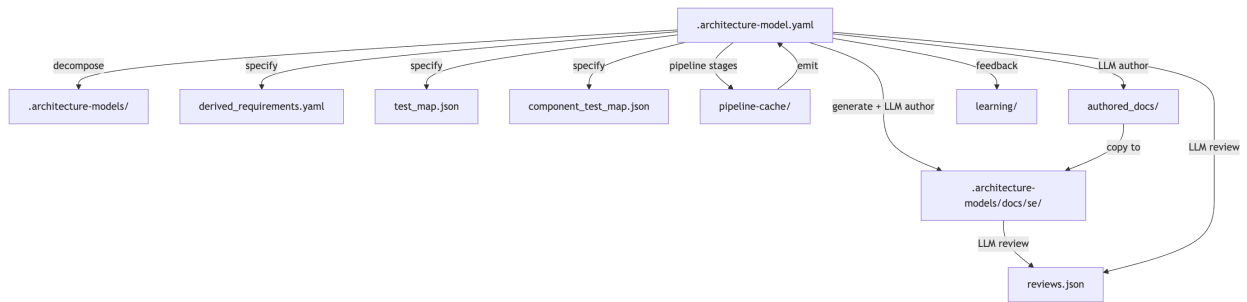


Figure 6: Diagram

3. Entity-to-Artifact Traceability Matrix

Artifact	Component	Capabilities	Behaviors	Interfaces	Constraints	Requirements	Actors	Layers
Behavior			25					
Flows								
Component	29							
Specs								
ConOps		15					3	
Functional		15	25					
Analysis								
Interface	29			43				
Specifi- cation								

Artifact	Component	Capabilities	Behaviors	Interfaces	Constraints	Requirements	Actors	Layers
Logical Architecture	29			43				—
Maintenance Manual	29							
Operations Manual	29							
Requirements Analysis		15			2	45		
Risk Assessment					2			
Use Cases			25				3	
Verification & Validation			25			45		

4. Relationship Distribution

Relationship Type	Count	Connects
satisfies	49	Component → Requirement
exposes	43	Component → Interface
depends-on	26	Component → Component
contains	18	Component → Component
realizes	15	Component → Capability

5. Traceability Gaps

- **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