

Concept of Operations: Manifest Subsystem

1. System Overview

The **Manifest** subsystem (COMP-3) performs source code scanning and structural fact extraction — the “reality” layer of the Architecture Model Standard. It parses source files via AST analysis, resolves import relationships, extracts behavioral signals, and groups modules into logical components. The output is a typed **Manifest** dataclass containing **ModuleInfo** records, **InterfaceEdge** relationships, and **MetricsResult** data.

The subsystem realizes **CAP-4: Generate Reality Manifest** and is composed of three sub-components:

Sub-component	ID	Responsibility
Scanners	COMP-3.1	Language-specific AST parsing (Python, TypeScript, Kotlin)
Graph & Analysis	COMP-3.2	Import resolution, call graphs, behavior extraction
Grouping & Generation	COMP-3.3	Component grouping, manifest assembly, recursive scanning

2. Stakeholders & Actors

Actor	Interaction
CLI (COMP-8)	Triggers manifest generation via runner CLI (IF-2)
Observe stage (COMP-2.2)	Consumes scanner output for code facts
Extract (COMP-6)	Uses scanners for deeper code analysis
Enrichment (COMP-5.1)	Reads manifest data to augment the architecture model
Gate Check (COMP-7)	Reads manifest to validate boundary coherence (REQ-19)
Config (COMP-9)	Provides exclusion patterns, functional block definitions
Developers	Consume manifest output for architecture understanding

3. Operational Scenarios

3.1 Full Project Scan

A developer runs the CLI to generate an initial architecture model. `generate_manifest(project_root)` is invoked, which:

1. Loads `ProjectConfig` from `.architecture-model.yaml` via COMP-9
2. Calls `compute_metrics(root, config)` for project-level stats
3. Iterates `config.source_block_dict`, calling `process_block()` for each functional block

4. Each block triggers `scan_file()` per Python file, producing `ModuleInfo` records with `FunctionInfo`, `ClassInfo`, and `ImportDetail` entries
5. `derive_interfaces()` resolves imports into `InterfaceEdge` objects
6. Returns a complete `Manifest` with all modules, interfaces, and metrics

Satisfies: REQ-8 (complete file scanning), REQ-9 (import edge resolution).

3.2 Incremental Scan with Caching

`ScanCache` (in `scan_cache.py`) tracks previously scanned files. On re-invocation, `generate_manifest` instantiates a `ScanCache` and skips files whose modification time hasn't changed. Only modified or new files are re-parsed, and the cached `ModuleInfo` records are merged with fresh results. This reduces wall-clock time on large repositories.

3.3 Multi-Language Scanning

`scan_all_languages(root)` in `multi_scanner.py` detects languages by file extension and dispatches to the appropriate scanner:

- `.py` → `generate_manifest()` → `SourceGraph.from_manifest()`
- `.kt` / `.java` → `scan_kotlin()` / `scan_java()` via tree-sitter
- `.ts` / `.tsx` → `scan_typescript_fallback()` via regex

Results are merged into a unified `SourceGraph` via `_merge_graphs()`. Cross-language edges are **not** resolved. JVM build directories (`build`, `.gradle`, `generated`) and `node_modules` are excluded.

Satisfies: REQ-10 (multi-language scanning).

3.4 Recursive Manifest for Subsystem Decomposition

`generate_block_manifest(root, block_id, block_def)` produces a scoped `Manifest` for a single functional block. The recursive scanner in `recursive.py`:

1. Resolves files from `block_def["dirs"]` and `block_def["files"]`
2. Deduplicates and scans each file with `scan_file()`
3. Derives interfaces scoped to the block
4. Wraps the result in a `RecursiveManifest` linked to a `component_id` via `_block_id_to_component_id()`

This enables drill-down architecture views per component.

4. System Context

External dependencies:

- **Config** (COMP-9): exclusion patterns, functional block definitions, source block dict
- **Utils** (`architecture_model.utils.discovery`): `collect_py_files()` for file discovery
- **Monitoring** (`architecture_model.monitoring`): `@monitored` decorator on `generate_manifest`
- **tree-sitter** (optional): required for Kotlin/Java scanning

5. Operational Constraints

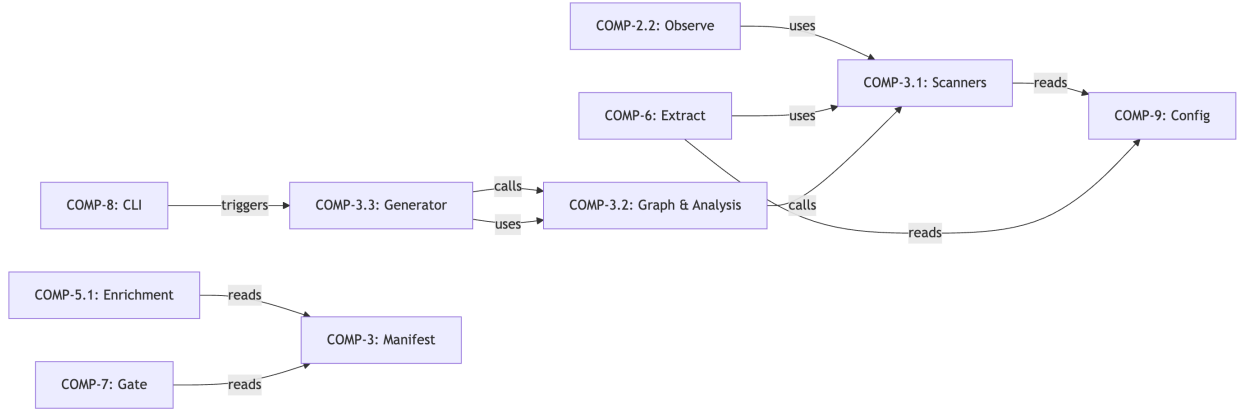


Figure 1: Diagram

Constraint	Detail
AST parse failures	Files with <code>SyntaxError</code> or <code>UnicodeDecodeError</code> fall back to regex extraction (<code>_RE_CLASS</code> , <code>_RE_FUNC</code> , <code>_RE_IMPORT</code>) and are marked <code>ModuleStatus.MISSING</code>
Trivial file exclusion	<code>__version__.py</code> , <code>__main__.py</code> , empty <code>__init__.py</code> (5 lines, no symbols), and vendor directories are filtered by <code>_is_trivial()</code>
Boundary coherence	REQ-19 requires >70% intra-component imports; validated by grouping in COMP-3.3
No cross-language edges	<code>multi_scanner.py</code> merges graphs but cannot resolve imports across language boundaries
Behavioral extraction scope	<code>behavior.py</code> performs single-function, single-file analysis only — no inter-procedural or type-inferred resolution
Builtins excluded	<code>extract_call_order</code> filters a <code>_BUILTINS</code> frozenset of 50+ names from call graphs

6. Data Flow

The pipeline flows left-to-right through the three sub-components: **Scanners** (COMP-3.1) produce `ModuleInfo` records, **Graph & Analysis** (COMP-3.2) resolves `InterfaceEdge` and `CallGraph` structures, and **Grouping & Generation** (COMP-3.3) assembles the final `Manifest` with grouped components and metrics.

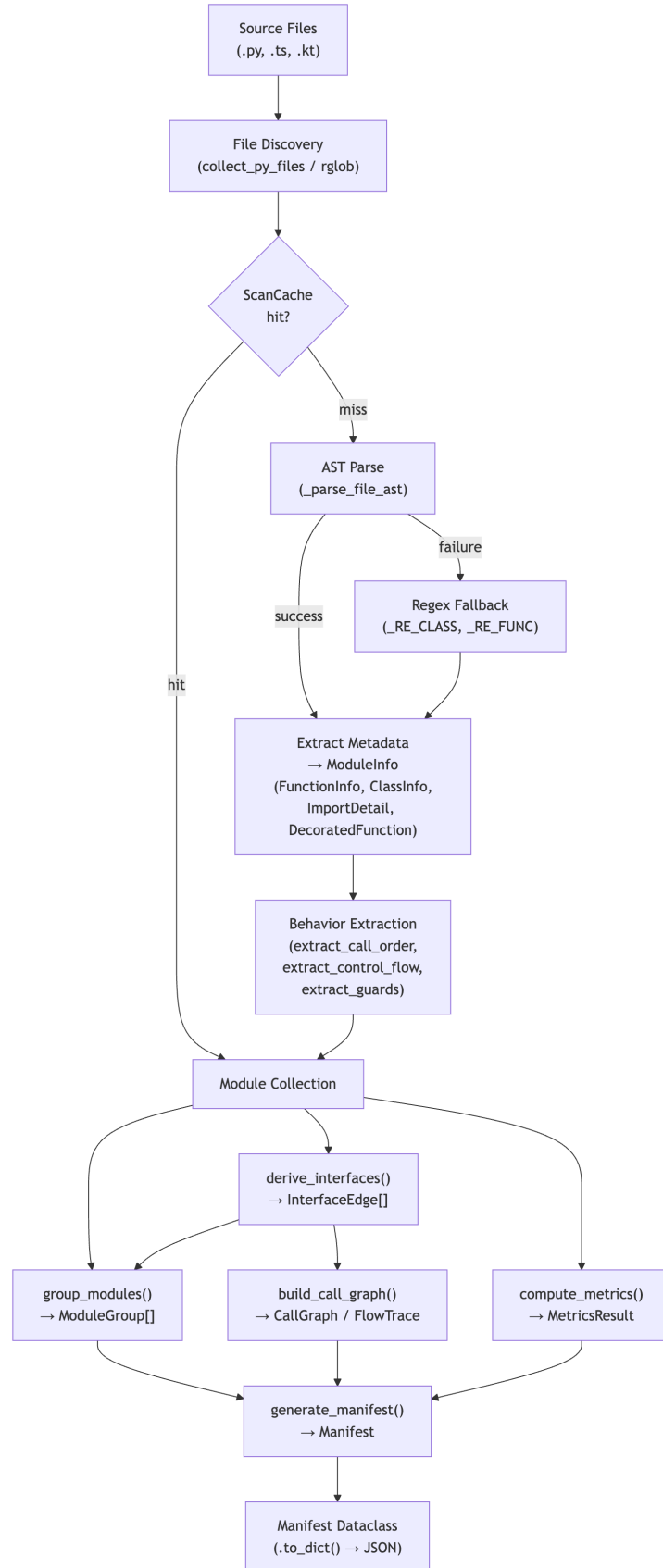


Figure 2: ₄ Diagram

Functional Analysis — Manifest Subsystem

1. Capability Inventory

CAP-4: Generate Reality Manifest decomposes into four sub-capabilities:

Sub-Capability	Description
CAP-4.1 Source Scanning	Parse source files via AST (Python) or regex/tree-sitter (TS, Kotlin) to extract FunctionInfo , ClassInfo , ImportDetail
CAP-4.2 Graph Construction	Resolve imports into edges, build call graphs, extract behavioral signals
CAP-4.3 Module Grouping	Cluster modules into logical components via directory, name-prefix, and import affinity
CAP-4.4 Manifest Assembly	Orchestrate scanning, interface derivation, metrics, and block processing into a typed Manifest

2. Functional Decomposition

COMP-3 — Manifest Core

Defines the type system: **ModuleInfo**, **FunctionInfo**, **ClassInfo**, **ImportDetail**, **DecoratedFunction**, **ModuleStatus**, **Manifest**, **ScanReport**, **MetricsResult**, **BlockManifest**, **RecursiveManifest**, **InterfaceEdge**. All pipeline stages accept and return these typed dataclasses.

COMP-3.1 — Scanners

- **scanner.py** — `scan_file(root, filepath) → ModuleInfo`. Parses Python via `ast.parse()` with regex fallback (`_RE_CLASS`, `_RE_FUNC`, `_RE_IMPORT`). Extracts functions, classes, imports, docstrings, line counts. Delegates behavioral enrichment to `behavior.py`.
- **multi_scanner.py** — `scan_all_languages(root) → SourceGraph`. Detects `.py`, `.kt`, `.java`, `.ts/.tsx` files, dispatches to language-specific scanners, merges via `_merge_graphs()`.
- **ts_scanner.py** — Regex-based TypeScript/JSX scanning.
- **kt_scanner.py** — Tree-sitter-based Kotlin/Java scanning.
- **scan_cache.py** — `ScanCache` for memoizing scan results.
- **metrics.py** — `compute_metrics(root, config) → MetricsResult`.
- **body_hints.py** — Additional heuristics for function body classification.
- **protocol.py** — Language-agnostic protocols: `SourceUnit`, `DependencyEdge`, `SourceGraph`.

COMP-3.2 — Graph & Analysis

- **call_graph.py** — `build_call_graph(manifest) → CallGraph`. Indexes all functions by qualified name (`file:func_name`), resolves import-based edges, supports multi-hop tracing via `FlowTrace`.
- **interfaces.py** — `derive_interfaces(modules, root) → list[InterfaceEdge]`. Resolves dotted import paths to file paths, deduplicates edges.

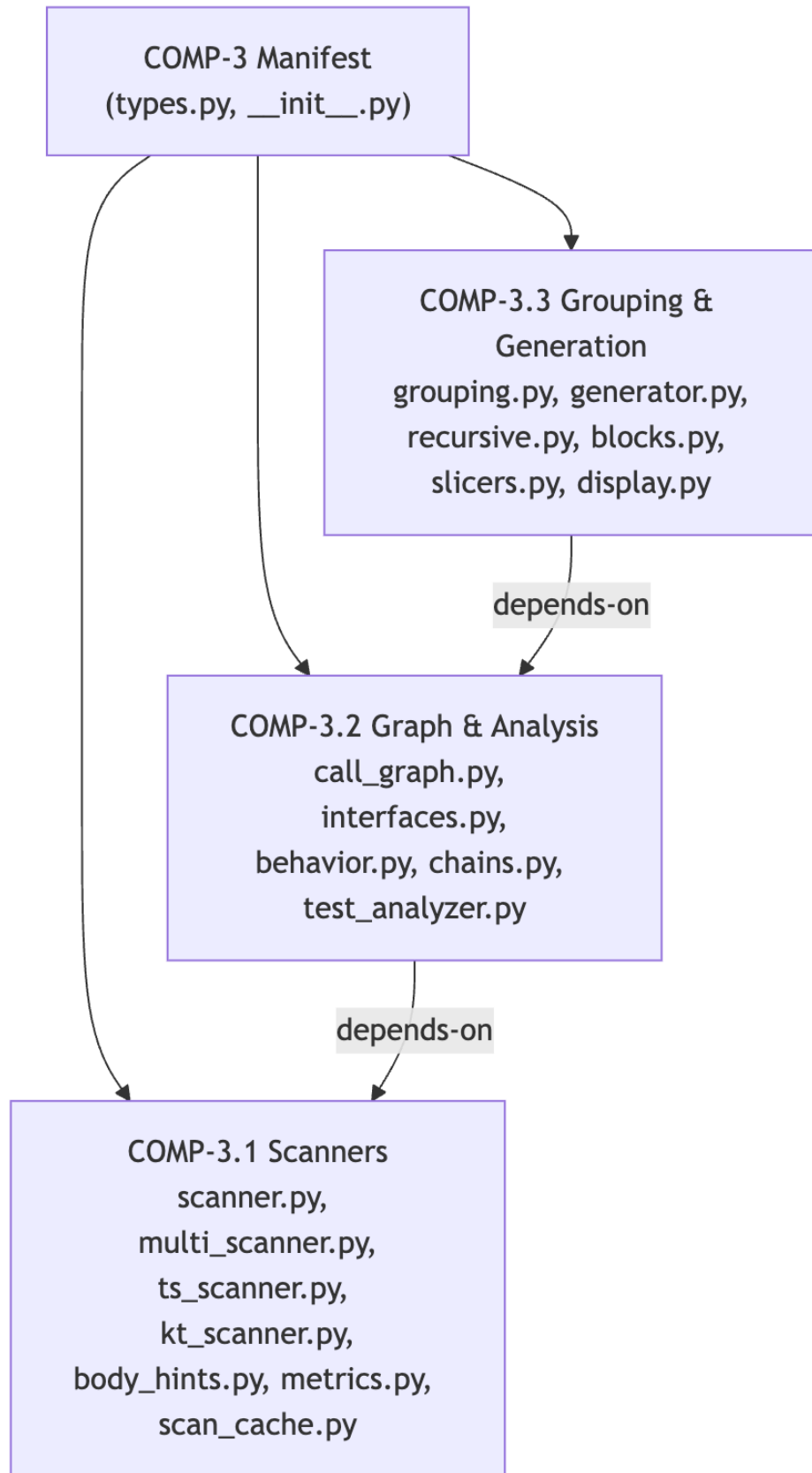


Figure 3: Diagram

- **behavior.py** — `extract_call_order()`, `extract_control_flow()`, `extract_guards()`. Walks AST bodies in execution order, filters builtins via `_BUILTINS` frozenset.
- **chains.py** — Dependency chain analysis.
- **test_analyzer.py** — Test file detection and coverage mapping.

COMP-3.3 — Grouping & Generation

- **generator.py** — `generate_manifest(project_root, config?)` → Manifest. Top-level orchestrator decorated with `@monitored`.
- **grouping.py** — `group_modules(modules, interfaces, ...)` → list[ModuleGroup]. Three-signal affinity: subdirectory, name-prefix, import density. Filters trivials via `_is_trivial()`.
- **recursive.py** — `generate_block_manifest(root, block_id, block_def)` → Manifest. Per-F-block deep scans scoped to configured dirs/files.
- **blocks.py** — `process_block()`, `_get_functional_blocks()`. Block-level processing.
- **slicers.py** — Manifest slicing utilities.
- **display.py** — Human-readable manifest formatting.

3. Capability-Component Mapping

Sub-Capability	Realizing Component	Key Functions
CAP-4.1 Source Scanning	COMP-3.1	<code>scan_file()</code> , <code>scan_all_languages()</code> , <code>scan_kotlin()</code> , <code>scan_typescript_fallback()</code>
CAP-4.2 Graph Construction	COMP-3.2	<code>build_call_graph()</code> , <code>derive_interfaces()</code> , <code>extract_call_order()</code>
CAP-4.3 Module Grouping	COMP-3.3	<code>group_modules()</code>
CAP-4.4 Manifest Assembly	COMP-3.3	<code>generate_manifest()</code> , <code>generate_block_manifest()</code> , <code>process_block()</code>

4. Behavioral Flows

4.1 `generate_manifest()` Flow

4.2 `scan_file()` Flow

4.3 `group_modules()` Flow

1. Filter trivial modules via `_is_trivial()` (excludes `__version__.py`, empty `__init__.py`, vendor dirs)
2. Compute **subdirectory affinity** — files sharing a parent directory
3. Compute **name-prefix affinity** — underscore-prefixed files in same directory
4. Compute **import affinity** — mutual import count from `InterfaceEdge` list
5. Merge signals, form `ModuleGroup` instances with `primary_file` set to largest by `line_count`

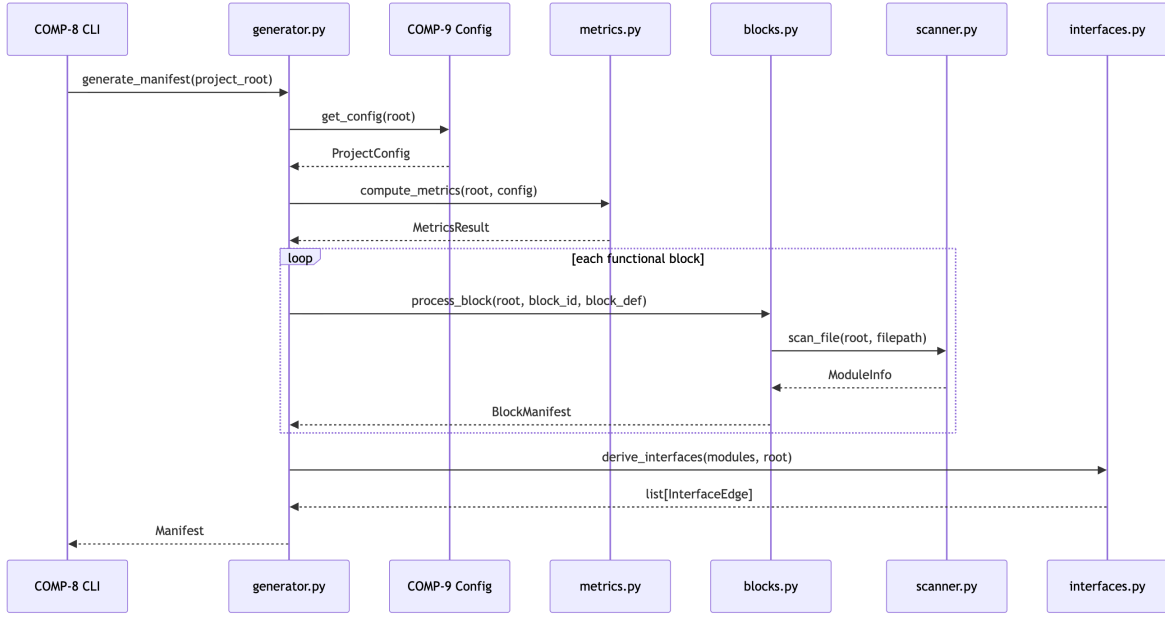


Figure 4: Diagram

4.4 build_call_graph() Flow

1. Index all `ModuleInfo.functions` by qualified name (`file:func_name`) into `CallGraph.functions`
2. Build `mod_path_to_file` mapping (file path $\rightarrow$ dotted Python path)
3. For each module, parse its `imports` list to identify imported modules
4. Resolve imported modules to files, then link caller functions to callee functions via `CallGraph.edges`
5. `FlowTrace` enables multi-hop tracing from an entry point, tracking `components_crossed` and `depth`

5. Requirements Satisfaction

Requirement	Satisfying Component	Mechanism
REQ-8 Complete file scanning	COMP-3.1	<code>scan_file()</code> processes every <code>.py</code> file discovered by <code>collect_py_files()</code> . Regex fallback handles unparseable files. <code>ScanReport</code> tracks <code>files_attempted</code> vs successful.
REQ-9 Import edge resolution	COMP-3.2	<code>derive_interfaces()</code> resolves dotted imports against the scanned module set via <code>_resolve_dotted()</code> , producing <code>InterfaceEdge</code> for every resolvable cross-module import.

Requirement	Satisfying Component	Mechanism
REQ-10 Multi-language scanning	COMP-3.1	<code>multi_scanner.scan_all_languages()</code> dispatches by extension: <code>.py</code> → AST scanner, <code>.kt/.java</code> → tree-sitter via <code>kt_scanner</code> , <code>.ts/.tsx</code> → regex via <code>ts_scanner</code> . All produce <code>SourceGraph</code> .
REQ-19 Boundary coherence	COMP-3.3	<code>group_modules()</code> uses import affinity to maximize intra-group edges. The >70% threshold is validated by comparing intra-component imports against total edges per <code>ModuleGroup</code> .

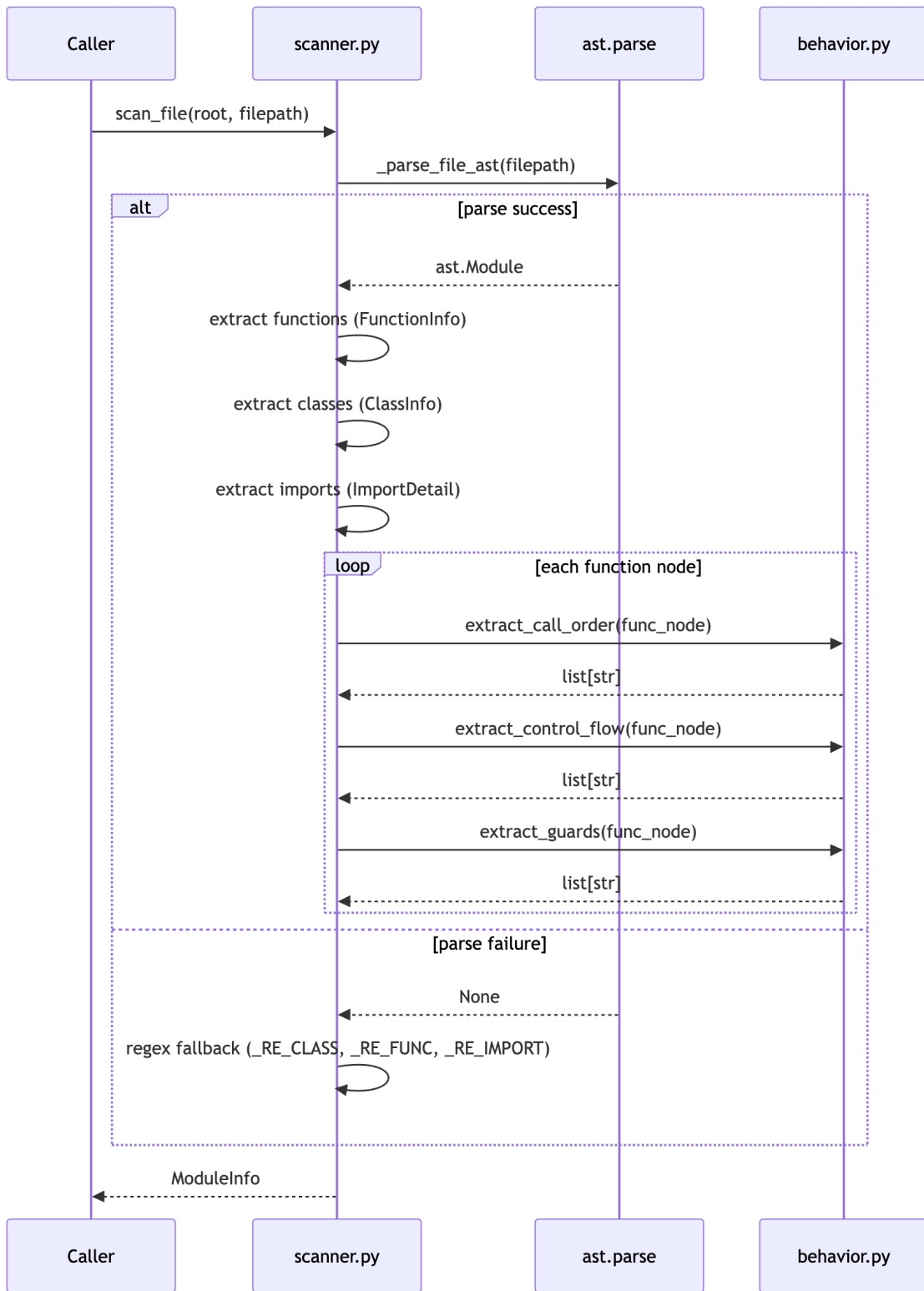


Figure 5: Diagram

Logical Architecture — Manifest Subsystem (COMP-3)

Overview

The Manifest subsystem performs source code scanning via AST analysis, import resolution, and module grouping to produce a typed structural representation of a codebase. It supports Python, TypeScript, Kotlin, and Java, and serves as the primary fact-extraction layer for the Architecture Model Standard.

Component Structure

All components reside in the **domain** layer and are contained within COMP-3.

Component	Responsibility	Key Files
COMP-3.1 Scanners	Language-specific AST parsing, file discovery, metrics computation, scan caching	<code>scanner.py</code> , <code>multi_scanner.py</code> , <code>ts_scanner.py</code> , <code>kt_scanner.py</code> , <code>scan_cache.py</code> , <code>protocol.py</code>
COMP-3.2 Graph & Analysis	Import resolution into edges, call graph construction, behavioral extraction, interface derivation	<code>call_graph.py</code> , <code>interfaces.py</code> , <code>behavior.py</code> , <code>chains.py</code> , <code>test_analyzer.py</code>
COMP-3.3 Grouping & Generation	Multi-signal module grouping, manifest generation/caching, recursive per-block scanning	<code>grouping.py</code> , <code>generator.py</code> , <code>recursive.py</code> , <code>blocks.py</code> , <code>slicers.py</code>

Dependency Graph

Internal flow: Scanners produce `ModuleInfo` lists → Graph & Analysis resolves imports into `InterfaceEdge` and builds `CallGraph` → Grouping & Generation assembles the final `Manifest`.

External consumers: CLI (COMP-8) triggers generation, Gate (COMP-7) and Enrichment (COMP-5.1) read manifest output, Observe (COMP-2.2) and Extract (COMP-6) use scanners directly. Scanners depend on Config (COMP-9) for exclusion patterns.

Interface Specification

Interface	Exposed By	Type	Description
IF-2 runner CLI	COMP-3	internal	CLI entry point for manifest generation
IF-4 Library API	COMP-3	internal	5 public symbols from <code>__init__.py</code> (e.g., <code>generate_manifest</code> , <code>scan_file</code>)

Interface	Exposed By	Type	Description
IF-auto-COMP-3.1 Scanners API	COMP-3.1	internal	<code>scan_file()</code> , <code>scan_all_languages()</code> , <code>ScanCache</code> , <code>compute_metrics()</code> , <code>SourceGraph/SourceUnit</code> protocol types
IF-auto-COMP-3.2 Graph & Analysis API	COMP-3.2	internal	<code>derive_interfaces()</code> , <code>build_call_graph()</code> , <code>extract_call_order()</code> , <code>extract_control_flow()</code> , <code>extract_guards()</code>
IF-auto-COMP-3.3 Grouping & Generation API	COMP-3.3	internal	<code>generate_manifest()</code> , <code>generate_block_manifest()</code> , <code>group_modules()</code> , <code>process_block()</code>

Key Data Types

All types are defined as typed dataclasses in `manifest/types.py` and related modules.

Type	Module	Purpose
Manifest	<code>types.py</code>	Top-level output: modules, interfaces, metrics, functional blocks
ModuleInfo	<code>types.py</code>	Per-file scan result: functions, imports, classes, line count, <code>ModuleStatus</code>
FunctionInfo	<code>types.py</code>	Function name, signature, calls, docstring, raises, plus behavioral fields (<code>call_order</code> , <code>control_flow</code> , <code>guards</code> , <code>data_in</code> , <code>data_out</code>)
ClassInfo	<code>types.py</code>	Class with bases, methods, decorators, attributes, <code>method_details</code> : <code>list[FunctionInfo]</code>
ImportDetail	<code>types.py</code>	Structured import: module, symbols, <code>is_relative</code>
InterfaceEdge	<code>types.py</code>	Directed edge: <code>source</code> → <code>target</code> file with <code>import_path</code>
CallGraph / FlowTrace	<code>call_graph.py</code>	Resolved call edges and multi-hop flow traces with component crossing detection
ModuleGroup	<code>grouping.py</code>	Logical group: name, member file paths, <code>primary_file</code>
BlockManifest	<code>types.py</code>	Per-functional-block manifest with sub-functions

Type	Module	Purpose
<code>RecursiveManifest</code>	<code>types.py</code>	Deep per-block scan linked to parent model via <code>component_id</code>
<code>ScanReport</code>	<code>types.py</code>	Bookkeeping: <code>files_attempted</code> , <code>blocks_processed</code>
<code>SourceGraph</code> / <code>SourceUnit</code> / <code>DependencyEdge</code>	<code>protocol.py</code>	Language-neutral protocol types for multi-language scanning

Requirements Traceability

Requirement	Satisfied By	Mechanism
REQ-8 Complete file scanning	COMP-3.1	<code>scan_file()</code> + <code>collect_py_files()</code> recursively covers all non-trivial Python files
REQ-9 Import edge resolution	COMP-3.2	<code>derive_interfaces()</code> resolves dotted paths and relative imports to <code>InterfaceEdge</code>
REQ-10 Multi-language scanning	COMP-3.1	<code>scan_all_languages()</code> dispatches to Python/TypeScript/Kotlin/Java scanners
REQ-19 Boundary coherence	COMP-3.3	<code>group_modules()</code> uses directory, name-prefix, and import affinity signals

Design Decisions

1. **Typed dataclasses over raw dicts.** All pipeline functions accept and return typed objects (`ModuleInfo`, `Manifest`, etc.), enforced by the `types.py` contract: *“Every function in the manifest pipeline should accept and return typed objects, not raw dicts.”*
2. **Regex fallback scanning.** When `ast.parse()` fails (syntax errors, encoding issues), `scanner.py` falls back to regex patterns (`_RE_CLASS`, `_RE_FUNC`, `_RE_IMPORT`) to extract partial structural facts rather than producing nothing.
3. **Scan caching.** `ScanCache` in COMP-3.1 avoids re-parsing unchanged files across incremental runs, instantiated per `generate_manifest()` invocation.
4. **Multi-signal grouping.** `group_modules()` combines three affinity signals — subdirectory structure, name-prefix patterns, and import density — with a trivial-file filter (`_is_trivial`) that excludes `__version__.py`, empty `__init__.py`, and vendored code.
5. **Language-neutral protocol.** `SourceGraph`/`SourceUnit`/`DependencyEdge` in `protocol.py` provide a common representation that `multi_scanner.py` merges across languages, even though cross-language edges are not resolved.

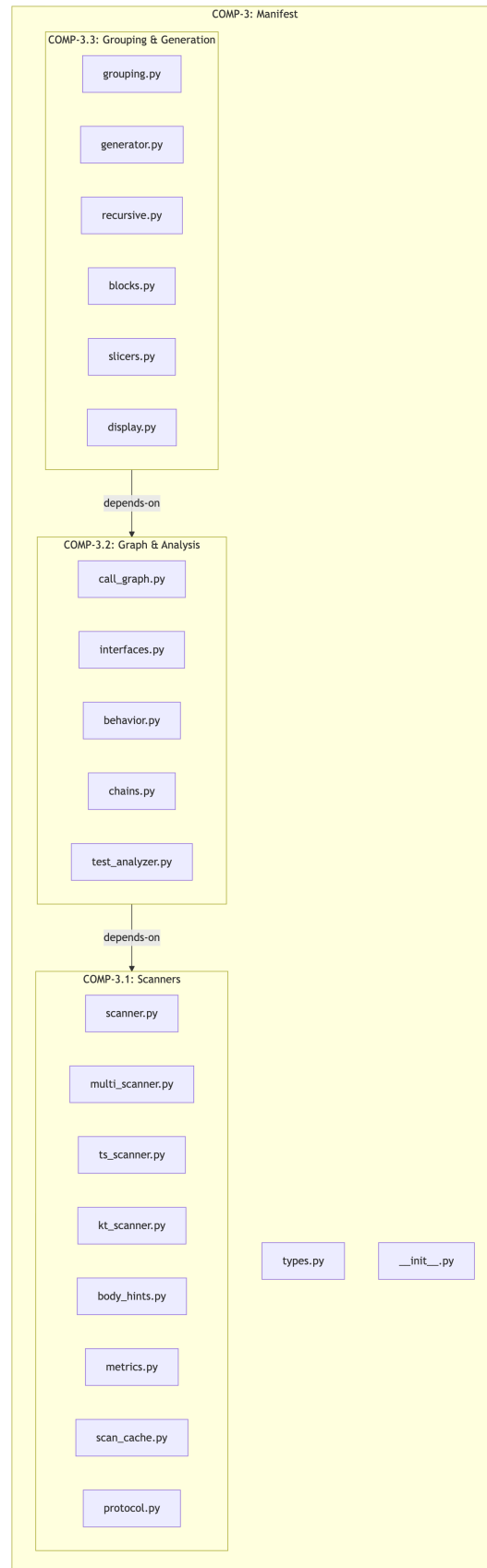


Figure 6: Diagram

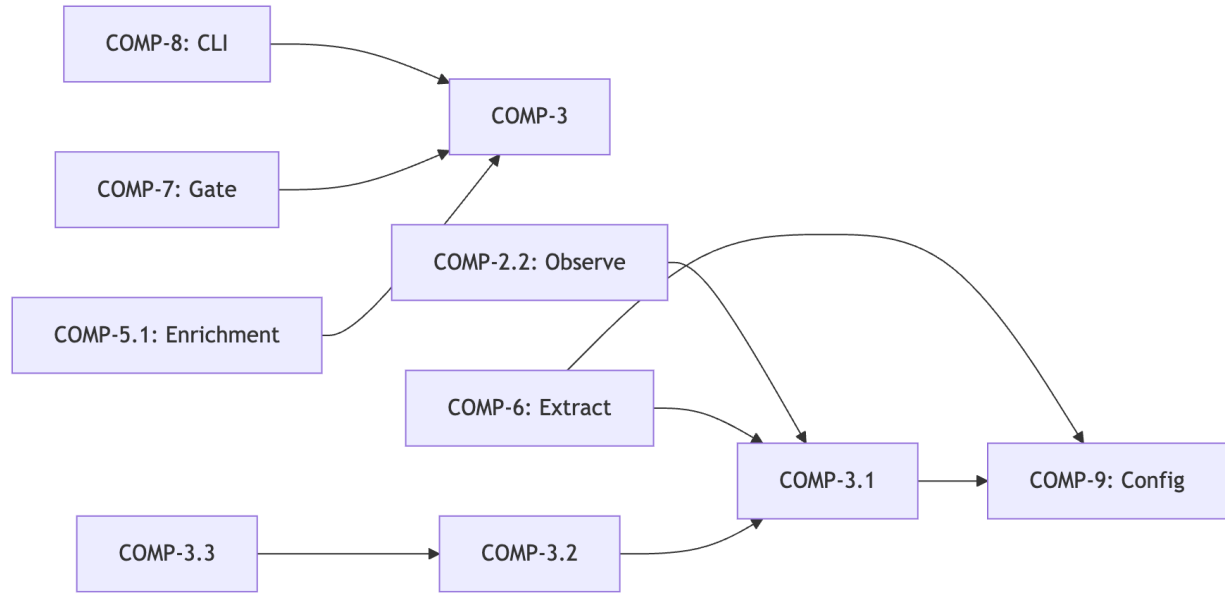


Figure 7: Diagram

6. **Recursive per-block manifests.** `generate_block_manifest()` produces scoped `Manifest` instances per functional block, linked to the architecture model via `_block_id_to_component_id()`, enabling component-level drill-down.

Use Cases — Manifest Subsystem

Use Case Diagram

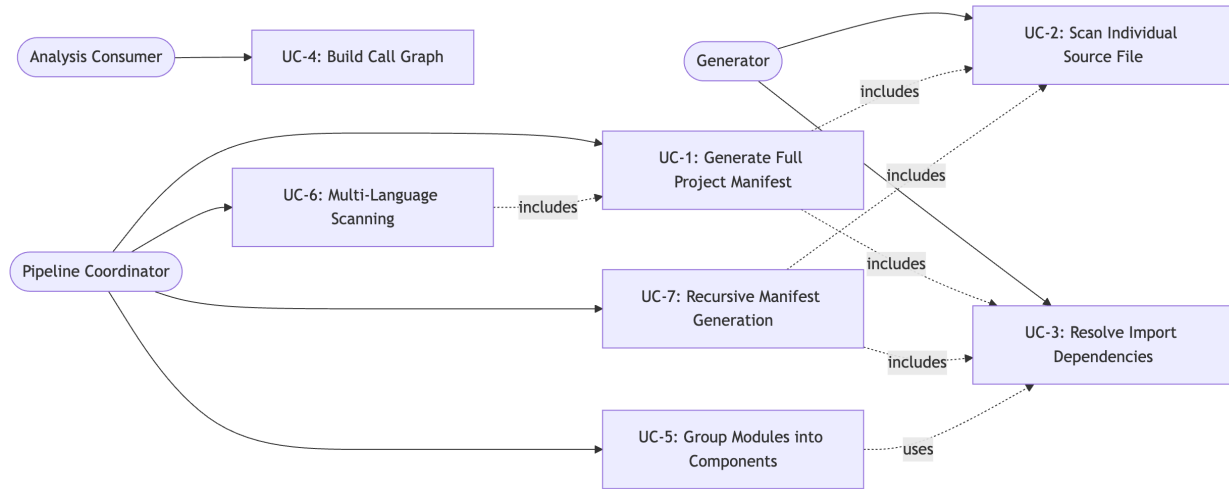


Figure 8: Diagram

UC-1: Generate Full Project Manifest

Field	Value
Actor	Pipeline Coordinator (COMP-2.2 Observe stage, COMP-8 CLI)
Capability Satisfies	CAP-4 — Generate Reality Manifest REQ-8 (complete file scanning), REQ-9 (import edge resolution)

Preconditions - `project_root` is a valid directory containing source files. - `.architecture-model.yaml` config is loadable (or passed explicitly).

Main Flow 1. Actor invokes `generate_manifest(project_root, config)` in `generator.py`.
2. If `config` is `None`, `get_config(root)` loads project configuration (COMP-9 dependency).
3. `compute_metrics(root, config)` produces a `MetricsResult` with project-level statistics.
4. `config.source_block_dict` is read to enumerate functional blocks. 5. For each block, `process_block(root, block_id, block_def, sub_block_configs)` is called, which internally triggers **UC-2** for each file. 6. `derive_interfaces(modules, root)` resolves import edges (**UC-3**). 7. A `Manifest` dataclass is assembled with `modules`, `interfaces`, `metrics_result`, and `functional_blocks`. 8. The `@monitored` decorator logs `module_count` and `parse_failures`.

Postconditions - Returns a typed `Manifest` object. Consumers call `.to_dict()` for serialization.
- `ScanReport` tracks `files_attempted` and `blocks_processed`.

Error Handling - Individual file parse failures are counted (`ModuleStatus.MISSING`) but do not abort generation. - Config load failures propagate as exceptions.

UC-2: Scan Individual Source File

Field	Value
Actor	Generator (internal to COMP-3.1 Scanners)
Satisfies	REQ-8, REQ-10

Preconditions - `filepath` exists and is a supported source file (`.py`). - `root` path is provided for relative path computation.

Main Flow 1. `scan_file(root, filepath)` is called from `scanner.py`. 2. `_parse_file_ast(filepath)` reads the file and calls `ast.parse()`. 3. AST visitors extract: - `FunctionInfo` objects (name, signature, calls, docstring, raises). - `ClassInfo` objects (name, bases, methods, decorators, attributes, `method_details`). - `ImportDetail` entries (module, symbols, `is_relative`). - `DecoratedFunction` entries for non-trivial decorators. 4. `extract_call_order()`, `extract_control_flow()`, and `extract_guards()` from `behavior.py` populate behavioral fields on each `FunctionInfo`. 5. Line count determines `ModuleStatus` (ACTIVE / DORMANT / MISSING). 6. A `ModuleInfo` is returned with all extracted metadata.

Postconditions - A fully populated `ModuleInfo` dataclass with typed fields.

Error Handling - `SyntaxError`, `UnicodeDecodeError`, `OSError` in `_parse_file_ast()` → logs warning, returns `None`. - Regex fallback scanners (`_RE_CLASS`, `_RE_FUNC`, `_RE_IMPORT`) extract partial data when AST parsing fails.

UC-3: Resolve Import Dependencies

Field	Value
Actor	Generator (COMP-3.2 Graph & Analysis)
Satisfies	REQ-9 — all resolvable imports produce edges

Preconditions - A list of `ModuleInfo` objects from scanning is available. - `root` path for relative import resolution.

Main Flow 1. `derive_interfaces(modules, root)` is called from `interfaces.py`. 2. A `file_to_module` map is built: file path → dotted module name (e.g., `src/foo/bar.py` → `src.foo.bar`). 3. The inverse `module_to_file` map is constructed. 4. For each `ModuleInfo`, simple imports are matched against known modules (Pass 1). 5. `_resolve_dotted(dotted)` handles prefix stripping (`src/`), `__init__` package resolution, and direct slash-based matching. 6. Deduplication via `seen: set[tuple[str, str]]` prevents duplicate edges. 7. Each resolved dependency produces an `InterfaceEdge(source, target, import_path)`.

Postconditions - Returns `list[InterfaceEdge]` representing all inter-module dependencies.

Error Handling - Unresolvable imports (external packages) are silently skipped — no edge created.

UC-4: Build Call Graph

Field	Value
Actor	Analysis Consumer (COMP-5.1 Enrichment, COMP-6 Extract)

Preconditions - A `Manifest` object with populated `modules` and `FunctionInfo.calls`.

Main Flow 1. `build_call_graph(manifest)` in `call_graph.py` is invoked. 2. All functions are indexed by qualified name (`file:func_name`) into `CallGraph.functions`. 3. A `mod_path_to_file` map is built from `ModuleInfo.file` paths (stripping `.py`, handling `__init__`). 4. For each module, imports are parsed to determine `imported_files`. 5. Edges are resolved: each call in `FunctionInfo.calls` is matched against functions in imported modules. 6. The resulting `CallGraph` contains `edges: dict[str, list[str]]`, `functions`, and `locations`. 7. Consumer calls `trace_flow()` with an entry point to produce a `FlowTrace` (BFS via `deque`), yielding `steps`, `components_crossed`, and `depth`.

Postconditions - `CallGraph` with resolved edges. `FlowTrace` includes `truncated: bool` for depth-limited traces.

Error Handling - Ambiguous function names (multiple modules) are resolved to all candidates in `name_index`.

UC-5: Group Modules into Components

Field	Value
Actor	Pipeline Coordinator (allocate stage)
Satisfies	REQ-19 — boundary coherence > 70% intra-component imports

Preconditions - `list[ModuleInfo]` and `list[InterfaceEdge]` from UC-2/UC-3. - Optional `target_groups` and `min_group_size` parameters.

Main Flow 1. `group_modules(modules, interfaces, target_groups, min_group_size)` in `grouping.py` is called. 2. Trivial files are filtered via `_is_trivial(mod): __version__.py`, empty `__init__.py`, vendored code, modules with zero functions/classes. 3. **Signal 1 — Sub-directory affinity**: files in the same directory are grouped. 4. **Signal 2 — Name-prefix affinity**: underscore-prefixed files in the same directory cluster together. 5. **Signal 3 — Import affinity**: `InterfaceEdge` data identifies high mutual-import pairs. 6. Groups are assembled as `ModuleGroup(name, modules, primary_file)` where `primary_file` is the largest by `line_count`.

Postconditions - Returns `list[ModuleGroup]`. Each group maps to a potential `Component`.

Error Handling - If all modules are trivial, returns an empty list. - `min_group_size` prevents degenerate single-file groups when not desired.

UC-6: Multi-Language Scanning

Field	Value
Actor	Pipeline Coordinator
Satisfies	REQ-10 — Python, TypeScript, Kotlin support

Preconditions - Repository root contains source files in one or more supported languages.

Main Flow 1. `scan_all_languages(root)` in `multi_scanner.py` is invoked. 2. Language detection by file extension: - `.py` → calls `generate_manifest(root)` then `SourceGraph.from_manifest()` (UC-1). - `.kt` → calls `scan_kotlin(kt_root)` from `kt_scanner.py` (tree-sitter based). - `.java` → calls `scan_java(java_root)` from `kt_scanner.py`. - `.ts/.tsx` → calls `scan_typescript_fallback(root)` from `ts_scanner.py` (regex-based). 3. JVM files in `_JVM_EXCLUDE` directories (`build`, `.gradle`, `generated`, etc.) are filtered. 4. `_find_jvm_source_root()` locates the appropriate source root for Kotlin/Java. 5. `_merge_graphs(graphs)` combines all `SourceGraph` instances into a unified graph.

Postconditions - Returns a merged `SourceGraph` with `SourceUnit` and `DependencyEdge` entries across all languages. - Cross-language edges are **not** resolved.

Error Handling - `ImportError` for tree-sitter → Kotlin/Java scanning silently skipped. - Any scanner exception → that language is skipped, others proceed.

UC-7: Recursive Manifest Generation

Field	Value
Actor	Pipeline Coordinator (decompose stage)
Component	COMP-3.3 Grouping & Generation

Preconditions - An `ArchitectureModel` or config with `functional_blocks` defined. - Each block specifies `dirs` and/or `files`.

Main Flow 1. `generate_block_manifest(root, block_id, block_def)` in `recursive.py` is called per F-block. 2. Files are collected from `block_def["dirs"]` via `collect_py_files(dir_path)` and `block_def["files"]`. 3. Deduplication by resolved path ensures no file is scanned twice. 4. Each file is scanned via `scan_file(root, filepath)` (UC-2), populating modules. 5. `derive_interfaces(modules, root)` resolves intra-block import edges (UC-3). 6. `_block_id_to_component_id(block_id, config, model)` maps the block to a `Component.id` — preferring model lookup over naming convention. 7. A `RecursiveManifest` is assembled, linked to its parent model via `component_id`.

Postconditions - Returns a `Manifest` (or `RecursiveManifest`) scoped to the block's files with full function-level detail. - `ScanReport.files_attempted` reflects the block's file count.

Error Handling - Missing directories are silently skipped (`if dir_path.is_dir()`). - Missing files are skipped (`if fp.is_file()`). - Individual scan failures are handled per UC-2 error handling.

Artifact Traceability Map: architecture-model-standard / Manifest

1. Entity Inventory

Entity Type	Count	Feeds SE Documents
Components	4	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	5	Interface Specification, Logical Architecture
Constraints	0	Requirements Analysis, Risk Assessment
Requirements	4	Requirements Analysis, Verification & Validation
Actors	0	ConOps, Use Cases
Layers	0	Logical Architecture

2. Artifact Dependency Graph

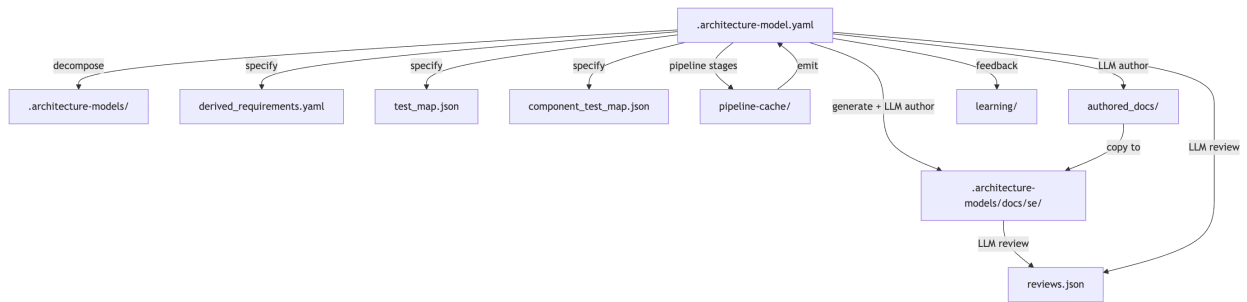


Figure 9: Diagram

3. Entity-to-Artifact Traceability Matrix

Artifact	Component	Capabilities	Behaviors	Interfaces	Constraints	Requirements	Actors	Layers
Behavior			—					
Flows								
Component	4							
Specs								
ConOps		1					—	
Functional		1	—					
Analysis								
Interface	4			5				
Specifi- cation								

Artifact	Component	Capabilities	Behaviors	Interfaces	Constraints	Requirements	Actors	Layers
Logical Architecture	4			5				—
Maintenance Manual	4							
Operations Manual	4							
Requirements Analysis		1			—	4		
Risk Assessment					—			
Use Cases			—				—	
Verification & Validation			—			4		

4. Relationship Distribution

Relationship Type	Count	Connects
depends-on	10	Component → Component, Component → Unknown, Unknown → Component, Unknown → Unknown
exposes	5	Component → Interface
satisfies	4	Component → Requirement
contains	3	Component → Component
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